03099

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

"ALGORITMOS GENÉRICOS: HERENCIA Y POLIMORFISMO EN EL ANÁLISIS DE ALGORITMOS"

T E S I S

QUE PARA OBTENER EL GRADO DE:

DOCTORA EN CIENCIAS (COMPUTACIÓN)

P R E S E N T A:

ELISA VISO GUROVICH

DIRECTOR DE TESIS:

SERGIO RAJSBAUM GORODEZKY

México, D. F. 2005.

m340637

Universidad Nacional Autónoma de México

Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Autorizo a la Dirección General de Bibliotecas de la
[illegible] impreso el
contenido de mi trabajo recepcional.
NOMBRE: ELISA VISO GUROVICH
FECHA: ENERO 27, 2005
FIRMA: [firma]

ESTA TESIS NO SALE
DE LA BIBLIOTECA

Agradecimientos:

Deseo agradecer profundamente al Dr. Sergio Rajsbaum la oportunidad que me brindó cuando aceptó dirigirme este trabajo, de aprender de él tantos y tan variados aspectos de las ciencias de la computación, y en particular de algoritmos.

Con la Dra. Atocha Aliseda quedaré eternamente en deuda por haberme impulsado a buscar el grado, y por su supervisión siempre rigurosa pero amistosa; por estar siempre presente y atenta.

Al Dr. Jorge Urrutia le agradezco su participación en mi comité tutoral, las llamadas de atención que me hizo, siempre correctas y respetuosas, para que lograra un mejor desarrollo de mi trabajo; aprecio sobre todo sus opiniones respecto al mismo, que de muchas maneras ampliaron mi perspectiva en el tema.

A todo el jurado de mi trabajo, formado por, además de los tres miembros de mi comité tutoral arriba mencionados, los doctores Ángel Kuri, Guillermo Morales Luna, Pablo Barrera y Criel Merino, antes que nada les pido una disculpa por haberlos sometido a revisar un trabajo tan extenso. Deseo también agradecerles su cuidadosa revisión que sin duda ayudó a que el trabajo mejorara.

A mi esposo, Mario Magidin, quien ha sabido ser un extraordinario compañero, y durante toda nuestra vida como pareja siempre me ha impulsado a querer ser más. Con todo mi amor para él, cuya presencia en mi vida fue elemento determinante para que haya logrado llegar a este momento.

A mi hijo Arturo Magidin, quien aunque pudiera parecer paradójico, ha sido un ejemplo para mí de vida académica productiva y seria. Le agradezco enormemente sus palabras de aliento.

A mi "pequeña" Mónica (Magidin), mujer fuerte, capaz, dulce y excelente madre. Su (no siempre justificada) confianza en mí fue un factor muy importante en que no cejara yo de alcanzar esta meta.

A ellos, los tres integrantes de mi familia, por motivarme a no seguir siendo la única sin doctorado en mi familia.

Este trabajo fue apoyado por una beca de la Dirección General de Asuntos del Personal Académico, Programa de Superación del Personal Académico.

Presentación

Motivación

Una de las tareas fundamentales de las ciencias de la computación es la *abstracción* de problemas. Desde que se concibió la programación estructurada, ha habido un esfuerzo constante por definir patrones de proceso que ayuden a entender, usar e implementar distintos tipos de soluciones. Por ejemplo, en el área de ingeniería de software en 1994 se presenta la idea de *patrones de software* en [GHJV94], que pretenden agrupar maneras de diseñar sistemas, asociándolos a los problemas que se pretenden resolver. En la enseñanza de programación, sobre todo con el lenguaje Scheme, tenemos libros que enseñan a programar partiendo de abstracciones, para especializar después esas abstracciones a problemas particulares [ASS96, FFFK01]. Esto no sólo se aplica a la programación. En realidad, desde sus inicios las ciencias de la computación han estado preocupadas por catalogar y clasificar los distintos problemas del área, bajo abstracciones comunes a una cierta clase.

Simultáneamente, el desarrollo en los paradigmas de computación, fundamentalmente con la Orientación a Objetos, también lleva hacia definir los problemas en términos de abstracciones, clasificando las herramientas para su solución en un esquema jerárquico, donde en la base de esa jerarquía no se presenta más que una idea general de *qué* es lo que se debe hacer, dejando para niveles inferiores en la jerarquía le especificación precisa de *cómo* y *con qué* hacerlo. Uno de los productos directos del análisis y diseño orientado a objetos es la clasificación de aplicaciones y herramientas en conjuntos llamados precisamente *clases*. Estas clases pueden representar superconjuntos de otras, o bien conjuntos con una cierta intersección con otros conjuntos. En cualquiera de los casos, una clase representa atributos y comportamiento iguales entre los objetos que pertenecen a ella.

Este esfuerzo por clasificar aplicaciones ha estado dirigido, fundamentalmente, hacia las estructuras de datos sobre las que trabajan los algoritmos, y es una consecuencia natural del concepto de estructuras abstractas de datos. Se establecen propiedades para los datos abstractos y se verifica que las distintas implementaciones cumplan con esas propiedades. También ha habido, en orientación a objetos, un esfuerzo por desarrollar aplicaciones o programas que puedan trabajar a nivel de la estructura abstracta, y que al concretar estas estructuras el desempeño del programa así obtenido sea tan eficiente como el programa original y se pueda aplicar a distintos tipos de datos con una abstracción común.

Sin embargo estos esfuerzos no se han reflejado en el campo de algoritmos, particularmente en lo que corresponde al análisis de los mismos.

Contribución

En este trabajo presentamos una manera de abstraer algoritmos que se apoya en la orientación a objetos, y en los conceptos de herencia y polimorfismo. La herencia nos habla de la construcción de nuevas clases a partir de una superclase, de tal manera que las nuevas clases comparten (heredan) los atributos y el comportamiento de la superclase. La herencia de comportamiento es un mecanismo mediante el cual a nivel de la superclase se especifica cuál va a ser la respuesta de la invocación de un cierto método, y a nivel de la subclase, siempre y cuando se cumpla con esta respuesta, la implementación del método puede cambiar, ya sea para tratar de manera distinta las estructuras heredadas, o para hacer trabajo adicional sobre nuevos atributos que la superclase no proporcionaba. El polimorfismo se refiere a las distintas formas (implementaciones) que toma en las subclases el método definido a nivel de la superclase.

Utilizamos la orientación a objetos para clasificar algoritmos, de tal manera que esta clasificación conforme familias de algoritmos. Cada familia corresponde a un árbol jerárquico, en términos de la herencia presente en la orientación a objetos. En la raíz de la jerarquía tenemos a lo que denominamos un *algoritmo genérico*, que describe, en un método que no se puede redefinir, una estrategia común a los algoritmos de la familia, y que invoca a métodos abstractos; lo más importante de nuestro trabajo es que esta superclase presenta propiedades no triviales respecto a la ejecución del algoritmo genérico, esto es, aun a este nivel abstracto contempla invariantes que se deben cumplir en la ejecución del algoritmo, y que de respetarse en la implementación de los métodos, garantizan que el objetivo del algoritmo se cumplirá. Asimismo, el algoritmo genérico nos permite calcular la clase de complejidad a la que pertenece la familia de algoritmos, como función del costo que presentan los métodos concretos invocados en las especializaciones.

La contribución principal de este trabajo es el uso de la herencia y el polimorfismo de la orientación a objetos en el establecimiento de la correctez y el cálculo de complejidad de cada uno de los algoritmos presentes en una familia. Se establecen propiedades para la superclase de la familia en términos de: "si la implementación de este método cumple con la propiedad p, entonces la ejecución de la estrategia general, en este punto, cumplirá con la invariante q". En otras palabras, establecemos la correctez de la estrategia planteada en la superclase sujeta a que cada una de los métodos invocados desde ella cumpla con determinadas propiedades. De esta manera, heredamos la demostración de la superclase, y para cada especialización, que corresponde a una clase que herede de la superclase, únicamente se tiene que demostrar que los métodos, definidos como abstractos en la superclase y concretados (implementados) en la subclase, cumplan con las propiedades *requeridas* de ellos.

Esta forma de abstraer familias de algoritmos permite un manejo más *económico* de las demostraciones de correctez, ya que permite la reutilización de las mismas; reduce la complejidad de la demostración de un algoritmo particular pues en el marco de la demostración para la superclase quedaron demostradas ya propiedades generales y, para demostrar la correctez de una subclase, únicamente tenemos que verificar los métodos que se concretaron en la especialización.

No es posible calcular de manera precisa la clase de complejidad a la que un algoritmo abstracto pertenece, ya que muchos aspectos de la implementación quedan, precisamente,

a un nivel abstracto. Pero de la misma manera en que trabajamos en las demostraciones de correctez bajo la premisa *requiere/proporciona*, en el caso del cálculo de la complejidad podemos dar una fórmula *genérica*, donde los términos de la misma están, simplemente, como una función de la complejidad de los métodos abstractos. Esto lo podemos hacer porque tenemos una estrategia común a todas las especializaciones, que describe a nivel abstracto el devenir de cualquiera de las especializaciones.

Un aspecto interesante de esta manera de abstraer familias de algoritmos es que promueve el diseño de algoritmos nuevos, pues es fácil proponer variantes para la implementación de los métodos polimorfos. Se hace énfasis en las coincidencias y quedan claras las diferencias entre una especialización y otra.

La enseñanza de análisis de algoritmos se simplifica. Al tomar este enfoque se identifican características comunes de un conjunto de algoritmos, y al estudiar cada una de las especializaciones únicamente hay que hacer énfasis en qué es lo que lo distingue de otra especialización, y de qué manera la especialización adquiere propiedades adicionales que descansan en las implementaciones de los métodos polimorfos.

También es una contribución de este trabajo presentar la unificación de algoritmos que hasta ahora se han presentado de manera aislada, o con muy poca relación entre ellos, y que aun cuando se hace notar alguna relación, ésta se da en un cierto nivel, como el de establecer los objetivos generales o el que trabajan sobre las mismas estructuras, donde esta asociación no proporciona una mayor percepción del modo de operar de cada uno de los algoritmos. Al unificar algoritmos proporcionamos un marco de referencia común que permite compartir objetivos, demostraciones de correctez y cálculos de complejidad.

Uno de nuestros objetivos al hacer la programación de estas familias ha sido el mostrar cómo la estructura formal de demostraciones de correctez corre paralela a la implementación de los distintos algoritmos. También consideramos muy importante el cumplir con los requisitos de tiempos de acceso a las estructuras de datos que definen, en gran medida, las clases de complejidad a las que muchos de los algoritmos pertenecen. Es vital garantizar esta clase de eficiencia[1], sobre todo si se está trabajando con algoritmos. La biblioteca de clases para todos los temas que revisamos en este trabajo, y donde cada una de las clases puede ser reutilizada, refinada y extendida para obtener variaciones sobre el tema, es otra contribución relevante de este trabajo. Se cuenta, entonces, con una biblioteca de clases que implementan tanto a las estructuras de datos requeridas para los distintos algoritmos, como a los algoritmos mismos, en el marco del esquema jerárquico propuesto.

Presentamos en 1999 [RV99] la unificación de los algoritmos para exploración en gráficas, la primera familia que trabajamos. En junio de 2003 presentamos un resumen extendido de este trabajo en la conferencia *Principles and Practice of Programming in Java, 2003*[RV03]. Un artículo que presenta los resultados de este trabajo será publicado en la revista *Science of Computer Programming*, de Elsevier, en enero de 2005[RV05].

[1]En este trabajo usamos el término *eficiencia* únicamente como una medida del costo de la ejecución del algoritmo, de forma *cuantitativa*, no como una calificación de que el algoritmo tiene un costo bajo en su ejecución

Objetivos

El objetivo general de este trabajo es el de proponer un marco de referencia alternativo para el estudio de algoritmos, que permita de manera más sencilla la identificación de características y estrategias comunes, y apoyada en predicados formales comunes a todos los algoritmos de una familia.

Para mostrar que este enfoque funciona, este trabajo se planteó examinar algoritmos de diversos temas, clasificarlos en el sentido de la orientación a objetos, y determinar la estructura de correctez y complejidad para cada familia. Una de estas familias es la que constituyen los algoritmos que, de una manera o de otra, exploran gráficas. Otra familia importante es la de ordenamientos. El problema del cálculo de flujo en redes también conforma una familia interesante. Estudiamos también algoritmos de apareamiento exacto de cadenas. En todos estos casos nuestro interés principal fue el de encontrar el marco formal para una familia que permita heredar las demostraciones y reutilizar el cálculo de la complejidad dados para la superclase.

En este trabajo buscamos hacer aportaciones en dos aspectos: el primero de ellos es llevar la idea de abstracción y clasificación al análisis de algoritmos, presentando una nueva metodología de tratar este tema, tanto para enseñanza como para la investigación y demostración de algoritmos que resulten de la especialización de estrategias genéricas; el segundo consiste en la identificación y unificación de algoritmos – la mayoría de los cuales se revisan aisladamente – para estudiarlos bajo el paradigma del algoritmo genérico y en este paradigma establecer la correctez y complejidad en los términos arriba expuestos. Habiendo manejado ya varios temas en este sentido, estamos convencidos de que este enfoque facilita la manipulación de este tipo de temas, además de propiciar la creatividad en cuanto a especializaciones particulares respecto a los distintos algoritmos genéricos.

El estado del conocimiento

Los primeros trabajos en el sentido de encontrar familias de algoritmos, aunque con el paradigma funcional, se da en [Mer85], en programación automática, donde el *qué* se especifica en términos de predicados que deben cumplir el estado inicial y final de los procesos, ejemplificándolo con algunos algoritmos de ordenamiento y clasificándolos de acuerdo a dos procesos: el proceso de *dividir* y el proceso de *unir*, pudiendo cada uno de estos procesos hacerse de manera fácil o difícil; las especializaciones se dan como las combinaciones de estas dos posibilidades. Merritt también trabaja poco después con árboles generadores [Mer89] bajo el mismo esquema. Los predicados que se establecen al principio en el nivel jerárquico más abstracto simplemente especifican el estado final deseado, sin apuntar de ninguna manera a la forma de alcanzar ese estado final.

En la literatura aparece muchas veces el término *algoritmo genérico* para referirse a algoritmos con un propósito específico pero que pueden adaptarse a tipos variados de datos. Tal es el caso de ciertas bibliotecas de C++[MN98] donde lo genérico se refiere a que pueden manejar números, cadenas, arreglos, etc., pero no definen niveles de especialización o niveles

de abstracción aplicables a las diferentes instancias de datos, sino que aplican un mismo algoritmo optimizado.

Biedl *et al* en [BCD+01] agrupa ciertas clases de algoritmos de ordenamientos que se llevan a cabo por intercambio, proponiendo nuevos algoritmos que aunque no son algoritmos más eficientes que los conocidos (con menor costo) dan un panorama claro de qué es lo que está involucrado en un ordenamiento de este tipo. Sin embargo, a pesar de dar un marco de referencia para demostrar que estos algoritmos son correctos, esto sólo lo hace a nivel de exigir que al final del algoritmo los datos queden ordenados; no especifica invariantes que unifiquen la ejecución de estos algoritmos.

En muchos casos se agrupan algoritmos para su presentación como en [CLRS01, AMO93], pero se trabaja sobre todo en términos de posibles parámetros a las funciones o procedimientos, cuando esto es posible, o simplemente se presenta un marco conceptual común, pero cada algoritmo es presentado individualmente.

Se pueden ver esfuerzos en esta dirección en el trabajo de Boyer *et al* [Boy04], donde se da un marco de referencia formal para la unificación de tres algoritmos lineales para determinar la planaridad de gráficas [LEC66, BM99, SH99], donde el marco conceptual permite establecer la correctez de la manera en que se implementan estos tres algoritmos; otro esfuerzo en este mismo tema es el trabajo de Small [Sma93] donde se busca un marco de referencia común para el algoritmo de Lempel, Even y Cederbaum [LEC66] y el de Hopcroft y Tarjan [HT74]. El trabajo de Boyer recién citado es el único presentado en términos de métodos y donde se da una estrategia general que invoca a métodos que cumplen con las especificaciones de tipo *requiere/provee* [KV00], estrategia seguida en nuestro trabajo.

En el primer capítulo tratamos más extensamente este tema, comparando nuestro trabajo con trabajo relacionado, citando de manera precisa las diferencias y las coincidencias con los trabajos que se listan bajo el tema de programación genérica o algoritmos genéricos.

La abstracción como prerrequisito para la reutilización

La reutilización en su sentido más amplio es requisito indispensable para el avance de la ciencia. Todo desarrollo está cimentado en resultados anteriores más sencillos.

La OO propone una reutilización descendente, donde lo que se reutiliza en muchos momentos es el marco abstracto, un patrón. En este trabajo se hace énfasis en la *reutilización descendente*, tanto de demostraciones como de código. Para hacer posible esta reutilización es importante definir familias de algoritmos que respondan a una misma estrategia genérica.

El estudio de algoritmos es central para las ciencias de la computación. El conocer caracterizaciones de algoritmos en cuanto a la clase de complejidad a la que pertenecen, establecer la correctez de un algoritmo o, finalmente, establecer la imposibilidad de su existencia, son conocimientos y habilidades que deben formar parte del bagaje de todo profesional de las ciencias de la computación. Estas tareas se facilitan y promueven bajo el enfoque de familias de algoritmos que comparten una estrategia genérica y poseen un marco formal común. En este trabajo se propone usar el concepto de *algoritmo genérico* para agrupar diversos algo-

ritmos en familias, proporcionando para cada una de éstas el marco formal que acabamos de describir. Los algoritmos específicos siguen esta estrategia genérica, especializando algunos de los pasos especificados de manera abstracta, y presentan, al menos, las propiedades del algoritmo genérico.

Un *algoritmo genérico* consiste de una jerarquía de soluciones para problemas afines, o que responden a una misma estrategia de solución. En la parte superior de la jerarquía se tiene a un algoritmo abstracto que, como mencionamos, plantea una estrategia general para la solución de una clase de problemas. Esta estrategia se refina o *especializa*, mediante la definición precisa de ciertos métodos invocados por ella. Esto es en cuanto al diseño de algoritmos. Para establecer su correctez, se establece a la estrategia general como correcta, siempre y cuando cada uno de los métodos invocados cumpla con determinados predicados. A continuación, para cada especialización – que corresponde a un algoritmo concreto – se procede a demostrar que la implementación particular de los métodos en efecto cumple con las especificaciones exigidas por la estrategia general.

Para lograr estos objetivos utilizamos el enfoque de la orientación a objetos (OO). Tradicionalmente, la OO se usa en programación para definir estructuras de datos abstractas, que cumplen con proporcionar los servicios asociados a la estructura. El programador tiene la libertad de implementar estas estructuras como mejor le convenga, siempre y cuando la estructura proporcione los servicios que anuncia. En algunas ocasiones se ponen restricciones adicionales a los servicios, en términos de medidas de complejidad que deben presentar. La preocupación principal es en la especialización de estas estructuras abstractas de datos, de tal manera que se pueda tener un programa genérico que sea capaz de trabajar con cualquiera de ellas. Podemos pensar que en este enfoque se hace una abstracción de lo particular a lo general, ya que usualmente se proporcionan distintas implementaciones para las estructuras de datos, y la aplicación deberá decidir cuál de ellas usar en términos de sus necesidades particulares.

La manera en que utilizamos la OO es distinta. Procedemos a diseñar una estrategia genérica para una familia de algoritmos; a continuación establecemos las propiedades que las estructuras abstractas de datos y los métodos definidos a nivel abstracto deben cumplir, y asumiendo que las implementaciones particulares cumplirán con estas propiedades, procedemos a establecer propiedades de la estrategia genérica. Una vez hecho esto, para los algoritmos específicos que se presentan con la implementación de los métodos abstractos y de las estructuras de datos necesarias, se requiere únicamente demostrar que estos métodos presentan las propiedades requeridas. De la misma manera podemos dar una fórmula para la complejidad del algoritmo, en la que participan como términos por definir las funciones de complejidad de los métodos abstractos, o de aquellos métodos que representan servicios dados por las estructuras de datos utilizadas. En pocas palabras, introducimos los conceptos de herencia y polimorfismo a las demostraciones de correctez y los cálculos de complejidad. Hasta el momento estos conceptos únicamente se utilizan como mecanismos para el análisis y diseño de soluciones para problemas específicos, pero no en el trato formal de estas soluciones.

La implementación jerárquica de una estrategia abstracta se puede hacer directamente en lenguajes orientados a objetos. En la raíz del árbol jerárquico se define la estrategia genérica que se va a utilizar y en los descendientes de esa raíz se colocan implementaciones concretas de algoritmos. Estas especializaciones se pueden dar en más de un nivel. Por ejemplo, en el

caso de la exploración a profundidad de una gráfica (*Depth First Search*), este algoritmo es una especialización de un algoritmo genérico de exploración en gráficas, que a su vez tiene especializaciones como el ordenamiento topológico, o la numeración *st* – que se utiliza en el algoritmo de Lempel *et al* [LEC66] para determinar la planaridad en gráficas. Un primer nivel de especialización para el algoritmo genérico de exploración en gráficas es el de distinguir el tipo de gráficas que va a explorar, en términos de si las aristas tienen o no un peso uniforme asignado.

En las ciencias de la computación todo proceso ya sea de clasificación o de unificación es importante. Ayuda a vislumbrar tanto relaciones como diferencias entre problemas, que ayudan en el planteamiento de la solución de los mismos. Atacar el análisis de algoritmos a través de algoritmos genéricos promueve la abstracción y la identificación de aspectos comunes o de aspectos que separan a una solución de otra. Esto proporciona una mejor comprensión de los problemas y ayuda a adquirir una mejor estrategia para el diseño de soluciones.

En el mundo profesional de la computación hay un énfasis muy grande en la reutilización, tanto de diseños como de implementaciones. Con este trabajo se promueve la reutilización, no nada más en términos de estos dos aspectos, sino también en aquellos que tienen que ver con la formalización de la computación, y que se refiere a establecer la correctez y complejidad de los distintos algoritmos. Una vez enmarcado un algoritmo dentro de una cierta clase, el algoritmo "hereda" de la clase a la que pertenece la correctez de la estrategia general, quedando por demostrar únicamente que cada uno de los métodos que contribuyen a concretar esa estrategia genérica cumplen con lo que se exige de ellos en aquélla. En cuanto a la complejidad, se especifica de manera similar: se establece un costo de la estrategia genérica, que estará en términos de los costos de los métodos especializados, por lo que al calcular la complejidad de un algoritmo específico únicamente se tiene que revisar lo correspondiente a cada uno de los métodos. Aun cuando se utilice complejidad amortizada, el foco deberá estar sobre cada uno de los métodos, lo que facilita el análisis.

Metodología

El trabajo empezó con una inquietud por usar la orientación a objetos para unificar algoritmos que, de una manera u otra, exploran gráficas. Se buscó contar con una estructura jerárquica dentro de estos algoritmos, en los que la herencia y el polimorfismo jugaran un papel fundamental, mediante la definición de un algoritmo abstracto y las distintas maneras de concretar este algoritmo. Se identificaron las coincidencias en algoritmos que clasificamos como de exploración, y determinamos asimismo sus diferencias. Un aspecto importante fue el poder representar las diferencias en términos de implementaciones distintas de métodos abstractos. Una vez definidas las coincidencias, propusimos una estrategia común para los algoritmos concretos, que invocara a métodos que respondían a las diferencias. En este punto, también reconocimos las estructuras de datos requeridas por los algoritmos, para encontrar, de manera similar a como hicimos con la estrategia, estructuras abstractas de datos que pudieran concretarse de manera adecuada para cada uno de los algoritmos concretos.

La siguiente familia de algoritmos con la que trabajamos fue la de ordenamientos. Es

importante mencionar que uno de los requisitos más importantes de nuestro enfoque para poder considerar a un grupo de algoritmos como una familia, es que presenten coincidencias en el algoritmo en sí, esto es, que el algoritmo abstracto planteado presente a nivel de la estrategia genérica propiedades no triviales: no podemos agrupar a todos los algoritmos de ordenamiento conocidos porque no podríamos plantear una estrategia general para todos ellos, de tal manera que esta estrategia presente invariantes a nivel de los métodos invocados. En otras palabras, una estrategia genérica que lo único que hace es invocar a un método abstracto, que se encarga de absolutamente todo el trabajo, no aporta nada a nivel de la estrategia general para entender, demostrar o probar variaciones sobre parte de la estrategia.

Para la familia de algoritmos de ordenamiento, además de la estructura jerárquica inherente a este enfoque, implementamos en Java los algoritmos en base a *comparadores*, de tal manera que los objetos a comparar pudieran ser de cualquier tipo. Asimismo, aun si la implementación básica está hecha en términos de arreglos con acceso directo, se pueden especializar los comparadores para que manejen cualquier estructura de datos que se desee, manteniendo presente siempre el costo de los accesos a dichas estructuras.

Como resultado de someter los resultados parciales que teníamos a árbitros externos, incursionamos en el tema de patrones, relacionado íntimamente hoy en día con la orientación a objetos. Los patrones buscan establecer esquemas generales – de acá el término *patrones* – para enmarcar conjuntos de aplicaciones que tienen una manera similar de proceder. Estos patrones, inmersos en la orientación a objetos, se refieren más a la interacción de los distintos objetos en una aplicación que a una estrategia determinada. Hay tres patrones, que mencionamos en el Capítulo 1, que se refieren a estrategias, pero el patrón simplemente dice algo como "se invoca un cierto algoritmo", lo que no ayuda a inducir una familia de algoritmos: cualquier algoritmo puede caber dentro de cualquiera de estos tres patrones. Por ello, concluimos que este tema no juega un papel relevante en nuestro trabajo.

Consideramos que uno de los aspectos más importante en este trabajo es el de especificar algoritmos abstractos, pero con propiedades formales – invariantes – que se reflejan en exigencias sobre los métodos invocados; en trabajos relacionados con este tema, cuando se establecen las propiedades formales de las familias de algoritmos, el nivel más abstracto de la jerarquía únicamente especifica, como invariante, que el algoritmo obtenga el resultado deseado – como en [Mer85]. Para cada familia propusimos un algoritmo genérico en términos de invariantes relevantes y hereditarias, de tal forma que en los algoritmos concretos se pueda establecer su correctez simplemente demostrando que las implementaciones de los métodos abstractos cumplen con lo exigido por la estrategia genérica. Algunas veces empezamos con un grupo amplio de algoritmos para constituir una familia, y sin embargo al momento de tratar de establecer las propiedades de la estrategia genérica nos vimos en la necesidad de excluir algoritmos que no se prestaban a la estructura del algoritmo genérico.

Estructura

La Parte I corresponde a la introducción. En esta parte presentamos en el Capítulo 1 una discusión del concepto de abstracción, para ubicar en ese contexto a la clase de abstracción

que realizamos a lo largo de este trabajo. Introducimos entonces el concepto de *algoritmo genérico* en términos de este tipo de abstracción, y justificamos el porqué de lo nuevo de este enfoque, comparándole con enfoques que ostentan el calificativo de "genérico" pero con significado distinto. Asimismo revisamos los esfuerzos que ha habido en la unificación de algoritmos en familias, marcando también las diferencias con nuestro enfoque.

En el Capítulo 2 presentamos a una familia de algoritmos de ordenamiento en los términos de este trabajo: un algoritmo abstracto y aquellos algoritmos de ordenamiento que se ajustan a este esquema. Esta familia se presenta en términos de un algoritmo genérico con invariantes, que cada una de las especializaciones cumple.

La Parte II corresponde a exploración en gráficas y comprende los capítulos 3 a 8. En el Capítulo 3 presentamos un algoritmo genérico de exploración en gráficas. Se presenta la estrategia genérica y las invariantes a este nivel. Se hace una especialización básica, de la que heredarán los algoritmos concretos que vamos a presentar.

En el Capítulo 4 presentamos la especialización para la exploración en amplitud – en inglés, *Breadth First Search* – y sus aplicaciones. En el capítulo 5 presentamos la especialización para la exploración a profundidad – conocida por su nombre en inglés *Depth First Search* – y sus aplicaciones. En el capítulo 6 enmarcamos en esta misma familia al algoritmo de Euler para encontrar si una gráfica es o no euleriana, y si lo es, encontrar el circuito euleriano correspondiente. En el Capítulo 7, también dentro de esta familia, revisamos el algoritmo de distancias de Dijkstra. En el capítulo 8 presentamos el algoritmo de exploración con escrutinio – en inglés *Scan First Search*. Estas presentaciones se hacen a detalle respecto a la estructura de cada una de las especializaciones, sus demostraciones de correctez y el cálculo de su complejidad en los términos de herencia que describimos en el apartado anterior.

En la Parte III presentamos ejemplos adicionales de nuestro enfoque. Es así que en el Capítulo 9, de forma breve, presentamos una familia para la construcción de árboles generadores en gráficas. Incluimos algunos algoritmos considerados de exploración, para mostrar cómo un mismo algoritmo puede interpretarse bajo distintas estrategias genéricas. En esta familia se presentan dos niveles de especialización. En el primer nivel tenemos una subclasificación entre aquellos algoritmos que simplemente construyen un árbol generador, y aquéllos donde el árbol debe llenar requisitos adicionales, como lo es el que sea de peso mínimo. En el tercer nivel de la estructura se presentan ya algoritmos concretos para construir árboles generadores.

Incorporamos a este trabajo en este mismo capítulo una revisión de otras familias de algoritmos que se adaptan a nuestro enfoque. En este contexto presentamos una familia de algoritmos para apareamiento exacto de cadenas de texto que incluyen las especializaciones: ingenua; de Karp-Rabin; de proceso con un autómata finito; de Knuth-Morris-Pratt; y de Boyer-Moore.

También discutimos de manera somera una familia de algoritmos para encontrar el flujo máximo entero en redes de flujo. Esta familia se clasifica, en una primera instancia, en incremento de flujo por trayectorias aumentantes; e incremento de flujo usando preflujo. En la primera subfamilia presentamos tres especializaciones y en la segunda dos.

Finalmente, en la Parte IV presentamos las conclusiones de nuestro trabajo.

Incluimos como apéndices conceptos generales de análisis de algoritmos, una breve introducción a la complejidad amortizada y propiedades de árboles, además de conceptos generales que sirven como introducción a algunos de los temas.

Índice general

I Fundamentos 1

1. Algoritmos genéricos **3**

1.1. El concepto de abstracción 3

1.1.1. Distintos significados para abstracción en computación 3

1.1.2. El concepto de algoritmo 4

1.1.3. La abstracción en los lenguajes de programación 6

1.1.4. Un algoritmo genérico basado en las características de Algol 6

1.1.5. La programación estructurada como una abstracción 9

1.1.6. Estructuras de datos abstractas 10

1.2. La complejidad de un algoritmo 10

1.3. Generalización de algoritmos 11

1.3.1. Familias de algoritmos 11

1.3.2. La reutilización como objetivo 12

1.3.3. Programación genérica 13

1.3.4. Programación adaptiva 14

1.3.5. Unificación de algoritmos 14

1.4. Definición de algoritmo genérico 15

1.5. Uso de herencia en análisis y verificación 16

1.5.1. Beneficios del uso generalizado de la herencia 17

1.5.2. Los algoritmos genéricos en la docencia 17

2. Ordenamientos 19

2.1. Definición formal de ordenamiento . . . 19

2.2. Familias de algoritmos de ordenamiento . . . 20

2.3. Un algoritmo genérico para ordenamiento . . . 22

2.4. Correctez de ordenamiento por comparación . . . 25

2.4.1. Complejidad . . . 31

2.5. Especializaciones . . . 32

2.5.1. Ordenamiento por selección . . . 32

2.5.2. Correctez de la especialización del ordenamiento por selección . . . 33

2.5.3. Complejidad de la especialización para el ordenamiento por selección . . . 39

2.6. Especialización para el ordenamiento por inserción . . . 45

2.6.1. Correctez de la especialización de ordenamiento por inserción . . . 46

2.6.2. Complejidad de la especialización del ordenamiento por inserción . . . 49

2.7. Especialización para el ordenamiento de burbuja . . . 53

2.7.1. Complejidad para la especialización del ordenamiento de burbuja . . . 58

2.8. Especialización para ordenamiento de montículo . . . 63

2.8.1. Complejidad para la especialización del ordenamiento por montículo . . . 67

2.9. Conclusiones . . . 71

II Exploración en gráficas 73

3. Enfoque unificador 75

3.1. Problema general . . . 75

3.1.1. Estructuras de datos para exploración de gráficas . . . 76

3.1.2. Funcionamiento general de un algoritmo de exploración . . . 76

3.2. Propiedades 87
3.2.1. Invariantes para el estado de exploración 88
3.2.2. Invariantes para frontera 91
3.2.3. El atributo π 93
3.2.4. El atributo de distancias d 96
3.3. Correctez y complejidad 97
3.4. Complejidad del algoritmo genérico 99
3.4.1. Complejidad para la especialización de la exploración básica 102
3.5. Exploración en gráficas no dirigidas 108
3.6. Aplicaciones del algoritmo genérico de exploración 109
3.6.1. Ciclos en gráficas no dirigidas 109
3.6.2. Componentes conexas en una gráfica 109
3.6.3. Conexidad fuerte en digráficas 111
3.7. Conclusiones 112

4. Búsqueda en amplitud **115**

4.1. Descripción del problema 115
4.2. Especialización de ExploracionBasica: ExploracionBFS 116
4.3. Correctez de ExploracionBFS 121
4.3.1. El árbol G_π 124
4.3.2. Correctez de ExploracionBFS 125
4.4. Complejidad de ExploracionBFS 125
4.5. ExploracionBFS en gráficas no dirigidas 127
4.5.1. Complejidad de ExploracionBFS en gráficas no dirigidas 128
4.6. Aplicaciones 129
4.6.1. Gráficas bipartitas 129
4.6.2. Diámetro de un árbol 133
4.7. Conclusiones 133

5. Exploración a Profundidad **135**

5.1. DFS en digráficas 136

5.2. Propiedades de ExploracionDFS 140

5.2.1. Implementación eficiente de la pila para ExploracionDFS 143

5.2.2. ExploracionDFS en bosques 145

5.2.3. Complejidad de ExploracionDFS 145

5.3. Propiedades adicionales de ExploracionDFS 146

5.3.1. Propiedades de las marcas de tiempo 148

5.3.2. Complejidad de ExploracionDFS extendido 156

5.4. Aplicaciones en digráficas 157

5.4.1. Determinación de ciclos 157

5.4.2. Ordenamiento topológico 160

5.5. ExploracionDFS en gráficas no dirigidas 162

5.6. Aplicaciones en gráficas no dirigidas 164

5.6.1. Especialización para componentes no separables 165

5.6.2. Propiedades de ExploracionDFSPB 170

5.6.3. Componentes no separables 173

5.6.4. Direccionamiento de calles 176

5.7. Conclusiones 177

6. Especialización para circuitos eulerianos **179**

6.1. Trayectorias y circuitos eulerianos 179

6.2. ExploracionBasica para circuitos eulerianos 180

6.2.1. La frontera para circuitos eulerianos 180

6.2.2. Modificaciones para trayectorias eulerianas 194

6.3. Correctez de ExploracionEuleriana 195

6.4. ExploracionEuleriana en gráficas no dirigidas 197

6.5. Aplicaciones de la especialización de Euler 198

6.6. Conclusiones 198

7. Caminos más cortos **199**

7.1. Descripción general del problema 199

7.2. Especialización para caminos más cortos 201

7.3. Complejidad 215

7.3.1. Implementación de frontera con arreglo o lista ligada 215

7.3.2. Implementación de frontera con heaps binarios* 216

7.3.3. Implementación de frontera con heaps de Fibonacci** 217

7.4. Conclusiones 218

8. Exploración SFS **219**

8.1. El algoritmo SFS 219

8.2. La especialización de ExploracionBasica a SFS 230

8.2.1. Evaluación y reorganización de las aristas 235

8.3. Ejemplo de ejecución de SFS 238

8.4. Propiedades de esta implementación de SFS 255

8.5. Complejidad de SFS 266

8.6. Aplicaciones de SFS 267

8.6.1. Certificados delgados de k-conexidad 268

8.7. Conclusiones 271

III Ejemplos Adicionales **273**

9. Otras familias de algoritmos **275**

9.1. Árboles generadores 275

9.1.1. Algoritmo genérico 276

9.1.2. Árboles generadores de gráficas con pesos heterogéneos en sus aristas 280

9.1.3. Árboles generadores con peso homogéneo en las aristas 288

9.1.4. Especialización BFS 291

9.1.5. Relación entre los dos algoritmos abstractos que producen árboles generadores 291

9.2. Apareamiento de cadenas de texto 291

9.2.1. El algoritmo genérico para apareamiento de cadenas 292

9.2.2. La especialización para el apareamiento ingenuo 297

9.2.3. Autómatas finitos 301

9.2.4. Especialización Knuth-Morris-Pratt 306

9.2.5. Apareamiento de Boyer-Moore 310

9.3. Flujo máximo en redes 314

9.3.1. Algoritmo genérico para flujo máximo entero 315

9.3.2. Trayectorias aumentantes 317

9.3.3. Algoritmo de Dinic de trayectorias aumentantes 320

9.3.4. Especialización de trayectorias aumentantes de distancias mínimas . . 323

9.3.5. Especialización con Etiquetamiento 326

9.3.6. Obtención de flujo máximo por preflujo 327

9.3.7. Especialización con una cola para los nodos activos 331

9.3.8. Especialización de preflujo por etiquetas mayores 333

9.3.9. Otros enfoques para el problema de máximo flujo 335

9.4. Conclusiones 335

Conclusiones **339**

Apéndice A .

Análisis de algoritmos 345

A.1. Conceptos generales 345

A.2. Analizando algoritmos 346

A.3. Correctez 348

A.4. Costo 351

A.5. Orden de crecimiento 353
A.6. Notación asintótica 356
A.6.1. Consideraciones prácticas 360
A.7. Medición de procesos recursivos 363

Apéndice B .

Conceptos de ordenamientos 369
B.1. Ordenamientos 369
B.2. Algoritmos de ordenamiento por comparación 370

Apéndice C .

Conceptos de teoría de gráficas 373
C.1. Conceptos básicos 373
C.1.1. Caminos más cortos y distancias 373
C.2. Gráficas bipartitas 374
C.3. Conexidad en gráficas 377
C.3.1. Componentes no separables 377
C.3.2. Propiedades 378
C.4. Gráficas con peso en las aristas 381
C.4.1. Conexidad en términos de cortes 388
C.5. Propiedades de conexidad 391
C.6. Conexidad S-mixta 394
C.6.1. Propiedades de la $k-$conexidad 395

Apéndice D .

Colofón de exploración en gráficas 399
D.1. Exploración en amplitud 399
D.2. Exploración a profundidad 400
D.2.1. Ejecución de ExploracionDFS con origen alternativo 400
D.2.2. Ordenamiento topológico 403

D.2.3. DFS en el direccionamiento de calles 406
D.3. Gráficas eulerianas 407
D.4. Propiedades de gráficas eulerianas 411
D.5. Sucesiones de de Bruijn 412
D.5.1. Construcción de sucesiones de de Bruijn 413
D.5.2. Tambores o discos magnéticos 417
D.5.3. Secuencias para ADN 418

Apéndice E .

Árboles 423

E.1. Caracterización 423
E.2. Árboles generadores 427
E.3. Árboles dirigidos 428
E.4. Digráficas infinitas 432
E.4.1. Paradoja de la bolsa de monedas 435
E.4.2. Mosaicos: una aplicación del Lema del Infinito 435

Apéndice F .

Análisis amortizado 439

F.1. Costos amortizados y estructuras avanzadas 439
F.2. Colas de prioridades 443
F.2.1. Insertar un nodo en un heap binario 454
F.2.2. Obtener el nodo con el valor mínimo 463
F.2.3. Reducción del valor de la llave de un nodo en el heap 463
F.2.4. Eliminación de la raíz de un heap 465
F.2.5. Aplicaciones de heaps binarios: ordenamiento Heapsort 471
F.3. Operaciones adicionales 471
F.4. Árboles binomiales 479
F.4.1. Propiedades 482

F.5. Heaps binomiales 483

F.5.1. Métodos borra(Nodo v) y reduceLlave(Nodos v, int delta) 492

F.6. Heaps de Fibonacci 493

F.7. Implementaciones eficientes 500

Bibliografía **505**

Índice de figuras

2.1. Situación inicial del arreglo 22

2.2. Ordenamiento por selección: 4 pasadas 32

2.3. Permutación inicial para Mejor Caso en el ordenamiento por selección . . . 41

2.4. Ordenamiento por inserción: 4 pasadas 45

2.5. Primeras cuatro iteraciones del ordenamiento de burbuja 53

2.6. El maxheap obtenido después del constructor 63

2.7. Primera iteración con $i = n - 1$ 64

2.8. Segunda iteración con $i = n - 2$ 64

2.9. Tercera iteración con $i = n - 3$ 64

2.10. Cuarta iteración con $i = n - 4$ 65

3.1. Suponiendo que al finalizar el algoritmo quedan vértices sin explorar 92

3.2. Exploración desde distintos orígenes 95

4.1. Ejecución BFS en un punto intermedio del proceso 116

4.2. Funcionamiento de BFS 117

4.3. Aristas no incluidas en G_π 128

4.4. Aristas que cruzan en el mismo nivel 132

5.1. Digráfica a explorar con ExploracionDFS 137

5.2. Primera iteración de DFS: después de inicializar, se mete a v_2 a la pila . . . 137

5.3. Segunda iteración de DFS: se alcanza a v_3 y se le coloca en la pila 138

5.4. Tercera y cuarta iteración del algoritmo: se saca a v_3 y se mete a v_4 138

5.5. Dos iteraciones: se recorren las aristas desde v_4... 139

5.6. Tres iteraciones: se cierra a v_5 y se regresa a v_4... 139

5.7. Estado de la gráfica al terminar la ejecución de exploracionGenerica 140

5.8. Camino de longitud $k+1$ en G_π, desde v hasta u 142

5.9. Determinación de ciclos en digráficas mediante DFS 148

5.10. Anidamiento de intervalos en DFS 153

5.11. Distintos tipos de arcos con DFS 156

5.12. Imposibilidad de la existencia de aristas hacia adelante o de cruce 164

5.13. Detección de componentes no separables 165

5.14. Puntos bajos en una gráfica 167

5.15. Acciones al recorrer la arista 6 (hacia atrás) 169

5.16. Lema 5.22 172

5.17. Lema 5.23 173

6.1. Construcción de una trayectoria euleriana 181

6.2. Estructuras de datos para la especialización de Euler 187

6.3. Trayectoria euleriana representado en las estructuras de la Figura 6.2 187

6.4. Ejemplo de digráfica para la construcción de un circuito 188

6.5. Estructuras de datos en la construcción de un camino 189

6.6. Estructuras de datos al descubrir a v_6 y tenerlo como centro de acción 190

6.7. Estructuras de datos al llegar nuevamente a v_1 191

6.8. Estructuras de datos al regresar buscando un vértice GRIS 192

6.9. Estructuras de datos cuando el camino llega a v_8 desde v_{11} 193

6.10. Estructuras de datos al retroceder hasta v_8 para pintarlo de NEGRO 193

6.11. Recorrido realizado para la construcción del circuito euleriano 194

6.12. Ejemplo de digráfica para la construcción de un camino euleriano 195

7.1. Gráfica con pesos negativos. 201

7.2. Ejecución de BFS especializado con peso heterogéneo en los arcos 203

7.3. Ejemplo donde no funciona la estrategiaespecializada BFS. 204
7.4. Ejemplo con la estrategia BFS ajustada. 205
7.5. Camino más corto de s a v. 212

8.1. Construcción de los árboles de distintos niveles y con distintas raíces 221
8.2. Exploración SFS con s como origen. Se ingresa a s... 223
8.3. Se usa $s \to v_4$ y se ingresa a v_4 a la frontera con sus aristas evaluadas 223
8.4. Se usa $v_4 \to v_1$, se mete a v_1 a la frontera y se evalúan sus aristas 224
8.5. Se usa $v_1 \to v_2$ y se evalúa a las aristas sin evaluar de 224
8.6. Se usa $v_2 \to v_3$ y se evalúa a las aristas de v_3... 225
8.7. Se usa $v_3 \to v_7$ y se evalúa a las aristas de v_7,... 225
8.8. Se usa $v_7 \to v_6$ y se evalúa a las aristas restantes de v_6,... 226
8.9. Se usa $v_6 \to v_5$; se descubre a v_5 que ya no tiene aristas por evaluar 226
8.10. Se usa la arista $v_2 \to v_6$ 227
8.11. Se usa la arista $v_1 \to v_3$ 227
8.12. Se recorren las aristas $s \to v_1$, $v_3 \to v_5$ y $v_4 \to v_7$ 228
8.13. Árboles determinados por SFS 228
8.14. Se usan las aristas guiados por una exploración "al estilo" de DFS 229
8.15. Se usan las aristas guiados por una exploración "al estilo" de BFS 230
8.16. Ejemplo para aplicar SFS 238
8.17. Al terminar con la lista de incidencia de s 239
8.18. Al terminar de descubrir a v_2 240
8.19. Al llegar el camino a v_4 por primera vez 241
8.20. Al descubrir a v_6 desde v_4 242
8.21. Se descubre a v_3 al recorrer a e_7 desde v_6 243
8.22. Se extiende el camino con e_6, descubriendo a v_5 244
8.23. Se avanza desde v_3 hasta v_5, y se retrocede por v_3 hasta v_6 245
8.24. Se extiende el camino desde v_6 con e_{12} 246

8.25. Se agrega al camino e_{15} desde v_7 247
8.26. Desde v_8 se recorre la arista e_{16} y se descubre a v_9 248
8.27. Se retrocede hasta v_7 desde donde se recorre la arista e_{14} 249
8.28. Se retrocede hasta v_4 desde donde se recorren las aristas e_9, e_{10} y e_{18} 250
8.29. Se extiende el camino desde v_{10} con la arista e_{17} y se descubre a v_{11} 251
8.30. Se extiende el camino desde v_{10} con la arista e_{17} y se descubre a v_{11} 252
8.31. Se retrocede desde v_4, pasando por v_2 hasta v_1, desde 253
8.32. Se retrocede desde v_4, pasando por v_2 hasta v_1, desde donde 254
8.33. Demostración del Lema 8.1 256
8.34. Ciclo de aristas con rango 1 263
8.35. Contradicción por la inexistencia de un camino dirigido 264
8.36. Existencia de arista orientada que debió ser incluida 265
8.37. Corte U que separa a S_1 y S_2 269
8.38. Corte S de vértices que separa a S_1 y S_2 270
8.39. Construcción de los árboles para el certificado de $k-$conexidad 271

9.1. Jerarquía de las clases para Árboles Generadores 281
9.2. Esquema Jerárquico para la familia de apareamiento de cadenas 297
9.3. AF que reconoce al patrón P=abaabca 302
9.4. Reacomodo del patrón para disminuir el número de comparaciones 303
9.5. Jerarquía de la familia de algoritmos de flujo máximo 316
A.1. Comparación de $T_a = 100n$ y $T_b = 2n^2$ 354
A.2. Comparación de complejidad de algoritmos 356
A.3. $3n^2 + 5n + 22 \in \Theta(n^2)$ 357
A.4. $\frac{1}{2}n^2 - 3n \in \Theta(n^2)$ 358
B.1. Árbol de comparaciones para ordenar tres elementos 370
C.1. Propiedad de la distancia 374
C.2. Gráfica con ciclo de longitud impar 375

C.3. Asignación de vértices en camino simple 376
C.4. Asignación de vértices en más de un camino simple 376
C.5. Componentes no separables de una gráfica 377
C.6. Componentes que resultan al quitar a un vértice de corte v 379
C.7. Vértice de corte v para más de una pareja 380
C.8. Gráfica con puentes 381
C.9. Conexidad local por aristas 383
C.10. Gráfica con puentes 384
C.11. Gráfica sin el puente v_4—v_7 384
C.12. Gráfica sin el vértice de corte v_4 385
C.13. Gráfica conexa 385
C.14. Gráfica con corte de aristas 386
C.15. Gráfica conexa con cortes de aristas 386
C.16. Gráfica conexa con cortes de aristas 386
C.17. Gráfica con corte de vértices 386
C.18. Gráfica conexa con cortes de vértices 387
C.19. Gráfica conexa con cortes de vértices alternativos 387
C.20. Gráfica conexa con cortes de aristas 388
C.21. Cortes de aristas minimal y mínimo 388
C.22. Conexidad por vértices y aristas 389
C.23. Ejemplo de conexidad por vértices 390
C.24. Ejemplo de conexidad por aristas 390
C.25. Ejemplo de conexidad por vértices y por aristas, agregando una arista . . . 390
C.26. Cortes de vértices y aristas mínimos 391
C.27. Relación entre $\kappa(G)$, $\lambda(G)$ y $\delta(G)$ 393
D.1. Estado inicial de DFS con v_6 como origen 400
D.2. Primera iteración con origen alternativo 401
D.3. Segunda iteración con $s = v_6$ 401

D.4. Dos iteraciones con $v_a = v_2$ 402
D.5. Una iteración con $v_a = v_4$ 402
D.6. Iteraciones con v_5 como centro de acción 403
D.7. Situación al vaciarse la pila 403
D.8. Gráfica con ordenamiento topológico 404
D.9. Seriación en la carrera de Ciencias de la Computación 404
D.10. Ordenamiento topológico usando ExploracionDFS 405
D.11. Los puentes de Königsberg 407
D.12. Gráfica correspondiente a los puentes de Königsberg 408
D.13. Algoritmo de construcción de circuito euleriano 410
D.14. Estructura de las etiquetas de los vértices y aristas 414
D.15. Gráficas para la construcción de las sucesiones de de Bruijn 414
D.16. Circuito euleriano en $G_{2,3}$ 415
D.17. Circuito euleriano para $G_{3,2}$ 415
D.18. Un disco magnético con ocho sectores y tres terminales 417
D.19. Gráfica de de Bruijn para el problema de los sectores de disco 417
D.20. Gráfica $G(\mathfrak{L})$ correspondiente a $\mathfrak{L}$ 420
E.1. Al agregar una arista a un árbol 424
E.2. Dos caminos simples en una gráfica conexa 424
E.3. Dos caminos dirigidos de r a v 429
E.4. Digráfica con $ingrado(\mathsf{v}) = 2$ 430
E.5. Construcción de camino dirigido de r a v 430
E.6. Digráfica infinita que cumple con (v) sin ser árbol 431
E.7. Digráfica infinita arbitrada, sin raíz 431
E.8. Tipos de digráficas infinitas 433
E.9. Construcción de camino dirigido infinito 434
E.10. Árbol infinito sin camino infinito 434
E.11. Tipos de mosaicos y enlosetado de 2×3 436

E.12. Nivel 0 y 1 del árbol para enlosetar 437
E.13. Un vértice en el nivel 3 del enlosetado 437
E.14. Árbol que corresponde a enlosetados válidos 438
F.1. Árbol parcialmente ordenado 444
F.2. Propiedades de un heap 445
F.3. Ejemplo de un maxheap 446
F.4. Organización de un heap en una lista con acceso directo 447
F.5. Propiedad de heap en subárbol de dos niveles 447
F.6. Garantizando propiedad de heap en árbol con profundidad mayor a 2 448
F.7. Intercambios necesarios hasta garantizar un maxheap 450
F.8. Intercambio de dos nodos en un heap 457
F.9. Al inicio de las operaciones en el heap 457
F.10. (1) Al insertar el nodo con llave=5 458
F.11. (2) Al insertar el nodo con llave=3 458
F.12. (3) "Flotamos" al valor 3 hacia la raíz del heap 458
F.13. (4) Inserción del nodo con llave=8 459
F.14. (5) Insertamos al nodo con llave 4 459
F.15. (6) El nuevo nodo "flotado" un nivel hacia arriba 460
F.16. (7) Inserción del nodo con llave=2 460
F.17. (8) Intercambio del nuevo nodo con su padre 460
F.18. (9) Sigue la flotación del nuevo nodo hacia la raíz 461
F.19. (10) Inserción del nodo con llave=6 461
F.20. (11) El nuevo nodo ya cumple la propiedad de heap 462
F.21. (12) Inserción del nodo con llave=7 462
F.22. Eliminando simplísticamente al mínimo de un heap binario 465
F.23. Eliminación de la raíz de un heap 466
F.24. Reorganización de un heap binario 467
F.25. Se intercambia a la raíz con la última hoja 469

F.26. Se intercambia a la nueva raíz con su hijo izquierdo 470
F.27. Se intercambia al nodo en la posición 2 con su hijo derecho 470
F.28. Incrementando el valor de una llave 472
F.29. Se intercambia con el último y se poda al último 473
F.30. Se compara con sus hijos y se sumerge un nivel 473
F.31. Se compara con sus hijos y ya no se sumerge 473
F.32. Construcción de un heap binario a partir de un arreglo 474
F.33. Propiedad de heap en el subárbol con raíz en $i = \lfloor 13/2 \rfloor = 6$ 475
F.34. Propiedad de heap en el subárbol con raíz en $i = 5$ 475
F.35. Propiedad de heap en el subárbol con raíz en $i = 4$ 476
F.36. Propiedad de heap en el subárbol con raíz en $i = 3$ 476
F.37. Propiedad de heap en el subárbol con raíz en $i = 2$ 477
F.38. Propiedad de heap en el subárbol con raíz en $i = 1$ 477
F.39. Árboles binomiales 480
F.40. Niveles en árboles ligados 483
F.41. Heap binomial para acomodar a $27 = 2^4 + 2^3 + 2^1 + 2^0$ nodos 484
F.42. Heap binomial para acomodar a $296 = 2^3 + 2^5 + 2^8$ nodos 485
F.43. Representación de un heap, donde a lo más hay un único B_i 488
F.44. Representación de un heap binomial con lista circular doblemente ligada . . 492
F.45. Acción de podado en un árbol de Fibonacci 494
F.46. Acción de podado en un árbol de Fibonacci 495
F.47. Acción de podado en un árbol de Fibonacci 495
F.48. Acción de podado en un árbol de Fibonacci 497
F.49. Acción de podado en un árbol de Fibonacci 497
F.50. Acción de podado en un árbol de Fibonacci 498
F.51. Algunos árboles de Fibonacci 499

Parte I

Fundamentos

Capítulo 1

Algoritmos genéricos

Un algoritmo genérico es una abstracción respecto a una familia de algoritmos que comparten una estrategia general para su solución. Esta estrategia general está dada a un nivel tal que presenta propiedades que se heredan a cada uno de los algoritmos específicos que conforman a la familia.

En este capítulo hablaremos sobre distintos significados que se dan al concepto de abstracción, y cómo se relacionan estos significados con el quehacer en computación; revisaremos aquellas abstracciones que guardan relación con nuestros algoritmos genéricos, para terminar con una definición más precisa y completa que la que acabamos de dar de lo que entendemos por algoritmo genérico.

1.1. El concepto de abstracción

1.1.1. Distintos significados para abstracción en computación

Una de las tareas fundamentales de las ciencias de la computación es la *abstracción* de problemas y su solución. Al respecto, en su libro "Structure and Interpretation of Computer Programs" [ASS96], los autores Abelson y Sussman inician el primer capítulo con una cita del filósofo John Locke en su trabajo sobre el entendimiento humano que data de 1690[Loc90]:

> *"Los actos de la mente, en la medida en que ejercen su poder sobre ideas simples, son fundamentalmente estos tres: 1) Combinar varias ideas simples en una compleja, y de esta manera se forman todas las ideas complejas. 2) La segunda es la de considerar dos ideas, sean éstas simples o complejas, una al lado de la otra para poder observarlas simultáneamente, sin combinarlas en una sola, y de esta manera obtener todas sus ideas de interrelaciones. 3) La tercera es separarlas de todas aquellas ideas que las acompañan en su existencia real: a esto se le llama abstracción y es como todas sus ideas generales se hacen."*

Esta cita de Locke nos indica que la abstracción empieza en ideas simples y progresa hacia lo complejo mediante la combinación y comparación de ideas simples o complejas, para

finalmente distinguir y aislar las características individuales de cada idea. En efecto, esta es la forma en que los autores de este libro sobresaliente atacan su objetivo: transmitir las ideas principales sobre la programación de computadoras, construyendo primero herramientas sencillas y combinándolas posteriormente para obtener utensilios cada vez más complejos y poderosos. También es con este mecanismo de abstracción que enfrentan el problema de medir la complejidad de los sistemas de cómputo y demostrar su correctez: hacerlo para componentes simples de manera exhaustiva, y posteriormente dejar que los sistemas más complejos se expresen en términos de sus componentes, en cuanto a la complejidad se refiere – ver [ASS96, Prefacio, tercer párrafo].

Hay otras maneras de ver el proceso de abstracción, que van de lo complejo hacia lo simple, como se expresa en la cita de Burloud que aparece en [IHB+00, Pág. 353] – tomada de [Bur27]:

> *"Lo* abstracto *propiamente dicho empieza con la comprensión de similitud y diferencia. La abstracción deberá distinguir las características comunes compartidas por varios objetos de aquellas exclusivas de un objeto en particular."*

En la cita de Ribot que sigue a ésta en [IHB+00, Pág. 353], tomada de [Rib97] se expresa un mecanismo similar:

> *"[...] en términos estrictamente científicos la abstracción tiene que descubrir las características que son constitutivas para un grupo ... que representan a un grupo y nos permiten pensar respecto al grupo como tal."*

Hay un contraste fuerte entre la cita de Locke y las de Burloud y Ribot. Mientras que el primero parte de lo simple hacia lo complejo, los segundos pretenden observar lo complejo para de ahí deducir propiedades simples. En general, si consultamos en diccionarios, o en los múltiples trabajos científicos que hablan al respecto, la idea de abstracción se refiere más al significado dado por Burloud y Ribot, que al dado por Locke.

1.1.2. El concepto de algoritmo

La abstracción fundamental que permite desarrollos en matemáticas y, de manera importante, en computación, es la dada por el álgebra. Con el uso de variables simbólicas en lugar de números se avanza en la dirección de dar soluciones a problemas en términos de recetas, o mejor aún, algoritmos – no se puede pensar en el desarrollo de lenguajes de programación sin el uso de variables simbólicas. Un algoritmo, entonces, se expresa en términos de abstracciones, ya que no concreta en ningún momento a los datos sobre los que actúa.

A la par con el desarrollo de algoritmos para calcular, se da un desarrollo en implementos (herramientas físicas) de cómputo capaces de llevar a cabo estos algoritmos. En el desarrollo de los implementos de cómputo, el desarrollo se da en el sentido de Locke: de lo particular a lo general; se diseñan equipos que resuelven operaciones elementales, para más tarde integrar estos equipos en equipos cada vez más complejos y que automatizan una porción mayor del

cómputo [IHB+00, Cer03]. Desde la época de Babbage se da la noción de una máquina capaz de llevar a cabo algoritmos que correspondan a cálculos.

En 1900, en el Congreso Internacional de Matemáticas en París, Hilbert plantea lo que él considera serán los 23 problemas más importantes que van a ocupar a los matemáticos en el Siglo XX. Algunos de ellos son muy amplios y nunca van a ser considerados como terminados, como la axiomatización de la Física (problema 6). Varios de los problemas expuestos tienen un tema común: definir algoritmos para una gama de actividades matemáticas como las demostraciones de teoremas y el cálculo de ciertas funciones. Empieza entonces un esfuerzo por definir qué es un algoritmo, en términos de qué es automatizable, y cuál debe ser el mecanismo que lleve a cabo la automatización.

El siguiente paso natural es el abstraer a los algoritmos mismos, dando descripciones más o menos informales e intuitivas de lo que es un algoritmo: un proceso o método para calcular algo, donde cada paso es ejecutable por un ser humano, y que no requiere de inventiva o creatividad por parte del ejecutante para llevarlo a cabo. Aun cuando se reconocen recetas para resolver un sinfín de problemas y cuyos orígenes datan de la Antigüedad, es en la primera mitad del siglo XX de nuestra era en que se intenta formalizar el concepto de computabilidad, y con él el de algoritmo, fundamentalmente en lo que respecta a la restricción de que el proceso debe poderse aplicar mecánicamente. Una parte muy importante es la de definir qué es un paso básico o sencillo, y en términos de éstos expresar algoritmos. En este sentido se llega a distintas abstracciones, como la Máquina de Turing, Algoritmos de Markov, Cálculo Lambda de Church y Kleene, Sistemas de Post, y el lenguaje de operaciones de Davis – ver entre otros [Dav58, Koz97, Tay98, BBJ02] donde aparecen la mayoría de estas formalizaciones – por mencionar algunos de los más relevantes.

Cada una de estas abstracciones, aunque en el fondo se preocupan por el tema de computabilidad, plantean aspectos distintos de un mismo problema. Por ejemplo, Turing plantea la existencia de una máquina que pueda realizar cómputos, y cuyos pasos discretos son muy elementales; los Algoritmos de Markov y los Sistemas de Post presentan una sucesión de reglas, donde en teoría cualquiera de ellas, si es aplicable, puede ser seleccionada como siguiente, pero que su aplicación está determinada realmente por la historia de ejecución que se haya presentado, o en otras palabras, del estado en el que se encuentre el sistema en ese momento; el Cálculo Lambda define funciones computables en términos aplicación y composición de funciones básicas; y mencionaremos por último, el lenguaje de operaciones de Davis define un conjunto de cuatro operaciones básicas tales que con combinaciones de ellas se puede calcular cualquier función computable.

Todas estas abstracciones que mencionamos pretenden formalizar al mismo objeto: un cómputo. La abstracción que se considera la representante de estas formalizaciones es la Máquina de Turing. La tesis de Church-Turing relaciona entre sí a estas distintas formalizaciones, y se refiere al tema de computabilidad, aseverando que las Máquinas de Turing y el Cálculo Lambda representan una abstracción correcta de lo que es una función computable, y que ambas se refieren al mismo conjunto de funciones.

Alrededor de estos temas se precisa de mejor manera lo que es un algoritmo, en términos de poderse representar en alguna de las abstracciones expuestas.

1.1.3. La abstracción en los lenguajes de programación

Con el advenimiento de los equipos de cómputo poco antes de empezar la segunda mitad del Siglo XX, y una vez definido el concepto de algoritmo, se desarrolla un esfuerzo por ejecutar algoritmos en computadoras. En el desarrollo de implementos de cómputo se dan dos tipos de esfuerzo: el primero de ellos, frente a un tipo particular de calculadora o computadora, se procede a "traducir" el algoritmo que se desea ejecutar a pasos elementales en el equipo de cómputo con que se cuenta; este proceso, no importa desde qué punto de vista lo veamos, es el inverso de la abstracción, ya que desmenuzamos aún más las ideas para expresarlas en un lenguaje que no mantiene relación con el algoritmo original. El segundo esfuerzo se da en el sentido de mantener un cierto nivel de abstracción al expresar algoritmos, independientemente de la manera en que serán llevados a cabo, pero buscando claridad y transmitir lo que pudiéramos entender como la "esencia" del algoritmo. Es claro que la especificación de un algoritmo en términos de un programa para una Máquina de Turing, si bien es correcta, no ayuda a entender el funcionamiento del algoritmo. Para expresarlo, siguiendo la noción de Locke de lo que constituye la abstracción, dada en la sección 1.1.1, es necesario agrupar los pasos individuales para obtener procesos más complejos, que a su vez se combinen para alcanzar áun mayor complejidad.

La historia de los lenguajes de programación se ve, por supuesto, influida por ese deseo de poder expresar cómputos de una manera abstracta, sin preocuparse por llevarlos a ser ejecutado en un dispositivo de cómputo particular. El diseño de lenguajes ensamblador pretende en su momento subir el nivel de abstracción de los programas ejecutables, de tal manera que el lector pueda reconocer grupos de instrucciones con un fin particular. Los lenguajes de programación de alto nivel, al ser descriptivos de una cierta área de aplicación, pretenden mantener un cierto nivel de abstracción en el cómputo, para no obligar al científico – al programador – a preocuparse de la computadora particular a usar, sino únicamente de la sucesión de pasos, en términos matemáticos, necesaria para llevar a cabo el cómputo.

El siguiente paso en la abstracción de algoritmos se da en términos de parametrización de funciones, agrupando problemas y jugando con los parámetros para que una misma función pueda realizar cómputos de diversa naturaleza. Esto se lleva a cabo con el diseño de lenguajes de programación que observan mecanismos de paso de parámetros sofisticados, como es el caso de Algol-60. En este lenguaje se maneja el paso de parámetros por valor y por nombre. El primero es el usual, donde se entrega al procedimiento o función una copia del argumento; el segundo corresponde a una evaluación del "texto" dado para el parámetro lo más tarde posible, esto es hasta que se usa dentro de la función invocada.

1.1.4. Un algoritmo genérico basado en las características de Algol

Uno de los primeros ejemplos de intentos por tener algoritmos genéricos (en ese entonces llamados "generales") se presenta en el caso de lo que Knuth y Merner llaman "modestamente" su Solucionador General de Problemas (en inglés *General Problem Solver*) – ver [KM61] – en el que tenemos en un código muy sencillo, que aparece en el Listado 1.1, una función que, usada de manera no trivial, calcula entre otros el producto interno de dos vectores (Lista-

do 1.2); la multiplicación de dos matrices dejando el resultado en una tercera (Listado 1.4); y el m-ésimo número primo (Listado 1.7). Además de que este ejemplo ilustra el poderío introducido por el mecanismo de paso de parámetros por nombre, también pretende convertirse en un modelo más de funciones computables, y por ende, de algoritmo.

Listado 1.1: Listado para GPS.

```
1    real procedure GPS(I, N, Z, V); real I, N, Z, V;
2    begin
3        for I := 0 step 1 until N do Z := V;
4        GPS := 1
5    end;
```

Al observar el Listado 1.1 hay que tener presente el mecanismo de paso por nombre de los parámetros. El cuerpo del GPS es simplemente una iteración sencilla en la que se hace una asignación. Pero observemos el Listado 1.2, donde para el cuarto parámetro, V, aparece la expresión, Z + A[I] * B[I].

Listado 1.2: Producto interno de dos vectores.

```
1    /* Producto interno de dos vectores con N elementos. */
2    Z := 0;
3    I := GPS(I, N, Z, Z + A[I] * B[I]);
```

Esto hace que la función se traduzca a como se ve en el Listado 1.3.

Listado 1.3: GPS con parámetros sustituidos.

```
1    /* Con esta llamada: */
2    GPS(I, N, Z, Z + A[I] * B[I])
3    /* Se sustituye en la función así: */
4    begin
5        for I := 0 step 1 until N do Z := Z + A[I] * B[I];
6        GPS := 1
7    end;
```

En el caso de la multiplicación de matrices, la invocación se hace como se ve en el Listado 1.4 En este caso es un poco más difícil de vislumbrar, aun dada la sangría. La multiplicación se obtiene de dos llamadas a GPS, cuyos resultados se multiplican entre sí. Sin embargo, nótese que GPS siempre regresa el valor 1, por lo que los resultados se están dejando realmente en los parámetros. En la primera llamada a GPS, línea (2), lo único que se hace es inicializar al primer elemento de la matriz C en ceros para acumular ahí el resultado de la multiplicación, como se puede ver de la sustitución que se hace con la llamada por nombre en el Listado 1.5.

Listado 1.4: Multiplicación de dos matrices.

```
1   /* Multiplicación de A[1:m, 1:n] por B[1:n, 1:p] dejando el resultado en C[1:m, 1:p] */
2     I := GPS(I, 1.0, C[1,1], 0.0) ×
3          GPS(I,
4              (m−1) ×
5              GPS(J, (p−1)×
6                     GPS(K, n, C[I, J], C[I, J] + A[I, K] × B[K, J]),
7                  C[I, J + 1], 0.0),
8              C[I + 1, 1],
9              0.0);
```

Listado 1.5: Primera llamada a GPS en Listado 1.4, línea (2).

```
1     real procedure GPS(I, 1.0, C[1,1], 0.0); real I, N, Z, V;
2     begin
3         for I := 0 step 1 until 1.0 do C[1,1] := 0.0;
4         GPS := 1
5     end;
```

En la segunda llamada en la línea (3), tenemos dos nuevas llamadas a GPS anidadas. Sin embargo, como ya mencionamos, los resultados parciales se van almacenando en los argumentos. Como el orden de evaluación será de adentro hacia afuera, traduzcamos primero la llamada que se hace en la línea (6) y que se muestra en el Listado 1.6.

Listado 1.6: Sustitución de invocación en línea (6).

```
1     real procedure GPS(K, n, C[I,J], C[I,J] + A[I,K] * B[K,J]);
2                             real I, N, Z, V;
3     begin
4         for K := 0 step 1 until n do
5             C[I,J] := C[I,J] + A[I,K] * B[K,J];
6         GPS := 1
7     end;
```

Como se puede ver de esta sustitución, esta llamada a GPS correspondería a la iteración más interna de la codificación común de una multiplicación de matrices. Si hacemos la sustitución en la siguiente llamada "hacia afuera" en la línea (5) veremos que se trata de la iteración para J, y la llamada de la línea (3) corresponde a la iteración sobre I. En la línea (8) es cuando se inicializa a la "siguiente" C[i+1, 1] por renglón, mientras que en la línea (7) se inicializa en ceros la "siguiente" columna para C[I, J+1].

Podemos observar un comportamiento parecido en la invocación que calcula el m-ésimo número primo. Hay que tener presente que la evaluación de los parámetros, en este caso el if aritmético, se llevará a cabo ya dentro de la invocación, y el número primo se dejará en el parámetro P, ya que GPS, insistimos, siempre regresa 1. Acá nuevamente, si observamos la llamada anidada, ésta se usa para detectar si el parámetro Z alcanza el valor de 1, para

probar con otros divisores. El explicar con detalle también este caso no aporta nada adicional para la comprensión del concepto de paso de parámetros por nombre, por lo que lo omitimos.

Listado 1.7: Cálculo del m-ésimo número primo.

```
1            /* Cálculo del m-ésimo número primo usando GPS: */
2       I := GPS( I ,
3                  if I = 0
4                  then −1.0
5                  else I ,
6                  P,
7                  if I = 1
8                  then 1.0
9                  else if GPS(A, I, Z, if A = 1
10                                      then 1.0
11                                      else if entier(A) ×
12                                              (entier(I) ÷ entier(A))
13                                              = entier(I) ∧ A < I
14                                          then 0.0
15                                          else Z) = Z
16                      then (if P < m
17                            then P + 1
18                            else I × GPS(A, 1.0, I, 0.0))
19                      else P);
```

De hecho, en el artículo antes citado los autores sostienen que con esta simple función pueden calcular cualquier función computable, y remiten al lector a [Dav58], quien establece que con cuatro operaciones básicas se puede calcular cualquier función computable. Éste es un ejemplo temprano de *reutilización*, ya que el procedimiento original se reutiliza de diversas maneras para obtener resultados diferentes, aprovechando el mecanismo de paso de parámetros por nombre definido para Algol-60. Podemos pensar en el GPS como un algoritmo genérico que puede realizar tareas muy disímiles entre sí, dependiendo del contexto en el que se le use. Sin embargo, dado lo simple del algoritmo, no presenta propiedades abstractas interesantes que no se deriven del mecanismo dado por el lenguaje, respecto a los problemas particulares que puede resolver, .

1.1.5. La programación estructurada como una abstracción

En este proceso de abstracción conseguido a través de lenguajes de programación, el siguiente paso es el de la programación estructurada, que pretende dar un desarrollo de los algoritmos de lo general a lo particular. En esta estrategia se sigue el principio de abstracción descrito por Burloud y Ribot y citado al principio de este capítulo, que consiste en reconocer características comunes en los problemas, sobre todo en las estructuras de control, y mediante parametrización, atacarlos con herramientas similares. Tal es el caso de, por ejemplo, la programación lineal, los algoritmos genéticos, los algoritmos que involucran a agentes, los algoritmos distribuidos, etc. En estos enfoques, se abstrae el problema particular a un marco conceptual específico, y desde ese marco se analiza y resuelve el problema. Una vez en un

cierto marco de referencia, los distintos problemas se resuelven en lo individual, siguiendo ciertos lineamientos generales. Es importante resaltar, sin embargo, que si bien se reutiliza la abstracción del problema, cada algoritmo particular se desarrolla independientemente, en algunos casos reescribiendo en términos de parametrización. También el establecer la correctez o la de complejidad se hacen de manera individual para cada algoritmo.

1.1.6. Estructuras de datos abstractas

Otra estrategia, que responde a la noción dada por Locke para abstracción y con la que empieza este capítulo, se refiere a observar un problema primero en términos de subproblemas más sencillos, de tal manera que la conjunción de estos problemas más sencillos dé solución al problema original. Este mecanismo se da ampliamente cuando se empieza a trabajar con estructuras de datos abstractas en los lenguajes de programación, ya que se definen primero estructuras de datos que cumplen con ciertas propiedades, y la solución al problema general se hace en términos del uso y explotación de estas estructuras de datos. Con la programación orientada a objetos se unifican los métodos sobre las estructuras con las estructuras mismas, logrando encapsulamiento y ocultando la implementación de las mismas – ver [Sha84]. El nivel de abstracción que se alcanza es, sobre todo, respecto a las estructuras de datos, mientras que en la programación estructurada es, como mencionamos, sobre las estructuras de control. Sin embargo, en esta etapa para establecer la correctez de los algoritmos se sigue un proceso de abajo hacia arriba, de lo particular – las estructuras de datos cumplen con ciertas propiedades – hacia el algoritmo general.

La orientación a objetos responde también, sin duda, a la concepción de abstracción de Burloud y Ribot, ya que lo primero que hace es reconocer características comunes y determinar cuáles son los atributos que unifican para posteriormente definir aquellos que distinguen. Sin embargo, hasta el momento se ha quedado un poco corto este enfoque, ya que clasifica y generaliza fundamentalmente alrededor de estructuras de datos, no de los algoritmos que las utilizan. También, algo que nos preocupa sobremanera en este trabajo y como se menciona en [Sha84], para establecer la correctez se sigue un proceso que va de lo particular a lo general.

1.2. La complejidad de un algoritmo

Para la computación en general no es suficiente que una función sea computable, ya que el cálculo de la misma puede llevarse, en la práctica, tiempos excesivos, por lo que profesionales de la computación (y las matemáticas) se dan a la tarea de clasificar funciones computables. En la década de los 60 surge el concepto de *complejidad* y se clasifican los algoritmos de acuerdo a ella. Se define la complejidad de un algoritmo en términos del tamaño de su entrada, y se da como una función de ese tamaño. Posteriormente, en 1963 Shepherdson y Sturgis presentan su abstracción de Máquina de Acceso Directo (en inglés *Random Access Machine*), que identifica operaciones básicas similares a las de una computadora con arquitectura de Von Neuman, y con la cual establecen la forma de medir la ejecución de un algoritmo. A

partir de este momento se empiezan a publicar algoritmos, la mayoría de ellos en uso ya muchos años, pero que no han aparecido hasta ese momento de manera formal, esto es, con la demostración de correctez y medida de complejidad. Hoy en día se usa a la Máquina de Turing para clasificar algoritmos en términos de su complejidad y tratabilidad a un nivel abstracto, aunque en la práctica generalmente se utiliza la máquina de acceso directo de Shepherdson y Sturgis.

En la década de los 70 deja de ser suficiente que una función sea computable – pueda evaluarse aplicando un algoritmo – sino que se hace necesario que el problema sea *tratable*, esto es, que la ejecución del algoritmo dado se lleve un tiempo *razonable*. Se define a un problema como tratable si es que el algoritmo dado para su solución tiene una función de complejidad asociada que es polinomial. Si la función de complejidad está por encima de un polinomio, se dice que el problema es *intratable*.

Por la manera en que se mide esta complejidad, en la práctica un algoritmo exponencial puede ser usado para ciertos casos, y también en la práctica un polinomio con término significativo con un exponente muy grande puede no ser utilizado. Sin embargo, hablamos de tamaños de entrada muy grandes; la medida de complejidad de un algoritmo se abstrae para calcularse precisamente en el límite. En ese punto, esta clasificación de algoritmos de acuerdo a su complejidad tiene muchísimo sentido.

En este trabajo usaremos indistintamente el término *eficiencia* y el de *clase de complejidad*, ya que nos estaremos refiriendo al desempeño de un algoritmo, medido en términos de su función de complejidad, no cualitativamente o en términos comparativos.

1.3. Generalización de algoritmos

En el área de algoritmos hay un esfuerzo constante por *unificar* algoritmos, no únicamente en cuanto a filosofías – como es el caso, por ejemplo, de la programación lineal o los procesos distribuidos con ficha – sino también en cuanto a distintos enfoques para resolver el mismo problema (*v. g.* [Sma93, Boy04]). No hay que desestimar que en el área de algoritmos, tres aspectos colaboran para determinar lo adecuado de un algoritmo: el diseño del algoritmo, la demostración de que el algoritmo es correcto y el cálculo de la clase de complejidad a la que el algoritmo propuesto pertenece.

1.3.1. Familias de algoritmos

Aun cuando la unificación y reutilización de algoritmos es una meta siempre presente en las ciencias de la computación, a partir de la programación estructurada el desarrollo de abstracciones se busca más eficientemente a través de los paradigmas de diseño y programación. En el área de la programación funcional, por ejemplo, el hecho de que las funciones sean entidades de primer orden y puedan ser pasadas como parámetros, provee mecanismos de abstracción y particularización en el sentido que buscamos y proveen marcos de desarrollo importantes para la programación automática. Con la orientación a objetos tenemos un

enfoque que se presta muy bien para diseñar esquemas abstractos de especificación, donde la herencia y el polimorfismo proveen los mecanismos para refinar un algoritmo de tal manera que las especializaciones correspondan a los algoritmos particulares que se agruparon. La orientación a objetos empuja hacia definir los problemas en términos de abstracciones, clasificando las herramientas para su solución en un esquema jerárquico, donde en la base de esa jerarquía no se presenta más que una idea general de *qué* es lo que se debe hacer, dejando para niveles inferiores en la jerarquía le especificación precisa de *cómo* hacerlo.

En el terreno de la abstracción de algoritmos, si usamos el enfoque de Burloud y Ribot dado en la sección 1.1.1, de primero agrupar y después distinguir, al buscar características comunes en diversos algoritmos se pretende definir familias de algoritmos que respondan a una estrategia general común.

Los primeros trabajos en el sentido de definir a una familia de algoritmos desde la raíz de un árbol jerárquico, aunque con el paradigma funcional, se dan, como mencionamos en la introducción de este trabajo, en [Mer85], en programación automática, donde el *qué* se especifica en términos de predicados que deben cumplir el estado inicial y final de los procesos. En el trabajo citado, este enfoque se usa para presentar algoritmos de ordenamiento, clasificándolos de acuerdo a dos procesos: el proceso de *dividir* y el proceso de *unir*, pudiendo cada uno de estos procesos hacerse de manera fácil o difícil; las especializaciones se dan como las combinaciones de estas dos posibilidades. Merritt también trabaja con árboles generadores [Mer89] bajo el mismo esquema. Para establecer la correctez de los algoritmos se recurre a que el planteamiento inicial está expresado en términos de los predicados generales que debe cumplir el algoritmo, lo que no colabora a establecer puntos intermedios de verificación.

1.3.2. La reutilización como objetivo

Con la abstracción de algoritmos se promueve la reutilización, ya que al refinar y concretar aspectos de una abstracción se definen módulos reutilizables en los niveles superior e intermedios. Siendo uno de los objetivos de diseño en la orientación a objetos la reutilización, se recurre a diversos mecanismos de abstracción para lograr aquélla.

En 1994 aparece el libro "Design Patterns, Elements of Reusable Object-Oriented Software" de Gamma *et al* [GHJV94], donde los procesos se abstraen un nivel más, definiendo patrones de programación que deben seguirse para implementar distintos tipos de aplicaciones. A partir de este momento, los esfuerzos en el desarrollo de aplicaciones con lenguajes orientados a objetos buscan adaptarse a alguno de los patrones descritos en ese libro, o a diseñar patrones nuevos para el desarrollo de aplicaciones. El patrón *Template* – en español plantilla – es el que más se acerca al concepto de algoritmo, ya que en él se delegan los pasos exactos de un algoritmo a las subclases; se queda un poco corto como plantilla de diseño para casos concretos, ya que no delimita aquello que se debe delegar. Otro de los patrones que pudiera usarse en el campo de análisis de algoritmos, el patrón *Strategy* – estrategia – simplemente encapsula a un algoritmo determinado en una clase. Tenemos presentaciones de algoritmos sencillos de ordenamiento y búsqueda para el segundo curso de programación, como en [NW01], que utilizando ya sea programación funcional o programación orientada a objetos, trata de encasillar a los algoritmos que revisa usando patrones de diseño. La preocu-

pación fundamental en este tipo de trabajo es el diseño, pero se continúa con un tratamiento de las demostraciones de correctez que no se benefician de esta abstracción o de la herencia.

El anhelo de tener reutilización en el código debe extenderse a la reutilización de todo el proceso de creación de los algoritmos, que involucra tanto el diseño como la verificación y el análisis. Cuando estamos hablando de algoritmos, no de aplicaciones, los patrones de software no aportan en esta dirección.

1.3.3. Programación genérica

Otro esfuerzo significativo en la reutilización de código es el de la programación genérica, definida de manera muy adecuada en [DS00, página 1] de la siguiente manera:

> *"La programación genérica depende de la descomposición de programas en componentes que pueden ser desarrollados por separado y combinados arbitrariamente, sujetos únicamente a interfaces bien definidas. [...]"*

C++ ha sido el lenguaje en el que más se ha trabajado en este sentido, aunque los resultados no han sido exclusivamente en el campo de la orientación a objetos. La biblioteca STL [MS96] para C++, así como muchos de los trabajos que se presentan en el taller de Dagstuhl[JLM00], se refieren a bibliotecas con tipos parametrizados y que constituyen el siguiente paso lógico después de las estructuras de datos abstractas [Sha84].

También en el contexto del taller de Dagstuhl, en el marco de la programación genérica, se presentan varios trabajos donde se utiliza la programación automática o programación adaptiva, en los que se plantea un programa genérico que se particulariza al adaptarse a datos específicos. En este tipo de programación genérica se genera, valga la redundancia, un programa "fresco" cada vez que se somete al programa genérico a un conjunto de datos. Mencionamos, para ejemplificar, el caso de apareamiento de cadenas [CD93], donde se define un programa *abstracto* del que, mediante evaluación parcial se *especializan* o generan programas particulares con las características de complejidad deseadas. Esto se logra generando programas concretos que toman en cuenta la entrada particular con la que van a trabajar. Específicamente en [ACDM01], las pruebas de correctez para los programas generados son sobre la base de los puntos en los que se generan funciones *ad hoc* para la entrada que se está considerando. Sin embargo, no se cumple en términos estrictos con los requisitos de medidas de complejidad y correctez en general, ya que los programas generados son eficientes, en el caso de apareamiento de cadenas, respecto a un patrón fijado para que se lleve a cabo la evaluación parcial, y es a este nivel en el que se puede demostrar la correctez y evaluar la clase de complejidad. Los resultados no pueden ser generalizados a cadenas de caracteres arbitrarias.

1.3.4. Programación adaptiva

En la literatura aparece muchas veces el término *algoritmo genérico* para referirse a algoritmos con un propósito específico pero que pueden adaptarse a tipos variados de datos, y que más bien responden a la parametrización de tipos de la que hablamos antes. Tal es el caso de ciertas bibliotecas de C++ ([MN98]) donde lo genérico se refiere a que los datos presentados a la aplicación pueden estar en estructuras distintas o ser de alguno de muchos tipos preestablecidos (números, cadenas, arreglos, etc.), pero no definen niveles de especialización o niveles de abstracción aplicables a los diferentes ejemplares de datos, sino que aplican un mismo algoritmo optimizado para que responda adecuadamente a cada uno de los posibles tipos. El algoritmo, para decirlo en otras palabras, es único pero trabaja sobre la definición abstracta de los datos. Un caso similar se da con algoritmos adaptivos, donde según la forma de los datos, elige una estrategia particular para hacer la tarea encomendada – por ejemplo, [ECW92, BK02, DLOM00] – pero todas las estrategias están al mismo nivel y se encuentran disponibles en el algoritmo adaptivo: simplemente, después de un análisis de la estructura de los datos, se elige la "mejor" estrategia de entre las disponibles. No se abstrae en ningún punto el diseño del programa, sino que se analizan conjuntos posibles de datos.

1.3.5. Unificación de algoritmos

Biedl *et al* en [BCD+01], como mencionamos en la presentación, introducen a un conjunto de algoritmos de ordenamiento por intercambio, que agrupan en una misma familia a algoritmos cuya forma de funcionamiento no es del todo compartida. Sin embargo, como ya mencionamos, a pesar de dar un marco de referencia para demostrar que estos algoritmos son correctos, esto sólo lo hace a nivel de exigir que al final del algoritmo los datos queden ordenados; no especifica invariantes que unifiquen la ejecución de estos algoritmos y que se puedan aplicar a todas las especializaciones.

También en la presentación hablamos del tipo de unificación que se hace en [CLRS01, AMO93], donde se trabaja sobre todo en términos de posibles parámetros. En cambio, en este trabajo se hace énfasis en la *reutilización*, tanto de demostraciones como de código.

Como también ya mencionamos en la presentación de este trabajo, un muy buen ejemplo de unificación de algoritmos lo tenemos en [Boy04], donde se da un marco de referencia formal para la unificación de tres algoritmos lineales para determinar si una gráfica es plana, y si lo es, proponer una planarización de la misma [LEC66, BM99, SH99]; en este trabajo el marco conceptual permite establecer la correctez de la manera en que se implementan estos tres algoritmos; como ya mencionamos, este trabajo coincide con el nuestro en términos de que da una estrategia general que invoca a métodos que cumplen con las especificaciones de tipo *requiere/provee* [KV00]. También mencionamos ya el trabajo de Small [Sma93] para unificar el algoritmo de Lempel, Even y Cederbaum [LEC66] con el de Hopcroft y Tarjan [HT74]. Esto lo puede hacer el autor ya que ambos algoritmos, desde sus orígenes, planean una estrategia muy parecida en la que lo que cambia es la manera cómo deciden cuándo y dónde agregar aristas a la gráfica plana que se está construyendo. También adopta la estrategia planteada en [KV00].

A pesar del éxito que ha tenido el paradigma de orientación a objetos, y de los mecanismos que provee para abstraer conducta y agrupar problemas similares, su uso no se ha permeado como se hubiera esperado en el área de análisis de algoritmos, sobre todo en lo que se refiere a las demostraciones de correctez y cálculo de clase de complejidad que pudieran derivarse a lo largo de las líneas de la herencia y el polimorfismo.

1.4. Definición de algoritmo genérico

Una definición de algoritmo genérico es la que presenta Austern en [Aus00]:

> *Un algoritmo genérico es lo que resulta de elevar a un algoritmo concreto al nivel más general posible sin que pierda eficiencia*[1]*; esto es, la forma más abstracta del algoritmo tal que cuando sea especializado de regreso al caso concreto el resultado sea un algoritmo tan eficiente como el original.*

Esta definición, que viene del área de programación genérica, tiene que ver, sobre todo, con el desempeño del algoritmo, no con la manera de "elevarlo" o la manera de concretarlo. Adicionalmente, el nivel más general posible es aquel que únicamente especifica el objetivo del algoritmo, por lo que esta definición, aún cuando va en la dirección correcta, no nos satisface ya que no se aplica al tipo de trabajo que realizamos. Veamos otra definición que aparece en [MSL00], tomada de [Col]:

> *El concepto de algoritmo como lo define Knuth en [Knu98a] se caracteriza por un conjunto de cinco propiedades: definición, finitud, entrada, salida y efectividad. Como observó Collins [Col], si relajamos el requisito de definición – de que cada paso del algoritmo esté definido de manera precisa – nos lleva al concepto de un* algoritmo abstracto *o un* algoritmo esquemático*, términos utilizados anteriormente a lo que se identifica hoy en día como un algoritmo genérico. Entonces un algoritmo en el sentido tradicional puede ser considerado como un algoritmo genérico para el cual todas las fuentes de indefinición – generalidad – han sido definidos.*

Siguiendo esta definición, proponemos el diseño de *algoritmos genéricos* que corresponden al nivel de clases abstractas en la orientación a objetos. Empezamos por agrupar a una cierta clase de algoritmos en una estructura jerárquica arborescente. La raíz de esta jerarquía, el nivel más abstracto, está representado por una clase abstracta (*algoritmo genérico*), donde se presenta la estrategia general en forma de algoritmo, invocando a métodos por definir, para resolver esa clase de problemas. Además de la estrategia general, en el algoritmo genérico se encuentran los atributos y campos comunes a los algoritmos de la clase, y quedan por definir implementaciones concretas de los métodos invocados por la estrategia general, que en ese punto se presentan como métodos abstractos. Como componente fundamental de

[1] Austern se refiere acá a la complejidad del algoritmo, considerando los términos menores y las constantes (ver Anexo A).

este enfoque, se utiliza la jerarquía dada por la herencia para definir propiedades a nivel del algoritmo genérico, que se heredan a las especializaciones. Asimismo, las medidas de complejidad se dan en términos de la estrategia general y como función de la complejidad de los métodos abstractos invocados.

Identificamos familias de algoritmos en términos de que sigan una misma estrategia genérica, y cumpliendo a ese nivel los mismos predicados. A la implementación de los métodos abstractos – que denominamos métodos polimorfos – se les exige que no violen las especificaciones dadas en el nivel abstracto. Al especializar (o extender) a la clase abstracta, se especifican predicados adicionales que debe cumplir el algoritmo particular.

En este trabajo un algoritmo genérico es representante de toda una familia de algoritmos concretos, aquellos que resultan de darle distintas implementaciones a los pasos genéricos del algoritmo abstracto. Puede haber niveles intermedios de abstracción que resultan de subfamilias con características en común. Siguiendo las definiciones de abstracción dadas por Ribot y Burloud y citadas al principio de este capítulo – donde la construcción de un algoritmo genérico, abstracto, empieza con el reconocimiento de características similares en diversos algoritmos, como se menciona en [Aus00] – buscamos que cada uno de los algoritmos concretos pueda ser elevado a una forma común a todos ellos. A esta forma común es a lo que llamamos el *algoritmo genérico*. Lo que resulta es un algoritmo en el sentido de la definición de Collins. A cada forma de definir los pasos abstractos del algoritmo genérico le llamamos una *especialización*, siguiendo la terminología de la orientación a objetos. Esta abstracción se encuentra en el marco de la orientación a objetos, tanto en la parte que corresponde a la programación de los algoritmos, como a la parte formal del análisis de los mismos, utilizando para ello los conceptos de herencia y polimorfismo. El algoritmo genérico corresponde a una clase abstracta, y la especialización a cada algoritmo concreto corresponde a una clase que hereda de la abstracta. Es importante mencionar que se pueden dar varios niveles de abstracción para una familia dada de algoritmos.

1.5. Uso de herencia en análisis y verificación

Como ya mencionamos, la abstracción es uno de los mecanismos fundamentales en toda labor científica y es una herramienta fundamental para la solución de problemas en las ciencias de la computación. Proponemos una manera novedosa de abstraer algoritmos que permite la reutilización no sólo de la estrategia general del algoritmo, algo que es natural para los algoritmos genéricos, sino también, y de manera preponderante, del trabajo que se hace para demostrar la correctez del mismo y calcular la clase de complejidad a la que pertenece.

Con el advenimiento de las estructuras abstractas de datos y, posteriormente, la programación orientada a objetos, se hace mucho trabajo en el sentido de agrupar y clasificar algoritmos y estructuras de datos. Sin embargo, el esfuerzo es más el de construir componentes sólidos, tanto en términos de desempeño como de correctez, para que al ser utilizados estos componentes en un algoritmo o en un sistema se pueda garantizar su desempeño [Sha84]. La orientación a objetos promueve el tipo de abstracción definido por Locke, de lo particular a lo general, sobre todo en lo que concierne a construir bibliotecas de clases, y a la utilización

de estructuras de datos abstractas. Más aún, los algoritmos a los que se refiere la programación genérica son algoritmos clásicos sobre estructuras de datos, como ordenamientos y búsquedas en contenedores de muy distintos tipos. En particular, el trabajo que hay respecto a algoritmos más elaborados, como los que trabajan con gráficas [LPS00, Kül00], abstracciones matemáticas como la geometría [KW00] o campos algebraicos [Sch00], se refieren a lo genérico de las estructuras de datos, más que a una estructura que extienda a un algoritmo abstracto.

1.5.1. Beneficios del uso generalizado de la herencia

Cuando logramos conformar una familia de algoritmos de la manera planteada en este trabajo, tenemos una mejor idea de cuáles son las características en común de los algoritmos que conforman una familia, y tal vez más importante aun, cuáles son las diferencias que provocan un comportamiento distinto para cada una de las especializaciones.

Este tratamiento de familias de algoritmos promueve el diseño de nuevos algoritmos, resultado de variaciones en las especializaciones. Una vez dada una jerarquía dentro de una familia, es relativamente fácil ser creativo para colocar un nuevo elemento en ese árbol jerárquico, dando implementaciones poco o muy distintas de los métodos abstractos a ese nivel. No sólo se obtienen nuevos algoritmos con muy poco esfuerzo, sino que además el contexto en el que se diseñan estos nuevos algoritmos permite la reutilización de las demostraciones de correctez y los cálculos de complejidad. Se requiere de poco trabajo adicional para que la demostración particular de correctez y el cálculo de la clase de complejidad queden completos.

1.5.2. Los algoritmos genéricos en la docencia

También en el área de docencia resulta muy útil este enfoque. Hay muchos y muy variados cursos de estructuras de datos y algoritmos, cuyo objetivo principal es el de ampliar el repertorio de recursos computacionales de los estudiantes y fomentar la creatividad en el diseño de nuevos algoritmos. Este tipo de cursos juega un papel mucho muy importante en la formación de profesionales de computación. Nuestra preocupación se centra, sin embargo, en cursos más especializados donde las nociones de correctez y complejidad sean centrales. Es en este tipo de cursos donde el tema de análisis de algoritmos no es un tema fácil de abordar a nivel licenciatura, ya que requiere de un bagaje matemático sólido. Nuestra propuesta aporta una manera novedosa en esta área, de analizar algoritmos que enfrenta por capas el difícil proceso de establecer su correctez, reutilizando de manera importante lo que se hace en capas más abstractas – y por lo tanto con predicados más sencillos – para únicamente agregar predicados en las capas más concretas. Nuestra propuesta de abstracción permite acceder de manera más sencilla a las demostraciones de correctez y complejidad de algoritmos, que consideramos importante en la formación de los profesionales en ciencias de la computación.

Nos hemos dado a la tarea de organizar material enfocado a que el estudiante adquiera conocimientos respecto a la metodología empleada en la demostración de correctez de algoritmos o en el cálculo preciso de su complejidad. Estos dos aspectos tienen que ver con

la calidad del algoritmo el primero, y con la cantidad de recursos necesarios el segundo. La calidad se refiere a la precisión del resultado, aspecto que se puede delimitar usando herramientas matemáticas que describan y garanticen la ejecución del algoritmo. En este contexto, no siempre es fácil decidir, entre otros aspectos, con cuáles algoritmos se ejemplifican los procesos de evaluación y demostración de correctez de los mismos; qué tanto se profundiza en ambos aspectos; cómo presentar los temas de tal manera que el estudiante vaya adquiriendo destreza en esta actividad. Para facilitar esta tarea recurrimos a un enfoque alternativo en la enseñanza del análisis de algoritmos, el de los *algoritmos genéricos*.

Trabajamos con distintos tipos de algoritmos, buscando conformar familias. En el resto de este trabajo a las familias seleccionadas, con una descripción genérica de la característica que unifica a los algoritmos de cada una de las familias, y cuáles algoritmos consideramos que pertenecen dentro de cada familia. Para todas estas familias se establece la estrategia general a seguir, las propiedades de esta estrategia en términos de predicados, y las aseveraciones que cada método invocado por la estrategia – métodos abstractos en este nivel – debe cumplir, en un formato de *requiere/cumple*. Asimismo, se especifica una función de complejidad en términos de la estrategia genérica con sub-funciones para calcular la participación de los métodos abstractos en esta fórmula. Una vez definidas las especializaciones, demostramos que las distintas especializaciones de los métodos abstractos cumplen con los requisitos dados por la estrategia genérica. También resolvemos las sub-funciones para obtener la clase de complejidad precisa para cada especialización.

Capítulo 2

Ordenamientos

En este capítulo revisaremos una familia de algoritmos de ordenamiento, a la que llamamos *ordenamientos por región*. En esta familia los ordenamientos se llevan acabo por intercambio simple, donde la lista de elementos a ordenar se encuentra dividida en dos regiones adyacentes, una ya ordenada y la otra por ordenar. Los algoritmos de ordenamiento que responden a esta clasificación, y que revisaremos, son: ordenamiento por inserción; ordenamiento por selección; ordenamiento de burbuja; y ordenamiento de montículo.

Plantearemos un algoritmo genérico en el sentido de la definición que dimos al inicio del Capítulo 1, y estableceremos las propiedades del algoritmo genérico, sujetas a que cada uno de los métodos abstractos cumpla con las propiedades requeridas de ellos. A continuación estableceremos que cada una de las especializaciones cumple, en efecto, con las propiedades exigidas de ellos.

2.1. Definición formal de ordenamiento

Empecemos por formalizar qué es un ordenamiento. Claramente nos referimos al concepto matemático de una *permutación*:

Definición 2.1 Sea $R = \langle r_1, \ldots, r_n \rangle$ una lista de elementos, donde r_i ocupa la posición i. Entonces, R' es una *permutación* de R si se cumple que el número de veces que aparece cada elemento en R es el número de veces que aparece en R', y viceversa. La especificación formal de este predicado se da con la Fórmula (2.1).

$$perm(R, R') \;\equiv\; \#(r_j = r_k) = \#(r_j = r'_k), \quad 1 \leq k \leq n, \quad 1 \leq j \leq n \tag{2.1}$$

Se conocen muchos algoritmos para ordenamiento. Formalicemos el problema – usamos la notación $\langle \ldots\ldots \rangle$ para denotar que el orden de los elementos listados es relevante:

Problema: Dada una sucesión de registros $R = \langle r_1, r_2, \ldots, r_n \rangle$, cada uno de ellos conteniendo una llave $(r_i.key)$ que pertenece a un conjunto finito ordenado K de llaves, obtener una

permutación $R' = \langle r'_1, \ldots, r'_n \rangle$ de R que cumpla:

$$perm(R, R') \wedge (r'_j.key \leq r'_{j+1}.key) \qquad 1 \leq j < n. \tag{2.2}$$

Nótese que en esta especificación del problema suponemos que estamos ordenando a un conjunto de *registros*, cada uno de ellos con un campo *key* de acuerdo al cuál se da el ordenamiento.

Datos: La sucesión de registros a ordenar

$$R = \langle r_1, \ldots, r_n \rangle$$

Salida: Una permutación de R, $R' = \langle r'_1, \ldots, r'_n \rangle$, que cumpla

$$perm(R, R') \wedge \left(r'_j.key \leq r'_{j+1}.key\right) \qquad \forall j \text{ con } 1 \leq j < n.$$

De este planteamiento del problema de ordenamiento es claro que tenemos que definir de manera precisa qué es una *permutación* de una lista de elementos, identificado cada uno de ellos por la posición que ocupan en la lista, empezando por la posición 1. Usaremos el nombre del predicado $perm(R, R')$ para denotar que R' es una permutación de R. Los algoritmos de ordenamiento pueden o no ser *estables*[1].

Una manera de medir que tan lejos de estar ordenada se encuentra una permutación es mediante una *tabla de inversiones* para la permutación. Sea $\langle a_1, a_2, \ldots a_n \rangle$ la permutación que estamos evaluando, y sea $\langle a'_1, a'_2, \ldots, a'_n \rangle$ la lista con los elementos ordenados. Si $\langle b_1, b_2, \ldots, b_n \rangle$ es la tabla de inversiones correspondiente a $\langle a_1, a_2, \ldots, a_n \rangle$, entonces b_i corresponde al número de elementos en $\langle a_1, \ldots, a_i \rangle$ tales que $a_j < a'_i$, $1 \leq j < i$. De alguna manera, las tablas de inversiones denotan que tan fuera de su lugar se encuentra el elemento a_i. Dada la definición, tenemos las siguientes propiedades para la tabla de inversiones:

$$b_1 = 0; \quad 0 \leq b_2 < 2; \quad 0 \leq b_i < i, \quad 1 \leq i \leq n$$

De esto, el número de inversiones presentes en una permutación está acotado por $(n-1) + (n-2) + \ldots 1 = n(n-1)/2$. El número de inversiones presentes en una lista ordenada es cero.

De lo anterior, se puede medir el avance en un algoritmo de ordenamiento ya sea verificando que (2.2) se cumpla para cada vez más elementos, o bien que el número de inversiones vaya decreciendo.

2.2. Familias de algoritmos de ordenamiento

Trabajaremos con lo que se conoce como *algoritmos internos* de ordenamiento[2]. Hay diversas clasificaciones de algoritmos de ordenamiento (e. g. [Woo93, Knu98b]). La primera de ellas, dada por Knuth en [Knu68], clasifica los algoritmos de ordenamiento en dos grandes

[1] Ver en el Apéndice B una definición de lo que es un ordenamiento estable.

grupos: los que trabajan haciendo comparaciones entre las llaves y los que no. Entre estos últimos se encuentran, por ejemplo, el ordenamiento por casilleros o cubetas (*Bucket Sort*) y el ordenamiento por índices (*Radix Sort*). La mayoría de los ordenamientos más usados son en base a comparaciones, como el ordenamiento por inserción, el *Quicksort* y el ordenamiento por intercalación (*Merge Sort*), ya que los ordenamientos como el *Bucket Sort* o el *Radix Sort* requieren una distribución particular de las llaves de los registros a ordenar, como por ejemplo que tengan valores transformables a un rango relativamente pequeño de enteros. Esta clasificación no impone, sin embargo, una estrategia genérica para todos los algoritmos de un mismo tipo, por lo que no nos será útil.

Otra clasificación que ha estado en uso los últimos años es la proporcionada por Merritt en [Mer85] y asumida por Wood en [Woo93], donde los algoritmos de ordenamiento se clasifican de acuerdo a dos procedimientos, el primero que *divide* a los datos y el segundo que los *pega* de regreso. Bajo esta taxonomía tenemos cuatro posibles grupos: división y pegado fáciles; división fácil y pegado difícil; división difícil y pegado fácil; y división y pegado difíciles. Esta clasificación se acerca más a nuestra manera de clasificar familias de algoritmos.

Nos ocuparemos en este capítulo únicamente de algunos algoritmos de ordenamiento por comparación. Para ilustrar las técnicas de análisis de algoritmos que proponemos, ejemplificaremos con una familia de algoritmos de ordenamiento, la que efectúa los ordenamientos a base de comparar e insertar. Los cuatro algoritmos que presentamos, el de burbuja, de inserción, selección y por montículo (*HeapSort*), presentan todos las siguientes características:

i. Todos trabajan *in situm*, esto es, no utilizan espacio adicional al que ocupa la lista de llaves (con sus referencias respectivas), más que en términos de un número constante de celdas para contadores, registros temporales, etc.

ii. Todos eligen o seleccionan a un elemento, y lo mueven al lugar que debe ocupar en una región previamente ordenada en el mismo arreglo.

iii. El movimiento se hace siempre intercambiando a registros en el arreglo.

iv. Conforme va avanzando la ejecución del algoritmo, existe una porción de lugares consecutivos que contienen elementos que ya se encuentran en orden. Esto implica que los algoritmos que se presentan parten al arreglo en dos regiones, la ya ordenada y aquella que está por ordenar, cada una de ellas ocupando posiciones consecutivas respectivamente.

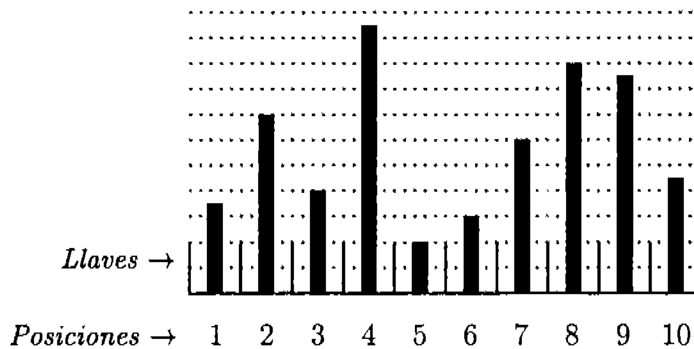
Hablaremos genéricamente de que los datos se encuentran en un arreglo, cuando en realidad podrían encontrarse en cualquier estructura de datos que tuviera acceso directo, como los arreglos. El acceso directo es un requisito para las medidas de complejidad que calculamos.

Para cada uno de estos cuatro algoritmos presentaremos primero el código del algoritmo original, y a continuación un ejemplo ilustrado de la ejecución. En los cuatro casos trabajaremos con un arreglo de 10 posiciones al ilustrar la ejecución. En la Figura 2.1 representamos al arreglo con celdas numeradas del uno en adelante, y la magnitud de cada llave aparece

[2] Ver Apéndice B para estas definiciones.

como la altura de la barra que se encuentra en la celda. A mayor altura de la barra, mayor magnitud (valor) de la llave.

Figura 2.1: Situación inicial del arreglo



Los cuatro ejemplos que presentaremos suponen la distribución de los valores en la Figura 2.1 al inicio de cada uno de los algoritmos. En ellos suponemos que $|R| = n$ ($|R|$ denota al número de registros en el arreglo).

2.3. Un algoritmo genérico para ordenamiento

Dimos en la introducción la definición de un algoritmo genérico como aquel que carece de especificidad en algunos de los pasos del algoritmo. Decimos entonces, en este contexto, que un algoritmo es *genérico* si establece la estrategia general de ejecución para una familia de algoritmos. Este enfoque se basa en la orientación a objetos que permite este tipo de algoritmos abstractos.

En un algoritmo genérico se establecen las características que deben cumplir los métodos del algoritmo y qué tipos de valores debe entregar, y se deja a cada algoritmo particular la especialización de estos métodos. Esto permite, en el análisis tanto de la complejidad como de la correctez, que los resultados del marco conceptual se hereden hacia las especializaciones (o extensiones) del algoritmo genérico.

Usaremos este enfoque para abstraer un algoritmo genérico de ordenamiento que pueda especializarse a los algoritmos que comparten las características mencionadas en la sección anterior, y que son los ordenamientos por inserción, selección, burbuja y de montículo (*heap*). Para la presentación de esta familia de algoritmos denominamos *región* a un conjunto de posiciones contiguas; la *región baja* del arreglo es aquella región cuyo primer elemento es el primero del arreglo, mientras que la región *alta* es aquella cuyo último elemento es el último elemento del arreglo.

De manera similar a la taxonomía de Merritt en [Mer85], la estrategia general de nuestro algoritmo genérico será la de mantener al arreglo dividido en todo momento en dos regiones, la baja en la que se encuentran los elementos sin un orden garantizado, y la alta en la que

los elementos se encuentran ya ordenados. El algoritmo procede a, en cada iteración, elegir un elemento de la región baja (select) y acomodarlo en la región alta (join), manteniendo el orden en esta región.

Proponemos la siguiente estructura genérica para lo que llamaremos *algoritmo genérico de ordenamiento por comparación* (AbstractSort). En este algoritmo genérico la estrategia consiste de dos partes: primero ubicar al registro que se desea mover (*select*); en la segunda parte, encontrarle su lugar definitivo entre los registros ya ordenados, recorriendo o intercambiando los elementos necesarios para ello y colocar al elemento en el lugar correspondiente (*join*). La especialización particular de cada uno de estos métodos dependerá del algoritmo específico que estemos implementando. Mostramos el algoritmo genérico en el Listado 2.1.

Listado 2.1: Algoritmo genérico para ordenamientos por comparación

```
 1 public abstract class AbstractSort {
 2     /* Lista de elementos a ser ordenados, almacenados en una   */
 3      * lista con acceso directo.                                */
 4   protected ArrayList objs;
 5     /* Número de elementos a ser ordenados.                     */
 6   protected final int N;
 7     /* Reglas para comparar los elementos de la lista.          */
 8   protected BasicComparator comp;
 9     /* Recibe un arreglo de registros a ordenar, y las reglas   *
10      * de comparación aplicables. Deja a los elementos del      *
11      * arreglo en la estructura de acceso directo seleccio-     *
12      * nada. Inicia el valor para N.                            */
13   protected AbstractSort(Object[] objs, BasicComparator comp) {
14     /* Hace lo necesario para dejar el comparador asociado a    *
15      * la lista de elementos.                                   */
16   }
17     /* Selecciona un objeto de la región baja.                  */
18   protected abstract int select(int first, int last);
19
20     /* Acomoda al elemento seleccionado en la región alta de    *
21      * la lista.                                                */
22   protected abstract void join(int sel, int low, int high);
23
24     /* La estrategia genérica para el ordenamiento, que con-    *
25      * siste en ir moviendo la frontera entre la región alta y  *
26      * la baja en una unidad en cada iteración.                 */
27   public final ArrayList genericSort() {
28       int k;
29       for (int i = N - 1; i > 0; i--) {
30           k = select(1, i);
31           join(k, i, N);
32       }
33       return objs;
34   }
35 }
```

En este código no mostramos el método *swap* que funciona de manera tradicional, intercambiando dos elementos.

En aras de que los cuatro algoritmos presentados se enmarquen de mejor manera en el algoritmo genérico, el ordenamiento por inserción trabajará de "arriba hacia abajo", esto es, irá suponiendo que la región ordenada del arreglo es la región alta e irá tomando los registros a acomodar desde la penúltima posición hacia abajo, contrario a la presentación tradicional donde se supone preordenada la región baja del arreglo. Es claro que esto no altera la ejecución del algoritmo y en cambio nos permite hablar uniformemente de la región ordenada del arreglo que ocupa las posiciones $i+1, \ldots, n$, donde i corresponde a la $(n-i)$-ésima pasada, mientras que la región que no ha sido ordenada se encuentra en las posiciones $1, \ldots, i$ del arreglo. Con este cambio podemos abstraer de los cuatro algoritmos las invariantes principales. Usaremos la siguiente notación para distinguir entre el contenido del arreglo antes y después de cada pasada:

R:	la permutación original de los datos.
$R^{(\ell)}$:	el arreglo resultado de la ℓ-ésima iteración, $\ell = 1, \ldots, n-1$.
$R^{(0)}$:	el arreglo después de la inicialización del algoritmo.
$R^{(n)}$:	el arreglo al terminar el algoritmo ($R^{(n)} = R^{(n-1)}$).
$R^{(\ell)}[j \ldots m]$:	el subarreglo que ocupa las posiciones j a m inclusives, de $R^{(\ell)}$.
$r^{(\ell)}[j]$:	el registro que se encuentra en la posición j en $R^{(\ell)}$.
n:	el número de registros en R.

Utilizaremos $r^{(i)}[j]$ para denotar, indistintamente, al registro completo o a la llave del registro que participa en las comparaciones. En general, cuando aparezca como superíndice un entero i, denotará que estamos trabajando sobre la organización del arreglo R en la i-ésima pasada. Si bien es costumbre identificar las iteraciones con el índice i, en nuestro caso, dado que la i varía de $n-1$ a 1, cuando hablamos de la i-ésima pasada se genera confusión. Por ejemplo, para la primera pasada i vale $n-1$ y no 1. Para evitar confusión contaremos las iteraciones con la literal ℓ que variará en sentido creciente de 1 a $n-1$, contando las veces que se ejecuta el bloque de la iteración. De esto, la ℓ-ésima pasada corresponde a aquélla en la que $i = n - \ell$.

Con esto podemos hacer ya las especificaciones del algoritmo genérico. Para toda pasada ℓ tal que $1 \leq \ell \leq n-1$, tenemos las siguientes invariantes:

A: Al iniciarse la ℓ-ésima pasada, con $\ell = n - i$,

i. El arreglo contiene una permutación de los datos originales:

$$perm\big(R, R^{(\ell-1)}\big) \tag{2.3.A.i}$$

ii. Los registros en las posiciones $i+1, \ldots, n$ se encuentran ordenados:

$$r^{(\ell-1)}[j] \leq r^{(\ell-1)}[j+1], \qquad \text{con } i+1 \leq j < n \tag{2.3.A.ii}$$

B: Al terminar la ℓ-ésima pasada:

i. Los registros que se encuentran en la región ya ordenada, permanecen ordenados, y el número de registros en ellas creció en una posición.

$$r^{(\ell)}[j] \leq r^{(\ell)}[j+1], \qquad i+1 \leq j < n \tag{2.3.B.i}$$

ii. Los elementos en $R^{(\ell-1)}[i+1\ldots n]$ permanecen en la región ordenada del arreglo en $R^{(\ell)}$:

$$r^{(\ell-1)}[j] = r^{(\ell)}[m], \quad \text{con } i+1 \leq j \leq n, \quad i \leq m \leq n \tag{2.3.B.ii}$$

iii. El arreglo resultante de la pasada $\ell - 1$ representa una permutación del arreglo al inicio de la ℓ-ésima pasada.

$$perm\big(R^{(\ell-1)}, R^{(\ell)}\big) \tag{2.3.B.iii}$$

En el código distinguiremos entre el registro y la llave identificando a esta última con el atributo key del registro (⟨registro⟩.key).

2.4. Correctez del algoritmo genérico de ordenamiento por comparación

La correctez de un algoritmo de ordenamiento está dada por la especificación del resultado que esperamos del mismo, a saber que el contenido del arreglo sobre el que trabajamos contenga una permutación del contenido original, de tal manera que se cumpla

$$perm(R, R^{(n-1)}) \wedge \big(r^{(n-1)}[j] \leq r^{(n-1)}[j+1]\big), \qquad \text{con } 1 \leq j < n \tag{2.4}$$

donde n representa al número original de registros. Para demostrar que lo que obtenemos es la permutación correcta, lo primero que haremos es probar que las invariantes (2.3.A.i) a (2.3.B.iii) son necesarias y suficientes para esto. Al demostrar que nuestra estrategia general funciona, tenemos que suponer las siguientes condiciones para cada uno de los métodos, abstractos o simplemente redefinibles, que definen al ordenamiento genérico:

AbstractSort(Object[] objs, BasicComparator comp): Corresponde al constructor de la clase abstracta. Se ejecuta una única vez al construirse a un ejemplar de una subclase, antes de empezar a iterar. Si se cumplen las precondiciones del método (ninguna), la implementación del constructor debe cumplir con las postcondiciones ASps-1 y ASps-2:

Precondiciones: Ninguna.
Postcondiciones: Al terminar la ejecución del constructor, se debe cumplir:

$$perm(R, R^{(0)}) \tag{ASps-1}$$

$$n = |R|, \text{ número de registros en } R^{(0)} \tag{ASps-2}$$

select(int first, int last): Sea $R^{(\ell-1)}$ la permutación antes de iniciarse la ejecución de este método en la ℓ-ésima pasada, y $R^{(\ell-1)'}$ la permutación que resulta de la ejecución de este método, con $1 \leq \ell < n$. Si se cumplen las precondiciones Spr-1 y Spr-2 antes de empezar a ejecutar select en la ℓ-ésima iteración, a saber:

Precondiciones:

i. Al empezar la ejecución, tenemos una permutación de los datos originales:

$$perm(R, R^{(\ell-1)}) \tag{Spr-1}$$

ii. Los elementos que ocupan las posiciones $i+1, \ldots, n$ se encuentran ordenados.

$$r^{(\ell-1)}[j] \leq r^{(\ell-1)}[j+1], \qquad \text{con } i+1 \leq j < n; \tag{Spr-2}$$

entonces, la implementación del método debe cumplir con las postcondiciones (Sps-1) a (Sps-4):

Postcondiciones:

i. El elemento elegido es uno de los que se encontraban en la región desordenada del arreglo.

$$1 \leq k \leq i. \tag{Sps-1}$$

ii. Se cumple que el subarreglo $R^{(\ell-1)'}[1 \ldots i]$ corresponde a una permutación de $R^{(\ell-1)}[1 \ldots i]$:

$$perm\big(R^{(\ell-1)}[1..i], R^{(i-1)'}[1..i]\big). \tag{Sps-2}$$

iii. Y que el subarreglo de los elementos ya ordenados permanece intacto:

$$r^{(\ell-1)}[j] = r^{(\ell-1)'}[j], \qquad \text{con } i+1 \leq j \leq n. \tag{Sps-3}$$

NOTA: Al cumplirse las postcondiciones (Sps-2) y (Sps-3) es inmediato que $R^{(\ell-1)'}$ es una permutación de $R^{(\ell-1)}$.

$$perm(R^{(\ell-1)}, R^{(\ell-1)'}). \tag{Sps-4}$$

join(int sel, int low, int high): Sea $R^{(\ell)}$ el resultado de ejecutar join sobre $R^{(\ell-1)'}$ en la ℓ-ésima pasada. Si antes de ejecutarse join se cumplen (Sps-1) a (Sps-4), entonces, la implementación de join debe garantizar que al terminar la ejecución de join en la ℓ-ésima pasada se cumplan los predicados (Jps-1) a (Jps-3):

Precondiciones: (Sps-1) a (Sps-4).

Postcondiciones:

i. El arreglo resultante $R^{(\ell)}$ es una permutación del arreglo al empezar la ejecución de este método:

$$perm(R^{(\ell-1)'}, R^{(\ell)}) \tag{Jps-1}$$

ii. La región de registros ordenados crece en una posición.

$$r^{(\ell)}[j] \leq r^{(\ell)}[j+1], \qquad \text{con } i \leq j < n \tag{Jps-2}$$

iii. La nueva región ordenada del arreglo contiene propiamente a los mismos elementos que la vieja región, en el mismo orden relativo.

$$\begin{aligned} &\big(r^{(\ell)}[j] = r^{(\ell-1)'}[m]\big) \wedge \big(r^{(\ell)}[j'] = r^{(\ell-1)'}[m']\big) \\ &\wedge \Big(\big(r^{(\ell-1)'}[m] \leq r^{(\ell-1)'}[m']\big) \Longrightarrow \big(r^{(\ell)}[j] \leq r^{(\ell)}[j']\big)\Big) \\ &\qquad \text{con } i \leq j < j' \leq n, \quad i+1 \leq m < m' \leq n \end{aligned} \tag{Jps-3}$$

Es claro que si se cumplen estas pre y post condiciones para $1 \leq \ell < n$, tendremos que las postcondiciones para join hacen que el predicado (2.4) se cumpla – el valor de i en la $\ell = n - 1$ pasada es 1.

El diseño de la especialización de cada uno de estos métodos deberá cumplir con estas especificaciones, lo que demostraremos en su momento.

Procedemos a demostrar primero que el arreglo R, al finalizar el algoritmo, contiene una permutación de los datos originales, lo que haremos en el Lema 2.1.

Lema 2.1 *Sea $R = r[1], \ldots, r[n]$ el arreglo original de registros, con $r[j]$ ocupando la posición j del arreglo. Entonces, al terminar la ejecución de* genericSort, *$perm(R, R^{(n-1)})$, con $R^{(n-1)} = r^{(n-1)}[1], \ldots, r^{(n-1)}[n]$ (donde $r^{(n-1)}[j]$ ocupa la posición j).*

Demostración:

Sea $R = r[1], \ldots, r[n]$ un arreglo con n registros. Es claro que el predicado de permutación tiene la propiedad de transitividad, esto es

$$perm(R, P) \wedge perm(P, Q) \Longrightarrow perm(R, Q). \tag{PTP}$$

Por lo que si demostramos

$$perm(R, R^{(0)}) \wedge perm(R^{(0)}, R^{(1)}) \wedge \cdots \wedge perm(R^{(n-2)}, R^{(n-1)}), \tag{gS-1}$$

por transitividad habremos demostrado que

$$perm(R, R^{(n-1)}).$$

Empecemos por demostrar que $perm(R, R^{(0)})$. Al iniciarse la ejecución del algoritmo genérico, se ejecuta una única vez el constructor AbstractSort. La implementación del constructor debe garantizar la postcondición (ASps-1):

$$perm(R, R^{(0)}),$$

por lo que, si exigimos a AbstractSort que cumpla con sus postcondiciones, este primer término de la disyunción es válido.

A continuación el algoritmo itera para $i = n - 1, \ldots, 1$ ($\ell = 1 \ldots n - 1$). Deberemos demostrar que a través de estas iteraciones se mantiene la invariante:

$$perm(R^{(\ell-1)}, R^{(\ell)}) \qquad \forall \ell = 1 \ldots n - 1 \qquad (i = n - 1 \ldots 1)$$

ya que entonces, por la transitividad de la que acabamos de hablar, junto con la inicialización, tendremos que podemos concluir

$$perm(R, R^{(n-1)})$$

Haremos la demostración por inducción en el número de pasadas del algoritmo, suponiendo que las especializaciones de los métodos **cumplen** con las especificaciones dadas. Nos fijaremos por el momento únicamente en aquellas invariantes que hablan sobre permutaciones.

Base: Para $\ell = 1$ $(i = n - 1)$.

$$(i)\ perm(R, R^{(0)}) \qquad \text{(de ASps-1)} \qquad (2.5)$$
$$n = |R| \qquad \text{(de ASps-2)}$$

(Como n ya no va a cambiar en el resto de la ejecución, no la volvemos a mencionar y trabajaremos únicamente con la primera de estas dos precondiciones.)

$$(ii)\ perm(R^{(0)}, R^{(0)'}) \qquad \text{(de Sps-4 con } \ell = 1 \quad (i = n - 1)) \qquad (2.6)$$
$$(iii)\ perm(R^{(0)'}, R^{(1)}) \qquad \text{de Jps-1 con } \ell = 1 \quad (i = n - 1) \qquad (2.7)$$

De (2.6) y (2.7) y de la propiedad de transitividad para permutaciones, tenemos que

$$(iv)\ perm(R^{(0)}, R^{(0)'}) \wedge perm(R^{(0)'}, R^{(1)}) \Longrightarrow perm(R^{(0)}, R^{(1)}) \qquad (2.8)$$

con lo que queda demostrado el caso base, a saber

$$perm(R^{(0)}, R^{(1)})$$

Inducción: Supongamos ahora que en la iteración $\ell - 1$ $(i = n - \ell + 1)$, $1 < \ell < n - 1$ se cumplió

$$perm(R^{(\ell-2)}, R^{(\ell-1)}) \qquad \text{(hipótesis de inducción)}$$

y demostraremos que al terminar la ℓ−ésima iteración $(i = n - \ell)$ se cumple

$$perm(R^{(\ell-1)}, R^{(\ell)}) \qquad \text{(por demostrar)}$$

En la ℓ-ésima iteración $(i = n - \ell)$ sabemos que:

$$perm(R, R^{(\ell-1)}) \qquad \text{(hipótesis de inducción y PTP)}$$
$$perm(R^{(\ell-1)}, R^{(\ell-1)'}) \qquad \text{(por Sps-4)}$$
$$perm(R^{(\ell-1)'}, R^{(\ell)}) \qquad \text{(por Jps-1)}$$
$$\therefore \quad perm(R, R^{(\ell)}) \qquad \text{(Por transitividad).}$$

De donde, para $\ell = n - 1$, al terminar las iteraciones,

$$perm(R, R^{(n-1)}).$$

❏

Demostraremos ahora en el Lema 2.2 que la segunda condición para que el algoritmo de ordenamiento implementado sea correcto también se cumple.

Lema 2.2 *Sea $R = r[1], \ldots, r[n]$ el arreglo original de registros, y $R^{(n-1)} = r^{(n-1)}[1], \ldots, r^{(n-1)}[n]$ la permutación que se obtiene al terminar de ejecutar* genericSort *sobre R, y que cumple con $perm(R, R^{(n)})$. Entonces se cumple que*

$$r^{(n-1)}[j] \leq r^{(n-1)}[j+1] \qquad con\ 1 \leq j < n \qquad (2.9)$$

Demostración:

Demostraremos por inducción en el número de iteraciones que el algoritmo ejecuta, que la región ordenada del arreglo al final de cada iteración cumple con:

$$r^{(\ell)}[j] \leq r^{(\ell)}[j+1] \qquad \text{con } i \leq j < n \qquad \text{(corresponde a Jps-1).}$$

Base: Para $\ell = 1$ $(i = n-1)$, suponemos que se cumplen:

i. Al terminar el constructor:

$$perm(R, R^{(0)}). \tag{2.10}$$

ii. Para $\ell = 1$ $(i = n-1)$, antes de ejecutar select se cumple:

$$r^{(0)}[j] \leq r^{(0)}[j+1], \qquad \text{con } n \leq j < n$$
$$\big(\text{Notar: } (i+1) = (n-\ell)+1 = (n-1+1) = n\big). \tag{2.11}$$

iii. Como se satisfacen (2.10) y por vacuidad (2.11), la implementación garantizará que se cumplan:

$$1 \leq k < (i+1) = (n-\ell+1) = n \qquad \text{(por Sps-1)}, \tag{2.12}$$
$$perm\big(R^{(0)}[1\ldots(i=n-1)], R^{(0)\prime}[1\ldots i=n-1]\big) \qquad \text{(por Sps-2)}, \tag{2.13}$$
$$r^{(0)}[j] = r^{(0)\prime}[j] \qquad \text{con } n \leq j \leq n \qquad \text{(por Sps-3)}. \tag{2.14}$$

iv. Como se cumplen (Sps-1) a (Sps-4) – este último por el Lema 2.1 – y suponemos que la implementación que demos para join garantiza que se cumplen (Jps-1) a (Jps-3) para $\ell = 1$ $(i = n-1)$:

$$r^{(1)}[n-1] \leq r^{(1)}[n], \qquad \text{(por Jps-2)}; \tag{2.15}$$

como hay un único elemento en la región ordenada, por vacuidad se cumple el que los elementos mantengan su orden relativo (Jps-3). Y por último,

$$r^{(1)}[j] \leq r^{(1)}[j+1], \qquad \text{con } n-1 \leq j \leq n-1 \qquad \text{(por Jps-1)} \tag{2.16}$$

$\therefore$ Al final de la primera iteración tenemos:

$$perm(R, R^{(1)}) \qquad \text{(por Lema 2.1)},$$
$$r^{(1)}[n-1] \leq r^{(1)}[n] \qquad \text{(por Jps-2 y Jps-3)},$$

y se cumple la postcondición (2.4) para $\ell = 1$ $(i = n-1)$.

Inducción: Supongamos ahora, como hipótesis de inducción, que al terminar la $(\ell-1)$-ésima iteración del algoritmo $(i = n-\ell+1)$, $1 < \ell < n-1$, se cumple:

$$perm(R, R^{(\ell-1)})$$
$$r^{(\ell-1)}[j] \leq r^{(\ell-1)}[j+1], \qquad \text{con } (i+1) \leq j < n \qquad (i = n-\ell+1)$$

y demostremos que esta invariante se mantiene al terminar la iteración ℓ, esto es, se cumple

$$perm(R, R^{(\ell)})$$
$$\text{y} \quad r^{(\ell)}[j] \leq r^{(\ell)}[j+1] \qquad \text{con } i \leq j < n.$$

Sigamos a los predicados que se van cumpliendo en la pasada ℓ:

i. Se cumplen las precondiciones para select por la hipótesis de inducción:

$$perm(R, R^{(\ell-1)}) \qquad \text{(HI)},$$
$$r^{(\ell-1)}[j] \leq r^{(\ell-1)}[j+1] \quad \text{con } i+1 \leq j < n \qquad \text{(HI)}.$$

ii. Como se cumplen estas precondiciones, se cumplen las postcondiciones del método select:

$$1 \leq k^{(\ell-1)} \leq i \qquad \text{(por Sps-1)},$$
$$perm\big(R^{(\ell-1)}[1..i+1], R^{(\ell-1)'}[1..i+1]\big) \qquad \text{(por Sps-2)},$$
$$r^{(\ell-1)}[j] = r^{(\ell-1)'}[j+1] \quad \text{con } (i+1) \leq j < n \qquad \text{(por Sps-3)} \quad \text{y}$$
$$perm(R^{(\ell-1)}, R^{(\ell-1)'}) \qquad \text{(por Sps-4)}.$$

iii. Como supusimos que se cumplen las postcondiciones de select en la pasada ℓ ($i = n - \ell$), y éstas son las precondiciones para join, podemos suponer que la implementación de este método será tal que las postcondiciones del mismo se cumplan, a saber:

$$perm(R^{(\ell-1)'}, R^{(\ell)}) \qquad \text{(por Jps-1)};$$
$$r^{(\ell)}[j] \leq r^{(\ell)}[j+1], \quad \text{con } i \leq j < n \qquad \text{(por Jps-2)};$$
$$\big(r^{(\ell)}[j] = r^{(\ell-1)'}[m]\big) \wedge \big(r^{(\ell)}[j'] = r^{(\ell-1)'}[m']\big) \qquad \text{(por Jps-3)},$$
$$\wedge \Big(\big(r^{(\ell-1)'}[m] \leq r^{(\ell-1)'}[m']\big) \Longrightarrow \big(r^{(\ell)}[j] \leq r^{(\ell)}[j']\big)\Big)$$
$$\text{con } i \leq j < j' \leq n, \quad \text{con } i+1 \leq m < m' \leq n$$

$\therefore$ tenemos para el proceso de inducción:

i. Por la base de la inducción, tenemos:

$$perm(R, R^{(1)})$$
$$r^{(1)}[n-1] \leq r^{(1)}[n]$$

ii. Tenemos que $perm(R, R^{(n-2)})$ por el Lema 2.1.

iii. Por la demostración del paso inductivo, tenemos que en la $(\ell - 1)$-ésima iteración los elementos que ya estaban ordenados en la parte alta del arreglo, permanecen en la parte alta del arreglo al terminar la ejecución de select:

$$r^{(\ell-1)}[j] = r^{(\ell-1)'}[j], \quad \text{con } i+1 \leq j \leq n, \qquad \text{(por Sps-3)};$$

y que al crecer la sección ordenada, se mantienen en ella todos los elementos que ahí estaban:

$$\big(r^{(\ell)}[j] = r^{(\ell-1)'}[m]\big) \wedge \big(r^{(\ell)}[j'] = r^{(\ell-1)'}[m']\big) \qquad \text{(por Jps-3)},$$
$$\wedge \Big(\big(r^{(\ell-1)'}[m] \leq r^{(\ell-1)'}[m']\big) \Longrightarrow \big(r^{(\ell)}[j] \leq r^{(\ell)}[j']\big)\Big)$$
$$\text{con } i \leq j < j' \leq n, \quad \text{con } i+1 \leq m < m' \leq n.$$

Por lo que una vez que entra un registro a la región ordenada, no vuelve a salir de ella, considerando la región aumentada en una posición al terminar la pasada ℓ del algoritmo:

$$\begin{aligned}
&\big(r^{(\ell)}[j] = r^{(\ell-1)}[m]\big) \wedge \big(r^{(\ell)}[j'] = r^{(\ell-1)}[m']\big) \qquad \text{(por Jps-3)}\\
&\wedge \Big(\big(r^{(\ell-1)'}[m] \le r^{(\ell-1)'}[m']\big) \Longrightarrow \big(r^{(\ell)}[j] \le r^{(\ell)}[j']\big)\Big)\\
&\qquad \text{con } i \le j < j' \le n, \quad \text{con } i+1 \le m < m' \le n.
\end{aligned}$$

De esto, se va agregando en cada pasada a un elemento al conjunto de ordenados, y tenemos al final de la pasada $\ell = n - 1$,

$$\begin{aligned}
&perm\big(R, R^{(n-1)}\big) \wedge \big(r^{(n-1)}[j] \le r^{(n-1)}[j']\big),\\
&\qquad \text{con } 1 \le j < j' \le n).
\end{aligned}$$

En particular, si $j' = j + 1$, tenemos,

$$r^{(n-1)}[j] \le r^{(n-1)}[j+1], \quad \text{con } (n-\ell) = n-(n-1) = 1 \le j < n.$$

$\therefore$ si cada uno de los métodos que participan cumplen con sus postcondiciones dadas sus precondiciones, se cumple que al terminar el algoritmo

$$perm\big(R, R^{(n-1)}\big)\wedge\big(r^{(n-1)}[j] \le r^{(n-1)}[j+1]\big), \qquad \text{con } 1 \le j < n.$$

❏

2.4.1. Complejidad

En ordenamientos es usual que las operaciones que se consideran para medir la complejidad de un algoritmo sean el número de accesos a los datos, de comparaciones y de asignaciones que se deben realizar. Por cada comparación o asignación que hagamos estamos haciendo dos accesos a los datos. Sin embargo, como estamos calculando complejidad asintótica, podemos contabilizar únicamente las dos últimas operaciones, ya que el resultado difiere únicamente en el coeficiente de los términos.

Realmente lo que podemos decir en este momento respecto a la complejidad del algoritmo genérico no es mucho, ya que desconocemos por completo la complejidad de cada uno de los métodos. Únicamente podemos decir que se deberá obtener la permutación $R^{(n-1)}$ para poder garantizar la que corresponde al arreglo ordenado.

Si definimos $\mathsf{f_{constr}(n)}$, $\mathsf{f_{sel}(i)}$ y $\mathsf{f_{join}(n-i)}$ como las complejidades del constructor y cada uno de los métodos invocados en la iteración, donde i va marcando la frontera entre las dos regiones y n es el número de elementos a ordenar, lo único que podemos decir, hasta en tanto no especifiquemos con más detalle estos métodos, es que

$$\textbf{Complejidad de genericSort} = \mathsf{f_{constr}(n)} + \sum_{\ell=1}^{n-1}\Big(\mathsf{f_{sel}(n-\ell)} + \mathsf{f_{join}(\ell)}\Big)$$

$$(\textit{donde} \quad i = n - \ell)$$

ya que el ciclo principal se ejecuta para $i = n-1$ hasta $i = 1$. Es claro que $\mathsf{f}_{\mathsf{constr}}(\mathsf{n}) \geq c \cdot n$, ya que el organizar a los datos para poder empezar, que se hace en el constructor, se va a llevar $O(n)$ tiempo – leerlos y colocarlos en la estructura de datos, lo que implica una cantidad constante de trabajo por cada elemento.

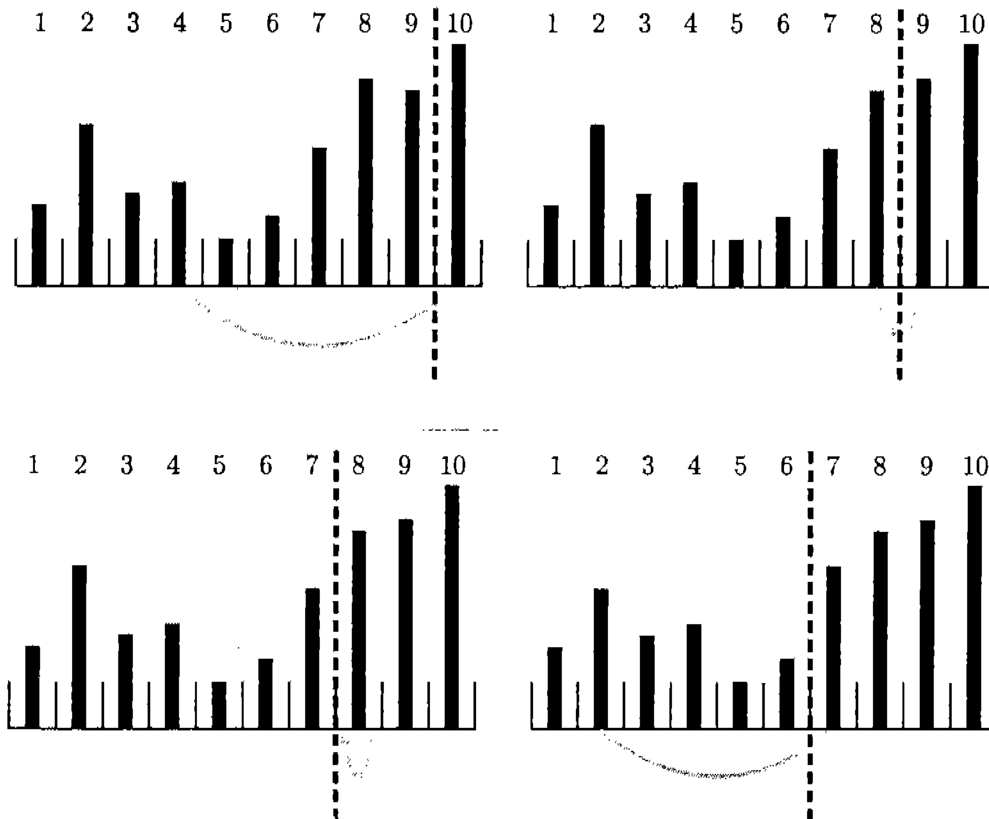
2.5. Especializaciones

Para cada uno de los cuatro ordenamientos elegidos deberemos especificar los tres métodos que quedaron nada más en términos de sus precondiciones y postcondiciones, repartiendo el trabajo de tal manera de garantizar que éstas se cumplen.

2.5.1. Ordenamiento por selección

Veamos primero, en la Figura 2.2, el funcionamiento de esta especialización.

Figura 2.2: Ordenamiento por selección: 4 pasadas



La manera de cumplir con las especificaciones dadas para los métodos select y join es la siguiente: le asignamos al constructor la responsabilidad de colocar al primer elemento (el máximo) en el último lugar del arreglo. De esa manera obligamos a este algoritmo a cumplir

con todas las especificaciones dadas, lo que demostraremos más adelante. La última iteración, para $i = 1$, simplemente se fija únicamente en el primer elemento, que ya es el mínimo (o máximo con respecto a sí mismo).

El trabajo queda de la siguiente manera, para cumplir con las precondiciones y postcondiciones dadas arriba:

1. En el constructor, además de colocar la lista de registros en el arreglo, construimos $R^{(0)}$ que contiene ya a un elemento "ordenado" en la posición n, que no abandonará la región alta en iteraciones sucesivas.
2. En select nos encargamos de extraer al registro con la la máxima llave en la región $1, \ldots, i$ y entregamos como resultado esta posición.
3. En join intercambiamos el registro en la posición i con el seleccionado.

La implementación correspondiente se encuentra en el Listado 2.2.

Listado 2.2: Especialización para el ordenamiento por selección

```
 1 public class SelectionSort  extends AbstractSort {
 2     public SelectionSort(Object[] objs, BasicComparator comp) {
 3       /* Encuentra el máximo en la región [1...n]        */
 4         super(objs, comp); // Constructor de la clase abstracta
 5         int posMax = select(1, N); // Selecciona al elemento mayor
 6         swap(posMax, N);             // Lo coloca en la última posición
 7     }
 8       /* Selecciona al máximo en la región baja.                    */
 9     protected int select(int first, int last) {
10         int posMax = first; // Propone al primero como máximo.
11         for (int i = first + 1; i <= last; i++) {
12       /* Si encuentra alguno mayor en la región baja, registra     *
13        * la nueva posición.                                        */
14             if (comp.gtr(objs.get(i), objs.get(posMax)))
15                 posMax = i;
16         }
17         return posMax;
18     }
19       /* Intercambia al elemento seleccionado con el que ocupa     *
20        * la primera posición de la región alta.                    */
21     protected void join(int sel, int low, int high) {
22         swap(sel, low);
23     }
24 }
```

2.5.2. Correctez de la especialización del ordenamiento por selección

Lo que debemos hacer ahora es demostrar que la manera como especializamos a los métodos SelectionSort (constructor), select y join cumplen con las propiedades asumidas para

cada uno de ellos cuando establecimos la correctez del algoritmo genérico. De esta manera heredamos la propiedad del algoritmo genérico de que produce un ordenamiento correcto. Estableceremos la correctez de cada uno de los métodos a través de varios lemas, que se encuentran a continuación.

El Lema 2.3 va a hacer menos extensa la demostración de la correctez del algoritmo, por lo que lo enunciamos antes que al teorema de correctez. Nos habla del orden entre los elementos que findMax va eligiendo.

Lema 2.3 *Sea $R[1\dots n]$ un arreglo. Sea i, $1 \le i < n$. Si se cumple que:*

i. La región alta está ordenada:

$$r[j] \le r[j+1], \qquad con\ (i+1) \le j < n. \tag{3.i}$$

ii. La región baja contiene elementos menores que cualquier elemento de la región alta; como se supone (3.i)*, esto se traduce a que son menores que el primer elemento de la región alta:*

$$r[j] \le r[i+1], \qquad con\ 1 \le j \le i. \tag{3.ii}$$

y se elige a un elemento $r[k]$, con $1 \le k \le i$, tal que:

iii. El elemento elegido es el máximo de la región baja:

$$r[j] \le r[k], \qquad con\ 1 \le j \le i, \tag{3.iii}$$

y se le intercambia con el elemento en la posición i, entonces se va a cumplir que:

iv. La región ordenada crece en una posición:

$$r[j] \le r[j+1], \qquad con\ i \le j < n. \tag{3.Ps(i)}$$

v. Siguen en la región baja elementos menores a los de la región alta extendida:

$$r[j] \le r[i], \qquad con\ 1 \le j < i. \tag{3.Ps(ii)}$$

Demostración:

Haremos la demostración por inducción sobre ℓ, el número de elementos en la región ordenada del arreglo ($i = n - \ell$). Siendo así, si tengo ℓ elementos ordenados, entonces las posiciones que ocupan esos elementos van de la $n - \ell + 1 = i + 1$ a la posición n.

Base: Para $\ell = 1$ ($i = n - 1$), suponemos que la posición n del arreglo cumple con (3.ii) (la desigualdad (3.i) no tiene caso verificarla; $i + 1 = n$, por lo que no existe j tal que $n \le j < n$ y se cumple por vacuidad), esto es, que en la posición n de la lista se encuentra el elemento con la llave mayor:

$$r[j] \le r[n], \qquad \text{con } 1 \le j \le n-1$$

Supongamos ahora que elegimos a $r[k]$ que cumple con (3.iii):

$$r[j] \le r[k], 1 \le k \le i = n-1, \text{con } 1 \le j \le i = n-1$$

y lo intercambiamos con $r[i = n-1]$. Como $r[k]$ es uno de los que cumplen (3.iii), se cumple (3.Ps(i)), que dice que

$$r[j] \leq r[j+1], \qquad \text{con } i = n-1 \leq j < n$$

o lo que es lo mismo, que

$$r[n-1] \leq r[n]$$

Por otro lado, como el elemento que se encuentra ahora en la posición $i = n-1$ fue elegido por tener valor máximo en el rango $[1 \ldots i = n-1]$, se cumple (3.Ps(ii)), que dice

$$r[j] \leq r[i = n-1], \qquad 1 \leq j < i = n-1$$

con lo que se verifica que se cumplen los predicados (3.Ps(i)) y (3.Ps(ii)) dados los predicados (3.i) a (3.iii).

Inducción: Supongamos ahora que el lema se cumple para ℓ posiciones ordenadas ($R[(i+1) = n-\ell+1 \ldots n]$) y veamos qué pasa cuando queremos trabajar con $\ell+1$ elementos ordenados (en las posiciones $i = n-\ell$ a n).

Por la hipótesis de inducción, tenemos que en los primeros ℓ, $1 < \ell < n-1$ elecciones e intercambios, quedamos con la siguiente configuración:

$$(a)\ r[j] \leq r[j+1], \qquad \text{con } (i+1) \leq j < n.$$
$$(b)\ r[j] \leq r[i+1], \qquad \text{con } 1 \leq j \leq i.$$

Supongamos ahora que escogemos $r[k]$ que cumple:

$$(c)\ r[j] \leq r[k],\ \text{con } 1 \leq k \leq i = n-\ell-1, \qquad \text{con } 1 \leq j \leq i.$$

Hagamos el intercambio, quedando $r[k]$ en la posición i del arreglo. Como el nuevo valor de $r[i]$ es uno de los que estaba en la parte baja del arreglo, cumple con

$$r[i] \leq r[i+1],$$

y por la hipótesis de inducción, tenemos

$$r[j] \leq r[j+1], \text{con } (i+1) \leq j < n.$$

De estas dos desigualdades, obtenemos

$$r[j] \leq r[j+1], \text{con } i \leq j < n.$$

Por otro lado, como $r[k]$ se eligió de tal manera que

$$r[j] \leq r[k], \text{con } 1 \leq j \leq i,$$

y como intercambiamos a $r[k]$ con $r[i]$, tenemos que esta desigualdad se convierte en:

$$r[j] \leq r[i], \text{con } 1 \leq j < i,$$

lo que demuestra el paso inductivo.

∴ al ir eligiendo de esta manera al máximo de lo que va quedando en la región baja se consigue ordenar al arreglo. ❏

Pasamos ahora a demostrar que las implementaciones dadas para cada uno de los métodos cumplen con las especificaciones requeridas.

Lema 2.4 (Correctez del constructor) *Sea $R = \langle r_1, \ldots, r_n \rangle$ la lista de registros que se le proporciona al constructor* SelectionSort, *tal que los registros contienen llaves válidas. Entonces $R^{(0)}$, el arreglo resultado de la ejecución del constructor SelectionSort, es tal que cumple*

i. $perm(R, R^{(0)})$, (ASps-1)

ii. $n = |R|$, *número de registros en* $R^{(0)}$. (ASps-2)

Adicionalmente a estos dos predicados, cumple

iii. $r[j] \leq r[n]$ *con* $1 \leq j < n$. (SSps-3)

Demostración:

Veamos que cada una de las postcondiciones se cumpla.

i. Veamos por qué esta invariante se cumple.
La llamada del constructor de la clase abstracta (AbstractSort) en la línea 4 del Listado 2.2 simplemente coloca a R en un arreglo y coloca en n el número de elementos del arreglo. La línea 5, que es una llamada a select, no modifica el contenido del arreglo – como demostraremos a continuación – y la línea 6 simplemente intercambia dos elementos del arreglo, por lo que el resultado es una permutación del arreglo original.
∴ se cumple que al terminar el constructor SelectionSort

$$perm(R, R^{(0)}).$$

ii. Este inciso se cumple porque el constructor de la superclase, AbstractSort, asigna a n el número de registros colocados, y en el resto del constructor SelectionSort a la variable n no se le asigna ningún nuevo valor; por lo que al terminar el constructor SelectionSort se cumple que

$$n = |R|.$$

∴ las postcondiciones para la implementación de SelectionSort se cumplen.

Finalmente, en la línea 5 se selecciona al elemento con valor máximo en el arreglo, al que se intercambia, en la línea 6, con el elemento en la última posición del arreglo. Suponiendo que select en efecto selecciona al elemento mayor de la región [first...last] – tenemos que demostrarlo – y como le pedimos que la región sea toda la lista, se elige al elemento mayor de la lista, y en la línea 6 se le intercambia con el de la última posición. Entonces se cumple:

$$r[j] \leq r[n] \qquad \text{con } 1 \leq j < n.$$

❏

Ahora veamos el comportamiento de los dos métodos que se ejecutan iterativamente, para poder demostrar el teorema de correctez para este algoritmo. Lo haremos a través de varios lemas. Utilizaremos a ℓ para contar las iteraciones, y a $i = n - \ell$ para delimitar la región baja del arreglo que aún no ha sido ordenada.

Lema 2.5 (Correctez de select) *Supongamos que las siguientes precondiciones se cumplen antes de la ℓ-ésima ejecución de* select, *con $i = n - \ell$:*

i. $perm(R, R^{(\ell-1)})$		(Spr-1).
ii. $r^{(\ell-1)}[j] \leq r^{(\ell-1)}[j+1]$,	*con* $(i+1) \leq j < n$	(Spr-2).
iii. $r^{(\ell-1)}[j] \leq r^{(\ell-1)}[n-\ell+1]$,	*con* $1 \leq j < n-\ell+1$	(SSps-3).

Entonces, después de la ℓ-ésima ejecución de select *los siguientes predicados se cumplen:*

iv. $1 \leq k \leq i$		(Sps-1).
v. $perm\big(R^{(\ell)}[1..i], R^{(\ell)\prime}[1..i]\big)$		(Sps-2).
vi. $r^{(\ell)}[j] = r^{(\ell)\prime}[j]$,	*con* $(i+1) \leq j \leq n$	(Sps-3).

Demostración:

Demostraremos primero que el código en las líneas (10) a (16) deja a la variable maxPos apuntando a la localidad que contiene al elemento mayor de la región [first... last]. Demostraremos por inducción sobre i, que al terminar la i-ésima iteración, posMax contiene la posición del elemento mayor de la región [first... i].

Base: Antes de que se ejecute el ciclo por primera vez, para $i =$ first $+ 1$, posMax contiene el valor de first, por lo que apunta al elemento mayor en la región [first... first]. Al ejecutarse el ciclo por primera vez, i toma el valor de first $+ 1$, y el ciclo hace que posMax quede apuntando al elemento mayor entre el que ocupa la posición first y el que ocupa la posición first+1, que será el máximo en la región [first... first $+ 1$], lo que demuestra el caso base.

Inducción: Supongamos que cuando entramos a la iteración para $i = k$ con $2 < k \leq$ last, posMax apunta a la posición que contiene al elemento mayor en la región [first... $i - 1$]. Si el elemento en la posición i es mayor que el elemento en la posición posMax, asignamos i a posMax. Esto es correcto ya que

$$r[k] \leq r[\text{posMax}], \text{ con first} \leq k < i, \tag{2.17}$$

y

$$r[\text{maxPos}] < r[i].$$

Al asignar maxPos ← i, conseguimos

$$r[k] \leq r[\text{posMax}], \text{ con first} \leq k \leq i, \tag{2.18}$$

lo que demuestra la inducción para uno de los casos. Para el caso en que $r[i] \leq r[\text{posMax}]$, no hacemos nada, pero de igual manera, combinada con (2.17) dado que posMax no se modifica, obtenemos (2.18).

De lo anterior, al terminar de ejecutarse esta iteración el valor de posMax corresponde a la posición en la que se encuentra el máximo valor de la región [first... last]. Como este ciclo no modifica a R, y esto es lo único que se ejecuta dentro de select, también tenemos que para cada invocación de select(1, k) se cumple (iv) – $perm(R^{(0)}, R^{(0)'})$. Y como el ciclo ni siquiera examina a los elementos más allá de k, los elementos en la región [last + 1 ... n] permanecen en su lugar, por lo que siguen ordenados entre sí. ❏

Lema 2.6 (Correctez de join) *Supongamos que antes de empezar la ℓ-ésima ejecución del método join en la línea* (31) *en el Listado 2.1 se cumplen los siguientes predicados:*

$$
\begin{array}{lll}
i. & perm(R^{(\ell-1)}[1\dots i], R^{(\ell-1)'}[1\dots i]) & \\
ii. & r^{(\ell-1)}[j] = r^{(\ell-1)'}[j] & con\ (i+1) \le j \le n \\
iii. & r^{(\ell-1)}[j] \le r^{(\ell-1)}[j+1] & con\ (i+1) \le j < n \\
iv. & r^{(\ell-1)'}[j] \le r^{(\ell-1)'}[k] & con\ 1 \le j \le i,\ \ 1 \le k \le i
\end{array}
$$

si se ejecuta la implementación de join presentada en las líneas (21) *a* (23) *del Listado 2.2, entonces al terminar la ejecución de join(k, i, n), los siguientes predicados se cumplen:*

$$
\begin{array}{llr}
v. & perm(R^{(\ell-1)'}, R^{(\ell)}) & \text{(Jps-1)} \\
vi. & r^{(\ell)}[j] \le r^{(\ell)}[j+1] \qquad con\ \ i \le j < n. & \text{(Jps-2)} \\
vii. & \left(r^{(\ell)}[j] = r^{(\ell-1)'}[m]\right) \wedge \left(r^{(\ell)}[j'] = r^{(\ell-1)'}[m']\right) & \text{(Jps-3)} \\
& \wedge\Big(\left(r^{(\ell-1)'}[m] \le r^{(\ell-1)'}[m']\right) \implies \left(r^{(\ell)}[j] \le r^{(\ell)}[j']\right)\Big) & \\
& \qquad con\ i \le j < j' \le n, & \\
& \qquad con\ i+1 \le m < m' \le n. &
\end{array}
$$

Demostración:

Las precondiciones para join son exactamente las postcondiciones para select, que ya establecimos en el Lema 2.5 que se cumplen.

El predicado (Jps-1) se cumple, pues lo único que hacemos en join es intercambiar dos elementos en el arreglo.

Para los predicados (Jps-2) y (Jps-3), las postcondiciones que debe cumplir join se parecen mucho a la consecuencia del Lema 2.3. Veamos si cumple el antecedente del lema. Lo haremos por inducción sobre ℓ, el número de pasada.

Base: Para $\ell = 1$ $(i = n-1)$, como tenemos un único elemento en la región ordenada del arreglo, se cumple el antecedente (3.i). El constructor colocó en la posición n al elemento mayor del arreglo, por lo que se cumple también el antecedente (3.ii). Como select es correcto, seleccionó a un elemento que cumple con (3.iii) en el constructor, y después en la primera pasada. En esta primera pasada, join coloca a $r^{(0)'}[k]$ en la posición $r^{(0)'}[n-1]$, por lo que, para $\ell = 1$ se cumple el inciso (3.Ps(i)). Como tenemos que los tres antecedentes del Lema 2.6 se cumplen, entonces se cumplen todos los consecuentes de ese lema, lo que hace verdaderas a las postcondiciones (Jps-1) y (Jps-2). Por último, como join no manipula al elemento en la posición n, éste conserva su posición relativa con respecto a sí mismo, cumpliéndose con esto la postcondición (Jps-3).

<u>Inducción:</u> Veamos que se cumplen los antecedentes del Lema 2.3.

i. Por la hipótesis de inducción, en la iteración $\ell - 1$ se dejó la región alta del arreglo ordenada:

$$r^{(\ell-1)}[j] \leq r^{(\ell-1)}[j+1] \qquad \text{con } (i+1) \leq j \leq n.$$

Acabamos de establecer que select deja a esa región alta intacta:

$$r^{(\ell-1)}[j] = r^{(\ell-1)'}[j] \qquad \text{con } i+1 \leq j \leq n.$$

De esto, tenemos que

$$r^{(\ell-1)'}[j] \leq r^{(\ell-1)'}[j+1] \qquad \text{con } (i+1) \leq j < n$$

ii. Por la hipótesis de inducción y la correctez de select, se cumple que en la iteración $\ell - 1$, select eligió un elemento en la posición k tal que

$$r^{(\ell-2)'}[j] \leq r^{(\ell-2)'} \quad \text{con } 1 \leq j \leq i+1, \;\; 1 \leq k \leq i+1.$$

En la iteración anterior, join intercambió los valores de $r^{(\ell-2)'}[k]$ con $r^{(\ell-2)'}[i+1]$, convirtiendo la desigualdad anterior en:

$$r^{(\ell-1)}[j] \leq r^{(\ell-1)}[i+1] \qquad \text{con } 1 \leq j \leq (i+1),$$

cubriéndose de esta manera el antecedente (ii).

iii. En esta iteración establecimos ya que select eligió k tal que

$$r^{(\ell-1)}[j] \leq r^{(\ell-1)}[k], \qquad \text{con } 1 \leq j \leq i, \; 1 \leq k \leq i,$$

y lo que hace join en la ℓ-ésima pasada es, precisamente, intercambiar $r^{(\ell-1)'}[k]$ con $r^{(\ell-1)'}[i]$.

Dado que la ℓ-ésima ejecución de join cumple con los antecedentes del Lema 2.3, podemos concluir que se cumple (Jps-2) al terminar la ejecución de join.

Por último, debemos establecer (Jps-3). Pero como la implementación de join no mueve a los registros $i+1\ldots n$, estos registros no sólo conservan el orden relativo que tenían antes de ejecutarse join, sino que están en exactamente las mismas posiciones. ❏

2.5.3. Complejidad de la especialización para el ordenamiento por selección

Como mencionamos antes, para determinar la complejidad del ordenamiento por selección se debe determinar la complejidad del constructor y de cada uno de los métodos, select y join. Dado que el constructor invoca a select, revisaremos primero la complejidad de este método.

Implementación del método select. En select lo único interesante es el ciclo de las líneas (11) a (16), ya que dependiendo de la organización de los datos cuando se llega al ciclo, la actualización dentro del ciclo se va o no a ejecutar. Tengamos presente que este método será llamado con valores particulares para first y last, que en el caso de esta especialización será siempre 1 e $i = n - \ell$ respectivamente.

select(int first, int last). (Líneas (9) a (18) del Listado 2.2).

Inicialización: Línea (10). Asignación a posMax.

Número de pasos:	1	en:	$O(1)$.

Iteración: líneas 11–16. Se ejecuta last − first veces.

Número de veces:	last − first	en:	$O(i)$.

- Control de la iteración en la línea 11.

Número de pasos:	1	en:	$O(1)$.

- La comparación en la línea 14.

Número de pasos:	1	en:	$O(1)$.

- La asignación de un nuevo máximo(*) - línea 15.

Número de pasos:	1	en:	$O(1)$.

Nota(*): La actualización del valor de posMax no siempre se lleva a cabo, ya que depende de los valores particulares en la lista. De acuerdo a eso, podemos definir un *mejor caso*, un *caso promedio* y un *peor caso*.

Mejor Caso: Se presenta cuando no se tiene que actualizar ni una vez el valor para posMax. Este caso se da cuando los datos vienen casi ordenados, con el elemento de mayor valor en la primera posición, y el resto de los elementos, a partir de la segunda posición, ordenados. En este caso siempre el primer elemento va a ser el mayor, por lo que no se tendrá que actualizar el valor de posMax dentro de la iteración – ver Figura 2.3.

Número de pasos:	1	en:	$O(1)$.

Peor Caso: Se presenta cuando los datos vienen ordenados, y entonces en cada iteración se toma al primero como propuesta, pero se tiene que intercambiar con cada uno de los elementos de la lista.

Número de pasos:	last − first	en:	$O(i)$.

Caso Promedio: Podemos suponer que cualquiera de las posibles presentaciones de los datos es igual de probable. Si esto es así, cada vez que se entra a la iteración se tendrán que realizar, en promedio, (last − first)/2 actualizaciones para posMax.

Número de pasos:	$\frac{\text{last} - \text{first}}{2}$	en:	O(i).

Complejidad para select:

Mejor caso:	Número de pasos:	2(last − first) + 2	en:	O(last − first).
Peor caso:	Número de pasos:	3(last − first) + 1	en:	O(last − first).
Caso promedio:	Número de pasos:	$\frac{5}{2}$(last − first) + 1	en:	O(last − first).

De donde, la complejidad para select está acotada de la siguiente manera:

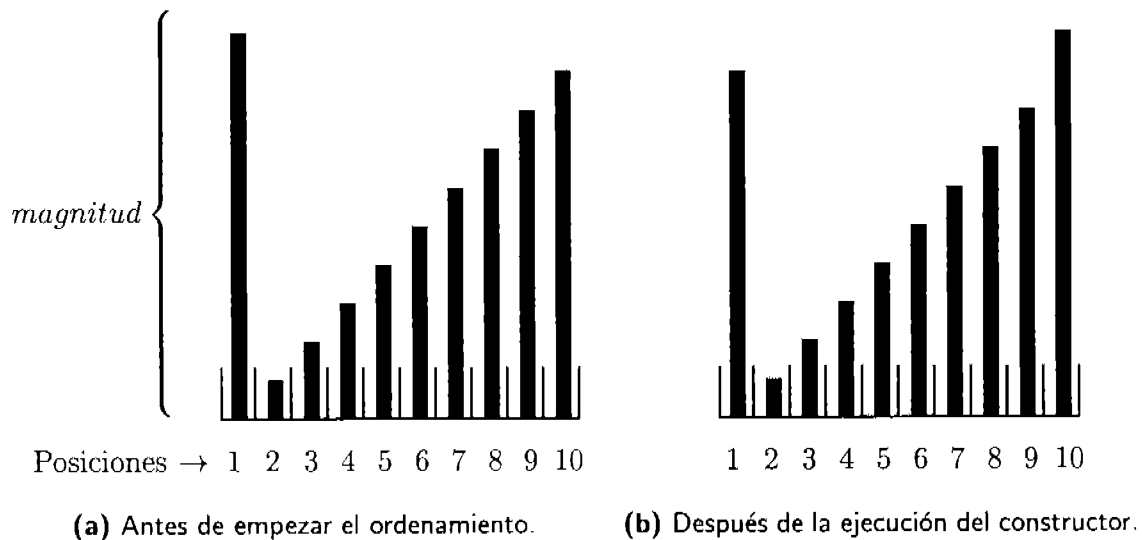
$$\mathbf{2} \cdot (\mathsf{last} - \mathsf{first}) + \mathbf{2} \leq \mathsf{f_{sel}}(\mathsf{last} - \mathsf{first}) \leq \mathbf{3} \cdot (\mathsf{last} - \mathsf{first}) + \mathbf{1},$$

lo que la coloca en

$$O(\mathsf{last} - \mathsf{first}).$$

Como podemos observar en la Figura 2.3 de la primera vez que se invoca a select, se selecciona al primer elemento, después de compararlo contra todos los demás, pero sin realizar ningún intercambio. Al invocar a join se intercambian el primero y el último elemento. En la siguiente iteración se va a intercambiar el primero con el penúltimo. En las siguientes iteraciones, se intercambia al que está en la primera posición, que es el mayor de la región baja, con el que se encuentra en la última posición de la región, por lo que siempre se colocará en el primer lugar al elemento mayor de la región; esto provocará que una vez que se proponga al primer elemento como el mayor, no será desplazado en ese ciclo.

Figura 2.3: Permutación inicial para Mejor Caso en el ordenamiento por selección. Cada vez que se invoca a select, el primer elemento resulta ser el mayor; se intercambia con el último de la región, y eso coloca al siguiente mayor en la primera posición nuevamente.



(a) Antes de empezar el ordenamiento. **(b)** Después de la ejecución del constructor.

Cabe aclarar que como lo único que aparece en este método es la parte correspondiente a la inicialización y la iteración, realmente el peso de la complejidad está dado por los distintos casos que examinamos.

Implementación del constructor (SelectionSort) En el constructor, la parte principal, heredada del constructor de la superclase, es la de configurar la estructura de datos sobre la

que se va a trabajar, para que los elementos queden en una estructura de datos dinámica, pero con acceso directo. Del constructor de la superclase se hereda la complejidad de esta parte, más lo que cueste la especialización particular.

SelectionSort(Object[] objs, Comparator comp) (Líneas (2) a (7)).

Inicialización: (líneas 102-105.)

- Construcción de la permutación original.
  Número de pasos: $c \cdot n$ en: $O(n)$.
- Asignación a n. Número de pasos: 1 en: $O(1)$.
- Asignación a posMax (invocación a select)(*):
  Número de pasos: $2 + 3 \cdot (n - 1)$ en: $O(n)$.
- Intercambio del máximo con la última posición.
  Número de pasos: 3 en: $O(1)$.

Nota(*): Dado que el método select presenta un mejor, peor y caso promedio, tenemos varias posibles evaluaciones para la llamada a select. Sustituyendo a first por 1 y a last por n obtenemos:

Peor caso:	Número de pasos:	$3n - 2$	en:	$O(n)$.
Mejor caso:	Número de pasos:	$2n$	en:	$O(n)$.
Caso promedio:	Número de pasos:	$\frac{5}{2}n + \frac{1}{2}$	en:	$O(n)$.

Total para el constructor:

Mejor caso:	Número de pasos:	$(c + 2)n + 4$	en:	$O(n)$.
Peor caso:	Número de pasos:	$(c + 3)n + 2$	en:	$O(n)$.
Caso promedio:	Número de pasos:	$\left(c + \frac{5}{2}\right)n + \frac{9}{2}$	en:	$O(n)$.

De donde, la complejidad para join está acotada de la siguiente manera:

$$\mathbf{(c + 2)n + 4} \leq \mathsf{f}_{\mathsf{constr}}(\mathsf{n}) \leq \mathbf{(c + 3)n + 2}$$

y es

$$O(n)$$

Complejidad de la implementación de join. Este método recibe dos argumentos, que corresponden a las posiciones que ocupan los registros a intercambiar, e independientemente de cuáles sean esas posiciones, procede a intercambiarlos. De esto, su complejidad es constante y está dada por tres operaciones:

join(int sel, int low, int high). (Líneas 21 a 23 del Listado 2.2).

Intercambio de 2 elementos: Número de pasos: 3 en: $O(1)$.

Complejidad para join:

$$\mathsf{f_{join}}(\mathsf{high} - \mathsf{low}) = 3$$

y está en la clase

$$O(1).$$

Pasemos al último paso en el cálculo de la complejidad de la especialización, que consiste en sustituir los valores para cada método en la fórmula general. Para ello, recordemos la fórmula general para la complejidad del algoritmo genérico:

$$\textbf{Complejidad de genericSort} = \mathsf{f_{constr}}(\mathsf{n}) + \sum_{\ell=1}^{n-1}\Big(\mathsf{f_{sel}}(\mathsf{n} - \ell) + \mathsf{f_{join}}(\ell)\Big).$$

Acabamos de mostrar la complejidad de cada uno de los componentes de esta ecuación para peor, mejor y caso promedio, sustituyendo en select a first por 1 y a last por $i = n - \ell$, que es como es invocado desde genericSort.

$$\begin{array}{lcll}
\mathsf{f_{constr}}(\mathsf{n}) & = & (c+3)n+2 & \text{Constructor.}\\
\mathsf{f_{sel}}(\mathsf{n}-\ell) & = & 3(n-\ell-1)+1 & \text{select.}\\
\mathsf{f_{join}}(\ell) & = & 3 & \text{join.}
\end{array}$$

Sustituyendo en la fórmula para el ordenamiento genérico, en el peor caso tenemos:

$$\begin{aligned}
T_{SelectionSort}(n) = &\quad (c+3)n + 2 + \sum_{\ell=1}^{n-1}\Big(3\cdot(n-\ell-1)+1+3\Big)\\
= &\quad (c+3)n + 2 + 3\cdot\sum_{\ell=1}^{n-1}(n-\ell-1) + \sum_{\ell=1}^{n-1}4\\
= &\quad (c+3)n + 3\cdot\sum_{i=1}^{n-1}(i-1) + 4n - 4 + 2\\
= &\quad (c+3)n + 3\cdot\sum_{i=0}^{n-2} i + 4n - 4 + 2\\
= &\quad (c+7)n + 3\Big(\frac{(n-2)(n-1)}{2}\Big) - 2\\
= &\quad \frac{3}{2}n^2 + (c+\frac{5}{2})n + 1\\
= &\quad O(n^2). \qquad \textbf{(Peor caso)}
\end{aligned}$$

Para el mejor caso, tenemos la siguiente sustitución

$$\begin{array}{lcll}
\mathsf{f_{constr}}(\mathsf{n}) & = & (c+2)n+4 & \text{Constructor.}\\
\mathsf{f_{sel}}(\mathsf{n}-\ell) & = & 2(n-\ell-1)+2 = 2i & \text{select.}\\
\mathsf{f_{join}}(\ell) & = & 3 & \text{join.}
\end{array}$$

Sustituyendo en la fórmula para el ordenamiento genérico, considerando el mejor caso, tenemos:

$$
\begin{aligned}
T_{SelectionSort}(n) &= (c+2)n+4+\sum_{\ell=1}^{n-1}\Big(2\cdot(n-\ell-1)+2+3\Big) \\
&= (c+2)n+4+2\cdot\sum_{i=1}^{n-1}(i-1)+\sum_{\ell=1}^{n-1}5 \\
&= (c+2)n+4+2\frac{(n-2)(n-1)}{2}+5n-5 \\
&= n^2+(c+4)n+1 \\
&= O(n^2). \qquad \textbf{(Mejor caso)}
\end{aligned}
$$

Para el caso promedio, y suponiendo que cualquier permutación de los datos es igual de probable al empezar el algoritmo, tendremos:

$$
\begin{aligned}
\mathsf{f}_{\mathsf{constr}}(\mathsf{n}) &= \Big(c+\frac{5}{2}\Big)n+\frac{9}{2} && \text{Constructor.} \\
\mathsf{f}_{\mathsf{sel}}(\mathsf{n}-\ell) &= \frac{5}{2}\Big(n-\ell-1\Big)+1 && \text{select.} \\
\mathsf{f}_{\mathsf{join}}(\ell) &= 3 && \text{join.}
\end{aligned}
$$

Sustituyendo en la fórmula para el ordenamiento genérico, considerando el caso promedio, tenemos:

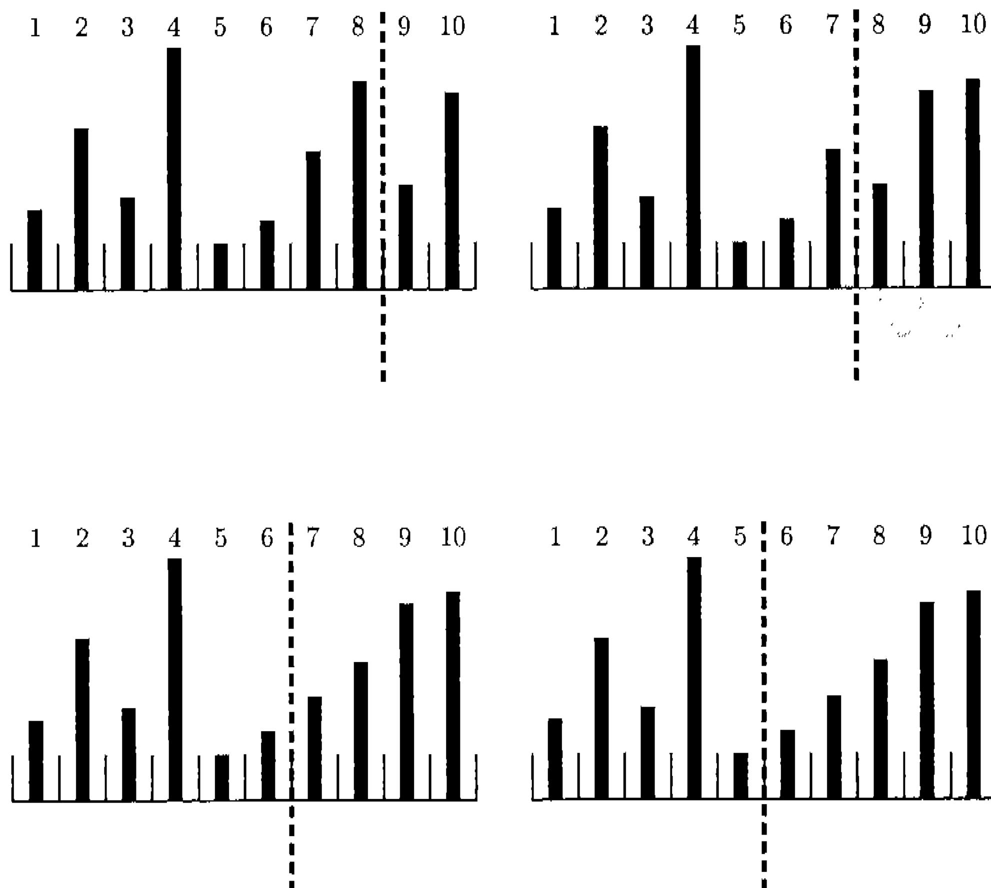
$$
\begin{aligned}
T_{SelectionSort}(n) &= \Big(c+\frac{5}{2}\Big)n+\frac{9}{2}+\sum_{\ell=1}^{n-1}\Big(\frac{5}{2}(n-\ell-1)+1+3\Big) \\
&= \Big(c+\frac{5}{2}\Big)n+\frac{9}{2}+\frac{5}{2}\cdot\sum_{\ell=1}^{n-1}(n-\ell-1)+\sum_{\ell=1}^{n-1}4 \\
&= \Big(c+\frac{5}{2}\Big)n+\frac{5}{2}\Big(\frac{(n-2)(n-1)}{2}\Big)+4(n-1)+\frac{9}{2} \\
&= \frac{5}{4}n^2+\Big(c+\frac{11}{4}\Big)n+3 \\
&= O(n^2). \qquad \textbf{(Caso promedio)}
\end{aligned}
$$

La complejidad está dada, realmente, porque la lista parcial se tiene que recorrer en cada iteración para encontrar al máximo elemento. El hecho de que se intercambie o no al máximo no afecta gran cosa el cómputo de la complejidad. Al eliminar las constantes y los términos menores nos queda, en los tres casos, un sumando en n^2, por lo que en los tres casos la complejidad es $O(n^2)$.

2.6. Especialización para el ordenamiento por inserción

La especialización para el ordenamiento por inserción (InsertionSort) resulta de elegir de manera muy sencilla en la región baja al elemento a mover para, de manera un poco más complicada, insertarlo en el lugar que le corresponde en la región alta. La ejecución de las primeras cuatro iteraciones de esta especialización se muestra en la Figura 2.4.

Figura 2.4: Ordenamiento por inserción: 4 pasadas



El código para la especialización se muestra en el Listado 2.3. En este caso no hay que hacer nada en el constructor, excepto lo que corresponde a armar la primera permutación, que es con la que llegan los datos. En el método `select (int first , int last)` simplemente elegimos al elemento en la posición `first`; y en el método `join(int sel , int low, int high)` deberemos buscarle el lugar que le corresponde de entre los elementos colocados en las posiciones low, ..., high, de tal manera de mantener las invariantes del algoritmo genérico.

Listado 2.3: Especialización para el ordenamiento por inserción

```
1  public class InsertionSort extends AbstractSort {
2    public InsertionSort(Object[] objs, BasicComparator comp) {
3      super(objs, comp);
4    }
5    protected int select(int first, int last) {
6      return last;
7    }
8    protected void join(int sel, int low, int high) {
9      for(int j = low; j < high &&
10         ((BasicComparator)comp).gtr(objs.get(j), objs.get(j + 1));
11                                                      j++) {
12         swap(j, j + 1);
13     }
14   }
15 }
```

2.6.1. Correctez de la especialización de ordenamiento por inserción

Como mencionamos antes (y lo hicimos en la sección anterior), lo único que tenemos que demostrar es que cada uno de los métodos que especializamos cumplen con las precondiciones y postcondiciones dadas para el AGOC.

Lema 2.7 (Correctez del constructor) *Al terminar la ejecución de InsertionSort (líneas 27–34 del Listado 2.3) se cumplen la Postcondiciones [ASps-1] y [ASps-2], a saber:*

i. $perm(R, R^{(0)})$

ii. $n = |R^{(0)}|$, *número de registros en* $R^{(0)}$

Demostración:

Este método lo único que hace es invocar al constructor del AGOC, que construye R. Entonces, la permutación $R^{(0)}$ al terminar la ejecución del constructor de esta clase es, precisamente, R,

$\therefore$ como $R^{(0)} = R$ y $perm(R, R)$, tenemos, $perm(R, R^{(0)})$. Asimismo, $n = |R|$, pues esto ya lo calcula bien el constructor de la super-clase. ❏

La correctez de select se demuestra de manera muy sencilla, ya que este método no hace casi nada:

Lema 2.8 (Correctez de select) *Si en la ℓ-ésima iteración de genericSort – líneas* (27) *a* (34) *– antes de invocarse a select en la línea* (30), *se cumple que:*

i. $perm(R, R^{(\ell-1)})$ (Spr-1)

ii. $r^{(\ell-1)}[j] \leq r^{(\ell-1)}[j+1]$ *con* $n - \ell + 1 \leq j < n$; (Spr-2)

entonces, inmediatamente después de la invocación de select se van a cumplir las postcondiciones (Sps-1) *a* (Sps-4).

Demostración:

Haremos la demostración por inducción sobre ℓ.

Base: Para $\ell = 1$ $(i = n - \ell)$, se cumple la precondición (Spr-1), ya que es la postcondición del constructor. Respecto a la precondición (Spr-2) también se cumple, por vacuidad, ya que no hay ningún valor para j en el rango $n \leq j < n$. Veamos ahora, una por una, que se cumplen las postcondiciones.

$$i.\ 1 \leq k \leq i = n - \ell \qquad \text{(Sps-1)}$$

El valor que se regresa, last $= i$, está en el rango especificado.

$$ii.\ perm(R^{(0)}[1 \ldots i], R^{(0)'}) \qquad \text{(Sps-2)}$$

Dado que no se movió a ningún elemento, tenemos $R^{(0)} = R^{(0)'}$, en particular la región baja se mantiene igual. También se cumple, nuevamente por vacuidad:

$$iii.\ r^{(0)}[j] \leq r^{(0)}[j+1] \qquad \text{con } n \leq j < n \qquad \text{(Sps-3)}$$

y por el mismo argumento, se cumple

$$perm(R^{(0)}, R^{(0)'}) \qquad \text{(Sps-4)}$$

Inducción: Supongamos ahora que las pre y postcondiciones se cumplen en las iteraciones con $1 \leq \ell < m$ y veamos qué pasa en la iteración para $\ell = m$.

Por la hipótesis de inducción, tenemos que las precondiciones se cumplen:

$$i.\ perm(R, R^{(m-1)}) \qquad \text{(Spr-1)}$$
$$ii.\ r^{(m-1)}[j] \leq r^{(m-1)}[j+1] \qquad \text{con } n - \ell + 1 \leq j < n; \qquad \text{(Spr-2)}$$

i. Para la postcondición (Sps-1), es claro que el valor que regresa el método corresponde a una posición en la región baja, ya que regresa, precisamente, la última posición de esta región.

ii. Como no se modifica nada en la permutación con la que se empieza, es claro que se cumple (Sps-2), que exige que la región baja resultante sea una permutación de la región baja original.

iii. Por la misma razón que el inciso anterior, si antes de ejecutarse select la región alta se encontraba ordenada, como no se movió nada, permanece ordenada al terminarse de ejecutar select, cumpliéndose la postcondición (Sps-3). Asimismo, como $R^{(m-1)} = R^{(m-1)'}$, se cumple la postcondición (Sps-4). ❑

Pasamos ahora a demostrar que join cumple con su especificación. Para que la demostración de que join cumple con sus especificaciones sea más clara, primero demostraremos un lema que nos servirá en la demostración de correctez para join.

Lema 2.9 *Sea* R[low ... high] *una lista de elementos con acceso directo, tal que* R[low + 1 ... high] *están ordenados. Entonces, si se ejecuta el ciclo en las líneas* (9) *a* (13) *sobre esta lista, el resultado es que* R[low ... high] *quedará ordenado.*

Demostración:

Haremos la demostración por inducción en el número de veces que se ejecuta el ciclo.

Base: Si el ciclo no se ejecuta ni una vez, quiere decir que

$$r[\mathsf{low}] \leq r[\mathsf{low}+1]$$

y como ya teníamos

$$r[\mathsf{low}+1] \leq r[\mathsf{low}+2] \leq \ldots \leq r[\mathsf{high}]$$

el caso base queda demostrado.

Inducción: Supongamos que si el ciclo se ejecuta k veces, el arreglo queda ordenado. Veamos qué pasa si se ejecuta $k+1$ veces $(0 < k+1 < \mathsf{high} - \mathsf{low})$. Como el ciclo no paró después de k iteraciones, quiere decir que, en el último ciclo antes del actual, el elemento en la posición $j = \mathsf{low} + k - 1$ es mayor que el elemento en la posición $j+1 = \mathsf{low} + k$, y que fueron intercambiados, con los elementos $R[\mathsf{low} \ldots \mathsf{low}+k]$ ordenados. En la iteración para $k+1$, nuevamente se intercambian los elementos en las posiciones $k+1$ y $k+2$, así que $r[j+1] \leq r[j+2$ al terminar la iteración. Como ya no se vuelve a ejecutar el ciclo, quiere decir que $r[j+2] \leq r[j+3]$, y como el resto de los elementos ya estaba ordenado, queda toda esta porción del arreglo ordenada. ❏

Lema 2.10 (Correctez de join) *Si en la ℓ-ésima iteración de genericSort – líneas* (27) *a* (34) *– antes de invocarse a join en la línea* (31), *se cumplen las precondiciones* (Sps-1) *a* (Sps-4), *entonces las postcondiciones* (Jps-1) *a* (Jps-3) *se van a cumplir.*

Demostración:

Demostraremos por inducción en el número de iteraciones que el lema se cumple.

Base: En la primera iteración $(\ell = 1, i = n-1)$, join es invocado con $\mathsf{sel} = n-1$, $\mathsf{low} = n-1$ y $\mathsf{high} = n$, por lo que el ciclo en las líneas (9) a (13) en el Listado 2.3 se ejecuta una única vez, comparando a los dos últimos elementos de la región alta. La primera disyunción de la condición se cumple, por lo que la segunda disyunción determina si se hace el intercambio o no de los elementos en las posiciones $n-1$ y n. Una vez terminado el ciclo, tenemos que el elemento en la posición $n-1$ es menor o igual al elemento en la posición n. Por lo tanto se cumplen:

$$i.\ perm(R^{(0)'}, R^{(1)}) \qquad \text{(Jps-1)}$$

ya que únicamente se intercambiaron dos elementos.

$$ii.\ r^{(1)}[n-1] \leq r^{(1)}[n] \qquad \text{con } n-1 \leq j < n \qquad \text{(Jps-2)}$$

por la ejecución del ciclo exactamente una vez y ya sea que se haya ejecutado el intercambio o no.

$$\begin{aligned} iii.\ & \left(r^{(\ell)}[j] = r^{(\ell-1)'}[m]\right) \wedge \left(r^{(\ell)}[j'] = r^{(\ell-1)'}[m']\right) \\ & \wedge\Big(\left(r^{(\ell-1)'}[m] \leq r^{(\ell-1)'}[m']\right) \Longrightarrow \left(r^{(\ell)}[j] \leq r^{(\ell)}[j']\right)\Big) \qquad \text{(Jps-3)} \\ & \qquad \text{con } i \leq j < j' \leq n, \quad i+1 \leq m < m' \leq n \end{aligned}$$

se cumple por vacuidad, ya que únicamente había un elemento en la región alta, y ese elemento conserva su posición relativa consigo mismo.

Inducción: Supongamos que las postcondiciones (Jps-1) a (Jps-3) se cumplen para $m < \ell = n - i$ y vamos a demostrar que entonces se cumplen para $m = \ell - n - i$. En la ℓ-ésima iteración tenemos que las precondiciones (Sps-1) a (Sps-4) se cumplen. Revisemos cómo es que se cumplen las postcondiciones (Jps-1) a (Jps-3).

$$i.\ perm(R^{(\ell-1)'}, R^{(\ell)}) \qquad \text{(Jps-1)}$$

En el ciclo de las líneas (9) a (13) lo único que se hace es, en todo caso, intercambiar elementos dentro del mismo arreglo, así que lo que resulta es una permutación de la lista antes del ciclo.

$$ii.\ r^{(\ell)}[j] \leq r^{(\ell)}[j+1] \qquad \text{con } n - \ell = i \leq j < n \qquad \text{(Jps-2)}$$

Por el Lema 2.9, y como se cumple la precondición (Sps-3), entonces esta postcondición también se cumple.

$$\begin{aligned} iii.\ & \left(r^{(\ell)}[j] = r^{(\ell-1)'}[m]\right) \wedge \left(r^{(\ell)}[j'] = r^{(\ell-1)'}[m']\right) \\ & \wedge\Big(\left(r^{(\ell-1)'}[m] \leq r^{(\ell-1)'}[m']\right) \Longrightarrow \left(r^{(\ell)}[j] \leq r^{(\ell)}[j']\right)\Big) \qquad \text{(Jps-3)} \\ & \quad \text{con } i \leq j < j' \leq n, \quad i+1 \leq m < m' \leq n \end{aligned}$$

Como los elementos que estaban en $R[i{+}1 \ldots \mathsf{high}]$ únicamente se mueven hacia la izquierda, si es que se mueven, conservan su posición relativa, y por lo tanto, su orden relativo. ❏

Pasamos ahora a evaluar la complejidad de esta especialización.

2.6.2. Complejidad de la especialización del ordenamiento por inserción

Nuevamente examinaremos únicamente la complejidad de cada uno de los métodos, para insertar esta complejidad en la fórmula para el algoritmo genérico.

Implementación del constructor (InsertionSort). Veamos la complejidad del constructor de esta especialización.

InsertionSort(Object[], Comparator) (Líneas (2)– (7) del Listado 2.3).

Invocación al constructor de la super clase:

Número de pasos: $c \cdot n$ en: $O(n)$.

Total para el constructor:

$$\mathsf{f}_{\mathsf{constr}}(\mathsf{n}) = \mathbf{c} \cdot \mathbf{n},$$

que está en

$$O(n).$$

Implementación del método select. En esta especialización, el método select únicamente selecciona al último elemento en la región baja – línea (6) del Listado 2.3.

select(int first, int last). (Líneas 5 a 7 del Listado 2.3).

Entregar el valor de last (línea (6)):

Número de pasos: 1 en: $O(1)$.

Total para select:

$$f_{sel}(\mathsf{last} - \mathsf{first}) = \mathbf{1},$$

y está en

$$O(1).$$

Implementación del método join. Este método es el que realiza prácticamente todo el trabajo, así que deberemos examinarlo con cuidado.

join(int sel, int low, int high) (Líneas 21 a 23 del Listado 2.2).

- Inicialización para el ciclo: Número de pasos: 1 en: $O(1)$.
- Evaluación de la condición del ciclo$^{(*)}$ con $0 \leq m \leq \mathsf{high} - \mathsf{low}$:
  Número de pasos: $m + 1$ en: $O(m)$.
- Intercambio de elementos$^{(*)}$:
  Número de pasos: m en: $O(m)$.

Nota$^{(*)}$: m depende en realidad de que tan fuera de su lugar esté el elemento que queremos insertar. La evaluación de la iteración se lleva a cabo una vez más que el intercambio de elementos, ya que se requiere de una evaluación para salir del ciclo. Presentamos a continuación los distintos casos que se pueden presentar.

Mejor caso: Éste se da cuando no se entra al ciclo ni una sola vez, cada vez que el método join es invocado. Si los datos vienen ya ordenados, la primera vez que se verifica la relación entre el elemento a insertar y el primero en la región alta, la respuesta será que el elemento está ya en su lugar, y el ciclo no se ejecutará ni una sola vez.

- Evaluación de la condición del ciclo:
  Número de pasos: 1 en: $O(1)$.
- Intercambio de elementos dentro del ciclo:
  Número de pasos: 0.

Peor caso: El peor caso es cuando cada vez que se invoca al método join, el elemento que se desea insertar se tiene que insertar en la última posición de la región alta. En este caso, en cada invocación de join, el ciclo se ejecutará $\mathsf{high} - \mathsf{low}$ veces, por lo que se harán $\mathsf{high} - \mathsf{low} + 1$ comparaciones y $\mathsf{high} - \mathsf{low}$ intercambios. Este caso se da si los elementos de la lista vienen ordenados en orden inverso, con el mayor siendo el primer elemento y

el menor ocupando la última posición.

- Evaluación de la condición del ciclo:
  Número de pasos: $\mathsf{high} - \mathsf{low} + 1$ en: $O(\mathsf{high} - \mathsf{low})$.

- Intercambio de elementos dentro del ciclo:
  Número de pasos: $3(\mathsf{high} - \mathsf{low})$ en: $O(\mathsf{high} - \mathsf{low})$.

Caso promedio: Si suponemos que cualquier permutación de los datos es igual de posible, podemos pensar que, en promedio, el ciclo se va a ejecutar $\frac{1}{2}(\mathsf{high} - \mathsf{low})$ veces, de donde el costo queda como sigue:

- Evaluación de la condición en el ciclo:
  Número de pasos: $\frac{\mathsf{high}-\mathsf{low}}{2}+1$ en: $O(\mathsf{high} - \mathsf{low})$.

- Intercambio de elementos dentro del ciclo:
  Número de pasos: $3\ \frac{\mathsf{high}-\mathsf{low}}{2}$ en: $O(n-i)$.

Complejidad para join:

Mejor caso:	Número de pasos:	2	en:	$O(1)$.
Peor caso:	Número de pasos:	$4 \cdot (\mathsf{high} - \mathsf{low}) + 1$	en:	$O(\mathsf{high} - \mathsf{low})$.
Caso promedio:	Número de pasos:	$2 \cdot (\mathsf{high} - \mathsf{low}) + 2$	en:	$O(\mathsf{high} - \mathsf{low})$.

De donde la complejidad para join queda acotada de la siguiente manera:

$$\mathbf{2} \leq \mathsf{f_{join}}(\mathsf{high} - \mathsf{low}) \leq \mathbf{2} \cdot (\mathsf{high} - \mathsf{low}) + \mathbf{2},$$

que se encuentra en

$$O(\mathsf{high} - \mathsf{low}).$$

Complejidad para la especialización en el ordenamiento por inserción: En el caso de esta especialización hay mucha diferencia entre el peor caso y el mejor caso, ya que mientras en el primero la complejidad de join, que es realmente la que pesa en la especialización, es de orden constante, en el segundo caso es de orden lineal. Veamos la complejidad de la especialización completa, y para ello recordemos nuevamente la fórmula general para la complejidad del algoritmo genérico:

$$\textbf{Complejidad de genericSort} = \mathsf{f_{constr}}(\mathsf{n}) + \sum_{\ell=1}^{n-1} \Big(\mathsf{f_{sel}}(\mathsf{n} - \ell) + \mathsf{f_{join}}(\ell)\Big).$$

Como la complejidad de select es de orden constante, no nos preocupamos por los parámetros que se le pasen. join es invocado con $k = i$, $\mathsf{low} = i$ y $\mathsf{high} = n$, por lo que sustituyendo estos valores en las fórmulas encontradas para el peor caso, tenemos:

$$
\begin{array}{lll}
\mathsf{f}_{\mathsf{constr}}(\mathsf{n}) & = c \cdot n & \text{Constructor.} \\
\mathsf{f}_{\mathsf{sel}}(\mathsf{i} = \mathsf{n} - \ell) & = 1 & \text{select.} \\
\mathsf{f}_{\mathsf{join}}(\ell = \mathsf{n} - \mathsf{i}) & = 4 \cdot (n - i) + 2 & \text{join.}
\end{array}
$$

Sustituyendo en la fórmula para el ordenamiento genérico, en el peor caso tenemos:

$$
\begin{aligned}
T_{InsertionSort}(n) = \quad & c \cdot n + \sum_{\ell=1}^{n-1} \Big(4 \cdot (n - \ell) + 2 + 1\Big) \\
= \quad & c \cdot n + 4 \cdot \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 3 \\
= \quad & c \cdot n + 4 \cdot \frac{n^2 - n}{2} + 3n - 3 \\
= \quad & 2n^2 + (c+1)n - 3 \\
= \quad & O(n^2). \qquad \textbf{(Peor caso)}
\end{aligned}
$$

Para el mejor caso, tenemos la siguiente sustitución

$$
\begin{array}{lll}
\mathsf{f}_{\mathsf{constr}}(\mathsf{n}) & = c \cdot n & \text{Constructor.} \\
\mathsf{f}_{\mathsf{sel}}(\mathsf{i} = \mathsf{n} - \ell) & = 1 & \text{select.} \\
\mathsf{f}_{\mathsf{join}}(\ell = \mathsf{n} - \mathsf{i}) & = 1 & \text{join.}
\end{array}
$$

Sustituyendo en la fórmula para el ordenamiento genérico, considerando el mejor caso, tenemos:

$$
\begin{aligned}
T_{InsertionSort}(n) = \quad & c \cdot n + \sum_{i=1}^{n-1} 2 \\
= \quad & c \cdot n + 2n - 2 \\
= \quad & (c+2)n - 2 \\
= \quad & O(n). \qquad \textbf{(Mejor caso)}
\end{aligned}
$$

Para el caso promedio, y suponiendo que cualquier permutación de los datos es igual de probable al empezar el algoritmo, tendremos:

$$
\begin{array}{lll}
\mathsf{f}_{\mathsf{constr}}(\mathsf{n}) & = c \cdot n & \text{Constructor.} \\
\mathsf{f}_{\mathsf{sel}}(\mathsf{i} = \mathsf{n} - \ell) & = 1 & \text{select.} \\
\mathsf{f}_{\mathsf{join}}(\ell = \mathsf{n} - \mathsf{i}) & = 4\frac{n-i}{2} + 2 = 2(n-i) + 2 & \text{join.}
\end{array}
$$

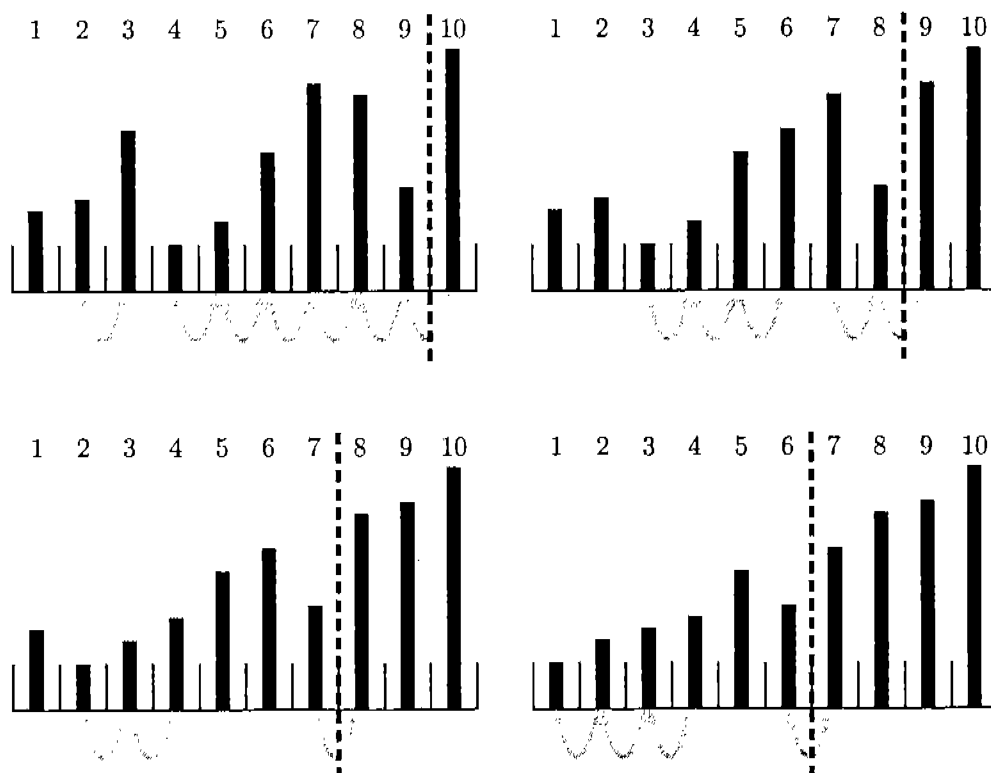
Sustituyendo en la fórmula para el ordenamiento genérico, considerando el caso promedio, tenemos:

$$\begin{aligned}
T_{InsertionSort}(n) &= c \cdot n + \sum_{\ell=1}^{n-1} \Big(2 \cdot (n - i = \ell) + 2 \Big) \\
&= c \cdot n + 2 \cdot \frac{n^2 - n}{2} + 2n - 2 \\
&= n^2 + (c+1)n - 2 \\
&= O(n^2). \qquad \textbf{(Caso promedio)}
\end{aligned}$$

2.7. Especialización para el ordenamiento de burbuja

El ordenamiento de burbuja puede ser visto también como una especialización del algoritmo genérico que estamos presentando, ya que, de acuerdo con la estrategia general presentada en genericSort, la lista de elementos está siempre dividida en dos regiones, y se mueve un elemento de la región baja a la región alta en cada iteración. Para seleccionar al elemento a mover, en el método select se barre la región baja, comparando elementos contiguos e intercambiándolos si es que están invertidos en cuanto al orden que deben tener. De esta manera se deja en la última posición de la región baja al elemento con mayor llave de esta región. Para integrar a ese elemento a la región alta, se compara a este elemento con el primero de la región alta, para definir el orden entre ellos, y eliminar la inversión si es que ésta existe.

Figura 2.5: Primeras cuatro iteraciones del ordenamiento de burbuja



Un esquema de la ejecución de las primeras cuatro iteraciones para esta especialización se muestran en la Figura 2.5.

La implementación para esta especialización se encuentra en el Listado 2.4.

Listado 2.4: Especialización para ordenamiento de burbuja

```
1   public class BubbleSort  extends AbstractSort {
2
3     public BubbleSort(Object[] objs, BasicComparator comp) {
4       super(objs, comp);
5     }
6
7     /* Burbujea hacia arriba el elemento mayor en               *
8      * [first..last] a la posición last.                        */
9     protected int select(int first, int last) {
10      for (int j = first; j < last; j++){
11        if (((BasicComparator) comp).gtr(objs.get(j),
12                                          objs.get(j + 1))) {
13          swap(j, j + 1);
14        }
15      }
16      return last;
17    }
18
19    /* Verifica si el nuevo máximo seleccionado, que se           *
20     * encuentra en la posición low, está ordenado con respecto  *
21     *  a la posición low+1.                                      */
22    protected void join(int sel, int low, int high) {
23      if (((BasicComparator) comp).gtr(objs.get(sel),
24                                        objs.get(low + 1))) {
25        swap(sel, low + 1);
26      }
27    }
28  }
```

Debemos demostrar, antes de proseguir, que el barrer una sublista de la manera en que lo hace select en efecto deja al elemento mayor de esa sublista en la última posición de la misma.

Lema 2.11 *Sea $R[1 \ldots k]$ una lista de elementos. Si se invoca al método* select *sobre esta lista, entonces al terminar la ejecución de* select *se cumple*

$$r[i] \leq r[k] \qquad \text{con } 1 \leq i < k$$

Demostración:

Haremos la demostración por inducción sobre k, el tamaño del arreglo.

Base: Para $k = 2$. Si únicamente hay dos elementos en la lista, el ciclo en las líneas (10) a (15) se ejecuta una única vez, con first $= 1$ y last $= 2$. Si $r[1] > r[2]$, estos dos elementos se intercambian, quedando la relación $r[1] \leq r[2]$, lo que demuestra el caso base.

Inducción: Supongamos que para las listas con i elementos, $R[1 \ldots i]$, al terminar la ejecución del ciclo, el elemento en la posición i es tal que

$$r[j] \leq r[i] \qquad \text{con } 1 \leq j < i$$

y veamos qué pasa si le agregamos al arreglo una posición. En las primeras $i-1$ iteraciones, por la hipótesis de inducción, tenemos en la posición i al elemento mayor de $R[1 \ldots i]$. En la siguiente iteración únicamente vamos a comparar al elemento en la posición i con el de la posición $i+1$, y si hay inversión, estos elementos se intercambian. Tenemos entonces,

$$\begin{array}{lll} r[i] \leq r[i+1] & & \text{por la última iteración;} \\ r[j] \leq r[i] & \text{con } 1 \leq j < i & \text{por la hipótesis de inducción.} \\ \therefore\ r[j] \leq r[i+1] & \text{con } 1 \leq j < i+1 & \end{array}$$

❏

Pasamos ahora a establecer la correctez de esta implementación.

Lema 2.12 (Correctez del constructor) *Al terminar de ejecutarse el constructor de la especialización del ordenamiento de burbuja (BubbleSort) se cumplen las postcondiciones* (ASps-1) *y* (ASps-2).

Demostración:

Este constructor únicamente llama al constructor de la superclase, que se encarga de armar la lista con acceso directo. ❏

Lema 2.13 (Correctez de select) *La implementación de select en la especialización para el ordenamiento de burbuja es correcta, esto es, si se cumplen las precondiciones* (Spr-1) *y* (Spr-2), *entonces las postcondiciones* (Sps-1) *a* (Sps-4) *se van a cumplir.*

Demostración:

La postcondición (Sps-1) se cumple, ya que el valor que regresa este método es la última posición de la región alta – línea (16).

Para demostrar que la postcondición Sps-2 también se cumple observemos el código de las líneas (10) a (15) y veremos que si este método mueve elementos en la región baja, lo que hace es intercambiarlos. El último intercambio, si se lleva a cabo, es para el último y el penúltimo elemento de la región baja, por lo que ningún elemento que estuviera en esta región antes de la ejecución de select sale de la región, y ningún elemento nuevo ingresa a ella.

Como no se toca la región alta, ésta permanece exactamente igual que como estaba antes de la invocación de select, cumpliéndose de esta manera la postcondición (Sps-3).

Finalmente, como lo que tenemos al terminar la ejecución de select en la región baja es una permutación de la misma, y en la región alta no se movió nada, la lista resultante es una permutación de la lista antes de la ejecución de select, cumpliéndose de esta manera la postcondición (Sps-4). ❏

Adicionalmente, tenemos el Corolario 2.14 del Lema 2.11:

Corolario 2.14 *Al terminar de ejecutarse el método select(first, last),* r[last] *cumple con*

$$r[j] \leq r[\text{last}] \qquad \textit{con } \text{first} \leq j < \text{last}$$

Pasemos a establecer que la implementación de join en esta especialización también es correcta. Para ello, apoyándonos en el Corolario 2.14 demostraremos primero un lema auxiliar, que nos indica la relación entre los elementos de la región baja y los que van quedando en la región alta.

Lema 2.15 *Si las precondiciones para el método join se cumplen, después de ejecutar el método join en la ℓ-ésima iteración ($i = n - \ell$) tenemos que la siguiente relación se cumple:*

$$r^{(\ell)}[j] \leq r^{(\ell)}[n - \ell + 1] \qquad \textit{con } 1 \leq j < (n - \ell = i) \tag{2.19}$$

Demostración:

Haremos la demostración por inducción sobre ℓ.

Base: La primera vez que join es invocado, $\ell = 1$ e $i = n - 1$, por lo que en la región alta del arreglo tenemos dos elementos. join, en la línea (24) pregunta si $r^{(0)'}[n-1] > r^{(0)'}[n]$, y si es así, los intercambia en la línea (25), de donde sabemos que al terminar su ejecución tendremos $r^{(1)}[n-1] \leq r^{(1)}[n]$. Además, por el Corolario 2.14 que nos dice que antes de ejecutarse join en $r^{(0)'}[n-1]$ tenemos al elemento mayor de la región $R^{(0)'}[1 \ldots n-1]$, tenemos que se cumple (2.19) para $\ell = 1$.

Inducción: Supongamos ahora que se cumple para $k < \ell$ que

$$r^{(k)}[j] \leq r^{(k)}[n - k + 1] \qquad \text{con } 1 \leq j < n - k$$

y demostremos que se cumple para $k = \ell$.

Por el Corolario 2.14, select deja en la posición $i = n - \ell$ a un elemento con el mayor valor de la región baja.

$$r^{(\ell-1)'}[j] \leq r^{(\ell-1)'}[i] \qquad \text{con } 1 \leq j \leq i$$

Por la hipótesis de inducción, sabemos que

$$r^{(\ell-1)'}[n - \ell = i] \leq r^{(\ell-1)'}[n - \ell + 2 = i + 2].$$

Al ejecutarse join tenemos dos posibilidades:

$$r^{(\ell-1)'}[i] > r^{(\ell-1)'}[i+1] \tag{a}$$
$$r^{(\ell-1)'}[i] \leq r^{(\ell-1)'}[i+1] \tag{b}$$

Si tenemos el caso (a), los elementos se intercambian, y tenemos que $r^{(\ell)}[i] \leq r^{(\ell)}[i+1]$; por la hipótesis de inducción, tenemos que

$$r^{(\ell)}[j] \leq r^{(\ell)}[i + 2] \qquad \text{con } 1 \leq j \leq i,$$

dado que el elemento en la posición $i + 1$ después de ejecutarse join, estaba en la posición i, por lo que el Corolario 2.14 se cumple para $r^{(\ell)}[i+2]$. Entonces tenemos que se cumple (2.19).

Si tenemos el caso (b), tenemos:

$$r^{(\ell)}[j] \leq r^{(\ell)}[i] \qquad \text{con } 1 \leq j \leq i \qquad \text{y}$$
$$r^{(\ell)}[i] \leq r^{(\ell)}[i+1].$$

De ambas relaciones, obtenemos

$$r^{(\ell)}[j] \leq r^{(\ell)}[i] \leq r^{(\ell)}[i+1] \qquad \text{con } 1 \leq j \leq i$$

que es más fuerte aún que lo que queríamos demostrar. ❑

Ahora sí pasamos a establecer la correctez del método join. Asociaremos un lema con cada postcondición.

Lema 2.16 (Postcondición Jps-1) *Si las precondiciones* (Sps-1) *a* (Sps-4) *se cumplen antes de ejecutarse join, entonces la postcondición* (Jps-1)*, que dice*

$$perm(R^{(\ell-1)'}, R^{(\ell)})$$

se cumple al terminar de ejecutarse join.

Demostración:

Para ver que la postcondición (Jps-1) se cumple, basta observar que esta implementación de join únicamente intercambia, si es que lo hace, al elemento $r[\text{low}]$ con $r[\text{low}+1]$, por lo que si teníamos una permutación de la lista original antes de ejecutarse join, la seguimos teniendo después. ❑

Lema 2.17 (Postcondición Jps-2) *Si las precondiciones* (Sps-1) *a* (Sps-4) *se cumplen antes de ejecutarse join, entonces la postcondición* (Jps-2)*, que dice*

$$r^{(\ell)}[j] \leq r^{(\ell)}[j+1] \qquad \textit{con } i \leq j \leq n$$

se cumple al terminar de ejecutarse join.

Demostración:

Veamos que la postcondición (Jps-2) también se cumple. Haremos la demostración por inducción sobre ℓ ($i = n - \ell$), el número de veces que join es invocado.

Base: La primera vez que join es invocado, $i = n - 1$, por lo que en la región alta del arreglo tenemos dos elementos. No podemos decir nada de $r^{(0)'}[n]$, pero como join, en la línea (24) pregunta si $r^{(0)'}[n-1] > r^{(0)'}[n]$, y si es así, los intercambia en la línea (25), sabemos que al terminar su ejecución tendremos $r^{(1)}[n-1] \leq r^{(1)}[n]$, por lo que para el caso base se cumple la postcondición.

Inducción: Supongamos que la última vez que join fue invocado (con $\ell - 1$, $1 < \ell \leq n-2$), la región alta del arreglo cumplió con:

$$r^{(\ell-1)}[j] \leq r^{(\ell-1)}[j+1] \qquad \text{con } i+1 \leq j < n.$$

En ℓ-ésima iteración, en la posición i se colocó a un elemento que cumple con

$$r^{(\ell-1)'}[j] \leq r^{(\ell-1)'}[i] \qquad \text{con } 1 \leq j \leq i.$$

Además, con esta relación junto con el Lema (2.15), tenemos que después de ejecutarse join,

$$r^{(\ell)}[j] \leq r^{(\ell)}[i+1] \qquad \text{con } 1 \leq j \leq i,$$

y en particular esto se da para $j = i$. De esto, y dado que join no tocó a los elementos más allá de la posición $i+1$, tenemos que

$$r^{(\ell)}[j] \leq r^{(\ell)}[j+1] \qquad \text{con } i \leq j < n,$$

que es lo que queríamos demostrar. ❏

Lema 2.18 (Postcondición Jps-3) *Si las precondiciones* (Sps-1) *a* (Sps-4) *se cumplen antes de ejecutarse join, entonces la postcondición* (Jps-3), *que dice*

$$\begin{gathered}\left(r^{(\ell)}[j] = r^{(\ell-1)'}[m]\right) \wedge \left(r^{(\ell)}[j'] = r^{(\ell-1)'}[m']\right) \\ \wedge \left(\left(r^{(\ell-1)'}[m] \leq r^{(\ell-1)'}[m']\right) \Longrightarrow \left(r^{(\ell)}[j] \leq r^{(\ell)}[j']\right)\right) \\ \text{con } i \leq j < j' \leq n, \quad i+1 \leq m < m' \leq n\end{gathered}$$

se cumple al terminar la ejecución de join

Demostración:

Dado que join únicamente trabaja sobre los elementos en las posiciones i e $i+1$, intercambiándoles en su caso, es claro que los elementos que estaban en $R^{(\ell-1)'}[i+1\ldots n]$ conservan su posición relativa entre sí, ya que lo único que se movió, si acaso, es el elemento en la posición $i+1$ a la posición i; por esto, también la relación de orden se mantiene entre los elementos de la región alta de la lista. ❏

Habiendo demostrado que el constructor de esta especialización, así como los métodos select y join cumplen con las especificaciones, podemos concluir esta sección con el Corolario 2.19.

Corolario 2.19 *La especialización para el ordenamiento de burbuja es correcta.*

2.7.1. Complejidad para la especialización del ordenamiento de burbuja

Una vez más calcularemos la complejidad de cada uno de los métodos concretados en esta especialización, para insertar posteriormente estos valores en la fórmula para la complejidad del algoritmo genérico.

Implementación del constructor (BubbleSort). Dado que lo único que hace este constructor es invocar al constructor de la superclase, su complejidad se reduce a la de la superclase.

BubbleSort(Object[] objs, Comparator comp) (Líneas (3) a (5) del Listado 2.4).

Invocación al constructor de la super clase:

Número de pasos: $c \cdot n$ en: $O(n)$.

Total para el constructor:

$$f_{constr}(n) = c \cdot n,$$

que está en

$$O(n).$$

Implementación del método select. En esta implementación de select, la región baja se recorre completa en cada invocación, para que el elemento mayor en la región llegue a la última posición de esta región. Mientras se recorre el arreglo, en ocasiones se intercambiará a los elementos del mismo, y en ocasiones esto no se hará.

select(int first, int last) (Líneas 9 a 17 del Listado 2.4).

- Inicialización del ciclo: Número de pasos: 1 en: $O(1)$.
- Número de veces que se lleva a cabo el ciclo: last − first
  - Evaluación de la condición.
    Número de pasos: 1 en: $O(1)$.
  - Intercambio de elementos(*).
    Número de pasos: 3 en: $O(1)$.
- Entregar el valor en la línea (16).
  Número de pasos: 1 en: $O(1)$.

Nota(*): Dado que el intercambio de elementos consecutivos únicamente se va a dar si existe inversión entre ellos, debemos considerar los tres distintos posibles casos para este valor.

Peor caso: Si los datos vienen en orden inverso, el elemento en la posición first de la región baja tendrá que desplazarse hasta la posición last, ejecutando un intercambio por cada par de elementos consecutivos que se examine. Si se tienen last − first + 1 elementos en la lista, el número de intercambios que se tiene que hacer es last − first.

Número de pasos: last − first en: $O(\text{last} - \text{first})$.

Mejor caso: Si los elementos vienen ya ordenados, no hay inversiones entre elementos consecutivos, por lo que no se efectuará ningún intercambio entre elementos consecutivos.

Número de pasos: 0

Caso promedio: Si suponemos que cualquier permutación inicial es igualmente posible para los datos, podemos suponer que en una lista con last − first parejas de elementos consecutivos, aproximadamente a la mitad de ellas habrá que intercambiarlas. De esto, en cada ciclo el número de intercambios que se deberán hacer es:

Número de pasos: $\frac{\text{last} - \text{first}}{2}$ en: $O(\text{last} - \text{first})$.

Total para select:

Peor caso:	Número de pasos:	$2 + 4 \cdot (\mathsf{last} - \mathsf{first})$	en:	$O(n)$.
Mejor caso:	Número de pasos:	$2 + \mathsf{last} - \mathsf{first}$	en:	$O(n)$.
Caso promedio:	Número de pasos:	$2 + \frac{5}{2}(\mathsf{last} - \mathsf{first})$	en:	$O(n)$.

De lo que la complejidad para esta implementación de select queda acotada de la siguiente manera:

$$\mathbf{2} + \mathsf{last} - \mathsf{first} \leq \mathsf{f_{sel}}(\mathsf{last} - \mathsf{first}) \leq \mathbf{2} + \mathbf{4} \cdot (\mathsf{last} - \mathsf{first}),$$

en la clase de complejidad de

$$O(n).$$

Implementación para el método join. En el método join únicamente se va a ejecutar, en su caso, un intercambio, que toma tres operaciones. Los valores quedan como sigue:

join(int sel, int low, int high) (Líneas 22 a 27 del Listado 2.4).

- Comparación en la línea (24):

  Número de pasos: 1 en: $O(1)$.

- Intercambio en la línea (25)(*):

  Número de pasos: 3 en: $O(1)$.

Nota(*): Por las características del algoritmo, este intercambio se puede o no dar; por lo tanto, como en casos anteriores, tenemos un mejor y peor caso. Para el caso promedio, aunque es un poco irreal hablar de que el intercambie se realice sólo para la mitad de los elementos, daremos esto como caso promedio. En el contexto de la ejecución del algoritmo, y a través de varias iteraciones, este valor toma sentido.

Peor caso: Se da cuando cada vez que este método es invocado, se lleva a cabo el intercambio. Si los datos originalmente vienen en orden inverso, ésta va a ser la situación:

Número de pasos: 3 en: $O(1)$.

Mejor caso: Se da cuando no hay necesidad de intercambiar estos dos elementos. Esta situación se presenta cuando los datos vienen ya ordenados.

Número de pasos: 3/2 en: $O(1)$.

Caso promedio: Como ya explicamos, supondremos que la mitad de las veces se tiene que llevar a cabo el intercambio.

Total para join:

Peor caso:	Número de pasos:	4	en:	$O(1)$.
Mejor caso:	Número de pasos:	1	en:	$O(1)$.
Caso promedio:	Número de pasos:	$\frac{5}{2}$	en:	$O(1)$.

De lo anterior, el costo para esta implementación de join queda acotado de la siguiente manera:

$$1 \leq \mathsf{f_{join}}(\mathsf{high} - \mathsf{low}) \leq 4,$$

quedando en la clase de complejidad de

$$O(1).$$

Complejidad para la especialización del ordenamiento de burbuja. Nuevamente vamos a integrar los valores para cada uno de los métodos en la fórmula general.

$$\textbf{Complejidad de genericSort} = \mathsf{f_{constr}}(\mathsf{n}) + \sum_{\ell=1}^{\mathsf{n}-1}\Big(\mathsf{f_{sel}}(\mathsf{i} = \mathsf{n} - \ell) + \mathsf{f_{join}}(\ell = \mathsf{n} - \mathsf{i})\Big).$$

Como la complejidad de select es de orden constante, no nos preocupamos por los parámetros que se le pasen. join es invocado con $k = i$, $\mathsf{low} = i$ y $\mathsf{high} = n$, por lo que sustituyendo estos valores en las fórmulas encontradas para el peor caso, tenemos:

$\mathsf{f_{constr}}(\mathsf{n})$	$= c \cdot n$	Constructor.
$\mathsf{f_{sel}}(\mathsf{i} = \mathsf{n} - \ell)$	$= 2 + 4 \cdot (i - 1)$	select.
$\mathsf{f_{join}}(\ell = \mathsf{n} - \mathsf{i})$	$= 4$	join.

Sustituyendo en la fórmula para el ordenamiento genérico, en el peor caso tenemos:

$$\begin{aligned}
T_{BubbleSort}(n) = & \quad c \cdot n + \sum_{i=1}^{n-1}\Big(4 \cdot (i-1) + 2 + 4\Big) \\
= & \quad c \cdot n + 4 \cdot \sum_{i=1}^{n-2} i + \sum_{i=1}^{n-1} 6 \\
= & \quad c \cdot n + 4 \cdot \frac{n^2 - 3n + 2}{2} + 6n - 6 \\
= & \quad 2n^2 + cn - 2 \\
= & \quad O(n^2). \qquad \textbf{(Peor caso)}
\end{aligned}$$

Para el mejor caso, tenemos la siguiente sustitución

$\mathsf{f_{constr}}(\mathsf{n})$	$= c \cdot n$			Constructor.
$\mathsf{f_{sel}}(\mathsf{i} = \mathsf{n} - \ell)$	$= 2 + i - 1$	$=$	$i + 1$	select.
$\mathsf{f_{join}}(\ell = \mathsf{n} - \mathsf{i})$	$= 1$			join.

Sustituyendo en la fórmula para el ordenamiento genérico, considerando el mejor caso, tenemos:

$$\begin{aligned} T_{BubbleSort}(n) = & \quad c \cdot n + \sum_{i=1}^{n-1}(i+1) \\ = & \quad c \cdot n + \frac{n^2+n}{2} - 1 \\ = & \quad \frac{1}{2}n^2 + \left(c + \frac{1}{2}\right) \cdot n - 1 \\ = & \quad O(n^2). \end{aligned} \qquad \textbf{(Mejor caso)}$$

Para el caso promedio, y suponiendo que cualquier permutación de los datos es igual de probable al empezar el algoritmo, tendremos:

$\mathsf{f_{constr}(n)}$	$= c \cdot n$	Constructor.
$\mathsf{f_{sel}(i = n - \ell)}$	$= 2 + \frac{5}{2}(i-1)$	select.
$\mathsf{f_{join}(\ell = n - i)}$	$= \frac{5}{2}$	join.

Sustituyendo en la fórmula para el ordenamiento genérico, considerando el caso promedio, tenemos:

$$\begin{aligned} T_{BubbleSort}(n) = & \quad c \cdot n + \sum_{i=1}^{n-1}\left(2 + \frac{5}{2}(i-1) + \frac{5}{2}\right) \\ = & \quad c \cdot n + \frac{5}{2}\sum_{i=1}^{n-1}(i-1) + \sum_{i=1}^{n-1}\frac{9}{2} \\ = & \quad c \cdot n + \frac{5}{2}\sum_{i=1}^{n-2} i + \sum_{i=1}^{n-1}\frac{9}{2} \\ = & \quad \frac{5}{4}n^2 + \left(c + \frac{3}{4}\right)n - 2 \\ = & \quad O(n^2). \end{aligned} \qquad \textbf{(Caso promedio)}$$

Es importante notar que el algoritmo usual para el ordenamiento de burbuja no lleva a cabo todas las iteraciones de la estrategia genérica, sino que si en una de las iteraciones ya no hay ningún intercambio, la ejecución termina. En el peor caso, sin embargo, si es que los elementos vienen en orden inverso, la complejidad va a ser exactamente la misma. En el mejor caso tendremos una ejecución de orden lineal, ya que la lista será recorrida únicamente una vez. El caso promedio es mucho más difícil de calcular. En [Knu98b] se presenta un análisis detallado de este algoritmo.

2.8. Especialización para el algoritmo de montículo (*HeapSort*)

El siguiente algoritmo que revisaremos es el algoritmo que usa un montículo (*heap*) binario como estructura de datos. A continuación damos una definición más precisa:

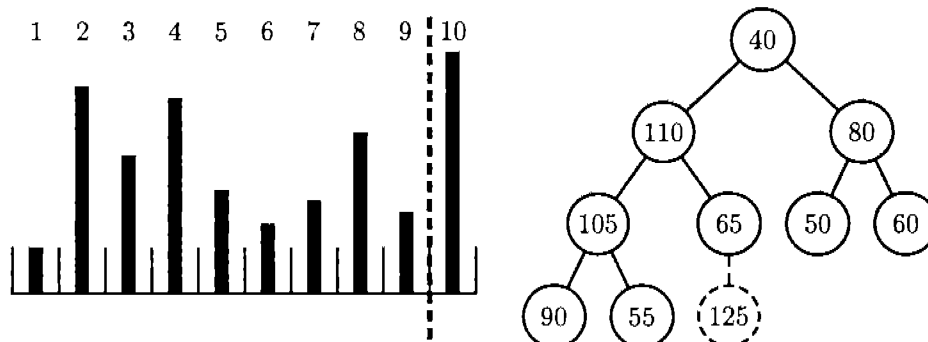
Definición 2.2 Un ***montículo***[3] es un árbol parcialmente ordenado, donde en cada nodo del árbol se encuentra un valor, tomado de un conjunto de valores con un orden total definido entre ellos. Cada subárbol cumple con la condición de que el valor en la raíz es mayor o igual que los valores en cada uno de sus hijos, de acá el concepto de que está parcialmente ordenado[4].

En el caso del ordenamiento por montículo usaremos un heap binario, esto es, un árbol binario parcialmente ordenado que está equilibrado y es casi completo, es decir, se va llenando por niveles de izquierda a derecha. Para representarlo utilizaremos un arreglo con acceso directo, tal como lo presentamos en el Apéndice F.

Este ordenamiento se parece mucho al ordenamiento por selección: en la ℓ-ésima iteración de genericSort se elige al elemento mayor de $R[1 \ldots n - \ell = i]$, y se coloca en la posición i. La diferencia fundamental es que al usarse un heap, el elemento mayor del subarreglo se encuentra en la primera posición del mismo, reduciendo con esto la complejidad del método select.

En las Figuras 2.6 a 2.10 podemos ver el cuadro con las cuatro iteraciones para esta especialización dada para el ordenamiento por montículo.

Figura 2.6: El maxheap obtenido después del constructor



[3]Usaremos el término *heap*, ya que es de uso mucho más extendido.

[4]**Nota:** La definición que acabamos de dar se refiere a lo que se conoce como un *max-heap*. Si la condición para cada subárbol es que el valor en la raíz sea menor o igual que los valores en cada uno de sus hijos, entonces estamos hablando de un *min-heap*. Para un trato más detallado de árboles parcialmente ordenados consultar el Apéndice F.

Figura 2.7: Primera iteración con $i = n - 1$

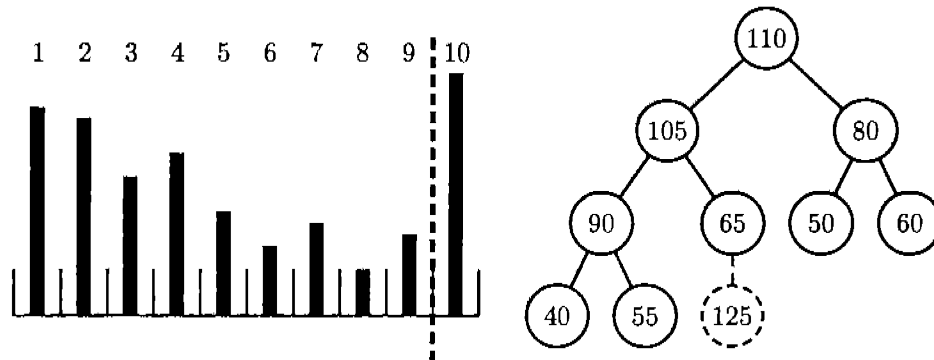


Figura 2.8: Segunda iteración con $i = n - 2$

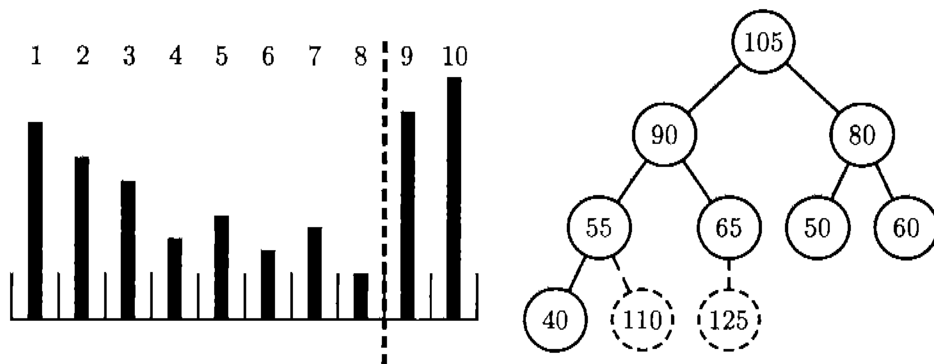


Figura 2.9: Tercera iteración con $i = n - 3$

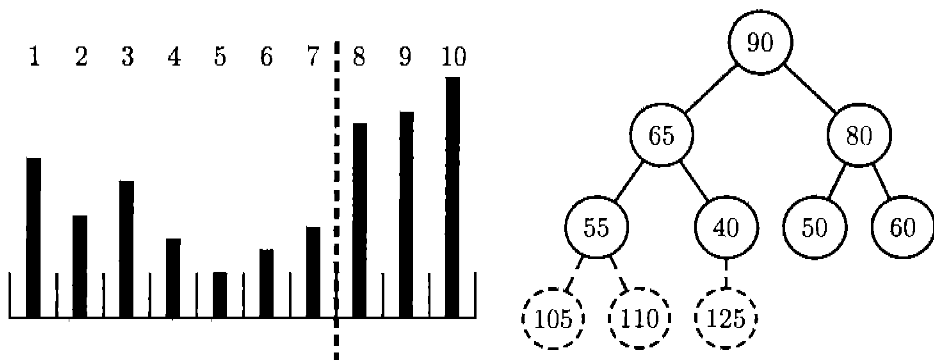
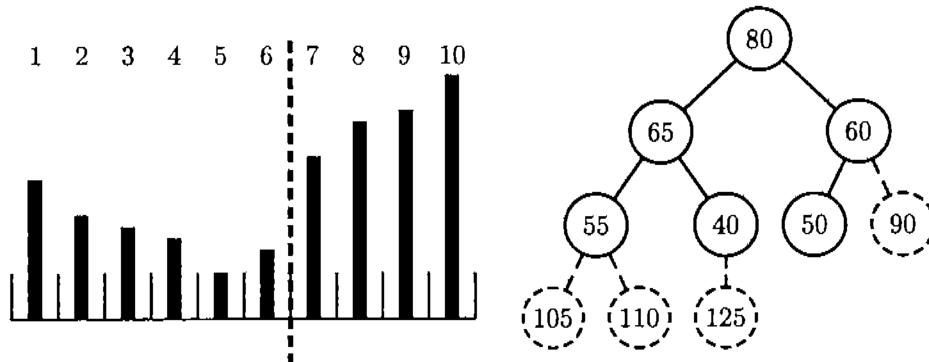


Figura 2.10: Cuarta iteración con $i = n - 4$



La especialización para HeapSort trabaja, como ya lo dijimos, de manera similar al ordenamiento por selección: el método select se encarga de encontrar el máximo en la región baja, organizando a los elementos de esta región en un max-heap, con el elemento con máximo valor ocupando la primera posición. join intercambia al elemento en la primera posición con el elemento en la posición a integrarse a la región alta. De esta manera, cada vez que select es invocado se encuentra con una estructura que cumple la propiedad de max-heap para los subárboles enraizados en la posición 1, pero en esta posición se encuentra un elemento cuya relación con el resto de la región baja no se sabe con certeza, por lo que tiene que restaurar la propiedad de max-heap. El código para esta especialización se encuentra en el Listado 2.5.

Listado 2.5: Especialización para el ordenamiento por montículo 1/2

```
1 public class HeapSort extends AbstractSort {
2      /* Construye la lista de objetos con acceso directo y        *
3       * establece el comparador llamando al constructor de la     *
4       * superclase. A continuación, establece la propiedad de     *
5       * heap en la lista, colocando al elemento mayor en la       *
6       * última posición.                                          */
7     public HeapSort(Object[] objs, BasicComparator comp) {
8         super(objs, comp);
9         buildHeap();
10        swap(1, N);
11    }
12     /* Mueve al elemento con el valor mayor a la raíz del heap.  */
13    protected int select(int first, int last) {
14        heapify(first, last);
15        return 1;
16    }
17     /* Intercambia al primer elemento del heap con el último.    */
18    protected void join(int sel, int low, int high) {
19        swap(sel, low);
20    }
```

Listado 2.5: Especialización para el ordenamiento por montículo 2/2

```
21      /* Verifica la condición de heap del subárbol almacenado en  *
22       * [i..n].                                                    */
23     private void heapify(int i, int n) {
24         int l = 2 * i;            // Referencia a hijo izquierdo
25         int r = 2 * i + 1;        // Referencia a hijo derecho
26         int largest;              // Referencia al de mayor valor
27
28      /* Si existe hijo izquierdo y es mayor, referirlo.            */
29         if (l <= n && comp.gtr(objs.get(l), objs.get(i)))
30             largest = l;
31         else
32             largest = i;
33      /* Si existe hijo derecho y es mayor, referirlo.              */
34         if (r <= n && comp.gtr(objs.get(r), objs.get(largest)))
35             largest = r;
36
37      /* Si el mayor valor no está en la raíz, intercambiarlos      *
38       * y proseguir hacia abajo del árbol.                         */
39         if (i != largest) {
40             swap(i, largest);
41             heapify(largest, n);
42         }
43     }
44      /* Organiza el recorrido del heap desde el último subárbol    *
45       * hacia el primero.                                          */
46     private void buildHeap(int N) {
47         for (int i = ((int) Math.floor(N / 2)); i > 0; i--)
48             heapify(i, N);
49     }
50 }
```

Pasamos ahora a demostrar que tanto el constructor, como los métodos select y join cumplen con las especificaciones dadas por el algoritmo genérico.

Lema 2.20 (Correctez del constructor) *Al terminar de ejecutarse el constructor de la especialización del ordenamiento por montículo (HeapSort) se cumplen las postcondiciones* (ASps-1) *y* (ASps-2).

Demostración:

Este constructor invoca al constructor de la superclase, por lo que al terminarse de ejecutar el constructor de la clase tenemos se cumplen ambas postcondiciones. Es claro que la postcondición (ASps-2) se sigue cumpliendo después de la ejecución de las líneas (9) y (10), ya que estas dos llamadas no modifican el valor de N. Falta corroborar que buildHeap() no provoca que se deje de cumplir la postcondición (ASps-1). Pero eso es claro de los cuerpos de buildHeap y heapify, ya que únicamente llevan a cabo intercambios dentro del arreglo.

Veamos ahora que también select cumple con sus especificaciones.

Lema 2.21 (Correctez de select) *La implementación de select en la especialización para el ordenamiento con montículo es correcta, esto es, si se cumplen las precondiciones* (Spr-1) *y* (Spr-2)*, entonces las postcondiciones* (Sps-1) *a* (Sps-4) *se van a cumplir.*

Demostración:

Es fácil ver que la postcondición (Sps-1) se cumple, ya que select regresa siempre 1 en la línea (15).

La postcondición (Sps-2) también se cumple, pues select, al invocar a heapify en la línea (14), lo único que hace es posibles intercambios entre los elementos de la región baja, preservando de esta manera la propiedad de que al terminar select, lo que tenemos es una permutación del contenido de la región baja antes de la ejecución.

También es claro que la región alta permanece intacta – postcondición (Sps-3) – ya que heapify no la toca.

Por último, dado que se cumplen las postcondiciones (Sps-2) y (Sps-3), al salir de select tenemos una permutación de la lista original – (Sps-4). ❏

Para establecer la correctez de join utilizaremos el Lema 2.3.A.i y las propiedades de un max-heap, presentadas con detalle en el Apéndice F.

Lema 2.22 (Correctez de join) *Si se cumplen las postcondiciones de select* (Sps-1) *a* (Sps-4)*, en cada iteración antes de ejecutarse join, entonces, al terminarse de ejecutar este método se cumplen las postcondiciones* (Jps-1) *a* (Jps-3).

Demostración:

La demostración es exactamente la misma que para el ordenamiento por selección, ya que en el Lema F.1 establecimos que heapify coloca en la posición 1 de la lista al elemento con mayor valor de la región baja. ❏

Pasamos ahora a calcular la complejidad de esta especialización. Para ello utilizaremos los cálculos de complejidad amortizada que hicimos en el Apéndice F para los métodos buildHeap y heapify.

2.8.1. Complejidad para la especialización del ordenamiento por montículo

Para calcular la complejidad de esta especialización, como lo hemos hecho hasta ahora, calcularemos únicamente la complejidad de los tres métodos invocados e insertaremos esta complejidad en la fórmula que tenemos para el algoritmo genérico. Dado que dos de los tres métodos en cuestión utilizan al método heapify, usaremos el cálculo que hacemos en el Apéndice F para buildHeap y heapify e insertar estos valores en la complejidad de esta especialización. Estos cálculos son:

Costo para el método heapify:

$$7 \leq \mathsf{f}_{\mathsf{heapify}}(\mathsf{n}-\mathsf{i}) \leq 9\log_2 n,$$

y por lo tanto en la clase

$$O(\log_2 n).$$

Costo para el método buildHeap:

$$\mathbf{2n} \leq \mathsf{f}_{\mathsf{buildHeap}}(\mathsf{n}) \leq \mathbf{5(n\log_2 n - 1{,}433n)},$$

quedando este método en

$$O(n\log_2 n).$$

Con estas medidas podemos ya analizar la complejidad de los tres métodos concretados en esta especialización.

Implementación del constructor (HeapSort). Esta implementación del constructor, además de llamar al constructor de la superclase, invoca al método **buildHeap**, que a su vez invoca a **heapify** para la mitad de los nodos del árbol, aquellos que son la raíz de un subárbol no trivial.

HeapSort(Object[] objs, Comparator comp): (Líneas (7) a (11) del Listado 2.5).

- Invocación al constructor de la super clase:
  Número de pasos: $c \cdot n$ en: $O(n)$.
- Invocación a buildHeap(1, n)$^{(*)}$.
  Número de pasos: $5(n\log_2 n - 1{,}433n)$ en: $O(n\log_2 n)$.

Nota$^{(*)}$: Copiaremos acá los valores calculados para **buildHeap**, ya que tenemos los tres distintos casos.

Total para el constructor:

Peor caso:	Número de pasos:	$5(n\log_2 n + (c-1{,}433)n)$	en:	$O(n\log_2 n)$.
Mejor caso:	Número de pasos:	$(c+2)n$	en:	$O(n)$.
Caso promedio:	Número de pasos:	$3(n\log_2 n + (c-1{,}433)n)$	en:	$O(n\log_2 n)$.

De lo anterior, la complejidad del constructor queda acotada de la siguiente manera:

$$\mathbf{(c+2)n} \leq \mathsf{f}_{\mathsf{constr}}(\mathsf{n}) \leq \mathbf{5(n\log_2 n + n(c-1{,}433))}$$

y en la clase de complejidad:

$$O(n\log_2 n).$$

Implementación del método select. En esta implementación de select, la región baja se recorre completa en cada invocación, para que el elemento mayor en la región llegue a la última posición de esta región. Mientras se recorre el arreglo, en ocasiones se intercambiará a los elementos del mismo, y en ocasiones esto no se hará.

select(int first, int last) (Líneas 9 a 17 del Listado 2.4).

- Invocación a heapify(*):	Número de pasos:	$9\log_2(\mathsf{last})$	en:	$O(\log_2(\mathsf{last}))$.
Total para select:				
Peor caso:	Número de pasos:	$9\log_2(\mathsf{last})$	en:	$O(\log_2(\mathsf{last}))$.
Mejor caso:	Número de pasos:	7	en:	$O(1)$.
Caso promedio:	Número de pasos:	$\frac{9}{2}\log_2(\mathsf{last})$	en:	$O(\log_2(\mathsf{last}))$.

De lo anterior, el costo para select queda acotado de la siguiente manera:

$$7 \leq \mathsf{f_{sel}}(\mathsf{last} - \mathsf{first}) \leq 9\log_2(\mathsf{last})$$

lo que nos da la clase de complejidad de:

$$O(\log_2(\mathsf{last}))$$

Implementación para el método join. En el método join únicamente se va a ejecutar, un intercambio entre el elemento que ocupa la primera posición de la lista, y el último elemento en el heap, que toma tres operaciones. Los valores quedan como sigue:

join(int sel, int low, int high): (Líneas 22 a 27 del Listado 2.4).

- Intercambio en la línea (19):	Número de pasos:	3	en:	$O(1)$.
Total para join:				
Peor caso:	Número de pasos:	3	en:	$O(1)$.
Mejor caso:	Número de pasos:	3	en:	$O(1)$.
Caso promedio:	Número de pasos:	3	en:	$O(1)$.

De lo anterior, el costo de este método queda como sigue:

$$\mathsf{f_{join}}(\mathsf{high} - \mathsf{low}) = 3$$

en la clase:

$$O(1)$$

Complejidad para la especialización en el ordenamiento por montículo: En esta especialización hay mucha diferencia entre el peor y el mejor caso, aunque el peor caso y el caso promedio se comportan prácticamente igual. Sin embargo, lo que es el mejor caso al construir por primera vez el heap no lo es al invocar a heapify, ya que si los datos vienen en orden inverso, al intercambiar al mayor con el último, tendremos en la raíz el menor elemento, y eso hará que se tenga que recorrer toda la profundidad del árbol para sumergirlo a una

posición correcta. En cambio, si los datos vienen ya ordenados, en cuyo caso el constructor va a estar en su peor caso, las distintas invocaciones a select no van a estar en su peor caso. Veamos cómo se comporta la fórmula general en esta especialización.

$$\textbf{Complejidad de genericSort} = \mathsf{f}_{\mathsf{constr}}(\mathsf{n}) + \sum_{\ell=1}^{\mathsf{n}-1}\Big(\mathsf{f}_{\mathsf{sel}}(\mathsf{i} = \mathsf{n} - \ell) + \mathsf{f}_{\mathsf{join}}(\ell = \mathsf{n} - \mathsf{i})\Big).$$

Como la complejidad de select es de orden constante, no nos preocupamos por los parámetros que se le pasen. join es invocado con $k = i$, low $= i$ y high $= n$, por lo que sustituyendo estos valores en las fórmulas encontradas para el peor caso, tenemos:

$\mathsf{f}_{\mathsf{constr}}(\mathsf{n})$	$=$	$5(n\log_2 n + n(c - 1{,}433))$	Constructor.
$\mathsf{f}_{\mathsf{sel}}(\mathsf{i} = \mathsf{n} - \ell)$	$=$	$9\log_2 i$	select.
$\mathsf{f}_{\mathsf{join}}(\ell = \mathsf{n} - \mathsf{i})$	$=$	3	join.

Para el caso de las sumas acotaremos por arriba con la integral correspondiente. Siendo así podemos obtener cotas superiores para todos los casos.

Sustituyendo en la fórmula para el ordenamiento genérico, en el peor caso tenemos:

$$\begin{aligned}
T_{HeapSort}(n) &= 5\big(n\log_2 n + (c - 1{,}433)n\big) + \sum_{i=1}^{n-1}(9\log_2 i + 3) \\
&= 5\big(n\log_2 n + (c - 1{,}433)n\big) + 9\cdot\sum_{i=1}^{n-1}\log_2 i + \sum_{i=1}^{n-1} 3 \\
&\leq 10\big(n\log_2 n + (c - 1{,}433)n\big) + 3n - 3 \\
&= 10\big(n\log_2 n + (c + 1{,}567)n\big) - 3 \\
&= O(n\log_2 n). \qquad \textbf{(Peor caso)}
\end{aligned}$$

Para el mejor caso, tenemos la siguiente sustitución

$\mathsf{f}_{\mathsf{constr}}(\mathsf{n})$	$=$	$(c + 2)n$	Constructor.
$\mathsf{f}_{\mathsf{sel}}(\mathsf{i} = \mathsf{n} - \ell)$	$=$	7	select.
$\mathsf{f}_{\mathsf{join}}(\ell = \mathsf{n} - \mathsf{i})$	$=$	4	join.

Sustituyendo en la fórmula para el ordenamiento genérico, considerando el mejor caso, tenemos:

$$
\begin{aligned}
T_{HeapSort}(n) = &\quad (c+2)n + \sum_{i=1}^{n-1} 11 \\
= &\quad (c+2)n + 11n - 11 \\
= &\quad (c+13)n - 11 \\
= &\quad O(n). \qquad\qquad \textbf{(Mejor caso)}
\end{aligned}
$$

Para el caso promedio, y suponiendo que cualquier permutación de los datos es igual de probable al empezar el algoritmo, tendremos:

$$
\begin{aligned}
\mathsf{f_{constr}(n)} &= 3(n\log_2 n + n(c-1{,}433)) && \textsf{Constructor.}\\
\mathsf{f_{sel}(i = n - \ell)} &= \frac{9}{2}\log_2 i && \textsf{select.}\\
\mathsf{f_{join}(\ell = n - i)} &= 3 && \textsf{join.}
\end{aligned}
$$

Sustituyendo en la fórmula para el ordenamiento genérico, considerando el caso promedio, tenemos:

$$
\begin{aligned}
T_{HeapSort}(n) = &\quad 3\big(n\log_2 n + (c-1{,}433)n\big) + \sum_{i=1}^{n-1}\Big(\frac{9}{2}\log_2 i + 3\Big) \\
= &\quad 3\big(n\log_2 n + (c-1{,}433)n\big) + \frac{9}{2}\sum_{i=1}^{n-1}\log_2 i + \sum_{i=1}^{n-1} 3 \\
\leq &\quad 6\big(n\log_2 n + (c-1{,}433)n\big) + 3n - 3 \\
\leq &\quad 6\big(n\log_2 n + (c+1{,}567)n\big) - 3 \\
= &\quad O(n\log_2 n). \qquad\qquad \textbf{(Caso promedio)}
\end{aligned}
$$

Creemos necesario comentar que es difícil que se dé el peor caso tanto en el constructor como en el ordenamiento en sí, ya que si el constructor hace muchos intercambios dejará un cierto orden en el heap, por lo que heapify en select ya no tendrá que hacer tantos intercambios. También sucede algo parecido cuando el constructor tiene el mejor caso respecto a select.

2.9. Conclusiones

Los ordenamientos presentados en este capítulo, aunque la mayoría no son de desempeño óptimo, ilustran la herencia y reutilización de las demostraciones de correctez, y el uso de la fórmula general de la complejidad, sustituyendo la complejidad de los métodos concretos por la función dada en la fórmula genérica. Este capítulo constituye una introducción al enfoque presentado en este trabajo.

Al estudiar cuáles algoritmos incluir en esta familia nos percatamos que muchas veces no es obvia la manera en que los distintos algoritmos de ordenamiento se organizan para trabajar. Por ejemplo Quicksort también trabaja a partir de dos regiones, pero no se puede garantizar el orden en una de ellas sino hasta que el algoritmo termina. En general, la selección de los algoritmos que presentaban propiedades comunes en la estrategia genérica, y que estas propiedades nos permitieran demostrar la correctez del ordenamiento, nos obligó a profundizar en las características digamos genéricas de los distintos algoritmos. Incursionamos también en ordenamientos por intercambio, donde la invariante fuera que el número de inversiones se redujera en cada iteración, y nos sorprendimos de ver algunos algoritmos que hubiéramos colocado en esta familia intuitivamente no cumplían con esta simple invariante, como es el caso del ordenamiento por montículo. En resumen, el simple hecho de establecer invariantes en la estrategia genérica proporciona una mejor comprensión de los distintos mecanismos que se presentan en los algoritmos de ordenamiento.

Parte II

Exploración en gráficas

Capítulo 3

Descripción general del problema de exploración

En este capítulo planteamos un algoritmo genérico para una familia de algoritmos que lleva a cabo una exploración en gráficas y gráficas dirigidas. Se hace una implementación básica del algoritmo de exploración, que va a ser extendida y especializada en varios algoritmos conocidos como la búsqueda en amplitud, la búsqueda a profundidad y la determinación de caminos más cortos, entre otros.

Se establecen propiedades de esta implementación básica, como lo son la construcción de un árbol generador de la exploración, y el número de veces que se revisa cada vértice y cada arista de la gráfica. Se obtiene, asimismo, una fórmula general para la complejidad de los algoritmos que identificamos en esta familia.

Aun a nivel de la implementación básica de este algoritmo genérico, tenemos aplicaciones como la de encontrar componentes conexas en la gráfica o determinar si tiene ciclos.

3.1. ¿Qué es un algoritmo de exploración?

Uno de los problemas fundamentales con gráficas son las distintas maneras que tenemos de garantizar que todos los vértices de una gráfica (o que todas las aristas) son revisados, con un cierto objetivo en mente. Esto tiene muchas aplicaciones inmediatas y obvias, como puede ser contar el número de aristas presentes en una gráfica, buscar vértices con alguna propiedad, verificar que en una red de computadoras (Internet) todas las conexiones están funcionando.

Los algoritmos de exploración que vamos a revisar tienen todos un funcionamiento local; esto es, para llevar a cabo la exploración no es necesario tener presente toda la gráfica, sino únicamente, en cada momento, al vértice desde el que se hace la exploración en esa iteración, junto con los vecinos de ese vértice, por lo que estos algoritmos pueden ir recibiendo la gráfica conforme van procesando. Esto es importante en algunas aplicaciones, por ejemplo de inteligencia artificial, donde la gráfica es gigantesca y no es posible guardarla toda en memoria, ya que se va generando conforme se va leyendo. Algunos de los algoritmos, incluso,

se pueden aplicar a gráficas infinitas. Sin embargo, a menos que especifiquemos lo contrario, trabajaremos con gráficas finitas y presentes en memoria al iniciar la ejecución del algoritmo.

Hay muy distintos algoritmos para explorar gráficas, cada uno de ellos con distinta complejidad, y que logra el objetivo dando ciertas garantías y en un orden determinado. Revisaremos varios de ellos, analizando su complejidad y mostrando distintas áreas de aplicación para los mismos. Dependiendo de la forma particular o el orden en que un algoritmo explore una gráfica, descubre cierta estructura de ésta, que resulta tener aplicaciones que van más allá de simplemente explorar la gráfica.

3.1.1. Estructuras de datos para exploración de gráficas

En el contexto de la OO, utilizaremos básicamente listas de incidencia para representar a las gráficas para los algoritmos de exploración. La familia de algoritmos para la exploración conformará una estructura jerárquica arborescente, donde cada algoritmo específico estará representado por una clase. Las estructuras de datos canónicas para representar gráficas serán extendidas para contemplar atributos que permitan la exploración. Algunas de las especializaciones requieren de mayor información, ya sea en los vértices o en los arcos, que la que contiene la clase básica. Para agregar atributos (o métodos que tengan acceso a ellos) tenemos dos opciones: extender las clases para vértices, arcos y gráficas, o bien ir modificando las clases básicas conforme se vaya requiriendo. Para claridad de la ejecución de los algoritmos hemos elegido esta última opción. Por lo tanto, presentaremos únicamente aquellos aspectos relevantes para el nivel de especialización que se esté revisando, omitiendo los aspectos de implementación de estas clases básicas que garantizan la complejidad de orden constante en los accesos a las mismas[1].

3.1.2. Funcionamiento general de un algoritmo de exploración

El algoritmo requiere para trabajar, además de una digráfica $G = (V, E)$ a explorar[2], el vértice inicial a partir del cual se va a hacer la exploración de la gráfica, al cual llamamos s. Este último puede elegirse de distintas maneras, una de ellas, por ejemplo que sea proporcionado por el usuario. La digráfica tiene asociada una función de *peso*, $w : E \to \mathbb{N}$, que asigna a cada arco (arista) un peso con valor real. Vamos a suponer, a menos que indiquemos explícitamente lo contrario, que G es una gráfica dirigida con $|V| = n$ y $|E| = m$. Por el momento, el peso asociado a cada arco es un valor unitario, aunque la exploración será especializada posteriormente a manejar pesos arbitrarios.

En cada estado de la exploración (en cada iteración), se tiene un vértice v_a que es el *centro de acción*, aquél desde el cual se va a realizar un paso de *explorar*, que consiste en *alcanzar* a algún vecino u, recorriendo el arco $\mathsf{v}_a \to \mathsf{u}$. Un vértice u es *descubierto* cuando es

[1] La implementación completa de todos los algoritmos se encuentra en `lambda.fciencias.unam.mx/generic/`, donde se pueden ver los mecanismos usados para garantizar la complejidad en los accesos, inserción y eliminación sobre las distintas estructuras de datos.

[2] V es un conjunto finito de vértices y E es un conjunto finito de arcos, cada arco una pareja ordenada de vértices

alcanzado por primera vez desde v_a, el centro de acción en ese momento. Ningún arco puede ser recorrido más de una vez. Se termina la exploración de un vértice y se considera *explorado* si ya no tiene arcos incidentes desde él (que tengan a ese vértice como origen) que no hayan sido recorridos. Conforme se vaya alcanzando a cada vértice se le coloca en una estructura de datos que llamamos la *frontera* de la exploración, desde la cual se elige al siguiente vértice a ser centro de acción. Un vértice permanece en la frontera en tanto no se haya determinado que todos los arcos incidentes desde él ya fueron utilizados en la exploración.

Definimos a la clase Digrafica, que representa a una digráfica, como base de nuestra jerarquía de gráficas. Una digráfica consiste de nodos (representados en la interfaz Nodo y en la clase más elemental que la implementa, NodoBasico) y arcos (representados por la interfaz Arista y la clase básica que la implementa, AristaBasica). Las estructuras de datos que albergan a las digráficas nos garantizan, como se explica en la página mencionada anteriormente, costos de orden constante en el acceso y modificación de los elementos de la digráfica. Asimismo, para poder llevar a cabo la exploración en la manera en que acabamos de describir, requerimos de los nodos y arcos de la gráfica lo siguiente:

Para la clase de vértices (Nodos):

a. Todo vértice v tiene tres posibles estados: cuando no ha participado en la exploración porque no ha sido alcanzado; cuando ya ha sido alcanzado pero todavía no se termina de procesarlo, por lo que está activo; y cuando ya se ha terminado con su proceso. El nodo pasará del primer estado al segundo, y del segundo al tercero, según sea su situación dentro del proceso de la gráfica. Al inicio del proceso ningún vértice ha sido alcanzado. El estado del nodo se registra como un atributo del mismo, mediante constantes simbólicas de color: BLANCO para cuando el vértice no ha sido descubierto; GRIS para cuando el vértice ya fue descubierto pero no se ha terminado con su proceso; y NEGRO cuando el vértice ya ha sido completamente explorado.
b. Todo vértice $v \in V$ tiene un atributo π (pi), su vértice *padre* que define cuál es el centro de acción cuando es descubierto (esto es, desde cuál vértice fue descubierto). El valor inicial de este tributo, dado que ningún vértice ha sido descubierto, es la constante simbólica NULL, que indica que el vértice no tiene aún padre.
c. Se extiende a los vértices $v \in V$ con un atributo d que contiene la longitud[3] de un camino de s a v. Ese camino está dado, en sentido inverso, por los arcos $v.\pi \rightarrow v$, donde v es, sucesivamente, v, $v.\pi$, $v.\pi.\pi$, $\ldots$, s. Dado que al empezar el algoritmo no se sabe aún si hay camino de s a $v \in V$, ∞ es el valor inicial para d.
d. Cada vértice contendrá atributos de trabajo, como el lugar que ocupa en la estructura de la gráfica, una referencia a su posición en la frontera y otros más que garantizan la clase de eficiencia en los accesos a las distintas estructuras, pero que no se van a exhibir en los códigos ya que no aportan nada para la exposición de los distintos algoritmos.

La implementación para la clase que representa a los vértices se hace sobre una interfaz de Java, Nodos, a la que se da una implementación básica en la clase NodoBasico. No mostramos más que los encabezados de los distintos métodos de la clase, ya que supondremos que satisfacen las propiedades requeridas y el mostrar el código completo no aporta al tema que

[3]Ver conceptos y resultados básicos de teorías de gráficas en el Apéndice C.

nos ocupa. Además, eliminamos del listado todo aquello que tiene que ver con la visualización de los distintos estados por los que pasan los nodos. Como vamos a utilizar estos nodos para su exploración, es conveniente observar los atributos declarados y el estado inicial de los mismos – ver Listado 3.1.

Listado 3.1: Atributos y métodos de la clase NodoBasico

```
1 public class NodoBasico implements Nodos {
2      /* Estado inicial: no descubierto.                              */
3    protected Color color = BLANCO;
4      /* La referencia al padre en el árbol generador.                */
5    protected Nodos pi = null;
6      /* Distancia a la raíz de este nodo. Inicia en infinito.        */
7    protected int d = INFINITY;
8      /* Lista de incidencia de los arcos.                            */
9    protected ListaDeAristas adjList;
10     /* Actualiza la distancia en el vértice.                        */
11   public void setD(int d);
12     /* Obtiene la distancia del vértice.                            */
13   public int getD();
14     /* Establece al padre del nodo en el árbol.                     */
15   public void setPi(Nodos padre);
16     /* Regresa la referencia al padre del nodo.                     */
17   public Nodos getPi();
18     /* Reporta si el nodo tiene más arcos sin usar en su            *
19      * lista de adyacencias.                                        */
20   public boolean masAristas();
21     /* Obtiene un arco de la lista de adyacencias.                  */
22   public Aristas getArista();
23     /* Reporta si el nodo ya ha sido alcanzado.                     */
24   public boolean haSidoAlcanzado();
25     /* Reporta si el nodo ha sido alcanzado y está activo
26      * todavía.                                                     */
27   public boolean estaEnLaFrontera();
28     /* Reporta si el nodo ya fue procesado completamente.           */
29   public boolean yaExplorado();
30     /* Establece el estado del nodo como no alcanzado todavía.      */
31   public void setNoAlcanzado();
32     /* Establece el estado del nodo como alcanzado y activo.        */
33   public void setAlcanzado();
34     /* Establece el estado del nodo como completamente              *
35      * procesado.                                                   */
36   public void setYaExplorado();
37     /* Regresa la lista de incidencia de los arcos (aristas).       */
38   public ListaDeAristas getListaIncidencia();
39     /* Establece a la lista que se le pasa como la lista de         *
40      * incidencia del vértice.                                      */
41   public void setListaIncidencia(ListaDeAristas lista);
42 }
```

Una vez dados los valores por omisión para los vértices y la información que debemos mantener, pasamos a revisar la definición de los arcos.

Para la clase de arcos (Aristas):

a. Toda arista tiene dos vértices asociados. Si se trata de un arco, uno de ellos es el vértice origen y el otro es el vértice destino.
b. Toda arista indica si juega el papel de arista o arco.
c. Toda arista $e = u - v \in E$ tiene un atributo, usada, que determina si e ya fue usada en un paso de exploración desde u o desde v. Este atributo se usa para evitar que ningún arco sea recorrido más de una vez.
d. Toda arista registra el peso asociado a ella.
e. Las aristas, al igual que los vértices, tienen atributos adicionales que ayudan con la eficiencia de los accesos a las distintas estructuras de datos, como el anotar si está considerada en el árbol generador, la posición que guarda dentro de la estructura de aristas de la gráfica, y otros más. Dado que, al igual que con los vértices, esta información no aporta a la estructura general de los algoritmos, la omitiremos en este trabajo.

Como en el caso de los vértices, la implementación se hace definiendo primero una interfaz, Aristas, y después una clase AristaBasica que la implementa. Mostramos esta implementación básica en el Listado 3.2. Al igual que en el caso de los nodos, omitimos aquellos métodos que sirven para mostrar el estado de las aristas, o que tienen que ver con detalles de implementación que no contribuyen a la claridad de los algoritmos que se presentarán.

Podemos observar en el Listado 3.2 los atributos que se declaran en la implementación básica de esta interfaz y los valores por omisión asignados a estos atributos, así como los distintos métodos definidos.

Listado 3.2: Implementación básica para las aristas de la gráfica (AristaBasica) 1/2

```
1  public class AristaBasica implements Aristas {
2        /* Nodo origen del arco.                                    */
3      protected Nodos vDesde;
4        /* Nodo destino del arco.                                   */
5      protected Nodos vHacia;
6        /* Para registrar si el arco ya fue utilizado.              */
7      protected boolean usada = false;
8        /* El peso por omisión es unitario.                         */
9      protected int peso = 1;
10       /* Identifica como arco (true) o arista (false).            */
11     protected boolean dirigida = false;
12
13       /*  Reporta si el arco ya fue usado en la exploración.      */
14     boolean getUsada();
15       /*  Establece al arco como ya usado.                        */
16     void setUsada();
17       /* Regresa el nodo origen de este arco.                     */
18     Nodos getDesde();
```

Listado 3.2: Implementación básica para las aristas de la gráfica (AristaBasica) 2/2

```
19        /* Establece al parámetro como el nodo origen.                    */
20      void setDesde(Nodos desde);
21        /* Regresa el nodo destino de este arco.                          */
22      Nodos getHacia();
23        /* Establece al parámetro como el nodo destino de este            *
24         * arco.                                                          */
25      void setHacia(Nodos hacia);
26        /* Regresa el nodo en el otro extremo de un arco. Si el           *
27         * parámetro es el nodo origen, regresa el destino y              *
28         * viceversa.                                                     */
29      Nodos getOtroNodo(Nodos v);
30        /* Regresa el valor del atributo de peso.                         */
31      int getPeso();
32        /* Establece el valor del atributo de peso.                       */
33      void setPeso(int w);
34        /* Indica si el arco es dirigido o no (una arista).               */
35      boolean esArco();
36        /* Orienta a la arista desde u.                                   */
37      void orientaArco(Nodos u, Nodos v);
38 }
```

Con estas especificaciones para los nodos y los arcos, veamos ahora cómo se especifica la clase básica para las digráficas, que se encuentra en la raíz de la jerarquía de los distintos tipos de gráficas que vamos a usar en este estudio. Esta especificación se puede ver en el Listado 3.3.

Listado 3.3: Métodos y atributos de una gráfica dirigida (*Digrafica*) 1/2

```
 1 public class Digrafica {
 2        /* Estructura lineal dinámica con acceso constante para           *
 3         * los nodos.                                                     */
 4      protected ArrayList nodos;
 5        /* Estructura lineal dinámica con acceso constante para           *
 6         * los arcos.                                                     */
 7      protected ArrayList aristas;
 8        /* Número de nodos en la digráfica.                               */
 9      protected int N;
10        /* Número de arcos en la gráfica.                                 */
11      protected int M;
12        /* Da el número de nodos en la gráfica.                           */
13      public int getN();
14        /* Para registrar el número de veces que un origen se se-         *
15         * lecciona.                                                      */
16      protected int indiceOrigen = 1;
17        /* Regresa el número de arcos en la digráfica.                    */
18      public int getM();
```

Listado 3.3: Métodos y atributos de una gráfica dirigida (*Digrafica*) 2/2

```
19      /* Marca al nodo, cuya posición está dada en el parámetro,   *
20       * como no alcanzado.                                        */
21      public void setNoAlcanzado(int dnde);
22      /* Marca al nodo, cuya posición está dada en el parámetro,   *
23       * como alcanzado.                                           */
24      public void setAlcanzado(int dnde);
25      /* Marca al nodo, cuya posición está dada en el parámetro,   *
26       * como totalmente procesado.                                */
27      public void setYaExplorado(int dnde);
28      /* Regresa la referencia del i-ésimo nodo en la gráfica.     */
29      public NodoBasico getNodo(int i);
30      /* Regresa la referencia del i-ésimo arco en la gráfica.     */
31      public Aristas getArista(int i);
32      /* Marca al iésimo arco como ya usado.                       */
33      public void setUsada(int i);
34      /* Regresa el estado del i-ésimo arco respecto a si ya fue   *
35       * usado o no.                                               */
36      public boolean getUsada(int i);
37      /* Regresa la estructura lineal de acceso directo en la      *
38       * que se almacena a los nodos.                              */
39      public Nodos[] getTodosLosNodos();
40      /* Regresa la estructura lineal de acceso directo en la      *
41       * que se almacena a los arcos.                              */
42      public Aristas[] getTodasLasAristas();
43 }
```

Presentamos únicamente los encabezados de estos métodos, ya que su implementación no es relevante para presentar el algoritmo. Omitimos los constructores de las digráficas y los métodos relacionados con esto, y simplemente garantizamos que la complejidad de construir una digráfica es $O(n+m)$. También omitimos aquellos métodos que muestran el estado de la gráfica, o que se refieren a aspectos de programación que garantizan la complejidad de los accesos, pero que no contribuyen a la claridad del algoritmo.

La frontera: Los vértices que son alcanzados por primera vez – cuando su estado cambia de no alcanzados a alcanzados – son almacenados en una estructura de datos frontera (una pila, cola, conjunto, etc.) que dependiendo del algoritmo particular, definirá el orden en que los vértices en la frontera serán elegidos para ser centro de acción del algoritmo. La manera como se maneje a la frontera es una de las características que distinguen a las distintas especializaciones. Cuando un vértice v es descubierto, se le mete a la frontera. Cuando se termina de explorar, se le saca de la frontera. A la frontera se le denomina de esta manera pues en ella se encuentran aquellos vértices susceptibles de convertirse en centro de acción, desde los cuales se realizan los pasos sucesivos de exploración. La interfaz para la clase que representa a la frontera se muestra en el Listado 3.4. La frontera se implementa tomando como base a una interfaz de Java, que define los métodos que debe contemplar cualquier clase que juegue el papel de frontera. Para cada algoritmo concreto presentaremos una clase que

implemente a esta frontera.

Listado 3.4: Interfaz para la clase que representa a la frontera (*Frontera*)

```
 1 public interface Frontera {
 2       /* Agrega un nodo a la frontera.                          */
 3     void agregaNodo(Nodos v);
 4       /* Elige un elemento de la frontera sin modificar a la    *
 5        * misma.                                                  */
 6     Nodos elige();
 7       /* Elimina a un nodo de la frontera.                      */
 8     void quitaNodo(Nodos v);
 9       /* Consulta a la frontera para ver si existen nodos en    *
10        * ella.                                                   */
11     boolean esNoVacia();
12 }
```

Podemos empezar a armar ya la clase que se va a encargar de instrumentar al algoritmo genérico de exploración, cuando menos en cuanto a los métodos que debe presentar. También en este caso trabajamos a partir de una interfaz de Java, aunque la estrategia genérica estará dada en una clase abstracta, que podemos ver en los listados 3.5, 3.6 y 3.7.

Listado 3.5: Clase abstracta para la exploración genérica (*ExploracionAbstracta*)

```
 1 public abstract class ExploracionAbstracta implements Explorador {
 2       /* Define la política para la selección del siguiente     *
 3        * centro de acción.                                       */
 4     protected Frontera frontera;
 5       /* Número de nodos en la gráfica.                         */
 6     protected int N;
 7       /* Número de arcos en la gráfica.                         */
 8     protected int M;
 9       /* Gráfica (digráfica, gráfica con pesos, etc.) a explorar. */
10     protected Digrafica G;
11       /* Nodos a ser incluidos en el árbol generador.           */
12     protected MiListaLigada nodosAlcanzados;
13       /* Arcos a ser incluidos en el árbol generador.           */
14     protected MiListaLigada aristasUsadas;
15       /* Árbol generador construido por la exploración.         */
16     protected Digrafica arbol;
17       /* Constructor que establece las condiciones iniciales    *
18        * para la exploración.                                    */
19     public ExploracionAbstracta(Digrafica g) {
20         G = g;
21         N = G.getN();
22         M = G.getM();
23         nodosAlcanzados = new MiListaLigada();
24         aristasUsadas = new MiListaLigada();
25     }
```

Cualquier implementación de esta interfaz tiene que aportar los constructores de los algoritmos concretos y, en particular, de uno genérico que lleve a cabo la exploración, pero sin ninguna política particular para seleccionar al vértice que es el centro de acción. Hemos elegido definir una clase abstracta que presente los principales atributos para la exploración y la implementación de aquellos métodos que van a permanecer inalterables en todas las especializaciones. Los atributos de cualquier algoritmo de exploración, así como el constructor básico se encuentran en el Listado 3.5. Es importante notar que todos los atributos tienen acceso protegido (se puede acceder a ellos únicamente desde clases que hereden a ésta).

A nivel de la clase abstracta podemos establecer la estrategia genérica de exploración, plasmada en el método exploracionGenerica. Como la estrategia genérica logrará marcar a todos los nodos accesibles desde el vértice origen, esta estrategia deberá ser capaz de explorar desde diversos orígenes, pasados éstos como argumento. La selección del origen se hará antes de invocar a este método. Si se desea explorar desde más de un origen, se tendrá que invocar a este método con los distintos orígenes deseados. Para posibilitar esto, junto con la estrategia genérica presentamos dos métodos, obtenOrigen y marcaComoOrigen, que tienen que ver con la selección del origen. El método obtenOrigen no se especifica, pues puede involucrar operaciones de entrada y salida que no vienen al caso para nuestros objetivos. Como en otros casos, sin embargo, deberemos garantizar la complejidad constante de este método.

Todo vértice que se elige como origen de una exploración es el primero que entra a la frontera en la ejecución de la estrategia genérica – líneas [28–45] en el Listado 3.6 – por lo que será considerado como primer centro de acción en el ciclo interno – líneas [33–42] en el mismo listado. Todo vértice s elegido como origen de la exploración es, por omisión, alcanzable desde él mismo por un camino de longitud 0. Por ello, se debe indicar esto en el atributo s.d cuando se le instala como origen de la exploración.

Listado 3.6: Método que maneja la exploración genérica

```
26      /* Manejador principal de la exploración. Permanece cons-    *
27       * tante para todas las especializaciones.                   */
28      public final Digrafica exploracionGenerica(Nodos s) {
29          ponComoOrigen(s);
30          nodosAlcanzados.agrega(s);
31          Nodos centroAccion = null;
32          Aristas e = null;
33          while (frontera.esNoVacia()) {
34              centroAccion = frontera.elige();
35              procesaCentroAccion(centroAccion);
36              if (centroAccion.masAristas()) {
37                  e = centroAccion.getArista();
38                  explora(centroAccion, e);
39              } else {
40                  cierraNodo(centroAccion);
41              }
42          }
43          cierraExploracion();
44          return arbol;
45      }
```

De acuerdo a nuestra descripción informal del algoritmo genérico, una vez iniciado el algoritmo mediante el constructor y elegido un vértice origen, debemos proceder a trabajar sobre la frontera de la gráfica, procesando los vértices de la manera que indicamos. Sucesivamente, se colocan en la frontera los vértices que se van descubriendo, y se elige un vértice en la frontera como centro de acción. Cuando se determina que el vértice ya no tiene arcos incidentes desde él que no hayan sido recorridos, se le saca de la frontera y se le marca como completamente procesado. El método exploracionGenerica() únicamente se encarga de manejar el proceso de exploración, sin atender, por lo pronto, otros objetivos que se pudieran tener durante la misma. Sin embargo, dejamos puestos los métodos que permitan marcar o procesar de alguna manera cada uno de las cuatro papeles que presenta un vértice: cuando es descubierto, cuando se le alcanza no por primera vez, cuando es centro de acción y cuando ya no tiene arcos sin usar incidentes desde él. En el Listado 3.7 se encuentran las definiciones básicas de los métodos invocados por exploracionGenerica().

Listado 3.7: Métodos en **ExploracionAbstracta** que son invocados desde **exploracionGenerica** 1/2

```
46       /* Mete al nodo origen a la frontera.                        */
47     protected void ponComoOrigen(Nodos v) {
48         v.setD(0);
49         v.setAlcanzado();
50         frontera.agrega(v);
51     }
52       /* Selecciona a un vértice para la exploración. Por ahora  *
53        * sólo le pide al usuario el origen — no se muestra.      */
54     public Nodos obtenOrigen(Digrafica graf) {
55       /* Se le solicita al usuario el vértice a usar como origen. */
56     }
57       /* El proceso que se realiza sobre un nodo cada vez que es *
58        * centro de acción.                                        */
59     protected abstract void procesaCentroAccion(Nodos centroAccion);
60       /* El proceso que se hace a un nodo cuando sale de la      *
61        * frontera, esto es, cuando se termina de explorar.       */
62     protected abstract void cierraNodo(Nodos centroAccion);
63       /* Ata todos los cabos sueltos del proceso y lo cierra.    */
64     protected void cierraExploracion() {
65         arbol = construyeArbol(nodosAlcanzados, aristasUsadas);
66     }
67       /* Las acciones a realizar en un paso de exploración desde *
68        * el centro de acción.                                     */
69     protected final void explora(Nodos centroAccion, Aristas e) {
70         Nodos u = e.getOtroNodo(centroAccion);
71         procesaArista(e);
72         if (!u.haSidoAlcanzado()) {
73             descubriendo(centroAccion, e, u);
74         } else {
75             yaDescubierto(centroAccion, e, u);
76         }
77     }
```

Listado 3.7: Métodos en ExploracionAbstracta que son invocados desde exploracionGenerica 2/2

```
78     /* Las acciones a realizar cuando se alcanza un nodo no      *
79      * por primera vez.                                          */
80     protected abstract void yaDescubierto(Nodos centroAccion,
81                                           Aristas e, Nodos u);
82     /* Las acciones a realizar cuando se alcanza un nodo por     *
83      * primera vez.                                              */
84     protected abstract void descubriendo(Nodos centroAccion,
85                                          Aristas e, Nodos u);
86     /* Las acciones que se realizan sobre un arco cuando éste    *
87      * es utilizado en un paso de exploración.                   */
88     protected abstract void procesaArista(Aristas e);
89 }
```

En la clase ExploracionBasica definimos el comportamiento mínimo que debe tener un algoritmo de exploración en los métodos que quedaron abstractos a nivel de la clase ExploracionAbstracta. Esta implementación básica se puede ver en el Listado 3.8.

Para efectos de la demostración de correctez de nuestro algoritmo genérico, y tomando en cuenta la implementación que acabamos de dar del mismo, presentamos las siguientes definiciones. En estas definiciones, sea $e \in E$ y $v \in V$.

Definición 3.1 (arco recorrido) Un arco e ha sido *recorrido* $\iff$ ya ha sido usado en un paso de exploración.

Definición 3.2 (vértice explorado) Un vértice v ha sido *explorado* $\iff$ $\forall$e tal que e incide en v, e ya ha sido usada.

Definición 3.3 (vértice no descubierto) Un vértice v *no ha sido descubierto* $\iff$ $v \neq s$ y no ha sido el extremo de un arco que ya fue recorrido.

Definición 3.4 (vértice alcanzado) Un vértice v *ha sido alcanzado (descubierto)* $\iff$ $\exists e = u \rightarrow v \in E$ y $u \in V$ tales que el arco e ya fue recorrido.

Definición 3.5 (vértice alcanzable) Un vértice $v \in V$ es alcanzable desde otro vértice $u \in V$ (no forzosamente $u \neq v$) $\iff$ $\exists$ un camino dirigido $u \rightsquigarrow v$.

Éste es, como dijimos, un algoritmo genérico. El método elige() de la interfaz Frontera hará su tarea dependiendo del algoritmo particular y la estructura de datos en la que esté guardada la frontera. Si bien puede tener complejidad arbitraria, a menos que indiquemos lo contrario supondremos, como ya mencionamos, que su complejidad es $O(1)$. De manera similar, el método explora(v_a, e) de la clase ExploracionAbstracta simplemente procederá a recorrer la arista indicada como argumento, marcándola como usada, y a decidir qué hace con el vértice en el otro extremo. Esta conducta es homogénea en todas las especializaciones, aunque lo que hace con el vértice del otro extremo depende tanto del estado de ese vértice como de la especialización particular dada. Su complejidad depende de la complejidad de los métodos invocados por él. Es necesario subrayar, sin embargo, que la complejidad del

Listado 3.8: Implementación mínima de los métodos abstractos en **ExploracionAbstracta**

```
1 public class ExploracionBasica extends ExploracionAbstracta {
2     /* Constructor que se encarga de decidir qué tipo de          *
3      * frontera usar.                                             */
4    public ExploracionBasica(Digrafica g) {
5       super(g);
6       frontera = new FronteraBasica();
7    }
8     /* Acciones básicas que se realizan cuando un nodo es         *
9      * alcanzado NO por primera vez. En la versión básica, no     *
10     * se hace nada.                                              */
11   protected void yaDescubierto(Nodos centroAccion,
12                                Aristas e, Nodos u) {
13   }
14    /* Acciones básicas que se llevan a cabo cuando un nodo es    *
15     * alcanzado por primera vez. Se actualizan sus atributos     *
16     * y se le ingresa a la frontera.                             */
17   protected void descubriendo(Nodos centroAccion,
18                               Aristas e, Nodos u) {
19      u.setPi(centroAccion);
20      u.setD(centroAccion.getD() + e.getPeso());
21      u.setAlcanzado();
22      frontera.agregaNodo(u);
23      Nodos lu = new NodoBasico(u);
24      nodosAlcanzados.agrega(lu);
25      Aristas le = new AristaBasica(e);
26      le.orientaArista(centroAccion);
27      aristasUsadas.agrega(le);
28   }
29    /* Acciones que se llevan a cabo en relación con el centro    *
30     * de acción. En este nivel de la jerarquía no se hace        *
31     * nada.                                                      */
32   protected void procesaCentroAccion(Nodos centroAccion) {
33   }
34    /* Acciones a realizar sobre el arco usado para la explo-     *
35     * ración. En el caso básico se le marca como usada.          */
36   protected void procesaArista(Aristas e) {
37      e.setUsada();
38   }
39    /* Acciones que se realizan cuando un nodo se termina de      *
40     * explorar: se marca NEGRO y se la saca de la frontera.      */
41   protected void cierraNodo(Nodos centroAccion) {
42      frontera.quitaNodo(centroAccion);
43      centroAccion.setYaExplorado();
44   }
45 }
```

algoritmo particular dependerá, entonces, de la complejidad de estos métodos, que estarán ligados a la estructura específica que se elija para almacenar a la frontera y los criterios, si es que los hay, bajo los cuales se decide cuál arco recorrer en un paso de explorar determinado, y qué tratamiento darle al vértice en el otro extremo.

Ahora procede establecer la correctez del algoritmo genérico y determinar su complejidad. Para ello, aún cuando el algoritmo puede parecer todavía muy vago, podemos determinar que nos da ciertas garantías respecto al proceso de exploración, las *invariantes* del algoritmo, que enunciaremos como lemas, y que utilizaremos posteriormente para establecer la correctez del algoritmo. En cuanto a la complejidad del algoritmo, examinaremos únicamente la relativa al tiempo y consideraremos como pasos elementales a contabilizar los accesos a los elementos de la gráfica (arcos/aristas, vértices), comparaciones entre elementos, asignaciones y preguntar sobre el estado de cualquier atributo o del contenido de un conjunto. Como mencionamos al principio de este capítulo, garantizamos complejidad de $O(1)$ para obtener una arista (o arco) de la gráfica, por lo que esto lo contabilizaremos con costo 1. Para tener una idea clara de nuestra meta, enunciaremos brevemente los puntos relevantes que establecen la correctez del algoritmo, aunque pospondremos su demostración hasta que hayamos terminado de colocar todos los peldaños que nos lleven a ella. Queremos establecer que:

A. El algoritmo siempre termina.

B. Al terminar el algoritmo:

   i. Exploró a todos los vértices alcanzables desde el origen s y recorrió a todas las aristas incidentes a estos vértices.
   ii. La gráfica dada por los arcos $\mathsf{v}.\pi \rightarrow \mathsf{v}$ es un árbol dirigido con raíz en s, G_π.
   iii. $\forall \mathsf{v} \in V$ queda registrada la longitud del camino (o el peso, según el caso) de ese vértice a s, en el árbol dirigido G_π.

C. Su complejidad es $O(|V| + |E|)$.

3.2. Propiedades del algoritmo genérico

Agruparemos las propiedades alrededor de los atributos que hemos establecido para los vértices y los arcos de las gráficas. Primero veremos aquéllas relacionadas con el estado del vértice en la exploración, esto es si ya fue descubierto, si está activo o si ya está totalmente explorado.

Cabe aclarar que estas invariantes del algoritmo no todas se cumplen indiscriminadamente en cualquier punto de la ejecución del algoritmo, ya que puede suceder que en determinado momento en la ejecución alguna de las invariantes no se cumpla. Seremos explícitos cuando esta situación se presente. En general supondremos que las invariantes se refieren a iteraciones completas dentro del método exploracionGenerica.

3.2.1. Invariantes para el estado de exploración

Veamos algunas propiedades relacionadas con el atributo que denota el estado del vértice en cuanto a si ha sido o no procesado. Para denotar a este estado utilizamos directamente el valor del atributo color, que a través de los métodos haSidoAlcanzado(), estaEnLaFrontera() y yaExplorado() informan a la aplicación del valor de este atributo. Asimismo, los métodos relacionados con este atributo se comportan como sigue:

setAlcanzado():	Da el valor de GRIS al atributo color.
setNoAlcanzado():	Da el valor BLANCO al atributo color.
setYaExplorado():	Da el valor NEGRO al atributo color.

Siendo el atributo color de acceso protegido, el valor del atributo únicamente se puede cambiar invocando alguno de estos tres métodos.

Lema 3.1 *Una vez construida* $G = (V, E)$, $\forall$ v $\in V$, v.color $\in$ {BLANCO, GRIS, NEGRO}, *esto es, se encuentra en alguno de los siguientes estados:*

- *No ha sido descubierto (*v.color=BLANCO*).*
- *Se encuentra activo (*v.color=GRIS*).*
- *Ya ha sido totalmente explorado (*v.color=NEGRO*).*

Demostración:

$\forall$ v $\in V$:

i. Al atributo color se le asigna el valor BLANCO como valor por omisión al construirse un nodo – línea [3] en el Listado 3.1 – por lo que desde el momento en que se construye el nodo contiene un valor válido.

ii. Como el atributo color es, como ya dijimos, de acceso protegido, la única manera de cambiar su valor es mediante los métodos que mencionamos arriba, setAlcanzado(), setNoAlcanzado() y setYaExplorado(), y los tres métodos asignan valores en {BLANCO, GRIS, NEGRO}.

iii. Éstos son los únicos métodos en la clase NodoBasico que modifican el atributo color.

Por lo que el atributo inicia con el valor BLANCO y se le asigna GRIS o NEGRO durante la ejecución del algoritmo. ❏

Lema 3.2 $\forall$ **v** $\in V$, **v.color** = GRIS $\iff$ **v** $\in$ **frontera**.

Demostración:

Un vértice es introducido a la frontera cuando se le elige como origen en ponComoOrigen() en el método exploracionGenerica de ExploracionAbstracta (línea [29], Listado 3.6) para el vértice origen s; y en el método descubriendo en la clase ExploracionAbstracta (línea [21] del Listado 3.8) para los vértices que se van descubriendo. Esta acción se ejecuta incondicionalmente después de pintar al vértice de GRIS para s en el método ponComoOrigen y para el resto de los vértices cuando son descubierto – por lo que cuando los vértices entran a la frontera lo hacen pintados de GRIS. Similarmente, estas dos operaciones aparecen una después de la otra, por lo que se pintan de GRIS si y sólo si entran a la frontera. Los otros

puntos donde se le cambia el color a un vértice es en la construcción de los nodos donde se les pinta de BLANCO, pero donde no hay ningún vértice en la frontera al terminar de crear la gráfica, o en el método cierraNodo, línea [43] del Listado 3.8 – que es invocado desde la línea [40] del método exploracionGenerica en el Listado 3.6 – y la acción de pintar al vértice de NEGRO ocurre *inmediata e incondicionalmente después* de sacar al vértice de la frontera (línea [43] de CierraNodo en el Listado 3.8), por lo que todos los vértices que están en la frontera en un momento dado de la iteración están pintados de GRIS. Y como acabamos de mencionar, el sacarlos de la frontera y pintarlos de NEGRO se ejecutan *siempre* ambos y uno después del otro, por lo que ningún vértice que sale de la frontera permanece pintado de GRIS. ❏

Corolario 3.3 $\forall$ **v** $\in V$, **v.color** $\in$ {BLANCO, NEGRO} $\Longleftrightarrow$ **v** $\notin$ **frontera**

Demostración:

Se sigue de los Lemas 3.1 y 3.2. ❏

Lema 3.4 *El color de los vértices empieza en* BLANCO, *cambia de* BLANCO *a* GRIS, *y de* GRIS *a* NEGRO, *en este orden y a lo más una vez.*

Demostración:

i. Como en la construcción de la gráfica todos los vértices inician con el color BLANCO, al empezar el algoritmo todos los vértices tienen este color. La construcción de los vértices es el único punto en el algoritmo en que se pinta a los vértices de este color, por lo que una vez que un nodo cambia de color, nunca vuelve a ser BLANCO.

ii. Los vértices cambian de color BLANCO a color GRIS cuando ingresan a la frontera. Para s esto sucede cuando el nodo es marcado como origen de la exploración cada vez que exploracionGenerica se invocado. Antes de esta inicialización, por la construcción de la gráfica, s estaba pintado de BLANCO.
Cualquier vértice puede cambiar a color GRIS al ser metidos a la frontera, pero esto sucede solamente en el caso de que estuvieran pintados de BLANCO – líneas [72–73] del método explora en la clase ExploracionAbstracta, Listado 3.7, desde donde se invoca al método descubriendo, que es donde se pinta al vértice de GRIS – y por el inciso (a) de este Lema, ningún vértice vuelve a ser pintado de BLANCO después del proceso de construcción y de inicialización. Por lo que un vértice va a pasar de BLANCO a GRIS únicamente una vez y simultáneamente a cuando entra a la frontera. Y por el Lema 3.2, mientras los vértices permanecen en la frontera están pintados de GRIS.

iii. La asignación del color NEGRO a un vértice se hace en la línea [43] del método cierraNodo (Listado 3.8). Este método es invocado desde el método exploracionGenerica (línea [40] en Listado 3.6) y la invocación está condicionada a:

- El vértice que se está pintando de NEGRO es el centro de acción actual y ya no tiene arcos incidentes desde él sin recorrer – líneas [36], [40] del método exploracionGenerica en el Listado 3.6.
- Para que un vértice sea centro de acción debe estar en la frontera, y por el Lema 3.2, está pintado de GRIS. De esto, va a pasar de GRIS a NEGRO.

Que esto sucede únicamente una vez es fácil de corroborar. Inmediatamente antes de pintar al vértice de NEGRO se le saca de la frontera – línea [42] del método cierraNodo en el Listado 3.8 – por lo que ya no se le puede volver a pintar de NEGRO en otra ocasión.

Por otro lado, ya vimos que tampoco se le puede volver a pintar de BLANCO o de GRIS (incisos (a) y (b) de este lema), de donde se le pinta de NEGRO una sola vez. ❑

Lema 3.5 $\forall$ v $\in V$, v *es pintado de* NEGRO $\iff$ v *es centro de acción y está explorado.*

Demostración:

- El vértice v es pintado de NEGRO $\iff$ se ejecuta cierraNodo.
- Se ejecuta cierraNodo $\iff$ la condicional de la línea [36] en el método exploracionGenerica, Listado 3.6, se evalúa a FALSO.
- La condicional se evalúa a FALSO $\iff$ $\nexists$ e = v→x tal que e.usado()=FALSO.
- $\nexists$ e tal que e.usado=FALSO $\iff$ v ya fue explorado (Definición 3.2).

Y por el Lema 3.4, una vez que es pintado de NEGRO ya nunca cambiará su color. ❑

Lema 3.6 *Sea* $S = \{$v $\in V \mid$ v.color = NEGRO$\}$ *al terminar el algoritmo. Entonces* $S \neq \varnothing$.

Demostración:

Por el Lema 3.4 si v.color=NEGRO al terminar el algoritmo, entonces debió pasar por estar pintado de BLANCO a GRIS y de GRIS a NEGRO. Al empezar el algoritmo todo vértice está pintado de BLANCO. Como por el Lema 3.2 un vértice estuvo pintado de GRIS si y sólo si estuvo en la frontera, debemos demostrar que al menos un vértice es introducido y sacado de la frontera durante la ejecución del algoritmo:

- El vértice s es introducido a la frontera cuando se invoca a exploracionGenerica y desde él, en la línea [29] al método ponComoOrigen – línea [50] del Listado 3.7 – por lo que la primera vez que la ejecución llega a la línea [33] del método exploracionGenerica en el Listado 3.6, entra a la iteración.
- Si el algoritmo termina, es porque la condición frontera.esNoVacia() de la línea [33] en el método exploracionGenerica se deja de cumplir, esto es la frontera se vacía. Y como al menos está s en la frontera al empezar, en algún momento se tuvo que sacar a s de frontera, y en ese momento se le pintó de NEGRO.

Por lo que S contiene al menos al vértice origen s. ❑

Lema 3.7 *Durante la ejecución del algoritmo,* v.color=BLANCO $\iff$ *no ha sido descubierto.*

Demostración:

$\Longrightarrow$ Supongamos que v.color=BLANCO. Por el Lema 3.4, el vértice v no ha cambiado de color. Por contradicción, supongamos que ya fue descubierto;esto quiere decir que jugó el papel del vértice u en las líneas [71–76] del método explora (Listado 3.7). Pero entonces, se le habría pintado de GRIS, lo que constituye una contradicción a que el vértice nunca cambió de color.

$\Longleftarrow$ Supongamos ahora que el vértice v no ha sido descubierto. Únicamente hay dos momentos en que el color de un vértice cambia, al ingresar a la frontera o al salir de ella. Para ingresar a la frontera, tiene que ser descubierto; para salir de la frontera tiene que haber entrado a ella; por lo tanto, si no ha sido descubierto conserva el color que se le dio en su constructor, que es el color BLANCO. ❑

3.2.2. Invariantes para frontera

Revisemos algunas propiedades relacionadas con el funcionamiento de la frontera, ya que será más claro establecer la correctez y complejidad del algoritmo. Pero antes estableceremos dos propiedades que relacionan a los vértices con los arcos.

Lema 3.8 *Cada arco es recorrido a lo más una vez.*

Demostración:

Para que un arco sea recorrido, se requiere que e.getUsada()=FALSO (método masAristas() en el Listado 3.1 de la clase NodoBasico) y entonces es seleccionado para ser recorrido. Cuando un arco es seleccionado (línea [37] del método exploracionGenerica en el Listado 3.6) inmediatamente al entrar al método explora, en la línea [71] del Listado 3.7, se procesa el arco (método procesaArista) y se le marca como ya recorrido (línea [37] del método procesaArista en el Listado 3.8). Y como, excepto por el método constructor del arco, en ningún otro paso se marca al arco como no recorrido, si es recorrido una vez, ya nunca más volverá a ser recorrido. ❑

Lema 3.9 *Sea $k = |\text{vértices negros}| + |\text{arcos recorridos}|$. Entonces, en cada iteración dentro de exploracionGenerica durante la ejecución del algoritmo la k se incrementa exactamente en 1.*

Demostración:

En cada iteración del ciclo en el método exploracionGenerica – líneas [33–42] en el Listado 3.6 – se usa un arco que no ha sido utilizado previamente y se le marca como usado – – línea [73] de explora – o bien se cierra un nodo en la línea [40] de exploracionGenerica, marcándole como explorado. En cualquiera de estos dos casos, la k se incrementa en 1. ❑

Lema 3.10 *Cada vértice es insertado en la frontera a lo más una vez.*

Demostración:

Al meter a un vértice a la frontera:

i. Siempre inmediatamente antes se le marca como descubierto (se le da el valor GRIS al atributo color):
- Para v = s, en la línea [49] de ponComoOrigen (Listado 3.7).
- Para v ≠ s, en las líneas [21] y [22] de descubriendo (Listado 3.8).

ii. Antes de meter al vértice v en la frontera siempre se verifica que no haya sido descubierto todavía (que su atributo color no tenga el valor distinto de BLANCO):
- Si v ≠ s, línea [72] de explora (Listado 3.7).
- Si v = s, al definir el origen de la exploración de la gráfica, en ponComoOrigen, que se ejecuta una sola vez por cada vez que se invoca a exploracionGenerica.

De donde una vez que ya se metió a un vértice a la frontera, y su atributo color perdió el valor BLANCO, nunca más se le va a considerar para volverle a meter. ❑

Lema 3.11 *Todo vértice que es introducido a la frontera es eventualmente sacado de ella.*

Demostración:

i. Por el Lema 3.10, todo vértice v es introducido a la frontera a lo más una vez.

ii. El valor de $exgrado$(v) es finito.

iii. El número de veces que la condición en la línea [36] del método exploracionGenerica se va a evaluar a verdadera para el vértice v es finito y es exactamente $exgrado$(v).

∴ eventualmente esta condición se evalúa a falso y se ejecuta la línea [40] del método exploracionGenerica, donde se saca al vértice v de la frontera. ❑

Queremos garantizar que se explora a toda la subgráfica susceptible de ser explorada.

Lema 3.12 *Al terminar la ejecución exploracionGenerica todos los vértices alcanzables desde* s *han sido alcanzados y todos los arcos incidentes desde y hacia esos vértices han sido recorridos.*

Demostración:

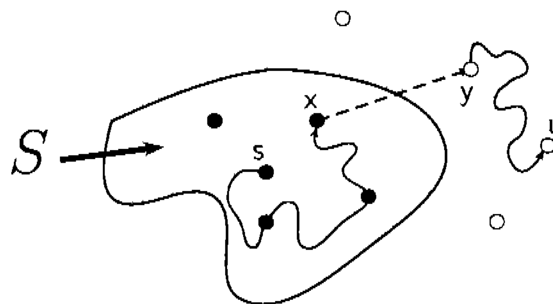
Supongamos por contradicción que la ejecución de exploracionGenerica terminó pero que dejó al menos un vértice alcanzable desde s sin explorar.

a. Para que termine la ejecución de exploracionGenerica, el ciclo de las líneas [33–42] debió haber terminado, y por lo tanto la frontera debió vaciarse.

b. Por el Lema 3.2, tendremos únicamente vértices no descubiertos (v.color=BLANCO) y vértices explorados (v.color=NEGRO).

Sea $S = \{\mathsf{v} \in V \mid \mathsf{v.color} = \text{NEGRO}\}$. Tenemos la situación que se muestra en la Figura 3.1. Queremos establecer que $S = \{\mathsf{v} \in V \mid \mathsf{s} \rightsquigarrow \mathsf{v} \text{ en } G\}$. Por el Lema 3.6, $S \neq \varnothing$. Sea u un vértice alcanzable desde s que quedó sin explorar y que por los Lemas 3.3 y 3.5 u.color= BLANCO. Como u es alcanzable desde s, existe un camino entre s y u. Sea $P = \mathsf{s} \rightsquigarrow \mathsf{u}$; sea x el último vértice ya explorado que toca P y sea y el primer vértice no explorado que toca este camino (y pudiera ser u y x pudiera ser s). Es obvio que también y es alcanzable desde s por el camino $P = \mathsf{s} \rightsquigarrow \mathsf{x} \rightarrow y \rightsquigarrow u$.

Figura 3.1: Suponiendo que al finalizar el algoritmo quedan vértices sin explorar



i. Como y no fue descubierto, quiere decir que el arco e=x → y no fue recorrido (Definición 3.3).

ii. Si quedó un arco e con e.getUsada()=FALSO tal que e.getDesde()=x, el algoritmo no pudo sacar a x de la frontera (línea [36] de exploracionGenerica), por lo que no pudo darlo por explorado sin alcanzar a y.

iii. Si alcanzó a y, éste fue introducido a la frontera y pintado de GRIS (método descubriendo).

iv. Por el Lema 3.11 y fue sacado de la frontera.

v. Por los Lemas 3.3 y 3.4, y fue pintado de NEGRO.

vi. Por el Lema 3.5, y tomó el estado de explorado (fue pintado de NEGRO).

Por lo que si y es alcanzable desde s, y no puede quedar sin descubrir (pintado de BLANCO), y por lo tanto, sin explorar, contradiciendo el que al finalizar el algoritmo quede pintado de BLANCO. ❏

Pasamos a establecer las propiedades de los atributos π y d.

3.2.3. El atributo π

El atributo v.π registra al vértice desde el cuál un vértice v fue descubierto. En el teorema de correctez del algoritmo genérico dijimos que este atributo es el que representa al árbol generador resultado de la exploración y presenta varias propiedades que vamos a revisar.

Definición 3.6 Definimos a la *digráfica de padres* $G_\pi = (V_\pi,\ E_\pi)$ de la siguiente manera:

$$\begin{aligned} V_\pi = &\ \{\mathsf{v} \in V \mid \mathsf{v.color} = \text{NEGRO}\} \\ E_\pi = &\ \{\mathsf{v}.\pi \rightarrow \mathsf{v} \mid \mathsf{v}, \mathsf{v}.\pi \in V_\pi \text{ y } \mathsf{v}.\pi \rightarrow \mathsf{v} \in E\} \end{aligned}$$

La digráfica de padres G_π corresponde a la subgráfica que resulta de tomar a todos los vértices alcanzables desde s y las aristas que van del padre de un vértice al vértice, donde el padre de un vértice es el vértice desde el cuál se le descubrió. Hay propiedades interesantes del algoritmo de exploración que se refieren a la gráfica de padres G_π.

Lema 3.13 *Al terminar el algoritmo para todo* v $\neq$ s *en* V_π, v.$\pi \neq$ NULL; s.π=NULL.

Demostración:

- Por el Teorema 3.18 inciso *B.b* al terminar el algoritmo todo vértice v $\in V$ alcanzable desde s fue explorado y por el Lema 3.5 está pintado de NEGRO.
- Pero para que fuera pintado de NEGRO tuvo que haber estado pintado de GRIS (Lema 3.4) una única vez.
- A todos los vértices alcanzables desde s, menos a s, se le pintó de GRIS al introducirlo a la frontera en el método descubriendo, y en ese momento también se le colocó un valor en v.π (líneas [19] y [21] en el Listado 3.8.
- Éste es el único enunciado del algoritmo en el que se modifica al atributo v.π.

∴ a todo vértice v que es introducido a la frontera en la línea [19] de descubriendo se le colocó un valor distinto de NULL en v.π.

El caso del vértice s es distinto, ya que s no es metido a la frontera en el método descubriendo, sino al entrar a la ejecución de exploracionGenerica.

- Como antes de entrar al ciclo, desde donde se invoca a descubriendo, el vértice ya ingresó a la frontera y está pintado de GRIS, nunca se cumple para él la condición de la línea [72] del método explora.
- De esto, no se le coloca valor alguno en s.π y queda con el valor NULL que se les colocó a todos los vértices, incluyendo a s, en la construcción de cada nodo, y por ende en la construcción de la gráfica.
- Además de este punto del algoritmo, donde se modifica el valor de v.π, el único otro punto es en la construcción de cada uno de los nodos de la gráfica.

$\therefore$ al terminar el algoritmo s.π=NULL y v.$\pi \neq$ NULL para s $\neq$ v y v$\in V_\pi$. ❑

Se debe garantizar que la pareja ordenada (v.π,v) corresponde a un arco de G, lo que estableceremos en el siguiente lema:

Lema 3.14 $\forall$ v $\in V$, v *es descubierto después que el vértice indicado en* v.π; *más aún,* v *es descubierto desde* v.π *al recorrer el arco* v.$\pi \rightarrow$v.

Demostración:

El atributo π se asigna en la línea [19] del método descubriendo – Listado 3.8. Observemos lo siguiente:

i. Entre los argumentos de este método están el vértice de acción v_a y el arco e por explorar.
ii. Al entrar a explora se obtiene el vértice destino u del arco e (línea [70], Listado 3.7), donde e=$\mathsf{v}_a \rightarrow$u.
iii. u.π toma el valor del vértice de acción v_a.
iv. u se está descubriendo en ese momento, mientras que v_a=u.π ya fue descubierto antes de entrar en esta ocasión a explora.

$\therefore$ $\forall$ v $\in V$ tal que v.$\pi \neq$ NULL, el vértice denotado por v.π fue descubierto antes que el vértice v al recorrer el arco e=v.$\pi \rightarrow$v ❑

Estableceremos ahora una propiedad muy importante que caracteriza a la gráfica de padres como un árbol.

Lema 3.15 *La gráfica* G_π *definida por* $V_\pi \subseteq V$ *y* $E_\pi = \{$v.$\pi \rightarrow$v$\}$ *forma un árbol con raíz en* s.

Demostración:

Utilicemos la siguiente caracterización de un árbol: Una gráfica $G = (V, E)$ es un árbol con raíz en s si para todo v $\in V$, v es alcanzable desde s y existe un único camino de s a v.

i. Que v es alcanzable desde s se sigue de la definición de V_π.
ii. Vamos a establecer que existe un único camino entre s y v.
Por el Lema 3.13, $\forall$v $\in V_\pi$ con v $\neq$ s, v.$\pi \neq$ NULL. Entonces, tomamos al vértice v y el arco v.$\pi \rightarrow$v, y lo hacemos el último en nuestro camino. A continuación tomamos el arco v.π.$\pi \rightarrow$v.π y lo anteponemos al que ya tenemos y así sucesivamente. Obtenemos un camino dado por

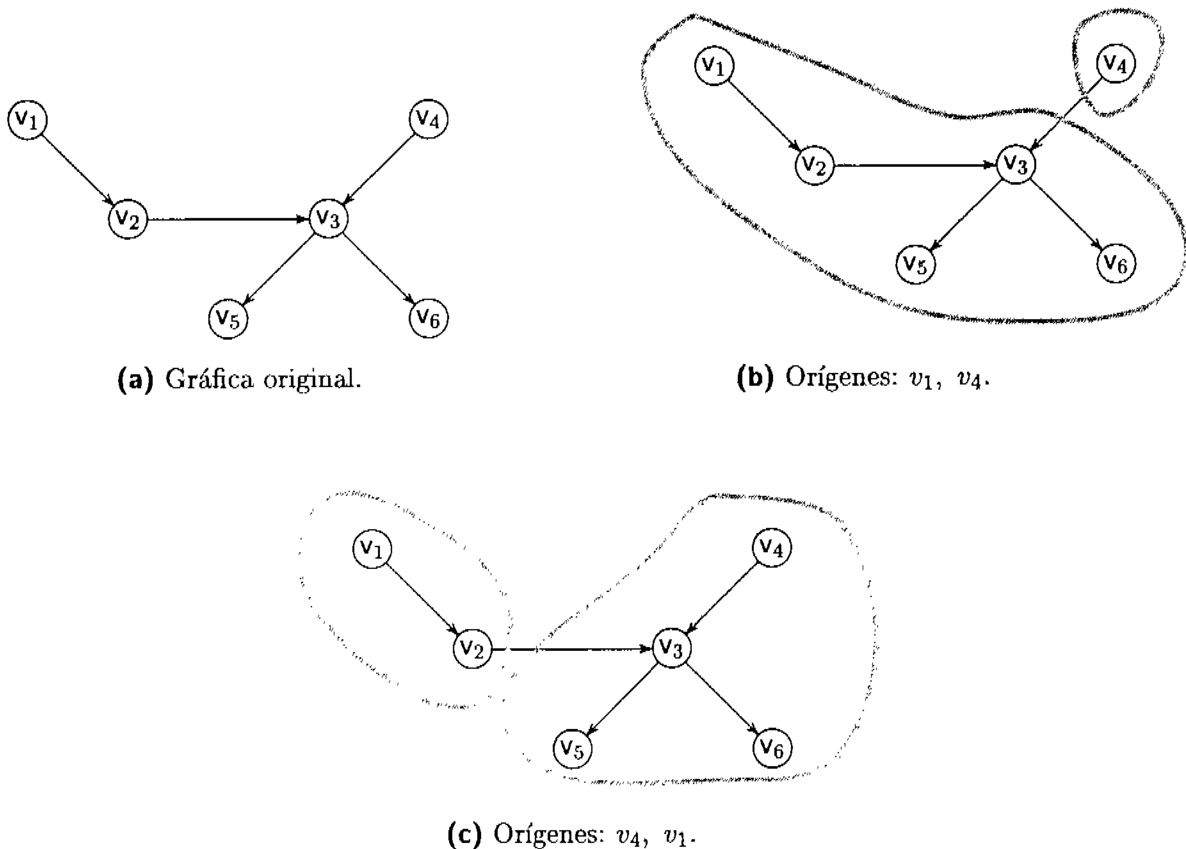
$$\mathsf{v}.\pi.\pi\ldots.\pi \rightarrow \ldots \rightarrow \mathsf{v}.\pi.\pi \rightarrow \mathsf{v}.\pi \rightarrow \mathsf{v}$$

Este camino es un camino simple, ya que cada padre fue descubierto antes que su hijo (Lema 3.14) y una única vez, y como G es finita tiene que terminar y sólo puede terminar

en s, ya que es el único que no tiene padre. Por lo que el camino es un camino s⇝v, y esto para cualquier v$\in V$. El camino es único porque a cada vértice v$\in V_\pi$ llega a lo más un arco, el que sale del vértice anotado en v.π, y cada vértice tiene un atributo único v.π. ❑

Cuando invocamos a exploracionGenerica con orígenes sucesivos de tal manera que explora a toda la digráfica, la selección de los vértices origen determina la estructura de las distintas gráficas G_π que se generan con cada origen seleccionado. Veamos como ejemplo la digráfica de la Figura 3.2.

Figura 3.2: Exploración desde distintos orígenes



(a) Gráfica original.

(b) Orígenes: v_1, v_4.

(c) Orígenes: v_4, v_1.

Como se ve en estas figuras, el orden en que se tomen los orígenes puede cambiar completamente la estructura de los árboles generadores resultantes. También, aunque no se ejemplifica, el número de árboles también puede cambiar mucho.

Si la digráfica es fuertemente conexa, todo vértice es alcanzable desde cualquier otro, por lo que presenta la siguiente propiedad.

Lema 3.16 *Si $G = (V, E)$ es una digráfica fuertemente conexa, el algoritmo genérico de exploración recorrerá todos los arcos de la digráfica y explorará todos los vértices, sin importar el vértice origen de la exploración.*

Demostración:

Se sigue del Lema 3.12. ❑

3.2.4. El atributo de distancias d

Establecimos en la sección anterior que la exploración de la gráfica determina caminos entre el origen de la exploración y los vértices alcanzables desde éste. En esta sección hablaremos un poco sobre las características de estos caminos, y la relación que existe entre la distancia desde s a cada uno de los vértices que se descubrió, y el atributo d de los vértices. Recordemos la definición de distancia:

Definición 3.7 (distancia) Sea $\delta(\mathsf{s},\mathsf{v})$ el mínimo número de arcos de un camino dirigido de s a v, o ∞ si no hay camino entre s y v. Nos referimos a $\delta(\mathsf{s},\mathsf{v})$ como la *distancia* de s a v.

De la Definición 3.7 es claro que $\delta(\mathsf{v},\mathsf{v}) = 0$.

Definición 3.8 (camino más corto) Un camino de s a v que tiene un número de arcos igual a $\delta(\mathsf{s},\mathsf{v})$ es un *camino más corto*. Puede haber más de un camino más corto entre cualesquiera dos vértices, pues puede ser que más de un camino tenga longitud $\delta(\mathsf{s},\mathsf{v})$.

En el Apéndice C damos algunas propiedades y definiciones básicas que tienen que ver con distancias y caminos más cortos.

Invariantes para el atributo d

Es indudable que para todo v$\in V$, el camino determinado por G_π es un camino de s a v. La manera como se calcula el atributo d al descubrir un vértice, nos da la longitud de *un* camino, por lo que podemos establecer que v.d es una cota superior para $\delta(\mathsf{s},\mathsf{v})$:

Lema 3.17 *Sea $G = (V, E)$ una digráfica sobre la que se ha ejecutado* exploracionGenerica, *con origen* $\mathsf{s} \in V$*. Entonces, es una invariante de este método, y por lo tanto del algoritmo, que*

$$\delta(\mathsf{s},\mathsf{v}) \leq \mathsf{v.d}, \qquad \forall\, \mathsf{v} \in V.$$

Demostración:

En el constructor de la clase se asigna el valor ∞ a v.d, $\forall \mathsf{v} \in V$ – construcción de los nodos, línea [7] en el Listado 3.1 – por lo que, claramente, $\delta(\mathsf{s},\mathsf{v}) \leq \mathsf{v.d} = \infty$, $\forall \mathsf{v} \in V$ al construirse la digráfica.

Esta estimación cambia cuando el vértice ingresa a la frontera. El algoritmo nos garantiza que cada vértice $\mathsf{v} \in V$ ingresa a la frontera a lo más una vez (Lema 3.10); además, la modificación del cálculo de la estimación v.d (la asignación a este atributo) se lleva a cabo únicamente en dos lugares más del algoritmo:

i. Para el vértice s en la línea [29] del Listado 3.6, al ponerlo como origen.

ii. En el método descubriendo – línea [20] del Listado 3.8 – que se ejecuta sólo cuando el vértice es descubierto (alcanzado por primera vez), inmediatamente antes e incondicionalmente cuando se le mete a la frontera, que por el Lema 3.10 sucede a lo más una vez.

Basta demostrar, entonces, que en todos estos lugares del algoritmo se sigue cumpliendo la invariante. Para ello nótese que tenemos dos tipos de vértices, aquellos que son alcanzables desde s y aquellos que no lo son.

Sea v=s. El único punto en el que se ingresa a s en la frontera es en el método exploracionGenerica al invocar a ponComoOrigen, inmediatamente después de haber asignado el valor 0 a s.d – línea [48] del Listado 3.7. Como s no vuelve a ser ingresado a la frontera, no se le vuelve a asignar valor al atributo s.d durante la ejecución del algoritmo, y el valor con el que se queda es 0. Dado que $\delta(\mathsf{v},\mathsf{v}) = 0\ \forall \mathsf{v} \in V$, tenemos la igualdad

$$\delta(\mathsf{s},\mathsf{s}) = 0 = \mathsf{s.d}$$

y se cumple la invariante.

Sea $\mathsf{v} \neq \mathsf{s}$ alcanzable desde s. Sabemos que todos los vértices alcanzables desde s van a ingresar a la frontera exactamente una vez, por lo que se les asignará una estimación de v.d en el método descubriendo, y esta estimación está dada por

$$\mathsf{v.d} \ \leftarrow\ \mathsf{v}.\pi.\mathsf{d} + \mathsf{e.peso}, \text{ con } \mathsf{e.peso} = 1$$

Sea u= $\mathsf{v}.\pi$. v es descubierto desde u, por lo que el camino $\mathsf{s} \rightsquigarrow \mathsf{u} \rightarrow \mathsf{v}$ tiene longitud mayor o igual a $\delta(\mathsf{s},\mathsf{v})$ y por el Lema C.1 se cumple la invariante

$$\delta(\mathsf{s},\mathsf{v}) \leq \mathsf{u.d} + 1 = \mathsf{v}.\pi.\mathsf{d} + 1 = \mathsf{v.d}.$$

Sea $\mathsf{v} \neq \mathsf{s}$ no alcanzable desde s . Por el Teorema 3.18, y los Lemas 3.2, 3.4 y 3.5, si v no es alcanzable desde s nunca ingresa a la frontera, por lo que la única asignación al atributo v.d que se lleva cabo es en el constructor de la gráfica – con la construcción de cada nodo – donde se le asigna el valor ∞. Con esto se cumple la invariante, presentándose la igualdad (por la definición de distancia):

$$\delta(\mathsf{s},\mathsf{v}) = \infty = \mathsf{v.d}.$$

$\therefore$ la invariante se cumple $\forall \mathsf{v} \in V$. ❏

3.3. Correctez del algoritmo genérico

Ya que tenemos bien marcadas las invariantes del algoritmo, apoyados en ellas procedemos a establecer la correctez y complejidad del algoritmo.

Un algoritmo es *correcto* si:

- Siempre termina.
- Frente a cualquier instancia del problema, da la respuesta correcta.

En esta sección demostraremos que el algoritmo genérico que presentamos en secciones anteriores cumple con estas dos propiedades. Y si bien es claro el significado de la primera propiedad, la segunda habla de intención y objetivos del algoritmo, que posiblemente haya necesidad de enunciar de manera precisa.

Hablamos al principio de este capítulo de que el objetivo general que nos planteamos es el de explorar una gráfica desde un un vértice origen s. Cada algoritmo específico de exploración sigue una estrategia particular para hacerlo, que queda plasmada en el manejo de la frontera del algoritmo (el tipo particular de estructura de datos que se utilice para almacenarla). El algoritmo genérico lo que debe garantizar, sin embargo, es que sujeto a esta estrategia, se van a explorar todos los vértices alcanzables y recorrer todos los arcos cuyo origen sea alguno de estos vértices, independientemente del orden que dicte el algoritmo particular. Pasemos a establecer la correctez del algoritmo.

Teorema 3.18 *El algoritmo genérico de exploración dado por el método* exploracionGenerica *de la clase* ExploracionAbstracta, *dada una gráfica dirigida $G = (V, E)$:*

A. Siempre termina.
B. Al terminar el algoritmo:
  i. Exploró a todos los vértices alcanzables desde el origen s *y*
  ii. recorrió a todas los arcos incidentes a estos vértices.
  iii. La gráfica G_π dada por los arcos v.π→v *es un árbol dirigido con raíz en* s.
  iv. ∀v ∈ *V en el atributo* d *queda registrada una cota superior para la distancia del vértice a* s.
C. Su complejidad es $O(|V| + |E|)$.

Demostración:

A. Dado que los primeros cuatro enunciados del método exploracionGenerica son únicamente llamadas a otros métodos y asignaciones, para garantizar que el algoritmo llega al ciclo de la línea [33] en el Listado 3.6, debemos exigir de estos dos métodos se ejecuten bien y terminen. Eso es claro puesto que dimos como especificaciones de ellos que:

  i. ponComoOrigen debe únicamente marcar al vértice y agregarlo a la frontera, ambas operaciones finitas y bien definidas.
  ii. El agregar un objeto a la lista ligada de los vértices usados como origen también está bien definido.

  Por lo anterior, podemos garantizar que la ejecución de este método alcanza el ciclo en la línea [33].

  Sabemos que al llegar la ejecución a la línea [33] para iniciarse el ciclo, la frontera contiene a un único elemento. Veamos por qué:

  i. Si se está invocando a exploracionGenerica por primera vez, la frontera se construye vacía. Los únicos métodos que pueden agregar objetos a la frontera son ponComoOrigen y explora, pero ambos métodos son protegidos, por lo que no se tiene acceso a ellos desde fuera de la clase. De esto, una vez obtenido el vértice origen, se tiene que invocar a exploracionGenerica para, desde ese método, tener acceso a la frontera. Y lo único que hace este método antes de entrar al ciclo es agregar a s a la frontera.

ii. Si dentro de la misma aplicación se invoca a exploracionGenerica más de una vez, para que se ejecute la invocación la i-ésima vez, $i > 1$, se tuvo que terminar la ejecución de la invocación $i - 1$, y esto sucede cuando el método sale del ciclo de las líneas [33–42] porque la frontera está vacía. Por lo que en la siguiente invocación vamos a tener nuevamente la frontera vacía antes de entrar a este método, y la situación se repite tal cual.

Nos falta establecer que la frontera en efecto se vacía, ya que ésta es la condición para que el ciclo deje de ejecutarse. Para garantizar esto basta notar lo siguiente:

- Por el Lema 3.10, cada vértice es incorporado a la frontera a lo más una vez.
- Por el Lema 3.11, todo vértice que es introducido a la frontera es eventualmente sacado de ella.

Y como la gráfica tiene un número finito de vértices, la frontera eventualmente quedará vacía.

De esto podemos garantizar que la condición para la iteración en la línea [33] de exploracionGenerica se cumple al menos una vez, y se va a dejar de cumplir, por lo que el algoritmo entonces terminará.

B. Respecto al inciso (B) podemos argumentar de la siguiente manera:

  i. Las propiedades (Bi) y (Bii) fueron establecidas en el Lema 3.12.
  ii. La propiedad (Biii) fue establecida en el Lema 3.15.
  iii. La propiedad (Biv) quedó establecida en el Lema C.1.

C. Dedicaremos la siguiente sección para establecer la complejidad del algoritmo.

Pasamos ahora a calcular una fórmula general para la clase ExploracionBasica, que será el marco de referencia para el cálculo de la complejidad de cada una de las especializaciones.

3.4. Complejidad del algoritmo genérico

Queremos obtener una fórmula genérica para la complejidad de la exploración abstracta, que estará en términos de los costos de cada uno de los métodos invocados. Para que en términos de esta fórmula podamos justificar que el algoritmo genérico es $O(|V| + |E|)$ (procederemos de la misma forma que lo hicimos para establecer las propiedades para el algoritmo genérico, donde supusimos la correctez de los métodos invocados; en el caso del cálculo de la complejidad daremos una fórmula genérica y resolveremos los costos de los métodos involucrados usando para ello la implementación básica que presentamos.

Revisando el método exploracionGenerica, que es el que da la estrategia genérica del algoritmo, identificamos los siguientes componentes de la complejidad del algoritmo.

$f_{constr}(n, m)$	Corresponde a la construcción de la gráfica y el marcado de los vértices y aristas (arcos), así como la construcción de la frontera.

$f_{daS}(n)$	Selecciona al origen del siguiente ciclo de exploración.
$f_{marcaS}(n)$	Costo de anotar al vértice y meterlo a la frontera.
$f_{F.noVacia}(n)$	Costo de verificar si la frontera está vacía.
$f_{elige}(n)$	Costo de seleccionar al siguiente centro de acción.
$f_{cAccion}(n, m)$	El proceso al centro de acción.
$f_{masAr}(m)$	Define si el vértice tiene más aristas disponibles.
$f_{daAr}(m)$	El costo de obtener una arista que no ha sido usada, incidente al vértice.
$f_{explora}(n, m)$	Costo de explorar desde un vértice y por una cierta arista.
$f_{cierra}(n)$	Costo de cerrar un vértice y sacarlo de la frontera.
$f_{fin}(n, m)$	Costo de finalizar la exploración.

Como se puede observar, cada uno de los componentes de la fórmula que describimos antes corresponde a una línea ya sea en el constructor de la clase o en el método exploracionGenerica, ya que en la clase abstracta tenemos como no modificables a la estrategia genérica (exploracionGenerica) y a la exploración de una arista (explora). Para los métodos abstractos podemos simplemente poner requisitos de implementación, de tal manera que la exploración tenga un cierto costo. También pondremos restricciones a los costos de los métodos que nos dan el acceso a las distintas estructuras de datos, sin necesidad de mostrar su implementación. Estableceremos asimismo que para la implementación básica dada, este cálculo es acertado. Recordemos que todos los algoritmos específicos van a extender esta implementación básica.

Para tener una ejecución de exploracionGenerica deberemos primero construir una instancia de la clase ExploracionBasica. De esto, y siguiendo el código dado para exploracionGenerica, la fórmula para la complejidad de este algoritmo genérico en términos de cada vértice origen seleccionado, está dado por la Ecuación (3.1), donde los límites de la suma indican que participarán los vértices de V alcanzables desde s y los arcos incidentes desde esos vértices y desde s – en cada iteración del algoritmo se elige un vértice y/o un arco. Cada una de las funciones de complejidad tiene como parámetro a n, m o ambas, indicando que el costo de ese método será una función del número n de vértices en la gráfica, del número m de aristas de la gráfica o de ambos. De esto sabemos que el límite superior de la sumatoria será $n + m$. Usamos el símbolo $\oplus$ para denotar que en cada iteración únicamente uno de los dos operandos participa en el cómputo.

$$\begin{aligned}\text{Complejidad de ExploracionAbstracta} &= f_{constr}(n, m) + f_{daS}(n) + f_{marcaS}(n) + \\ &+ \sum_{m+n} \Big(f_{F.noVacia}(n) + f_{elige}(n) + f_{cAccion}(n, m) + f_{masAr}(n, m) + \\ &+ \Big(\big(f_{daAr}(m) + f_{explora}(n, m)\big) \oplus f_{cierra}(n) \Big) \Big) + f_{fin}(n, m). \end{aligned} \tag{3.1}$$

Analizando la ecuación (3.1) podemos ver que, aun cuando vamos a ejecutar a lo más $n+m$ veces el ciclo, no todos los términos aportan en las $n+m$ iteraciones, podemos reescribir

la sumatoria clasificando aquello que se ejecuta $n + m$ veces, lo que se ejecuta únicamente n veces y lo que se ejecuta m veces, razonando de la siguiente manera:

- En cada iteración, de acuerdo al Lema 3.9, se elige un arco o se pinta un vértice de NEGRO. Por lo tanto, podemos elegir hasta m arcos y pintar hasta n vértices de NEGRO.
- Por lo anterior, las líneas [34–35] del Listado 3.6 se ejecutarán $n + m$ veces.
- Las líneas [37–38] se ejecutan por cada arista, por lo que esto será a lo más m veces.
- La línea [40] se ejecuta 1 vez por cada vértice, por lo que esto será a lo más n veces.

De lo anterior podemos reescribir la ecuación (3.1) como se muestra en la ecuación (3.2).

$$\begin{aligned}\mathsf{Complejidad\ de\ ExploracionAbstracta} &= \mathsf{f}_{\mathsf{constr}}(\mathsf{n},\mathsf{m}) + \mathsf{f}_{\mathsf{daS}}(\mathsf{n}) + \mathsf{f}_{\mathsf{marcaS}}(\mathsf{n}) + \\ &+ \sum_{n+m}\Big(\mathsf{f}_{\mathsf{F.noVacia}}(\mathsf{n}) + \mathsf{f}_{\mathsf{elige}}(\mathsf{n}) + \mathsf{f}_{\mathsf{cAccion}}(\mathsf{n},\mathsf{m})\Big) + \\ &+ \sum_{m}\Big(\mathsf{f}_{\mathsf{masAr}}(\mathsf{n},\mathsf{m}) + \mathsf{f}_{\mathsf{daAr}}(\mathsf{m}) + \mathsf{f}_{\mathsf{explora}}(\mathsf{n},\mathsf{m})\Big) + \\ &+ \sum_{n}\mathsf{f}_{\mathsf{cierra}}(\mathsf{n}) + \mathsf{f}_{\mathsf{fin}}(\mathsf{n},\mathsf{m}).\end{aligned} \tag{3.2}$$

En la ecuación anterior, sin embargo, hay que tener presente que si bien explora va a ser invocado m veces, el trabajo en cada una de esas veces dependerá de si el vértice en el otro extremo de la arista está siendo descubierto (n veces) o si se trata de alcanzar un vértice descubierto previamente ($m - n$ veces). Haremos esta distinción cuando revisemos el término $\mathsf{f}_{\mathsf{explora}}(\mathsf{n},\mathsf{m})$. Por lo pronto conviene separar este término del resto de la ecuación, quedando nuestra fórmula genérica como en (3.3).

$$\begin{aligned}\mathsf{Complejidad\ de\ ExploracionAbstracta} &= \mathsf{f}_{\mathsf{constr}}(\mathsf{n},\mathsf{m}) + \mathsf{f}_{\mathsf{daS}}(\mathsf{n}) + \mathsf{f}_{\mathsf{marcaS}}(\mathsf{n}) + \\ &+ \sum_{n+m}\Big(\mathsf{f}_{\mathsf{F.noVacia}}(\mathsf{n}) + \mathsf{f}_{\mathsf{elige}}(\mathsf{n}) + \mathsf{f}_{\mathsf{cAccion}}(\mathsf{n},\mathsf{m})\Big) + \\ &+ \sum_{m}\Big(\mathsf{f}_{\mathsf{masAr}}(\mathsf{n},\mathsf{m}) + \mathsf{f}_{\mathsf{daAr}}(\mathsf{m})\Big) + \sum_{m}\mathsf{f}_{\mathsf{explora}}(\mathsf{n},\mathsf{m}) + \\ &+ \sum_{n}\mathsf{f}_{\mathsf{cierra}}(\mathsf{n}) + \mathsf{f}_{\mathsf{fin}}(\mathsf{n},\mathsf{m}).\end{aligned} \tag{3.3}$$

Tenemos ya una fórmula general para clasificar la complejidad de las distintas especializaciones de ExploracionAbstracta. Por lo pronto analizaremos la complejidad de la implementación básica dada, que cumple con las propiedades exigidas para los métodos abstractos. Llenaremos cada uno de los valores para las funciones de complejidad que dimos.

3.4.1. Complejidad para la especialización de la exploración básica

Analizar la clase ExploracionBasica es muy importante, porque todas las variantes o especializaciones para la exploración extienden a esta clase, utilizando la estrategia genérica y extendiendo o redefiniendo a los métodos que son abstractos en ExploracionAbstracta.

Implementación del constructor ExploracionBasica. El constructor de esta exploración básica invoca al constructor de la superclase – línea [5] del Listado 3.8 – y después inicializa la frontera básica en la línea [6] del mismo listado. De esto, tenemos en la Ecuación (3.4) la conformación del término $\mathsf{f}_{\mathsf{constr}}(\mathsf{n},\mathsf{m})$:

$$\mathsf{f}_{\mathsf{constr}}(\mathsf{n},\mathsf{m}) = \mathsf{f}_{\mathsf{constrEA}}(\mathsf{n},\mathsf{m}) + \mathsf{f}_{\mathsf{constrFront}}(\mathsf{n},\mathsf{m}). \tag{3.4}$$

La composición de la fórmula para la construcción de la clase abstracta se muestra en la Ecuación (3.5).

$$\mathsf{f}_{\mathsf{constrEA}}(\mathsf{n},\mathsf{m}) = \mathsf{f}_{\mathsf{obtenN}}(\mathsf{n},\mathsf{m}) + \mathsf{f}_{\mathsf{obtenM}}(\mathsf{n},\mathsf{m}) + \mathsf{f}_{\mathsf{constrListas}}(\mathsf{n}) + \mathsf{f}_{\mathsf{constrListas}}(\mathsf{m}). \tag{3.5}$$

Calculemos la complejidad del constructor para la exploración abstracta. Como podemos observar, a pesar de que el término para el constructor está en términos de n y m, el tamaño de la gráfica, como ésta se entrega al algoritmo ya construida no influye en la complejidad del algoritmo, y únicamente tenemos que contabilizar el registro de la misma, que cuesta una unidad de tiempo.

La implementación de la gráfica tiene que ser tal que podamos obtener de manera directa. sin recorrer la gráfica y sin contar, tanto el número de vértices como el número de aristas, por lo que registrar estas dos cantidades también nos cuesta una unidad cada operación.

También la construcción de las listas debe ser tal que su costo sea bajo. En este caso, las listas que estamos usando requieren de tres asignaciones, ya que involucra únicamente la inicialización de las variables que se refieren al principio y final de la lista, y al tamaño de la lista en 0. En el cuadro siguiente resumimos estos cálculos.

public ExploracionAbstracta(Digrafica g). (Líneas [19] a [25] del Listado 3.5).

- Registro de la gráfica: Número de pasos: 1 en: $O(1)$.
- Obtener n y m: Número de pasos: 2 en: $O(1)$.
- Iniciar lista de nodos alcanzados.
  Número de pasos: 3 en: $O(1)$.
- Iniciar lista de aristas usadas.
  Número de pasos: 3 en: $O(1)$.

Total para ExploracionAbstracta:

Número de pasos: 9 en: $O(1)$.

De lo anterior, la complejidad para el constructor está dada por:

$$\mathsf{f}_{\mathsf{constrEA}}(\mathsf{n},\mathsf{m}) = \mathbf{9}$$

que está en la clase

$$\mathbf{O(1)}.$$

El costo de la construcción de la frontera se define para cada clase de frontera que se utilice. En el caso de la clase que corresponde a ExploracionBasica, la frontera es una lista ligada que requiere de tres asignaciones para su construcción. De todo esto, el costo del constructor para ExploracionBasica queda como sigue:

public ExploracionBasica(Digrafica g). (Líneas [4] a [7] del Listado 3.8).

- Constructor de la superclase:

  Número de pasos: 9 en: $O(1)$.

- Construcción de la frontera básica:

  Número de pasos: 3 en: $O(1)$.

Total para ExploracionBasica:

Número de pasos: 12 en: $O(1)$.

De esto, la complejidad para el constructor está dado por:

$$\mathsf{f}_{\mathsf{constr}}(\mathsf{n},\mathsf{m}) = \mathbf{12}$$

que es de

$$O(1).$$

Para obtener el origen del ciclo de exploración se debe tener un mecanismo eficiente para obtener un vértice que aún no ha sido explorado. En una primera instancia, se puede realizar esto únicamente una vez y pedirle al usuario que proporcione el vértice origen.

public protected Nodos obtenOrigen(Digrafica g) (Líneas [54] a [56] del Listado 3.6).

Obtener siguiente vértice no explorado:

Número de pasos: c_o en: $O(1)$.

De esto, la complejidad de este método está dada por:

$$\mathsf{f}_{\mathsf{daS}}(\mathsf{n}) = \mathbf{c_o}$$

y está en la clase

$$O(1).$$

El marcar a un vértice como el origen de la exploración es una actividad de orden constante, ya que involucra dos asignaciones y agregar al vértice a la frontera. Como la frontera, en este caso, es una lista ligada, agregar un vértice involucra tiempo constante, ya que se le agrega simplemente al final de la lista y no se tiene que recorrer la lista en ningún sentido.

protected void ponComoOrigen(Nodos v).: (Líneas [48] a [50] del Listado 3.6).

- Marcar la distancia como 0: Número de pasos: 1 en: $O(1)$.
- Marcar al nodo como descubierto: Número de pasos: 1 en: $O(1)$.
- Ingresarlo a la frontera: Número de pasos: 3 en: $O(1)$.
- Marcar el origen: Número de pasos: 5 en: $O(1)$.

De esto, la fórmula para la complejidad de este método es:

$$f_{\mathsf{marcaS}}(n) = \mathbf{5}$$

y está en la clase

$$O(1).$$

El verificar que la frontera no esté vacía es también una operación de orden constante, ya que basta verificar los atributos de la lista ligada, o el número de elementos registrados.

public boolean esNoVacia().: (Línea [11] del Listado 3.4).

Preguntar si no hay nadie en la lista:
Número de pasos: 1 en: $O(1)$.

De esto, la fórmula de complejidad para este método queda:

$$f_{\mathsf{F.noVacia}}(n) = \mathbf{1}$$

en la clase de complejidad de

$$O(1).$$

En esta implementación básica, no hay ninguna acción que se lleve a cabo con el centro de acción de la exploración, por lo que el método procesaCentroAccion(Nodos centroAccion) no contribuye a la complejidad del algoritmo.

El método elige es un poco más interesante, aunque nuevamente depende de la implementación de la frontera. En el caso de la exploración básica, la frontera es simplemente una lista circular que con cada solicitud devuelve al nodo al frente de la lista circular y avanza al siguiente.

public Nodos elige(). (Línea [6] del Listado 3.4).

- Anotar la referencia del vértice referido por la variable definida para ello.
Número de pasos: 1 en: $O(1)$.

- Avanzar la referencia en una lista circular.

Número de pasos: 1 en: $O(1)$.

Total para elige:

Número de pasos: 2 en: $O(1)$.

y la fórmula para este término queda:

$$\mathsf{f}_{\mathsf{elige}}(\mathsf{n}) = \mathbf{2}$$

en la clase de complejidad de

$$O(1).$$

El siguiente método que tenemos que analizar es el que realiza un proceso sobre el centro de acción, procesaCentroAccion, que es invocado en la línea [35] en el Listado 3.6. Pero en la especialización para ExploracionBasica este método no lleva a cabo ninguna acción, por lo que su complejidad no aporta nada a nuestra fórmula genérica.

El análisis para el método masAristas se reduce a exigirle a cada vértice que pueda responder a esta pregunta en tiempo constante. En el caso de nuestra implementación para los nodos, tenemos asociado a cada nodo una lista de adyacencia que contesta en $O(1)$ a la pregunta de si la lista tiene todavía aristas disponibles. Aún en el caso de que estemos trabajando con aristas y haya que recorrer la lista de adyacencia hasta encontrar una arista disponible, eliminando aristas disponibles desde el otro extremo, esto no puede suceder, en el total de los vértices, más de $2 \cdot m$ veces, pues es el número total de referencias en las listas de adyacencias. Dado esto, si amortizamos el recorrido para encontrar si quedan aristas disponibles entre todos los vértices, como está en una sumatoria con límite m, podemos pensar en que la complejidad es de 2 unidades en promedio por cada ejecución. Si la lista de adyacencia ya está vacía, esto puede ser determinado con una comparación.

public boolean masAristas(). (Línea [20] del Listado 3.1).

- Recorrer la lista mientras se presenten aristas ya usadas:

Número de pasos: 2 en: $O(1)$.

- Preguntar si la lista está vacía:

Número de pasos: 1 en: $O(1)$.

Total para masAristas:

Número de pasos: 3 en: $O(1)$.

De esto, la fórmula para este método queda:

$$\mathsf{f}_{\mathsf{masAr}}(\mathsf{n},\mathsf{m}) = \mathbf{3}$$

en la clase de complejidad

$$O(1).$$

Para el método getaArista() podemos hacer un análisis similar al que hicimos para el método masAristas, ya que, en principio, como tenemos una lista de incidencia para las

aristas asociadas a un vértice, obtener el siguiente arco cuesta un número constante de pasos, ya que la lista va avanzando conforme entrega arcos disponibles. Sin embargo, si estamos trabajando con una gráfica no dirigida, cada arista aparece en dos listas de incidencia, por lo que al utilizarse desde un vértice ya no queda disponible para el otro. Por lo tanto, de la misma manera que para masAristas, podemos repartir entre todas las peticiones de aristas el tener que eliminar a la mitad de ellas por haber sido ya utilizadas desde el vértice en el otro extremo. Como vamos a solicitar, a lo más, m aristas, podemos pensar en que el total de este método es $2 \cdot m$, y como este término está dentro de una sumatoria con límite m, si le asignamos un costo de 2 unidades a cada ejecución de este método, tendremos una cota superior para la complejidad del algoritmo.

public Aristas getArista(). (Línea [6] del Listado 3.4).

Avanzar a la siguiente arista disponible:

	Número de pasos:	2	en:	$O(1)$.

Total para getArista:

	Número de pasos:	2	en:	$O(1)$.

De esto, la complejidad para getArista queda:

$$\mathsf{f}_{\mathsf{daAr}}(\mathsf{m}) = \mathbf{2}$$

en la clase

$$O(1).$$

El método explora(Nodos, Aristas) es también un método que se hereda tal cual a las especializaciones dadas, y es el encargado realmente de hacer la exploración, agregando vértices a la frontera y marcando aristas como usadas. Hagamos el análisis de su complejidad. Si observamos la implementación de este método en las líneas [69–77] en el Listado 3.7, y dado que la implementación de este método permanece constante en todas las especializaciones que extienden a ExploracionBasica, tenemos la siguiente fórmula que nos da su complejidad:

$$\sum_{m} \mathsf{f}_{\mathsf{explora}}(\mathsf{n},\mathsf{m}) = \sum_{m} \Big(\mathsf{f}_{\mathsf{getOtroN}}() + \mathsf{f}_{\mathsf{proceAr}}() + \mathsf{f}_{\mathsf{yaUsado}}()\Big) + \sum_{n} \mathsf{f}_{\mathsf{descubriendo}}(\mathsf{n}) + \sum_{m-n} \mathsf{f}_{\mathsf{yaDescubierto}}(\mathsf{n}), \quad (3.6)$$

donde el significado de cada uno de los términos es el de la complejidad del método al que denotan. Dado que explora se va a ejecutar m veces, y que descubriendo se debe ejecutar n veces, mientras que yaDescubierto se debe ejecutar $m - n$ veces, en cada iteración se ejecuta exactamente uno de los dos. Con esta observación pasamos a calcular la complejidad del método explora.

public void explora(Nodos, Aristas). (Líneas [69–77] del Listado 3.7).

getOtroNodo: (línea [29] del Listado 3.2). Obtener el nodo en el otro extremo de la arista. La implementación nos debe garantizar complejidad constante. En nuestra implementación, este método ejecuta 3 pasos.
Número de pasos: 3 en: $O(1)$.

procesaArista(Aristas) (líneas [36–38] en el Listado 3.8), que en la especialización para ExploracionBasica únicamente invoca al método setUsada(), cuyo costo es la asignación del valor verdadero a este atributo.
Número de pasos: 1 en: $O(1)$.

haSidoAlcanzado() ([24] del Listado 3.1). Nuestra implementación únicamente pregunta por el estado de un atributo.
Número de pasos: 1 en: $O(1)$.

descubriendo(): (línea [17–28] del Listado 3.8). Todos los métodos invocados en este método, excepto por frontera.agregaNodo, nodosAlcanzados.agrega y aristasUsadas.agrega en la líneas [22], [24] y [27] respectivamente, son métodos que lo único que hacen es cambiar el valor de un atributo, por lo que tienen complejidad constante. Los tres métodos mencionados agregan a una clase de lista donde para agregar a un elemento requerimos de 3 asignaciones.
Número de pasos: 17 en: $O(1)$.

yaDescubierto no hace nada.
Número de pasos: 0

Quedando la fórmula para este método:

$$\sum_{\mathbf{m}} \mathsf{f}_{\mathsf{explora}}(\mathsf{n},\mathsf{m}) = \sum_{\mathbf{m}} \mathbf{5} + \sum_{\mathbf{n}} \mathbf{17} + \sum_{\mathbf{m-n}} \mathbf{0} = \mathbf{5m + 17n}$$

que está en la clase de complejidad de

$$O(n+m).$$

El valor para $\mathsf{f}_{\mathsf{fin}}(\mathsf{n},\mathsf{m})$ es $O(n+m)$ pues se encarga, a partir de las aristas incluidas en G_π, de construir una gráfica isomorfa a G_π.

Realmente no podemos precisar más la fórmula para exploracionGenerica en la clase ExploracionAbstracta precisamente porque quedan muchos aspectos por definir y precisar: la estructura de la frontera, la complejidad de los accesos a la misma, la forma de obtener el nodo origen, lo que se hace al descubrir un vértice, etc. Nos tendremos que conformar con la fórmula dada en la Ecuación (3.2), sustituyendo al término $\mathsf{f}_{\mathsf{explora}}(\mathsf{n},\mathsf{m})$ por la fórmula dada en la Ecuación (3.6). Sin embargo, para la implementación básica dada en la clase ExploracionBasica, donde todos los métodos tienen una implementación lo más simple posible,

sí podemos calcular de mejor manera la complejidad. Ésta está dada por la sustitución en la Ecuación (3.3) de los costos que calculamos para esta implementación. El resultado lo podemos observar en la fórmula (3.7).

$$
\begin{aligned}
\text{Complejidad de ExploracionBasica } &= \\
&= c_o + 9 + 5 + \sum_{n+m}(1+2+0) + \sum_{m}(3+2) + 5m + 17n \\
&= c_o + 14 + 3(n+m) + 5m + 5m + 17n \\
&= 20n + 13m + c_o + 14 + c_1(m+n) \\
&= O(n+m). \qquad (3.7)
\end{aligned}
$$

El teorema de complejidad para el algoritmo genérico de exploración queda de la siguiente manera:

Lema 3.19 *El algoritmo genérico de exploración, con las implementaciones dadas en la clase ExploracionBasica es de complejidad $O(n+m)$, donde $|V| = n$ y $|E| = m$.*

Demostración:

Está dada por las justificaciones dadas en los cálculos que acabamos de hacer, ❏

No podemos dejar de mencionar antes de dar por concluida esta sección, que el algoritmo puede dejar vértices sin explorar, ya que sólo explora a aquéllos que son alcanzables desde s. Para lograr la exploración de todos los vértices, se debe modificar ligeramente el algoritmo de tal manera que repita la exploración desde distintos orígenes, seleccionando sucesivamente a algún vértice que permanezca todavía sin descubrir.

3.5. Exploración en gráficas no dirigidas

Para utilizar el algoritmo genérico de exploración en una gráfica no dirigida (o simplemente gráfica) no se requiere de muchos cambios. Todo lo que se debe hacer es generar el objeto de exploración con una gráfica no dirigida como parámetro, cuya clase deberá heredar de la clase para la digráfica. Cuando a un nodo se le pida que entregue la siguiente arista (arco) sin usar, deberá verificar que el arco que está por entregar de su lista de adyacencia no haya sido ya utilizado desde otro nodo. Esta verificación se encuentra en el nodo básico. Simplemente, cuando la gráfica es dirigida, como cada arco aparece en una sola lista de adyacencia, esta verificación será redundante.

Dado que la definición de los nodos es exactamente la misma, y el único cambio es que al construir la gráfica no dirigida se marcará a los arcos como sin dirección, no es de hecho necesario ningún cambio en el algoritmo genérico o en la estructura de datos para la frontera.

El teorema de correctez y complejidad del algoritmo genérico para gráficas no dirigidas queda como sigue:

Teorema 3.20 *El algoritmo genérico de exploración dado por el método* exploracionGenerica *de la clase* ExploracionAbstracta, *dada una gráfica no dirigida* $G = (V, E)$ *cumple con las mismas propiedades expresadas en el Teorema 3.18.*

Demostración:

La demostración es similar a la del Teorema 3.18 y no aporta mucho, por lo que se omite. Vale la pena mencionar que cuando hicimos el cálculo del costo de obtener un arco no usado consideramos ya el caso de gráficas no dirigidas, en la que cada arista aparece en dos listas de incidencia. Así que la clase de complejidad no cambia para esta clase de gráficas. ❑

3.6. Aplicaciones del algoritmo genérico de exploración

Aún con el no determinismo que este algoritmo presenta, podemos ya utilizarlo para determinar propiedades de la gráfica explorada. Estas propiedades pueden ser muy útiles, sobre todo cuando la gráfica está representando alguna estructura sobre la que queremos trabajar, como lo es una red de computadoras o un diagrama de organización de tareas.

3.6.1. Ciclos en gráficas no dirigidas

Utilizando el método exploracionGenerica de la clase ExploracionBasica es fácil determinar si la gráfica contiene o no ciclos. Si al estar explorando la gráfica nos encontramos con una arista que vaya a un vértice ya visitado, ese arco cierra un ciclo. En este caso, el arco que se acaba de recorrer no será incluido en G_π. Estas observaciones se formalizan en el Lema 3.21:

Teorema 3.21 *Una gráfica* $G = (V, E)$ *es acíclica* $\iff$ $E = E_\pi$.

Demostración:

$\Longrightarrow$ Por contra positivo, supongamos que $E \neq E_\pi$ y consideremos una arista

$\mathsf{e} = \mathsf{u}\text{—}\mathsf{v} \in E - E_\pi$.

Si $\mathsf{e} \notin G_\pi$, en el momento de recorrerla quedó orientada desde un vértice en G_π a otro que también ya estaba en G_π. Esto quiere decir que hay un camino $\mathsf{s} \rightsquigarrow \mathsf{u}$ y otro camino $\mathsf{s} \rightsquigarrow \mathsf{v}$, que junto con la arista e cierra un ciclo dado por $\mathsf{v} \rightsquigarrow \mathsf{s} \rightsquigarrow \mathsf{u}\text{—}\mathsf{v}$.

$\Longleftarrow$ También por contra positivo, supongamos ahora que todas las aristas de E se encuentran en E_π. Mostraremos que G no tiene ciclos. Y en efecto, esto es así, ya que G_π es un árbol, y por lo tanto acíclica. Y como en G no hay ni una arista más o menos que las que hay en G_π, también G es un árbol y por lo tanto, acíclica. ❑

3.6.2. Componentes conexas en una gráfica

El algoritmo genérico puede ser utilizado, en el caso de gráficas no dirigidas, para determinar las componentes conexas de la gráfica. Esta aplicación requiere de pocas modificaciones al algoritmo. Para ello, basta notar lo siguiente:

1. Cada vez que se invoca a explorацionGenerica con un vértice s, la frontera debe empezar con ese único vértice y, como se demostró, se vacía al finalizar la ejecución de este método.
2. Los vértices que no pertenecen a la componente conexa de s quedan pintados de BLANCO ya que nunca fueron descubiertos.
3. La complejidad de cada ejecución del algoritmo depende del tamaño de la componente conexa a la que pertenece el vértice s.
4. Si para cada ejecución de exploracionGenerica se elige un vértice que aun no ha sido descubierto, el número total de vértices descubiertos será n, cada uno de ellos descubierto exactamente una vez, y el número total de aristas recorridas será m, también cada una de ellas usadas exactamente una vez.

Dadas estas observaciones, requerimos de esta especialización que resuelva el siguiente aspecto: el método que regresa el nodo origen de la exploración, debe elegir al siguiente vértice pintado de BLANCO, sin que tenga que estar revisando a todos los vértices en cada ocasión.

Podemos resolver esto de manera muy sencilla. Colocamos en la clase abstracta un contador que vaya señalando, en el orden en el que están los vértices en la estructura de nodos de la gráfica, al siguiente vértice – empezamos con el primero. Se construye al objeto de la exploración – el algoritmo – y después seleccionamos el siguiente vértice e invocamos a exploracionGenerica con él. La aplicación se vería como se muestra en el Listado 3.9.

Listado 3.9: Utilización para determinar componentes conexas

```
1   class Aplicacion {
2       public static void main(String[] args) {
3           Grafica graf = new Grafica();
4           Grafica arbol = null;
5           ExploracionBasica exploracion =
6                                   new ExploracionBasica(graf);
7           Nodos s;
8           while((s = exploracion.obtenOrigen()) != null)
9               arbol = exploracion.exploracionGenerica(s);
10              arbol.muestra();
11          }
12      }
```

Si queremos reconocer a las diferentes raíces en el bosque de exploración, es conveniente que construyamos un conjunto en el que se encuentren estas raíces. Llamemos a este conjunto Componentes, y agregamos a él todos y cada uno de los vértices utilizados como origen en la exploración.

Con estas estructuras de datos adicionales, debemos redefinir al método obtenOrigen para que siga entregando vértices como origen mientras haya vértices que no hayan sido descubiertos. En el Listado 3.10 mostramos una implementación para el método obtenOrigen, que es el único que tiene que ser redefinido.

Por último, mostramos el enunciado del Teorema de Correctez de este algoritmo:

Teorema 3.22 *El algoritmo genérico que encuentra las componentes conexas de una gráfica:*

A. Cumple con el Teorema 3.19.

B. Al terminar, el conjunto Componentes *contiene a un vértice s_i por cada componente conexa de la gráfica, y este vértice s_i es la raíz del árbol G_π que corresponde a esa componente.*

Demostración:

Lo único que hay que demostrar es que los árboles enraizados en los orígenes s_i son ajenos y cubren toda la gráfica. Esto es claro puesto que los vértices que ya se utilizaron para un árbol están pintados de NEGRO y no pueden ser tomados en cuenta una vez que ya fueron pintados de NEGRO. Lo mismo sucede con las aristas, que una vez recorridas ya no pueden volver a ser consideradas. De esto, los árboles enraizados en los s_i son ajenos entre sí.

Por otro lado, para el conjunto Componentes se eligen vértices que aún son BLANCOS, y obtenOrigen regresa un nodo mientras haya vértices sin alcanzar, por lo que todos los vértices serán incluidos en alguna componente. ❑

Listado 3.10: Especialización para componentes conexas

```
13 public class ConexExplora
14           extends ExploracionBasica    {
15     /* Cuenta los nodos que ha ido eligiendo.                          */
16     private int siguienteOrigen = 1;
17     /* Lista de orígenes.                                              */
18     LinkedList Componentes = new List();
19     public ConexExplora(Digrafica g) {
20         super(g);
21     }
22       /* Selecciona a un vértice para la exploración. Los toma          *
23        * verificando si el siguiente vértice es BLANCO.                 */
24     protected Nodos obtenOrigen(Digrafica graf) {
25         while ( siguienteOrigen <= N &&
26                   g.getNodo(siguienteOrigen).haSidoAlcanzado())
27             siguienteOrigen++;
28         if (siguienteOrigen > N)
29             return null;
30         Nodos s = g.getNodo(siguienteOrigen++);
31         Componentes.agrega(s);
32         return s;
33     }
34 }
```

3.6.3. Conexidad fuerte en digráficas

Recordemos primero que en una digráfica el concepto de conexidad se refiere a la gráfica no dirigida subyacente, mientras que si queremos expresar el concepto de que cada vértice de la digráfica sea alcanzable desde cualquier otro a través de un camino dirigido, nos referimos

a *conexidad fuerte*. Decimos que la gráfica es *fuertemente conexa* de darse la propiedad que acabamos de mencionar.

Un algoritmo ingenuo para determinar si una digráfica es o no fuertemente conexa consiste en ejecutar el algoritmo genérico tomando como origen de la exploración, sucesivamente, a cada uno de los vértices de la gráfica. Para cada vértice el algoritmo determina si cada uno de los vértices restantes en la digráfica es alcanzable desde el origen. Desafortunadamente este algoritmo es muy costoso, pues es $O(n^2)$. A continuación describiremos un algoritmo en $O(|V| + |E|)$ que resuelve el problema de determinar si una digráfica es o no fuertemente conexa.

1. Sea s un vértice arbitrario, que elegimos como origen de la exploración.
2. Ejecutamos el método exploracionGenerica una vez, y vemos si todos los vértices son alcanzables desde s.
3. Si no lo son, reportamos que la gráfica no es fuertemente conexa y salimos.
4. Construimos una nueva digráfica $G' = (V, E')$ con

$$E' = \{y \dashrightarrow x \mid x \dashrightarrow y \in E\}$$

5. Ejecutamos el método exploracionGenerica sobre esta nueva digráfica con s como origen.
6. Si todos los vértices son alcanzables desde s en esta nueva digráfica, la gráfica es fuertemente conexa.

La idea detrás de este algoritmo es la siguiente:

- La primera ejecución de exploracionGenerica establece caminos desde s a cada uno de los vértices de la digráfica.
- Al invertir la dirección de todos los arcos de la digráfica, lo que en la digráfica original era un camino dirigido v⇝u, en la nueva digráfica es un camino dirigido u⇝v.
- Si todos los vértices son alcanzables desde s en la nueva digráfica, existe un camino dirigido s⇝v, o sea un camino dirigido v⇝s en la digráfica original.
- Para encontrar el camino dirigido u⇝v para cualesquiera dos vértices u y v basta notar que, por la primer ejecución de exploracionGenerica tenemos un camino s⇝v, mientras que por la segunda ejecución tenemos un camino u⇝s. Juntando estos dos caminos tenemos u⇝s⇝v.

Por supuesto que debemos enunciar el algoritmo formalmente y demostrar su correctez, pero dado que es más laborioso que complicado, se omite por no presentar una aportación relevante.

3.7. Conclusiones

Terminamos este capítulo con algunas consideraciones respecto al algoritmo genérico desarrollado.

Al revisar este algoritmo podemos notar que hay varios puntos en él que si bien se encuentran completamente especificados – no sería algoritmo si no fuera así – tiene una estructura

no determinística, en el sentido de que permitimos que las acciones estén determinadas en términos únicamente de resultados, no del proceso. Por ejemplo, al elegir el siguiente vértice a explorar de la frontera, no especificamos de manera determinística *cómo* elegir, sino simplemente pedimos que se elija, no determinísticamente, a un vértice $\mathsf{v} \in L$. Lo mismo sucede con la elección de arco o arista por recorrer; simplemente exigimos que una vez elegido no vuelva a estar en el conjunto de elegibles. Esto hace de exploracionGenerica un algoritmo no determinístico, y de la manera como se defina, determinísticamente, no únicamente el resultado sino también el proceso mismo, especializaremos el algoritmo encontrando propiedades adicionales. Es importante insistir en que las especializaciones son posibles en aquellos puntos en que hay no determinismo.

Adicionalmente, muchos de los procesos que se llevan a cabo en cada punto de la exploración pueden ser extendidos a que realicen más actividades de las originalmente planeadas, pero siempre sobre atributos nuevos. Éste es un requisito *sine qua non* podemos garantizar que las extensiones que hagamos sobre ExploracionBasica preserven las propiedades del mismo. Esto es, las actividades especificadas sobre los atributos básicos deben mantenerse, y en caso de que algún método modifique a alguno de los atributos básicos de alguna manera no especificada en ExploracionBasica, deberá demostrarse que esta modificación no afecta las propiedades dadas, o bien determinar de qué manera las afecta.

Capítulo 4

Búsqueda en amplitud (BFS[1])

En este capítulo revisaremos la especialización del algoritmo genérico de exploración (la implementación básica) conocida como BFS, que revisa los vértices por niveles. Estableceremos que las implementaciones dadas para los métodos abstractos son correctas, y definiremos la manera en que estos métodos colaboran para la complejidad del algoritmo.

La exploración conocida como BFS trabaja alcanzando un vértice, y desde él explorando a todos los vértices adyacentes a él, e incluyéndolos en una cola, de manera que responde a la estrategia genérica de nuestro algoritmo de exploración.

Veremos algunas aplicaciones de esta exploración, presentándolas como especializaciones de la especialización. Entre estas especializaciones se encuentran: determinar si una gráfica es bipartita y calcular el diámetro de un árbol.

4.1. Descripción del problema

Un algoritmo que consideramos de exploración es el que explora los vértices siguiendo una estrategia en amplitud. Para ello, define "niveles" o capas de la gráfica, que tienen que ver con la distancia (en términos de número de arcos o aristas) a las que se encuentra cada vértice del vértice origen s. Es uno de los algoritmos más simples que se han diseñado para explorar gráficas, garantizándonos, como en todos los casos de algoritmos de exploración, que explora a todos los vértices alcanzables desde s y recorre cada arco incidente desde esos vértices exactamente una vez. El algoritmo es conocido por sus siglas en inglés como BFS (*Breadth First Search*). Este algoritmo se puede aplicar tanto a gráficas no dirigidas, como a digráficas, al igual que ExploracionBasica.

Por la forma en que BFS va explorando los vértices, en términos de su distancia a s, a través de G_π encuentra los caminos más cortos entre s y cada uno de los vértices que va

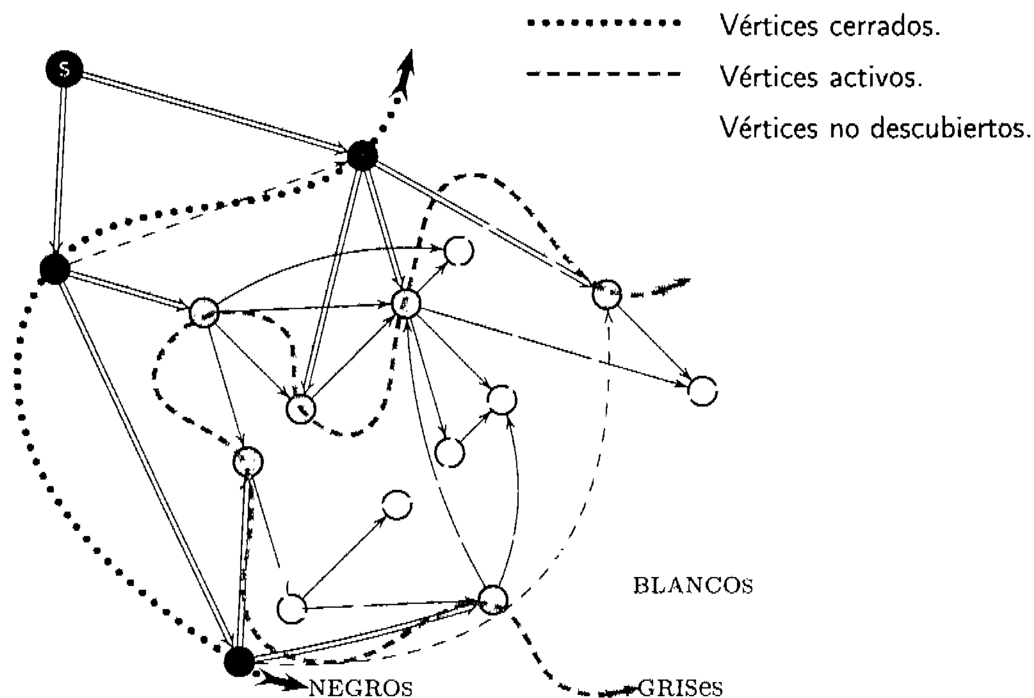
[1]En el desarrollo de este capítulo utilizaremos el nombre *BFS* para esta exploración, ya que es más compacto y se reconoce fácilmente.

alcanzando.

El nombre de "exploración en amplitud" se debe a que BFS expande la frontera entre vértices descubiertos y no descubiertos uniformemente. Es decir, descubre todos los vértices a distancia k antes de descubrir los que están a distancia $k+1$. Desarrolla la exploración *por capas*, saliendo de un vértice s. En la capa 1 están los vértices que están a distancia 1 de s. En la capa 2 están los que están a distancia 1 de algún vértice de la capa 1, y que no se encuentren ya en la capa 1; en la capa $k+1$ están los vértices que están a distancia 1 de algún vértice de la capa k y que no se encuentren ya en alguna de las k capas anteriores.

Como ya mencionamos y al igual que en ExploracionBasica, el árbol que ExploracionBFS construye corresponde al árbol G_π de ExploracionAbstracta, pero en el caso de ExploracionBFS este árbol es un árbol de distancias o de caminos mínimos. Por lo que, como producto lateral de la exploración determinaremos la distancia de todo vértice alcanzable desde el origen. Supondremos a lo largo de todo este capítulo que e.peso $= 1 \quad \forall$e $\in E$.

Figura 4.1: Ejecución BFS en un punto intermedio del proceso



4.2. Especialización de ExploracionBasica: ExploracionBFS

Problema: Dada una digráfica $G = (V, E)$ encontrar los caminos más cortos entre un vértice origen s y cada uno de los vértices de la gráfica.

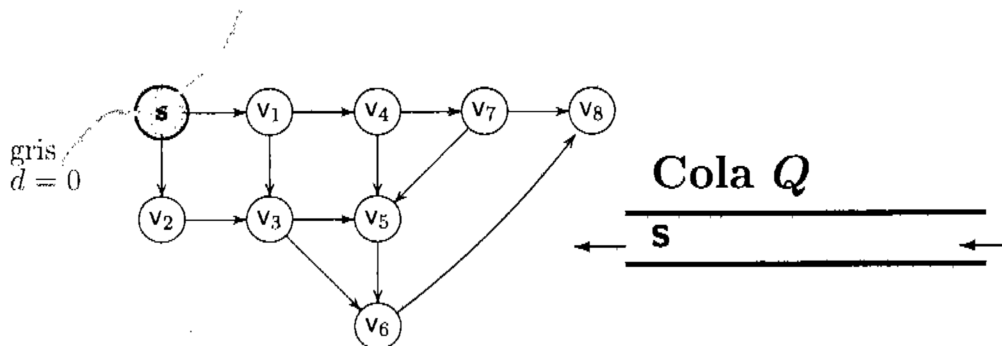
Entrada: Una digráfica $G = (V, E)$.

Salida: El árbol BFS y, para cada vértice, su distancia a s.

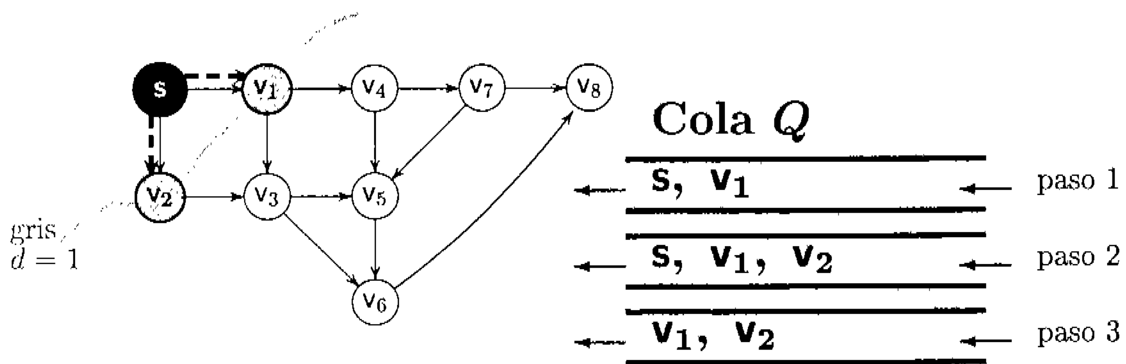
Estructuras de Datos: Para facilitar el acceso a los distintos elementos de la digráfica, especializaremos a la frontera para definir el orden en que se van explorando los vértices, que en el caso de BFS es por capas, como acabamos de mencionar. Este proceso se ilustra en la Figura 4.1. Si frontera corresponde a una cola, el orden en que se exploran los vértices corresponde a una búsqueda en amplitud o por capas.

Antes de describir con detalle el código de ExploracionBFS, veamos un ejemplo de cómo funciona en las Figuras 4.3(a) a 4.3(f). En cada una de las figuras se muestran varios pasos de la exploración, y a la derecha de la gráfica se encuentran uno o más diagramas del estado de la cola, con un diagrama por cada cambio que se verifica en ella en la transición entre figuras consecutivas.

Figura 4.2: Funcionamiento de BFS 1/3

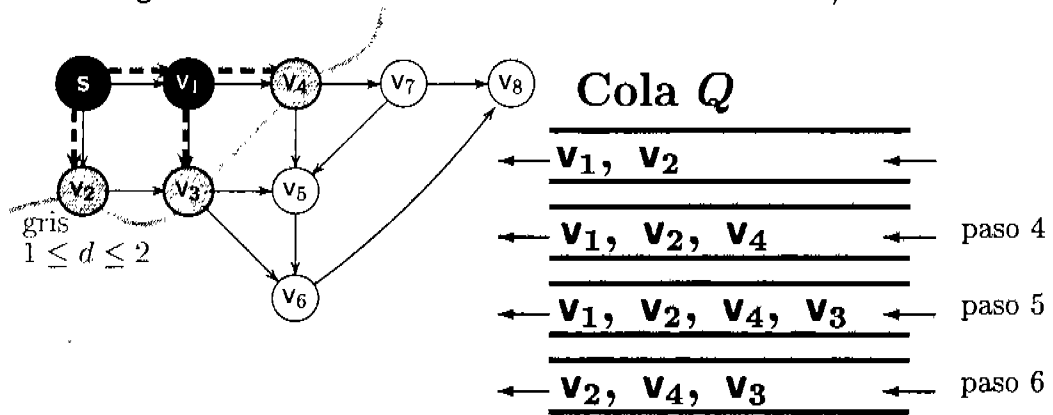


(a) Estado de la gráfica en el momento de iniciar la ejecución de BFS.

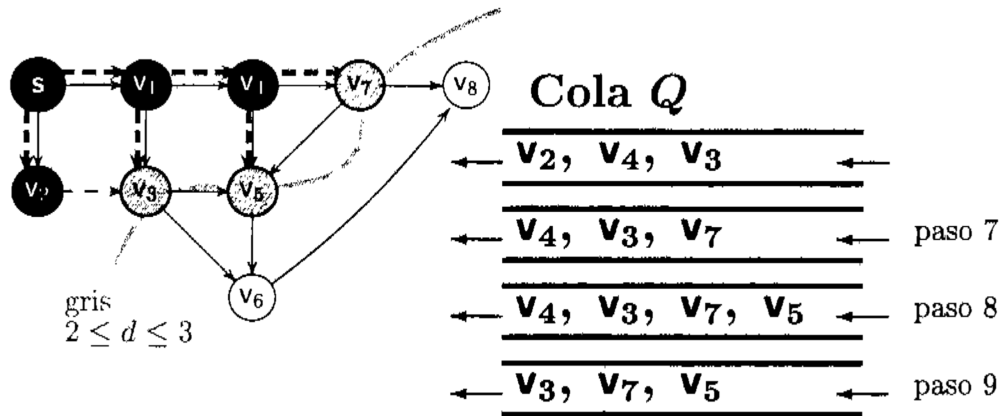


(b) Descubriendo los vértices a distancia 1, v_1 y v_2 (dos pasos)

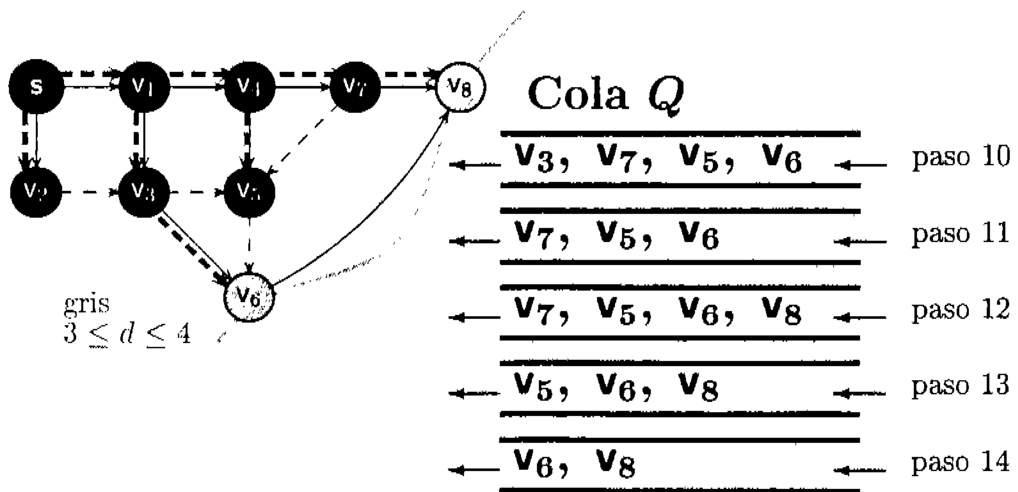
Figura 4.2: Funcionamiento de BFS 2/3



(c) Descubriendo los vértices v_3 y v_4 (tres pasos)

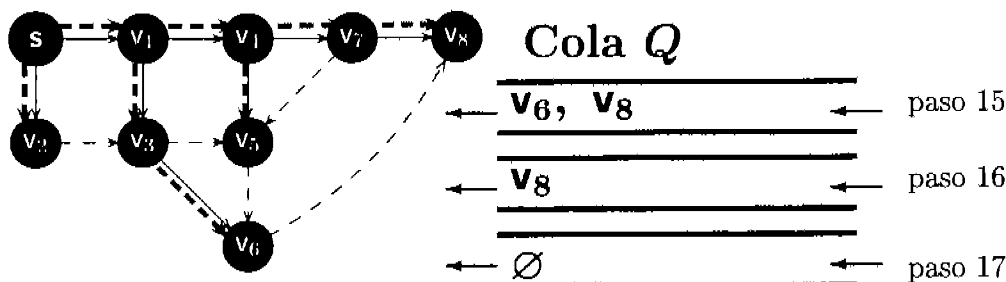


(d) Sacando a v_2 y descubriendo v_7 y v_5 (tres pasos).



(e) Descubriendo los vértices v_6 y v_8 (4 pasos)

Figura 4.2: Funcionamiento de BFS 3/3



(f) Habiendo explorado ya todos los vértices y todos los arcos

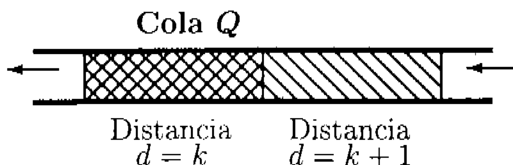
En las figuras anteriores, las flechas gruesas intermitentes indican la dirección del árbol BFS. En este ejemplo particular, quedaron cuatro arcos sin ser incluidos en el árbol y 8 arcos que forman parte del árbol. Dependiendo de la gráfica, el número de arcos incluidos en el árbol puede ser mucho menor que el número de arcos de la gráfica.

Listado 4.1: Especialización de la frontera

```
 1      /* La frontera se implementa como una cola, donde el         *
 2       * vértice que el método elige es el primero de              *
 3       * la lista.                                                 */
 4   public class FronteraEnCola extends FronteraBasica {
 5      /* Construye una frontera vacía y le asocia un iterador.     */
 6     public FronteraEnCola()
 7      {/* Operación que regresa al primero de la cola.   */   }
 8     public Nodos cabeza()
 9      {/* Regresa al último de la cola.                  */   }
10     public Nodos cola();
11   }
```

La especialización de la frontera se muestra en el Listado 4.1. No transcribimos la implementación particular de los métodos agregados, ya que esto se hace de manera tradicional. Además, como mencionamos antes, la definición de la estrategia de acceso a la frontera se encuentra en el iterador que recorre la lista.

Es importante notar que en cada iteración del algoritmo nunca se mezclan vértices de distintos niveles:



Esto se debe a que nunca tenemos en la cola vértices que son de más de dos niveles consecutivos, pues no se empieza a explorar a los vértices adyacentes a vértices del nivel $k+1$ (y por lo tanto, a meter vértices del nivel $k+2$) hasta que no se sacan de la cola a todos los vértices del nivel k.

Veamos ahora con más detalle la especialización de ExploracionAbstracta para obtener ExploracionBFS. Una vez especializada la frontera, pasamos a especializar los métodos que configuran el algoritmo, de la siguiente manera:

Definición de la clase ExploracionBFS

Para especializar a ExploracionBasica y obtener ExploracionBFS, lo único que tenemos que hacer es instanciar a la frontera como una cola – clase FronteraEnCola. En el Listado 4.2 mostramos los cambios que hacemos al constructor de ExploracionBFS, que se reducen a extender el constructor para que la frontera correcta tome su lugar.

Listado 4.2: Definición de la clase ExploracionBFS

```
1  public class ExploracionBFS extends ExploracionBasica {
2    /* Construye una instancia de ExploracionBFS,            *
3     * creando una frontera que es una cola.                 */
4    public ExploracionBFS(Digrafica g) {
5      super(g);
6      frontera = new FronteraEnCola();
7    }
8  }
```

Ninguno de los métodos abstractos de ExploracionAbstracta, implementados de manera básica en ExploracionBasica, se redefine o se extiende. Todos se heredan tal cual están en ExploracionBFS. Únicamente los métodos referentes al manejo de la frontera son redefinidos para que la frontera funcione como una cola.

frontera.elige() Dado que la frontera está especializada en una cola, el siguiente vértice a elegir será el que se encuentre al frente de la misma, y permanecerá como el elegido mientras no se le saque de frontera.

frontera.agrega(v): Se invoca el método propio de las colas para agregar elementos.

frontera.quitaNodo(v): Este método es invocado desde la línea [42 del Listado 3.8 en el Capítulo 3. Simplemente, como la frontera fue construida como una cola, el vértice que se elimina es el que está al frente de la cola.

4.3. Correctez de ExploracionBFS

Dado que ExploracionBFS es una especialización de ExploracionBasica, lo primero que debemos garantizar es que las redefiniciones o extensiones que hicimos en ExploracionBFS no provocan que se pierdan las propiedades que ya teníamos en la superclase, esto es:

i. Explora a toda los vértices alcanzables desde s y recorre a todos los arcos incidentes a estos vértices.
ii. Construye G_π correctamente..
iii. Calcula las distancias a cada uno de los vértices alcanzables desde s.

Estableceremos que estas propiedades se conservan enunciando algunos lemas. Observemos que en la especialización para ExploracionBFS lo único que cambiamos es la disciplina de acceso a la frontera, tanto para agregar como para sacar vértices de ella. Todos los métodos mantienen la implementación dada en ExploracionBasica.

Teorema 4.1 *La especialización ExploracionBFS, al invocar al método exploracionGenerica,*

A. Siempre termina.
B. Al terminar el algoritmo:
i. Exploró a todos los vértices alcanzables desde el origen s.
ii. Recorrió a todos los arcos incidentes en los vértices explorados.
iii. El árbol G_π es un árbol generador para la subgráfica que consiste de todos los vértices alcanzables desde s.
iv. El atributo v.d *registra la distancia desde* s *a* v *en G_π.*

Demostración:

La especialización ExploracionBFS hereda absolutamente todos los métodos de la clase ExploracionBasica, excepto la disciplina en la frontera. Pero aún así, la decisión de cuáles vértices entran a la frontera y cuáles son sacados de ella se establece independientemente de la disciplina de la frontera, por lo que estas decisiones las toma ExploracionBFS de la misma manera que las toma ExploracionBasica. Esto significa que hereda todas las propiedades para el atributo color, el atributo π y el atributo d, de los que depende que se cumpla el Lema 3.12, por lo que también se cumple en este caso. ❏

Invariantes para el atributo d

El atributo d calcula la longitud de un camino desde s, mientras que la distancia δ es la longitud de un camino más corto. Para ExploracionBasica establecemos que el valor del atributo d está acotado por abajo por el valor de la distancia al vértice origen de la exploración – Lema 3.17. Pero una de las propiedades adicionales de la exploración BFS es que el valor del atributo d es *exactamente* la distancia al origen. Procedemos a establecer que ExploracionBFS calcula bien $\text{v.d} = \delta(\text{s}, \text{v})$, mostrando únicamente que en esta especialización la relación $\text{v.d} \leq \delta(\text{s}, \text{v})$ se cumple. Para ello, conviene definir de manera más precisa el concepto de "capas", que se muestra en la Figura 4.1, mediante el cual el algoritmo va extendiendo la frontera.

Definición 4.1 Sea $V_k = \{\mathsf{v} \in V \mid \delta(\mathsf{s}, \mathsf{v}) = k\}$.

De la Definición 4.1, $V_0 = \{\mathsf{s}\}$, ya que s es el único vértice a distancia 0 de s.

Ahora estableceremos que en cualquier momento durante la ejecución del algoritmo, a lo más hay dos valores distintos para el atributo d de los vértices que se encuentran en la frontera, y su diferencia es 1, cuando la hay.

Lema 4.2 *Supongamos que durante la ejecución de explorationGenerica la cola contiene a* $\langle \mathsf{v}_{j_1}, \mathsf{v}_{j_2}, \ldots, \mathsf{v}_{j_r} \rangle$*, donde* v_{j_1} *es la cabeza de la cola y* v_{j_r} *es el último de la cola. Entonces.*

$$\mathsf{v}_{j_r}.\mathsf{d} \leq \mathsf{v}_{j_1}.\mathsf{d} + 1 \quad y \quad \mathsf{v}_{j_i}.\mathsf{d} \leq \mathsf{v}_{j_{i+1}}.\mathsf{d}, \qquad para\ \ \mathsf{i} = 1, \ldots, \mathsf{r} - 1$$

Demostración:

La demostración se hará por inducción en las operaciones que se realizan sobre Q, que corresponden a las iteraciones del método exploracionGenerica – líneas [33–42] del Listado 3.6.

Base: En el constructor de ExploracionBFS se construye una frontera vacía. En el método ponComoOrigen de la clase ExploracionBasica, se introduce a s a la frontera. Esto sucede antes de que se ejecute la iteración por primera vez. Cuando la ejecución del método llega por primera vez a la línea [33] del Listado 3.6, la frontera únicamente contiene a s, por lo que las desigualdades se cumplen trivialmente, independientemente de la clase que se haya construido para Q.

Inducción: Supongamos, como hipótesis de inducción, que en Q se encuentran $r \geq 1$ vértices para los que se cumplen las desigualdades de este lema. Tenemos que ver qué sucede, tanto al sacar un vértice como al meter un vértice a la cola, y establecer que en ambos casos se siguen cumpliendo las desigualdades que dimos antes.

Supongamos que v_{j_1} está al frente de la cola y en ese momento lo sacamos de la cola – línea [40] del Listado 3.6. Si v_{j_1} era el único vértice en Q y la cola se vacía, como no queda ningún vértice por examinar, la invariante se cumple. Supongamos que esto no sucede y que queda v_{j_2} como cabeza de la cola. Pero por hipótesis de inducción tenemos

$$\mathsf{v}_{j_r}.\mathsf{d} \leq \mathsf{v}_{j_1}.\mathsf{d} + 1$$

y como por la hipótesis de inducción se cumple que

$$\mathsf{v}_{j_1}.\mathsf{d} \leq \mathsf{v}_{j_2}.\mathsf{d} \leq \ldots \leq \mathsf{v}_{j_r}.\mathsf{d}$$

podemos simplemente sumarle 1 a cada miembro de la desigualdad, obteniendo

$$\mathsf{v}_{j_r}.\mathsf{d} \leq \mathsf{v}_{j_1}.\mathsf{d} + 1 \leq \mathsf{v}_{j_2}.\mathsf{d} + 1 \leq \ldots \leq \mathsf{v}_{j_r}.\mathsf{d} + 1$$

por lo que se sigue manteniendo

$$\mathsf{v}_{j_r}.\mathsf{d} \leq \mathsf{v}_{j_2}.\mathsf{d} + 1$$

y la relación que existe entre vértices consecutivos en la cola.

Veamos ahora qué pasa cuando metemos al vértice $v_{j_{r+1}}$ a la cola. Esto lo hacemos en explora(centroAccion, e), cuando v_{j_1} es el centro de acción. Esto quiere decir que $e = v_{j_1} \to v_{j_{r+1}} \in E$. Por la asignación que se hace en la línea [20] del método **descubriendo** en el Listado 3.8, tenemos que

$$v_{j_{r+1}}.d \;=\; v_{j_1}.d + e.peso \;=\; v_{j_1}.d + 1 \qquad \text{ya que } \forall e \in E, \quad e.peso = 1,$$

con lo que establecimos que se cumple la primera invariante, que dice que la estimación del último vértice de la cola es menor o igual a la estimación del vértice al frente de la cola incrementado en 1. Por otro lado,

$v_{j_r}.d \;\leq\; v_{j_1}.d + 1$	Hipótesis de inducción
$\quad\;\; = \; v_{j_{r+1}}.d$	Por la asignación en **descubriendo**

con lo que establecimos que se sigue cumpliendo la invariante entre vértices consecutivos en la cola. ❏

Deseamos ahora mostrar que si $v \in V_k$, entonces $v.d = k$, con V_k de acuerdo a la Definición 4.1.

Lema 4.3 *Sea $G = (V, E)$ una digráfica y supongamos que se ejecuta explorac ionGenerica – de la subclase ExploracionBFS – sobre G desde un vértice origen $s \in V$. Entonces,*

i. Al terminar exploracionGenerica de ExploracionBFS, $v.d = \delta(s, v)$.
ii. Más aún, $\forall v \in V$ alcanzable desde s, un camino más corto $s \rightsquigarrow v$ es un camino más corto $s \rightsquigarrow v.\pi$ seguido del arco $v.\pi \to v$.

Demostración:

Dado que $v \in V_k$ es equivalente a decir que $\delta(s,v)=k$, lo que queremos demostrar es

$$v \in V_k \iff v.d = k.$$

Si no existe ningún camino $s \rightsquigarrow v$, entonces $\delta(s.v) = \infty$. Como v no es alcanzable desde s el único momento en que se le va a asignar valor al atributo v.d es en el constructor de la gráfica – línea [7] del Listado 3.1 – quedando $v.d = \infty$, lo que satisface que $\delta(s, v) = v.d$.

Para aquellos vértices alcanzables desde s, demostraremos este lema por inducción sobre k, la distancia de s a v.

<u>Base:</u> Para $k = 0$, Sabemos que $V_0 = \{s\}$.

i. Por demostrar: $\delta(s, s) = s.d = 0$.
   - Al construirse la gráfica para dársela a ExploracionBFS, todos los vértices toman el valor ∞.
   - s.d toma el valor 0 en el método ponComoOrigen.
   - Como s no vuelve a ser pintado de GRIS (Lema 3.4), esto sucede una única vez y 0 es el valor que se queda en s.d.

   $\therefore \forall v \in V_0, \quad 0 = \delta(s, v) = \delta(s, s) = s.d$, cumpliéndose la primera parte del lema.

ii. Por demostrar: que un camino más corto $s \rightsquigarrow s$ pasa por $s.\pi$. Como $s.\pi$ es nulo, este inciso se cumple por vacuidad.

Inducción: Suponemos como hipótesis de inducción que para vértices en V_j con $j < k$ se cumple lo siguiente:

i. $j = \delta(\mathsf{s}, \mathsf{u}) = \mathsf{u.d}, \quad \forall \mathsf{u} \in V_j$.

ii. Un camino más corto a un vértice $\mathsf{u} \in V_j$ pasa por $\mathsf{u}.\pi$.

Y demostraremos estos dos incisos para vértices a distancia k.

i. Por demostrar: $\forall \mathsf{v} \in V_k$, $k = \delta(\mathsf{s}, \mathsf{v}) = \mathsf{v.d}$.
 - Como $\mathsf{v} \in V_k$, v es alcanzable desde s – tiene un valor finito para $\delta(\mathsf{s}, \mathsf{v})$ – y $\exists$ un camino s⇝v, por lo que v es pintado de GRIS en algún momento y metido a Q.
 - Cuando v es metido a Q, todos los vértices de V_{k-1} ya fueron metidos a Q. Veamos por qué, mostrando que $\forall \mathsf{u} \in V_{k-1}$, u fue metido a Q antes que $\mathsf{v} \in V_k$:
   – Por la hipótesis de inducción, $\mathsf{u.d} = k - 1$.
   – La asignación al atributo d de un valor distinto de ∞ sucede únicamente una vez, cuando el vértice es descubierto; esto es, cuando cambia de BLANCO a GRIS; y esto sucede únicamente una vez para los vértices alcanzables desde s – Lema 3.4.
   – Por el Lema 3.17, $\mathsf{v.d} \geq k$.
   – Supongamos, por contradicción, que u entró a Q después que v. Por el Lema 4.4, para todos los vértices que ingresen a Q después que v, se cumplirá que $\mathsf{v.d} \leq \mathsf{u.d} = k-1$, esto es $\mathsf{u.d} < k$, contradiciendo al Lema 3.17, que dice que $\mathsf{v.d} \geq k$.

   $\therefore$ cuando entra v a Q, todos los vértices en V_j, $j < k$, ya ingresaron a Q.
 - Debido a que $\mathsf{v} \in V_k$, $\exists \mathsf{u} \in V_{k-1}$ tal que $\mathsf{u} \rightarrow \mathsf{v} \in E$. $\therefore$ u es introducido a Q y pintado de GRIS antes que v. Sea u el primer vértice en V_{k-1} con $\mathsf{u} \rightarrow \mathsf{v}$ que entra a Q. Observemos entonces:
   – u debe aparecer en algún momento en la cabeza de la cola Q y ser el siguiente vértice a ser elegido.
   – Como u es el primer vértice de los que tienen algún arco que va a v, cuando u aparece en la cabeza de la cola v no ha sido descubierto todavía.
   – No se puede terminar de explorar a u sin descubrir a v.
   – Se va a descubrir a v siendo u centro de acción, por lo que en ese momento se harán las siguientes asignaciones:

```
19          u.setPi(centroAccion);
20          u.setD(centroAccion.getD() + e.getPeso());
21          u.setAlcanzado();
```

   con lo que queda demostrado el primer inciso.

ii. De lo anterior, podemos ver que $\mathsf{v} \in V_k \Rightarrow \mathsf{v}.\pi \in V_{k-1}$, ya que partimos de elegir a $\mathsf{u} = \mathsf{v}.\pi \in V_{k-1}$.

$\therefore$ podemos encontrar un camino más corto s⇝v yéndonos por un camino más corto s⇝v.π, y luego por el arco v.$\pi \rightarrow$v. ❑

4.3.1. El árbol G_π

Antes de enunciar el teorema de correctez para ExploracionBFS, trabajemos con el atributo v.π. Recordemos la definición de G_π, construido por ExploracionBasica (y por herencia, por

ExploracionBFS):

$$G_\pi = (V_\pi, E_\pi) \quad \text{donde}$$
$$V_\pi = \{\mathsf{v} \in V \mid \mathsf{v} \text{ es alcanzable desde } \mathsf{s}\}, \qquad E_\pi = \{\mathsf{v}.\pi \to \mathsf{v} \in E \mid \mathsf{v} \in V - \{\mathsf{s}\}\}$$

Definición 4.2 (Árbol BFS) El árbol de padres G_π es un *árbol* BFS si el único camino que existe entre s y cada uno de los vértices $\mathsf{v} \in V_\pi$ es un camino más corto de s a v en G.

Lema 4.4 *Después de ejecutar* BFS *sobre una gráfica $G = (V, E)$, el árbol G_π es un árbol* BFS.

Demostración:

Sabemos que los vértices que están en V_π son aquellos que tienen asignada una distancia finita, ya que cuando se les asignó el valor al atributo π también se le asignó valor al atributo d. Como G_π es un árbol – Lema 3.15 – existe un camino único entre s y $\mathsf{v} \in V_\pi$. Por el Lema 4.5, un camino más corto entre s y v pasa por $\mathsf{v}.\pi$, por lo que el camino en G_π es un camino más corto $\mathsf{s} \rightsquigarrow \mathsf{v}$, ya que el valor de v.d está dado precisamente por el camino sobre G_π. ❏

4.3.2. Correctez de ExploracionBFS

Estamos ahora sí, listos para enunciar el teorema de correctez de ExploracionBFS:

Teorema 4.5 (Correctez de ExploracionBFS) *Sea $G = (V, E)$ una digráfica y $\mathsf{s} \in V$ un vértice arbitrario de G. Entonces, al terminar el algoritmo se cumple que:*

i. $\forall \mathsf{v} \in V, \exists \mathsf{s} \rightsquigarrow \mathsf{v} \iff$ v.color = NEGRO; *y* $\forall \mathsf{e} \in E, \mathsf{v}, \mathsf{x} \in V$, $\mathsf{e} = \mathsf{v} \to \mathsf{x}$ *con* v.color = NEGRO $\iff$ e.recorrido = SI.

*(*BFS *exploró a todos los vértices de G alcanzables desde* s *y recorre a todos los arcos que inciden desde los vértices explorados).*

ii. $\forall \mathsf{v} \in V$, $\mathsf{v.d} = \delta(\mathsf{s}, \mathsf{v})$.

iii. *G_π es un árbol de caminos más cortos desde* s *a cada uno de los vértices explorados (es un árbol* BFS*).*

Demostración:

Usando las propiedades de ExploracionBasica y usando los lemas que vimos para propiedades de distancias, quedan demostrados los incisos i a iii de este teorema. ❏

4.4. Complejidad de ExploracionBFS

Enunciamos como teorema la clase de complejidad a la que ExploracionBFS pertenece:

Teorema 4.6 *Sea $G = (V, E)$ una digráfica y $\mathsf{s} \in V$ un vértice arbitrario de G. Entonces, ExploracionBFS es $O(|V| + |E|)$.*

Demostración:

Retomemos la fórmula que dimos en el Capítulo 3:

$$\begin{aligned} \text{Complejidad de ExploracionAbstracta} &= \mathsf{f}_{\mathsf{constr}}(\mathsf{n},\mathsf{m}) + \mathsf{f}_{\mathsf{daS}}(\mathsf{n}) + \mathsf{f}_{\mathsf{marcaS}}(\mathsf{n}) + \\ &+ \sum_{n+m}\Big(\mathsf{f}_{\mathsf{F.noVacia}}(\mathsf{n}) + \mathsf{f}_{\mathsf{elige}}(\mathsf{n}) + \mathsf{f}_{\mathsf{cAccion}}(\mathsf{n},\mathsf{m})\Big) + \\ &+ \sum_{m}\Big(\mathsf{f}_{\mathsf{masAr}}(\mathsf{n},\mathsf{m}) + \mathsf{f}_{\mathsf{daAr}}(\mathsf{m}) + \mathsf{f}_{\mathsf{explora}}(\mathsf{n},\mathsf{m})\Big) + \\ &+ \sum_{n}\mathsf{f}_{\mathsf{cierra}}(\mathsf{n}) + \mathsf{f}_{\mathsf{fin}}(\mathsf{n},\mathsf{m}). \end{aligned}$$

donde $\mathsf{f}_{\mathsf{explora}}(\mathsf{n},\mathsf{m})$ se descompone de la siguiente manera:

$$\begin{aligned} \sum_{m}\mathsf{f}_{\mathsf{explora}}(\mathsf{n},\mathsf{m}) = &\sum_{m}\Big(\mathsf{f}_{\mathsf{getOtroN}}() + \mathsf{f}_{\mathsf{proceAr}}() + \mathsf{f}_{\mathsf{yaUsado}}()\Big) + \\ &+ \sum_{n}\mathsf{f}_{\mathsf{descubriendo}}(\mathsf{n}) + \sum_{m-n}\mathsf{f}_{\mathsf{yaDescubierto}}(\mathsf{n}). \end{aligned}$$

Los cálculos para cada uno de los componentes de esta ecuación en el caso de ExploracionBasica es la siguiente:

$$\begin{array}{lcrcl} \mathsf{f}_{\mathsf{daS}}(\mathsf{n}) & = & c_o & = & O(1) \\ \mathsf{f}_{\mathsf{constr}}(\mathsf{n},\mathsf{m}) & = & 12 & = & O(1) \\ \mathsf{f}_{\mathsf{marcaS}}(\mathsf{n}) & = & 5 & = & O(1) \\ \mathsf{f}_{\mathsf{F.noVacia}}(\mathsf{n}) & = & 1 & = & O(1) \\ \mathsf{f}_{\mathsf{elige}}(\mathsf{n}) & = & 2 & = & O(1) \\ \mathsf{f}_{\mathsf{cAccion}}(\mathsf{n},\mathsf{m}) & = & 0 & = & -- \\ \mathsf{f}_{\mathsf{masAr}}(\mathsf{n},\mathsf{m}) & = & 3 & = & O(1) \\ \mathsf{f}_{\mathsf{daAr}}(\mathsf{m}) & = & 2 & = & O(1) \\ \mathsf{f}_{\mathsf{getOtroN}}() & = & 3 & = & O(1) \\ \mathsf{f}_{\mathsf{proceAr}}() & = & 1 & = & O(1) \\ \mathsf{f}_{\mathsf{yaUsado}}() & = & 1 & = & O(1) \\ \mathsf{f}_{\mathsf{descubriendo}}(\mathsf{n}) & = & 17 & = & O(1) \\ \mathsf{f}_{\mathsf{yaDescubierto}}(\mathsf{n}) & = & 0 & = & -- \\ \mathsf{f}_{\mathsf{cierra}}(\mathsf{n}) & = & 0 & = & -- \\ \mathsf{f}_{\mathsf{fin}}(\mathsf{n},\mathsf{m}) & = & c_1(n+m) & = & O(n+m) \end{array}$$

con lo que la complejidad de $\mathsf{f}_{\mathsf{explora}}(\mathsf{n},\mathsf{m})$ para ExploracionBasica se dio como

$$\sum_{\mathsf{m}}\mathsf{f}_{\mathsf{explora}}(\mathsf{n},\mathsf{m}) = \sum_{\mathsf{m}} 5 + \sum_{\mathsf{n}} 17 + \sum_{\mathsf{m-n}} 0 = 5\mathsf{m} + 17\mathsf{n} = \mathbf{O(n+m)}$$

Para la especialización dada en ExploracionBFS lo único que se hizo es redefinir la frontera, y con ello el cálculo de complejidad de los siguientes métodos:

$f_{constr}(n, m)$	Inicialización de frontera.
$f_{marcaS}(n)$	Agregar a s a la frontera.
$f_{F.noVacia}(n)$	Involucra directamente a la frontera.
$f_{elige}(n)$	Depende del acceso a la frontera.
$f_{descubriendo}(n)$	Involucra agregar a la frontera.

Sin embargo, dado que la frontera está implementada como una cola en una lista ligada, todos estos métodos conservan la complejidad dada para ExploracionBasica, y la clase de complejidad a la que pertenece ExploracionBFS es exactamente la misma que la clase a la que pertenece ExploracionBasica:

ExploracionBFS $\in O(|V| + |E|)$

4.5. ExploracionBFS en gráficas no dirigidas

En realidad no hay mucho que agregar respecto a BFS cuando lo aplicamos a una gráfica no dirigida, sobre todo en lo que respecta al teorema de correctez. Las propiedades de BFS para digráficas, principalmente las que tienen que ver con el comportamiento de la frontera, siguen siendo válidas, por lo que se heredan al algoritmo cuando lo aplicamos a gráficas no dirigidas. Hay que cuidar, como en el caso del ExploracionBasica, que las aristas sean recorridas una única vez.

Una de las razones por las que ExploracionBFS se comporta prácticamente igual en gráficas dirigidas que no dirigidas es el hecho de que al aplicar ExploracionBFS a una gráfica no dirigida estamos orientando a la gráfica, ya que cada arista va a tomar la dirección que le asigne ExploracionBFS. Esto es importante, sobre todo, en el contexto del árbol G_π que determina caminos más cortos entre el vértice origen y cualquiera de los vértices alcanzables de la gráfica.

Es interesante revisar, en este caso, qué sucede con las aristas que no están incluidas en G_π. Supongamos que nos encontramos explorando a un vértice $u \in V$ y elegimos a la arista e=u−v donde el vértice v fue descubierto en una exploración anterior. Si esto sucede, resulta que la diferencia entre $\delta(s, u)$ y $\delta(s, v)$ es a lo más de 1:

Lema 4.7 *Sea $G = (V, E)$ una gráfica no dirigida sobre la que se termina de ejecutar exploracionGenerica, y sea $e = u{-}v \in E$ tal que $e \notin E_\pi$. Entonces u.d y v.d difieren a lo más en una unidad.*

Demostración:

Supongamos que en un momento dado, exploracionGenerica se encuentra explorando al vértice u que se encuentra al frente de Q y elige la arista e=u−v para recorrer. Esta arista

no es ingresada a G_π en el caso de que v no esté pintado de blanco. Si v.color=GRIS, entonces, por las propiedades de ExploracionBasica, v se encuentra en Q, y por el Lema 3.5, como tanto u como v están en Q, y como u está en el frente de Q, v.d $\leq$ u.d+1 – Lema 4.2.

Supongamos que v.color=NEGRO (ya no está en Q). Observemos que:

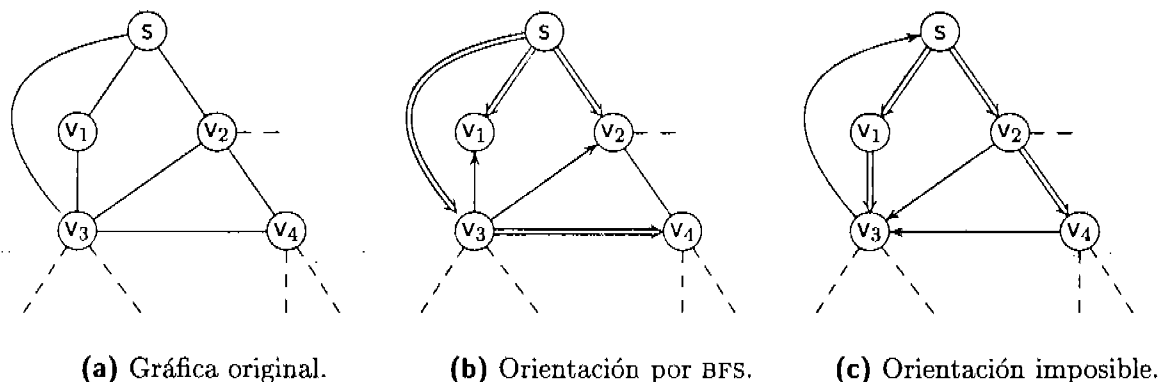
- Si ya no está en Q, quiere decir que ya fue explorado.
- Si ya fue explorado, todas las aristas incidentes a v ya fueron recorridas.

$\therefore$ e=u−v no pudo ser seleccionada, lo que nos lleva a una contradicción.

De lo anterior, si e no es incluida en G_π es porque el otro extremo de e, v, se encuentra en Q (v.color=GRIS) y entonces su distancia al origen es, a lo más, 1 más que la distancia al origen desde u. ❑

Lo que nos dice el lema anterior es que ninguna arista a la que orientemos "hacia atrás" (hacia un vértice ya descubierto) puede brincar más de un nivel. Por supuesto que puede haber más de una orientación de la gráfica original, dependiendo del vértice origen y del orden de las aristas en las listas de incidencias de los vértices. Veamos la Figura 4.3, donde se muestra la gráfica original, Figura 4.4(a), seguida de una orientación posible en la Figura 4.4(b), donde los arcos dobles son los de G_π mientras que los arcos sencillos son los que no quedaron incluidos, pero que fueron orientados. Finalmente, en la Figura 4.4(c) se muestra una orientación que no pudo haber sido generada por BFS.

Figura 4.3: Aristas no incluidas en G_π



(a) Gráfica original. **(b)** Orientación por BFS. **(c)** Orientación imposible.

4.5.1. Complejidad de ExploracionBFS en gráficas no dirigidas

Como en el caso de ExploracionBasica, y dado que se consideró el caso en que una misma arista aparezca en dos listas de incidencia, la clase de complejidad a la que pertenece ExploracionBFS cuando es aplicada a gráficas no dirigidas es la misma que en el caso de digráficas.

4.6. Aplicaciones de ExploracionBFS

En esta sección presentaremos varios problemas que se resuelven aplicando el algoritmo ExploracionBFS.

4.6.1. Gráficas bipartitas[2]

Podemos dar una especialización del algoritmo ExploracionBFS que determine de manera eficiente si una gráfica es bipartita o no. La idea general es la siguiente:

Procedemos a asignar a los vértices de G en dos conjuntos ajenos, S_1 y S_2, tales que $V = S_1 \cup S_2$. Agregamos a los vértices un atributo conjunto cuyo valor posible es 0 o 1. Si el valor de este atributo al terminar el algoritmo es 0, colocaremos a ese vértice en el conjunto S_2, y si es 1 lo colocaremos en S_1. La regla para asignar valor a conjunto es la siguiente:

$$\mathsf{v.conjunto} = \begin{cases} 0 & \text{si } \mathsf{v.d} \bmod 2 = 0 \\ \\ 1 & \text{si } \mathsf{v.d} \bmod 2 = 1 \end{cases}$$

Especializamos el algoritmo ExploracionBFS para que determine si una gráfica es bipartita o no (nos referiremos a él como ExploracionBFSBip) asignando a cada uno de los vértices de la gráfica a uno de dos conjuntos, S_1 y S_2 conforme el algoritmo los va descubriendo (cuando los mete a Q), de la siguiente manera:

i. Al vértice s se le coloca en el conjunto S_2 en ponComoOrigen – se redefine este método en ExploracionBFSBip.
ii. A cada vértice v del árbol G_π, en el momento en que es descubierto, se le coloca en el conjunto en el que no está $\mathsf{v}.\pi$.

El valor del atributo conjunto debe ser asignado a cada vértice una única vez durante el proceso, y resulta conveniente hacerlo al descubrir al vértice. En ese momento es cuando se tiene la información del conjunto al que está asignado su padre, que es adyacente a él. Por ello, es el método descubriendo el que se debe extender. Sin embargo, preferimos no extender a la clase Nodos con el atributo conjunto, pues eso acarrearía extender también a Digráfica para que esté compuesta por nodos con este atributo. Por lo tanto, simplemente creamos una estructura paralela a la de los nodos, con un único atributo, donde el atributo para el nodo v_i se encuentra en la misma posición relativa que el vértice v_i. Podemos ver, por lo pronto, como quedan ponComoOrigen y descubriendo, además de la declaración de la estructura que acabamos de mencionar en el Listado 4.3.

Al terminar de ejecutarse ExploracionBFSBip tendremos una asignación de los vértices de V relativa al árbol G_π, y como la asignación se hizo por niveles, no tenemos dos vértices adyacentes *en el árbol* que se encuentren en el mismo conjunto. Sin embargo, hasta ahora

[2]Una exposición detallada de lo que son las gráficas bipartitas y algunas de sus propiedades se presentan en el Apéndice C.

Listado 4.3: Extensión de ExploracionBFS a ExploracionBFSBip

```
 1 class ExploracionBFSBip extends ExploracionBFS {
 2         /* Para marcar a cuál conjunto pertenece.                     */
 3     int[] conjunto;
 4
 5         /* Constructor. Agrega la estructura de datos paralela.      */
 6     public ExploracionBFSBip(Digrafica G) {
 7         super(G);
 8         /* Hereda el valor de N, que es el número de nodos          *
 9          * en la gráfica                                            */
10         conjunto = new int[N + 1];
11     }
12         /* Redefinición de ponComoOrigen.                            */
13     public void ponComoOrigen(Nodos s) {
14         super.ponComoOrigen(s);
15         int i = s.getPos();    /* lugar de s en la estructura       */
16         conjunto[i] = 0;
17     }
18         /* Redefinición de descubriendo.                             */
19     protected void descubriendo(Nodos centroAccion, Arcos e,
20                                                   Nodos u) {
21         super.descubriendo(centroAccion, e, u);
22         int p = centroAccion.getPos(); /* lugar del padre           */
23         int h = u.getPos();            /* lugar del descubierto.    */
24         conjunto[h] = (conjunto[p] + 1) % 2;
25     }
36 }
```

no hemos tomado en cuenta las aristas que no quedaron en E_π. El problema con ellas es que los extremos de una o más de estas aristas pudieran estar en el mismo conjunto. Una arista no queda en E_π si cuando se le recorre desde el centro de acción v_a, el vértice u en el otro extremo de la arista no está pintado de BLANCO. Tanto en ExploracionBasica como en ExploracionBFS no se hace nada en este caso, pero en ExploracionBFSBip éste es el punto donde debemos verificar los conjuntos a los que están asignados los extremos de la arista en cuestión.

Dado que v_a está asignado ya a uno de los dos conjuntos (estaba en Q) y que u también está asignado ya a uno de los conjuntos (se le asignó en el momento en que fue descubierto), podemos simplemente ver si están o no asignados al mismo conjunto. Si sí, la gráfica no es bipartita ya que tenemos una arista cuyos extremos están en el mismo conjunto.

Si suspendemos la ejecución del algoritmo cuando éste se encuentre con la situación que acabamos de describir, emitiendo un mensaje que así lo indique, tendremos una manera de determinar si la gráfica es bipartita. La modificación que se debe hacer al algoritmo consiste en redefinir el método yaDescubierto, que hasta el momento no hacía nada, para que detecte si la arista que se está recorriendo es entre vértices asignados al mismo conjunto. El método redefinido se puede ver en el Listado 4.4. Se encuentra, por supuesto, en la clase

ExploracionBFSBip.

Listado 4.4: Detección de gráficas no bipartitas

```
27        /* Definición de yaDescubierto.                                  */
28     protected void yaDescubierto(Nodos centroAccion, Arcos e,
29                                                       Nodos u) {
30         int vPos = centroAccion.getPos();
31         int uPos = u.getPos();
32         if (conjunto[vPos] == conjunto[uPos]) {
33             throw new GraficaNoBipartitaException();
34         }
35     }
```

Una vez caracterizadas las gráficas bipartitas[3], debemos demostrar que la asignación que estamos haciendo de los vértices en la clase ExploracionBFSBip en efecto asigna bien a los vértices a los conjuntos S_1 y S_2. Enunciamos esto en el siguiente teorema:

Teorema 4.8 *Una gráfica $G = (V, E)$ es bipartita $\iff$ la invocación a explorationGenerica desde un objeto de la clase ExploracionBFSBip termina su ejecución sin reportar que la gráfica no es bipartita.*

Demostración:

$\Longrightarrow$ Supongamos que G es bipartita y debemos demostrar que el algoritmo termina sin reportar que G no lo es. Por contrapositivo, supongamos que exploracionGenerica (desde un objeto de la clase ExploracionBFSBip) reporta que la gráfica no es bipartita.

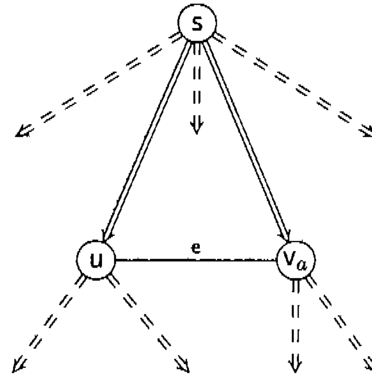
ExploracionBFSBip reporta que G no es bipartita si al elegir una arista $\mathsf{e} = \mathsf{v}_a\text{—}\mathsf{u}$ para recorrer, el vértice u no está pintado de BLANCO y $\mathsf{v_a}$.conjunto = u.conjunto. Esto sólo puede ocurrir si v_a y u se encuentran en el mismo nivel del árbol, o bien la arista va de un nivel impar a otro impar, o de un nivel par a otro par.

Supongamos que e incide en dos vértices del mismo nivel del árbol BFS, digamos k. e no va a ser incluida en E_π porque u ya fue descubierto. Tenemos la situación que se presenta en la Figura 4.4.
Obviamente, dado que u y v_a están en el nivel k, hay k aristas en el árbol de s a u y de s a v_a, por lo que tenemos $2k$ aristas en el camino que va de u a v_a pasando por s. Como además tenemos la arista e=u—v_a tenemos el ciclo $\mathsf{u} \overset{k}{\rightsquigarrow} \mathsf{s} \overset{k}{\rightsquigarrow} \mathsf{v}_a\text{—}\mathsf{u}$ de longitud impar, lo que contradice el que G sea bipartita. Por lo que si ExploracionBFSBip reporta que la gráfica no es bipartita en esta situación, en efecto no lo es.

Por el Lema 4.7 las aristas que no estén en G_π no pueden saltar más de un nivel, por lo que no pueden ir de un nivel par a otro par, ni de un nivel impar a otro impar (a menos que sea el mismo nivel, caso que ya vimos), sino que tienen que ir de uno par al inmediato impar posterior, o de uno impar al inmediato par posterior. Supongamos que v_a se encuentra en el nivel k. Entonces, si e brinca un nivel, u tiene que estar en el nivel $k+1$ (u no puede estar en el nivel $k-1$ porque existiría una arista incidente en un vértice del nivel $k-1$ que no ha sido recorrida, pero antes de empezar a explorar los vértices del nivel k tienen que

[3]Ver Apéndice C.

Figura 4.4: Aristas que cruzan en el mismo nivel



haberse explorado - pintado de NEGRO - todos los vértices de nivel $k-1$). Al evaluarse la condición de la línea [32] del método yaDescubierto en el Listado 4.4, el atributo conjunto va a tener distinto valor para v_a y u, por lo que la ejecución seguirá adelante. Si bien este tipo de aristas cierran ciclos de longitud par - $\delta(s, v_a) = k$, $\delta(s, u) = k+1$, por lo que la longitud del ciclo $u \rightsquigarrow s \rightsquigarrow v_a - u$ es $k + k + 1 + 1 = 2(k+1)$ - esto no implica que la gráfica no sea bipartita y la ejecución puede continuar. Como el nivel de v_a es k y el de u es $k+1$, tienen distinto valor en el atributo conjunto, por lo que la condición en la línea [32] del Listado 4.4 en la clase ExploracionBFSBip no se cumple para este tipo de aristas.

De lo anterior, la única vez que exploracionGenerica en la clase ExploracionBFSBip va a reportar que la gráfica no es bipartita es cuando tenga una arista entre vértices del mismo nivel que, como ya vimos, constituye un ciclo de longitud impar.

$\Longleftarrow$ Supongamos ahora que el algoritmo termina su ejecución sin emitir el mensaje referido. Debemos demostrar que G es bipartita, y eso lo logramos demostrando que no tiene ciclos impares.

Las aristas de G se clasifican en dos grupos: E_π y $E - E_\pi$. En G_π no hay ciclos, ya que es un árbol. Al agregarle a G_π cualquiera de las aristas de $E - E_\pi$ formamos un ciclo (propiedad de los árboles). Pero por el Lema 4.7 estas aristas sólo pueden conectar vértices en el mismo nivel o vértices en niveles consecutivos. Si una arista $e \in E - E_\pi$ conecta vértices en el mismo nivel, quiere decir que hay ciclos de longitud impar en G, y el algoritmo emite el mensaje; pero como la hipótesis es que el algoritmo termina sin emitir el mensaje no debemos considerar este caso acá. Si conecta vértices de niveles consecutivos, el ciclo que se forma es par, lo que no se contrapone a que la gráfica sea bipartita. De esto, si el algoritmo termina es porque no emitió el mensaje, y si no emitió el mensaje es porque no encontró ciclos de longitud impar. ❑

Terminamos esta sección con el Corolario 4.9:

Corolario 4.9 *Existe un algoritmo de complejidad $O(|E| + |V|)$ para decidir si una gráfica es bipartita.*

Demostración:

En la especialización ExploracionBFSBip redefinimos al constructor y a los métodos ponComoOrigen, descubriendo y yaDescubierto. En el primero, tenemos únicamente la construcción

del arreglo para el atributo conjunto, lo que lleva a lo más $O(n)$, influyendo únicamente en el coeficiente para n en la fórmula de complejidad; en el método ponComoOrigen agregamos dos asignaciones, que contribuyen también con una constante a la complejidad de este método, similar a lo que agregamos en descubriendo.

En el caso del método ya Descubierto, que en ExploracionBFS no hacía nada, agregamos dos asignaciones de costo constante, y la prueba para ver si se cierra un ciclo de longitud impar, lo que agrega a este método una complejidad de orden constante. En resumen, la especialización para ExploracionBFSBip únicamente aumenta los coeficientes de la fórmula dada para la complejidad de ExploracionBFS, por lo que su complejidad es también $O(|V|+|E|)$. ❑

4.6.2. Diámetro* de un árbol

Podemos utilizar a ExploracionBFS para obtener el diámetro de un árbol de la siguiente manera:

i. Ejecutamos exploracionGenerica de ExploracionBFS una vez usando como origen a cualquiera de los vértices del árbol.
ii. Sea u el último vértice explorado en esta primera ejecución.
iii. Ejecutamos exploracionGenerica de ExploracionBFS por segunda vez, con u como origen.
iv. Sea v el último vértice explorado en esta segunda ejecución.
v. El diámetro de G está dado por $\delta(\mathsf{u}, \mathsf{v})$.

La complejidad de este algoritmo es $O(|V| + |E|)$, ya que ejecutamos exploracionGenerica únicamente dos veces.

En el Lema D.1 se demuestra la correctez de esta especialización. Como esta demostración no hace uso de la manera cómo especializamos a BFS, sino a las propiedades del algoritmo, no la presentamos acá.

4.7. Conclusiones

El funcionamiento de BFS encaja perfectamente en el esquema genérico que vimos, facilitando de manera significativa la demostración de las propiedades de BFS. Más aún, algunas de estas propiedades se presentan ya en el algoritmo genérico, como la construcción del árbol o el cálculo de las distancias en términos de la distancia del vértice padre. Este enfoque da una visión muy intuitiva del funcionamiento de la especialización, facilitando enormemente la reutilización de las demostraciones de correctez dadas para el algoritmo genérico.

* Ver Apéndice C para las definiciones necesarias y el Apéndice D para la demostración de que ExploracionBFS calcula de manera correcta el diámetro de un árbol.

Capítulo 5

Exploración a profundidad (DFS[1])

En este capítulo implementaremos al algoritmo de exploración a profundidad que identificamos con el nombre DFS (del inglés **D**epth **F**irst **S**earch) como una especialización más de ExploracionBasica. El algoritmo DFS elige el centro de acción de tal manera de "bajar" por un camino en la gráfica y llegar tan lejos como sea posible. Al llegar a un vértice del que ya no se puede salir – porque no tiene arcos que salgan de él o porque los arcos de ese vértice van hacia vértices ya descubiertos – regresaremos al último visitado antes que éste, para intentar extender el camino desde él. El algoritmo termina cuando regresamos al origen y ya no se puede volver a salir de él. Revisaremos tanto el caso de gráficas no dirigidas como de gráficas dirigidas.

Revisaremos la implementación de esta especialización usando una pila implícita, dada por las referencias al padre, y que requiere de únicamente un registro para representar a toda la pila. A continuación daremos propiedades que se dan con esta especialización, como la propiedad de paréntesis. Terminaremos mencionando aplicaciones interesantes de DFS.

DFS proporciona información importante respecto a la estructura de la gráfica que no es proporcionada por BFS y se utiliza en muchas aplicaciones. Por ejemplo, para determinar caminos de ciertas características, como los que encuentran la salida en un laberinto o el camino simple más largo posible desde un vértice dado (la *excentricidad* del vértice). Esta especialización también nos permite determinar las componentes no separables en una gráfica – aunque esta propiedad no es tan fácil de intuir como la existencia de ciclos.

Se presenta primero la versión del algoritmo aplicada a digráficas y algunas aplicaciones interesantes, para pasar después, como lo hemos estado haciendo, a la versión del algoritmo para gráficas no dirigidas.

[1] Usaremos el nombre DFS, mejor reconocido en la disciplina.

5.1. La especialización ExploracionDFS

En esta presentación supondremos, a menos que indiquemos lo contrario, que tenemos una digráfica G tal que $\forall \mathsf{v} \in V$, v es alcanzable desde s. Suponemos también que la función $peso : E \to \mathbb{N}$ asigna el valor 1 a cada arco, como en el caso de ExploracionBFS. En la exploración a profundidad el algoritmo busca "llegar tan lejos como pueda", saliendo de un vértice dado y alcanzando algún vértice adyacente a él que no haya sido descubierto. Decimos que un vértice tiene una ***salida disponible*** cuando tiene algún arco incidente desde él que no ha sido recorrido y el vértice destino de ese arco no ha sido descubierto. Cuando se llega a un vértice que ya no tiene salidas disponibles, debe "retroceder" (*backtrack*) por el camino que lleva recorrido, hacia el vértice desde el cuál éste fue descubierto, hasta encontrar el primer vértice en ese camino de retroceso que tiene una salida disponible.

ExploracionBasica se especializa en ExploracionDFS cuando la frontera es una pila; esto garantiza el orden de exploración descrito arriba. Más adelante estableceremos esto y otras propiedades más.

La estructura que proponemos para frontera es una estructura de pila, como ya mencionamos – ver Listado 5.1 – en la que el último vértice descubierto es el siguiente desde el cual se va a intentar salir. Una vez explorado el vértice en el tope de la pila, éste se saca y el padre de éste pasa a ser el nuevo centro de acción. Se prosigue la exploración desde este vértice.

Listado 5.1: Extensión a la clase FronteraEnPila

```
1  public class FronteraEnPila extends FronteraBasica {
2      /* Construye una instancia de la frontera con disciplina   *
3       * de una pila. La inicia vacía.                            */
4      public FronteraEnPila() { /* ... */ }
5       /* Agrega un nodo a la pila.                               */
6      public void agregaNodo(Nodos v) { /* ... */ }
7  }
```

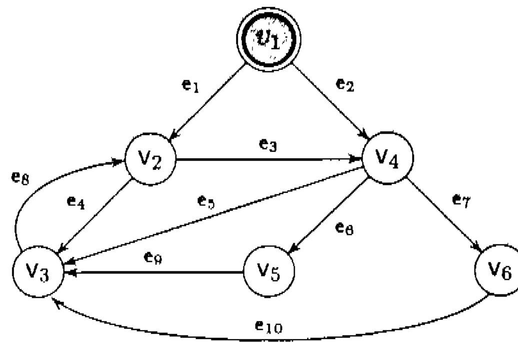
La especialización de ExploracionBasica para DFS es, simplemente, en el constructor de ExploracionDFS hacer que frontera sea un objeto de la clase FronteraEnPila. El código se puede ver en el Listado 5.2.

Listado 5.2: Especialización de ExploracionBasica a ExploracionDFS

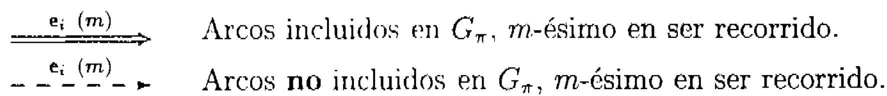
```
1 public class ExploracionDFS extends ExploracionBasica {
2        /* Construye el objeto para la exploración, pero asig-   *
3         * nando a la frontera un objeto de FronteraEnPila.      */
4     public ExploracionDFS(Digrafica g) {
5         super(g);
6         frontera = new FronteraEnPila();
7     }
8 }
```

Ejemplo Procedamos a ver un ejemplo de este tipo de exploración, usando para ello la gráfica en la Figura 5.1, con $s = v_1$. Por lo pronto no especializaremos a los métodos del algoritmo, sino que únicamente observaremos la forma que toma la exploración cambiando la disciplina de frontera a una disciplina de pila.

Figura 5.1: Digráfica a explorar con ExploracionDFS: al inicio de la estrategia genérica

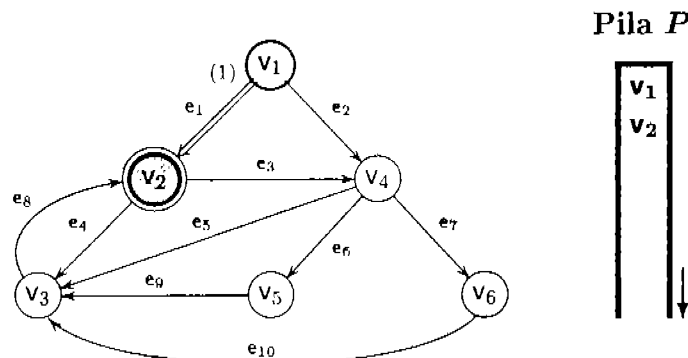


La ejecución del algoritmo se inicia con $s = v_1$, el centro de acción. En las figuras que siguen, los arcos que se incluyen en G_π estarán identificados con lineas dobles, y con línea intermitente aquéllos que no se incluyen en G_π, con el vértice que es el centro de acción encerrado en doble círculo. Indicaremos el orden en que se van recorriendo los arcos a continuación de la etiqueta del arco, entre paréntesis:



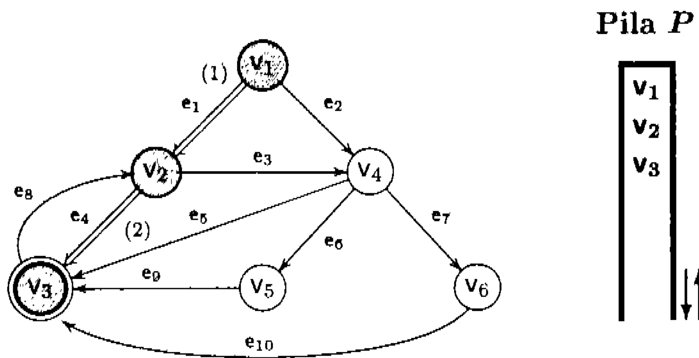
Al iniciarse la ejecución de DFS, se coloca al vértice v_1 como primer centro de acción, y desde él se descubre a v_2, a quien se procede a colocar en la pila, y pasa a ser el siguiente centro de acción, como se muestra en la Figura 5.2.

Figura 5.2: Primera iteración de DFS: después de inicializar, se mete a v_2 a la pila



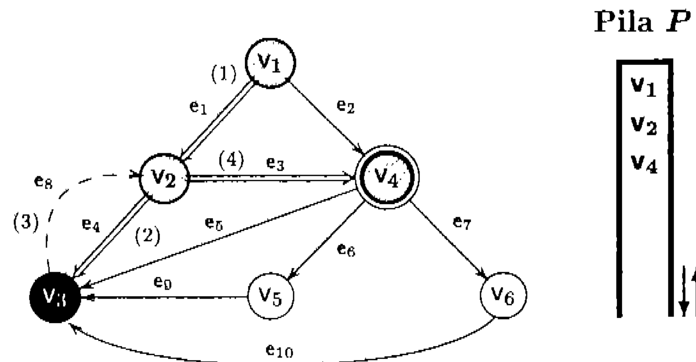
En la siguiente iteración, se descubre a v_3 desde v_2, se empuja a v_3 a la pila y se convierte en el siguiente centro de acción, como se muestra en la Figura 5.3.

Figura 5.3: Segunda iteración de DFS: se alcanza a v_3 y se le coloca en la pila



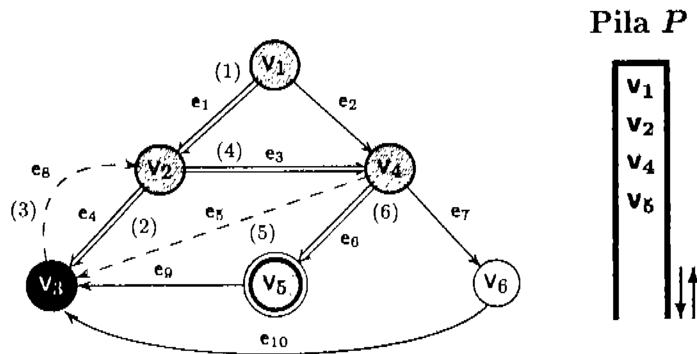
Desde v_3 se recorre el arco e_8, pero como su extremo es un vértice NEGRO no se hace nada, más que marcar el arco. En la siguiente iteración como no hay más arcos incidentes desde v_3, se procede a sacar a v_3 de la pila, como se muestra en la Figura 5.4, volviendo a quedar v_2 en el tope de la pila.

Figura 5.4: Tercera y cuarta iteración del algoritmo: se saca a v_3 y se mete a v_4



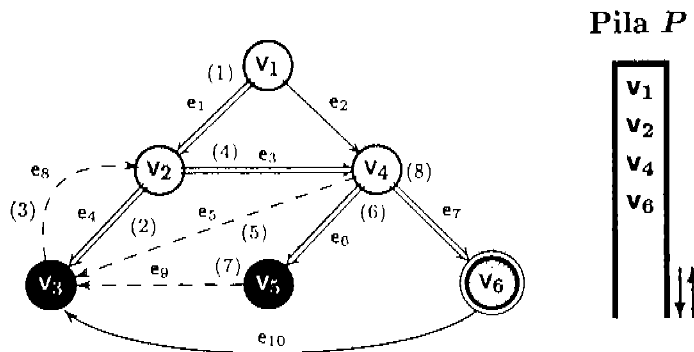
Desde v_2 descubre a v_4 y desde v_4 a v_5, colocándolos en la pila conforme los va descubriendo. Esto le lleva dos iteraciones del algoritmo, dejando al cabo de ellas a v_5 en el tope de la pila y los arcos clasificados como se muestra en la Figura 5.5.

Figura 5.5: Dos iteraciones sobre la digráfica: se recorren las aristas desde v_4 y se alcanza a v_5



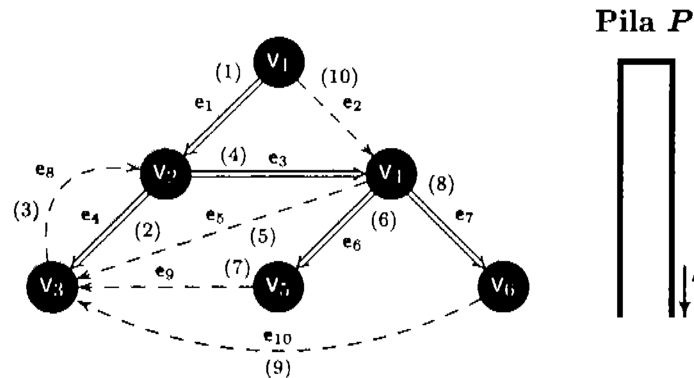
Desde v_5 recorre el arco e_9 cuyo destino es un vértice GRIS, por lo que no hace nada. En la siguiente iteración, dado que v_5 ya no tiene arcos disponibles, se le pinta de NEGRO y se le saca de la pila, dejando a v_4 como el siguiente centro de acción. Desde v_4 se descubre a v_6 y se le empuja a la pila, dejando a v_6 como el centro de acción para la siguiente iteración, como se muestra en la Figura 5.6.

Figura 5.6: Tres iteraciones más del algoritmo: se cierra a v_5 y se regresa a v_4 para alcanzar a v_6



Desde v_6 se recorre el arco e_{10} pero como su destino es un vértice NEGRO, se le deja como centro de acción y se vuelve a iterar. En la siguiente iteración, como v_6 ya no tiene arcos disponibles, se le pinta de NEGRO y se la saca de la pila, dejando a v_4 en el tope de la pila. En las siguientes iteraciones, como v_4, v_2 y v_1, en ese orden, ya no tienen arcos disponibles, se les pinta de NEGRO y se les saca de la pila, dejando a la pila vacía, como se puede observar en la Figura 5.7. Al quedar la pila vacía, termina la ejecución de exploracionGenerica.

Figura 5.7: Estado de la gráfica al terminar la ejecución de exploracionGenerica



Efectos de la elección del origen

Al igual que en BFS, es importante notar que el vértice que se toma como origen de la exploración puede cambiar muchísimo el resultado final de la misma, ya que si la gráfica no es fuertemente conexa, no todos los vértices son alcanzables desde cualquier vértice. También influye en el resultado de la exploración el orden en que los arcos se presenten en las listas de incidencia. En el Apéndice D, páginas 400 – 403 se puede ver una exploración con un origen distinto a v_1.

5.2. Propiedades de ExploracionDFS

Con la frontera del algoritmo especializándose como una pila, se cumplen propiedades interesantes de ExploracionDFS que no se cumplen ni en ExploracionBasica ni en ExploracionBFS, y que presentamos en los lemas que siguen.

Lema 5.1 *Al terminar la ejecución de exploracionGenerica en ExploracionDFS, dos vértices* u *y* v *ocuparon posiciones consecutivas en la pila, con* u *más cerca del fondo que* v *en* frontera, $\Longleftrightarrow$ *existe un arco* u$\rightarrow$v *en* G_π.

Demostración:

$\Longrightarrow$ Supongamos que dos vértices u y v se encuentran consecutivos en la pila. Observemos lo siguiente:

i. Para ingresar a v en la pila se requiere que el momento en que ingresa, v sea el destino de un arco, cuyo origen es el centro de acción. Por la disciplina de la frontera, un vértice u, para ser centro de acción, debe encontrarse en el tope de la pila. Por lo que v fue ingresado a la pila cuando el vértice u era centro de acción.

ii. Esto es, se recorrió el arco u $\rightarrow$ v, con u en el tope de la pila y v el destino del arco recorrido.

iii. Si se introdujo a v a la pila es porque estaba pintado de BLANCO, y se le pintó de GRIS.

iv. Al pintar de GRIS a v se asignó a su atributo π el valor de u.

$\therefore$ el arco $u \rightarrow v \in G_\pi$.

$\Longleftarrow$ Supongamos ahora que el arco $u \rightarrow v \in G_\pi$ y queremos demostrar que esto implica que los vértices u y v se encontraron consecutivos en la pila, en algún momento anterior. $u \rightarrow v \in G_\pi$ quiere decir que siendo u centro de acción, desde u se descubrió a v. Como u era centro de acción, al descubrir desde él a v, u se encontraba en el tope de la pila. Al descubrir a v se le colocó en el tope de la pila, dejando a u en la posición inmediata abajo del tope.

$\therefore$ u y v se encontraron en posiciones consecutivas de la pila, con u más cerca del fondo de la pila. ❑

Lema 5.2 *Sea $\langle v_{i_0}, v_{i_1}, \ldots, v_{i_k} \rangle$ el contenido de la pila en una iteración dada de la ejecución de explorationGenerica, con v_{i_0} en el fondo de la pila y v_{i_k} en el tope de la pila. Entonces, existe un camino dirigido desde v_{i_0} hasta v_{i_k} en G_π,*

$$v_{i_0} \rightarrow v_{i_1} \rightarrow \ldots \rightarrow v_{i_k}$$

Demostración:

Se sigue directamente del Lema 5.1, ya que para cada $j = 0, \ldots k-1$, como v_{i_j} y $v_{i_{j+1}}$ son vértices consecutivos en P, entonces $v_{i_j} \rightarrow v_{i_{j+1}} \in G_\pi$. ❑

Usaremos la notación $u \prec_\pi v$ en lugar de la más explícita $u \prec_{G_\pi} v$ para denotar la relación v *es descendiente*[2] *de* u en el árbol G_π.

Dado esto, podemos caracterizar a los vértices en la pila (en frontera) con el siguiente lema:

Lema 5.3 *Sea $\langle v_{i_0}, v_{i_1}, \ldots, v_{i_k} \rangle$ el contenido de la pila en una iteración dada de la ejecución de exploracionGenerica en ExploracionDFS, con v_{i_0} en el fondo de la pila y v_{i_k} en el tope de la pila. Entonces, $v_{i_j} \prec_\pi v_{i_\ell}$ para $0 \leq j < \ell \leq k$.*

Demostración:

Por hipótesis, el contenido de la pila es la sucesión de vértices $\langle v_{i_0}, v_{i_1}, \ldots, v_{i_k} \rangle$. Por el Lema 5.2 existe un camino dirigido $v_{i_0} \rightsquigarrow v_{i_k}$ en G_π. Por lo que para cualquier subsucesión de vértices $v_{i_j}, \ldots, v_{i_\ell}$, $0 \leq j \leq \ell \leq k$ existe un camino dirigido, subcamino del camino del Lema 5.2, cumpliéndose la Definición C.1.

$\therefore v_{i_{j+1}}, \ldots, v_{i_{j+1}}$ son descendientes de v_{i_ℓ} en G_π. ❑

Lema 5.4 *Sea $v \in V$ tal que $s \rightsquigarrow v \in G$ y sea $u \neq v$ tal que $\exists v \rightsquigarrow u \in G_\pi$. Entonces, en el momento en que se agregó u a frontera, todos los vértices en el camino de v a u en G_π se encontraban en la pila, en el orden dado por el camino.*

[2]Ver la definición precisa en el Apéndice C.

Demostración:

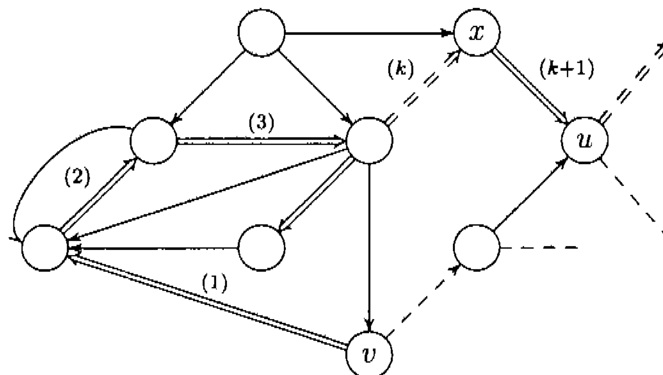
Si existe un camino dirigido de v a u en G_π, entonces v $\prec_\pi$ u. Demostraremos por inducción en la longitud ℓ de ese camino que u tuvo que estar, en algún momento, encima de v en la pila.

Base: Para $\ell = 1$. Supongamos que el camino de v a u tiene longitud 1, de donde u es descendiente directo de v en G_π. Por el Lema 5.1, u y v se encontraron en posiciones consecutivas de la pila, con v más cerca del fondo de la pila que u, cumpliéndose la conclusión del lema.

Inducción: Supongamos como hipótesis de inducción, que el lema se cumple para todo camino de longitud $0 \leq k \leq \ell$ y probaremos para caminos de longitud $\ell+1$. Podemos descomponer este camino de longitud $\ell + 1$ en dos partes: la primera de ella nos lleva al padre de u, digamos x (v $\rightsquigarrow$ x) y la segunda consiste del arco x$\rightarrow$u – ver Figura 5.8. Por la hipótesis de inducción podemos suponer que cada uno de los vértices en el camino de v a x se encuentran en la pila en el momento en que ingresa x a la misma. Como el arco x$\rightarrow$u $\in G_\pi$, por el Lema 5.1 x y u van a estar colocados en la pila, en lugares consecutivos, en el momento de ingresar a u a la pila.

$\therefore$ todos los vértices que se encuentran en el camino de v a u se encuentran en la pila, en el orden dado por el camino, con v más cerca del fondo y u en el tope, en el momento en que se ingresa a u a la pila. ❏

Figura 5.8: Camino de longitud $k + 1$ en G_π, desde v hasta u



Lema 5.5 *Un vértice* v *es pintado de* NEGRO $\iff$ *todos sus descendientes propios en* G_π *ya han sido pintados de* NEGRO.

Demostración:

$\Longrightarrow$ Supongamos por contradicción que existen k descendientes de v en G_π que, al pintar a v de NEGRO no han sido pintados de NEGRO. Como son descendientes de v, son alcanzables desde v, por lo que por el Lema 3.12 inciso (b), eventualmente serán pintados de NEGRO. Observemos la iteración en la que u, uno de esos vértices, es pintado de NEGRO. Para que se pinte a u de NEGRO debe encontrarse en el tope de la pila. Pero por el Lema 5.2, v tiene que estar en la pila, más cerca del fondo que u; por el Lema 3.2, v tiene que estar pintado de GRIS, y no de NEGRO como se había supuesto.

$\therefore$ todos los vértices descendientes de v deben haber sido sacados de la pila (y pintados de NEGRO) antes de sacar a v.

$\Longleftarrow$ Supongamos ahora que todos los vértices descendientes de un vértice v en G_π ya han sido pintados de NEGRO y debemos mostrar que entonces v será el siguiente vértice pintado de NEGRO. Observemos lo siguiente:

i. Por el Lema 3.8 los vértices de V entran a la pila – se pintan de GRIS – exactamente una vez.

ii. Como todos los vértices descendientes de v ya fueron pintados de NEGRO, ninguno de ellos se encuentra en la pila, pero estuvieron en algún momento.

iii. El vértice v se encuentra en la pila, pues ya fue descubierto pero no ha sido pintado de NEGRO.

iv. Por el Lema 5.2, el vértice v se encuentra en el tope de la pila, pues si hubiera algún otro vértice u encima de él en la pila, u tendría que ser descendiente de v, lo que contradice la hipótesis de que no hay ningún descendiente de v en G_π en la pila.

v. Al ser v centro de acción, ya no se descubre ningún otro vértice adyacente a él (si se descubriera, éste sería descendiente de v, contradiciendo el que ya todos los descendientes de v fueron explorados). Por lo que permanece como centro de acción, y en el tope de la pila, hasta que se recorran todos los arcos, si es que hay alguno, que inciden desde él y que tengan como destino vértices ya descubiertos o explorados. Y en ese momento, v va a ser pintado de NEGRO. ❏

5.2.1. Implementación eficiente de la pila para ExploracionDFS

El Lema 5.1 nos sugiere que en todo momento en la pila se encuentra el camino desde la raíz del árbol hasta el vértice que se encuentre en ese momento en el tope de la pila. Este camino está marcado en la gráfica por medio del atributo π, en sentido inverso. En [Eve79] se propone una estructura para la frontera que elimina a la pila como espacio adicional y la sustituye con un único registro. Este registro contiene, en todo momento, el vértice al final del camino actual, o sea el que estaría en el tope de la pila.

Iniciamos al registro con el vértice origen de la exploración. Se elige al centro de acción simplemente regresando el contenido del registro; para ingresar a un vértice a la frontera, después de asignar el atributo π, simplemente se sustituye en el registro al vértice que se encuentra en él con el que se está ingresando; para eliminar a un vértice de la frontera, se sigue el atributo π al padre del vértice en el registro, y se sustituye a éste con el padre. En el Listado 5.3 damos la implementación que se daría para una frontera con estas características.

Si bien esta frontera no contiene físicamente a los vértices que ya fueron descubiertos y que no han sido explorados completamente, – de hecho en todo momento sólo contiene a un vértice – se puede recuperar la pila física siguiendo las referencias al padre, empezando por el vértice que está en el registro, y tomando al padre, al padre de éste, y así sucesivamente hasta llegar a la raíz del árbol de exploración, que corresponde al vértice origen y que no tiene padre.

Listado 5.3: Frontera para una pila que utiliza un único registro

```
1 public class FronteraEnRegistro implements Frontera {
2     /* Registro para mantener al tope del camino.                  */
3     protected NodoBasico registro = null;
4
5     /* Agrega un nodo a la frontera.                                */
6     public void agregaNodo(Nodos v) {
7         registro = v;
8     }
9
10    /* Elige un elemento de la frontera sin modificar a la misma */
11    public Nodos elige() {
12        return registro;
13    }
14
15    /* Elimina a un nodo de la frontera.                            */
16    public void quitaNodo(Nodos v) {
17        if (registro.getPi() != null)
18            registro = registro.getPi();
19        else
20            registro = null;
21    }
22
23    /* Consulta a la frontera para ver si existen nodos en ella   */
24    boolean esNoVacia() {
25        return registro != null;
26    }
27 }
```

El resto de esta especialización queda exactamente igual que ExploracionDFS. Lo único que tendríamos que hacer es instanciar FronteraEnRegistro en lugar de FronteraEnPila al construir la instancia de este algoritmo. La implementación se puede ver en el Listado 5.4.

Listado 5.4: Especialización de ExploracionBasica a ExploracionDFS con ahorro de memoria

```
1 public class ExploracionDFSRegistro extends ExploracionBasica {
2         /* Construye el objeto para la exploración, pero asig-    *
3          * nando a la frontera un objeto de FronteraEnRegistro.   */
4     public ExploracionDFS(Digrafica g) {
5         super(g);
6         frontera = new FronteraEnRegistro();
7     }
8 }
```

Es claro que las propiedades demostradas para ExploracionDFS se siguen cumpliendo, dado que en todo momento se tiene acceso al camino que va desde la raíz del árbol de exploración hasta el vértice que se encuentre en el registro, que corresponde al tope de la pila.

5.2.2. ExploracionDFS en bosques

Como hemos visto, dependiendo del vértice origen de la exploración vamos a poder o no explorar todos los vértices. El planteamiento más general de DFS es aquel que procesa toda la gráfica, eligiendo como orígenes sucesivos alguno de los vértices que queden todavía pintados de BLANCO. Esto se consigue con una especialización del DFS similar a la que se hizo con ExploracionBasica para lograr esto mismo. Los cambios que se requieren se consiguen tomando la implementación de obtenOrigen en la clase ConexExplora – Listado 3.10 – y tomar como patrón de llamada el código para la función principal main el que dimos en el Listado 3.9. En lo que sigue toda referencia a ExploracionDFS será al algoritmo que, utilizando distintos orígenes, garantiza que explora a todos los vértices, generando un bosque con los G_π así obtenidos.

5.2.3. Complejidad de ExploracionDFS

Recordemos la fórmula que dimos para la complejidad de ExploracionAbstracta en el Capítulo 3:

$$\begin{aligned}\text{Complejidad de ExploracionAbstracta} &= f_{constr}(n,m) + f_{daS}(n) + f_{marcaS}(n) + \\ &+ \sum_{n+m}\Big(f_{F.noVacia}(n) + f_{elige}(n) + f_{cAccion}(n,m)\Big) + \\ + \sum_{m}\Big(f_{masAr}(n,m) + f_{daAr}(m) &+ f_{explora}(n,m)\Big) + \sum_{n} f_{cierra}(n) + f_{fin}(n,m),\end{aligned}$$

donde $f_{explora}(n,m)$ se descompone de la siguiente manera:

$$\begin{aligned}\sum_{m} f_{explora}(n,m) = &\sum_{m}\Big(f_{getOtroN}() + f_{proceAr}() + f_{yaUsado}()\Big) + \\ &+ \sum_{n} f_{descubriendo}(n) + \sum_{m-n} f_{yaDescubierto}(n).\end{aligned}$$

En esta fórmula, los cálculos hechos para ExploracionBasica se mantienen, excepto en aquellos términos que hacen referencia al manejo de la frontera, que es lo único que ha cambiado hasta el momento:

$f_{marcaS}(n)$	Agrega a s a la frontera.
$f_{F.noVacia}(n)$	Verifica si la frontera está vacía.
$f_{elige}(n)$	Elige al siguiente centro de acción.
$f_{cierra}(n)$	Saca al nodo de la frontera.

Pero la implementación de la frontera garantiza tiempo constante en el acceso a la misma, así como agregar o sacar a vértices de ella, por lo que a la complejidad de ExploracionBasica

se refiere no cambia nada para el cálculo de la complejidad de ExploracionDFS. Por lo tanto, podemos enunciar el Teorema 5.6.

Teorema 5.6 *La complejidad de exploracionGenerica en ExploracionDFS es* $O(|V| + |E|)$.

Demostración:

La demostración está dada por el argumento que se acaba de exponer. ❑

5.3. Propiedades adicionales de ExploracionDFS

Sabemos ya que exploracionGenerica de la clase ExploracionDFS cumple con las propiedades de correctez de ExploracionAbstracta, ya que respetamos las restricciones que marcamos al extender el algoritmo. Pero no únicamente nos interesa garantizar lo mismo que nos garantiza ExploracionAbstracta, sino, como mencionamos en la introducción de este capítulo, determinar características respecto a la estructura de la digráfica en cuestión. Para ello, deseamos determinar la relación que existe entre cualesquiera dos vértices de la gráfica, en términos de G_π: cuál vértice es descendiente o antepasado de cuál otro, o si no hay relación de este tipo entre ellos sin necesidad de observar la ejecución del algoritmo. ExploracionDFS logra determinar esta relación en G_π especializando el algoritmo agregando atributos a los vértices, además de especializar la disciplina para la frontera. Vamos a registrar en los vértices el momento en el que se les pinta de GRIS y el momento en que se les pinta de NEGRO. Después justificaremos el por qué esto determina la relación que estamos buscando. Nuevamente, para evitar cargar a la clase básica para Nodos de información que no es común a todas las especializaciones, y no tener que especializar a la digráfica, construiremos estructuras paralelas a las de los nodos en el algoritmo mismo. Con estas extensiones la clase ExploracionDFS queda como se muestra en el Listado 5.5, donde la clase MarcasDeTiempo tiene un atributo desc para cuando se descubre el vértice y otro atributo fin para cuando se cierra.

Listado 5.5: Extensión a los atributos de los vértices

```
1  public class ExploracionDFS    extends ExploracionBasica   {
2          /* Para registrar cuándo se descubre y se cierra cada    *
3           * vértice.                                              */
4      protected MarcasDeTiempo[] marcasDeTiempo;
5          /* Un reloj discreto para ''contar el tiempo''.           */
6      protected int reloj = 0;
7          /* En el constructor se agrega la construcción de las    *
8           * marcas de tiempo.                                      */
9      public ExploracionDFS(Digrafica g) {
10         super(g);
11         frontera = new FronteraEnPila();
12         marcasDeTiempo = new MarcasDeTiempo[N+1];
13     }
```

Para registrar estos dos momentos, mantendremos un contador reloj que se incrementa con cada vértice descubierto y con cada vértice explorado. El registro de estos tiempos se

plasma en la extensión de los métodos descubriendo y cierraNodo – ver Listado 5.6, líneas [16–20] y [23–26].

Listado 5.6: Redefinición de los métodos descubriendo y cierraNodo

```
14       /* Registra el valor de reloj en el momento en que el vértice  *
15        * es descubierto.                                              */
16   protected void descubriendo(Nodos centroAccion, Aristas e,
17                               Nodos u) {
18         super.descubriendo(centroAccion, e, u);
19         marcasDeTiempo[u.getPos()].setDesc(++reloj);
20     }
21       /* Registra el valor de reloj cuando se termina de explo-     *
22        * rar al vértice.                                              */
23     protected void cierraNodo(Nodos centroAccion) {
24         super.cierraNodo(centroAccion);
25         marcasDeTiempo[u.getPos()].setFin(++reloj);
26     }
```

Para manejar el caso del vértice origen de la exploración, que no es introducido a la frontera en descubriendo sino en el método ponComoOrigen cuando éste es invocado desde exploracionGenerica, se debe redefinir este método, como se muestra en el Listado 5.7, líneas [29–32]. En la línea [6] del Listado 5.5 aparece la declaración del contador reloj. El resto de los métodos, por el momento, quedan exactamente igual.

Listado 5.7: Redefinición de ponComoOrigen para registrar tiempos para s

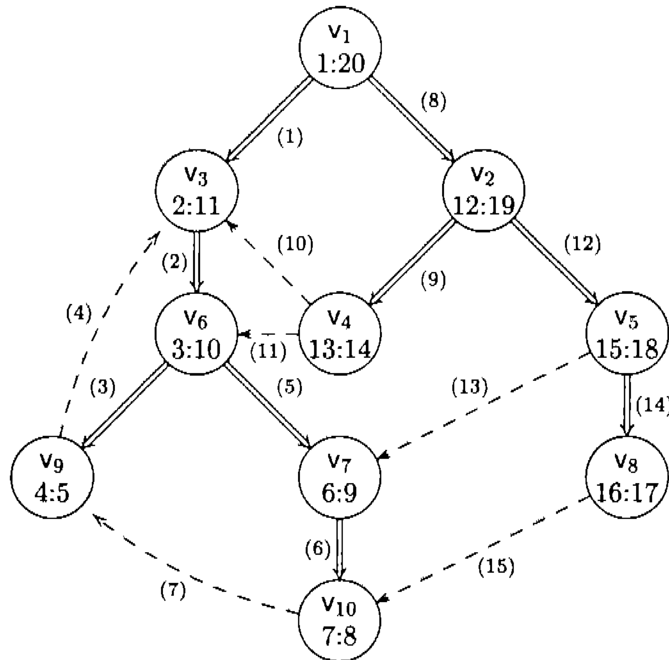
```
27       /* Mete al nodo origen a la frontera y registra el valor    *
28        * de reloj.                                                */
29     protected void ponComoOrigen(Nodos v) {
30         super.ponComoOrigen(v);
31         marcasDeTiempo[v.getPos()].setDesc(++reloj);
32     }
```

Ejemplo de ejecución con las marcas de tiempo

Ejecutemos el algoritmo en la digráfica de la Figura 5.9 y observemos el resultado al terminar éste. Los tiempos desc y fin se encuentran bajo la etiqueta del vértice con el formato desc:fin. El orden en que fueron considerados los arcos se da como etiqueta del arco correspondiente, entre paréntesis - (n) y, como lo hemos hecho hasta ahora, los arcos que pertenecen a G_π tienen línea doble, mientras que los que no pertenecen a G_π tienen línea intermitente. Veamos cuál es el uso que le damos a esta información. Concentrémonos, por lo pronto, en la determinación de si una digráfica es o no acíclica. Intuitivamente, existe un ciclo en la digráfica si durante la construcción de un camino conforme vamos descubriendo vértices, intentamos recorrer un arco cuyo destino es un vértice que ya fue descubierto, como se muestra en la Figura 5.9, donde el arco $e = v_9 \rightarrow v_3$, en efecto, cierra un ciclo. Sin embargo,

en esta misma figura, el arco $\mathsf{e} = \mathsf{v}_8 \to \mathsf{v}_{10}$ no cierra un ciclo y corresponde también a un arco cuyo destino es un vértice ya descubierto.

Figura 5.9: Determinación de ciclos en digráficas mediante DFS



Como se puede ver, el caso de ciclos en gráficas no dirigidas, que como demostramos en el Lema 3.19 es sencillo de detectar, en el caso de digráficas se complica un poco pues no todo arco que no esté en G_π cierra un ciclo.

En la Figura 5.9 podemos observar que mientras el arco $\mathsf{e} = \mathsf{v}_9 \to \mathsf{v}_3$ (cuarto en ser recorrido) en efecto, completa un ciclo ($\mathsf{v}_3 \to \mathsf{v}_6 \to \mathsf{v}_9 \to \mathsf{v}_3$), el resto de los arcos que no quedaron incluidos en G_π, por sí solos, no son parte de ningún ciclo en la gráfica. La diferencia entre el arco $\mathsf{e} = \mathsf{v}_9 \to \mathsf{v}_3$ y los otros arcos a los que hacemos referencia, es que el vértice etiquetado con v_3 es antecesor en el árbol del vértice etiquetado con v_9; esto es, el vértice destino del arco es antecesor del vértice origen del arco. Esto no sucede con ninguno de los otros arcos que no quedaron incluidos en G_π. Podemos intuir, de este ejemplo, que un arco cierra un ciclo si va desde un vértice hacia uno antecesor de éste en G_π.

5.3.1. Propiedades de las marcas de tiempo

Denotemos por [i, j], $0 < i < j$, al segmento de la ejecución de exploracionGenerica en ExploracionDFS que transcurre desde que el contador reloj toma el valor i y hasta que toma el valor j. Si suponemos que al construir a cada objeto de la clase MarcasDeTiempo tenemos los siguientes valores por omisión:

```
protected int desc = 0;
protected int fin  = INFINITO;
```

es fácil verificar el siguiente lema para la relación que existe entre $desc[v.getPos()]$ y $fin[v.getPos()]$. Usaremos desc[v] y $finv$ para hacer más clara la exposición.

Lema 5.7 $\forall$ v $\in V$, desc[v] < fin[v].

Demostración:

Si v no es alcanzable desde s, desc[v] = 0 y fin[v.getPos()] = ∞ en el constructor, y como no es alcanzable nunca es descubierto, por lo que estos valores no cambian y el lema se cumple.

Si s⇝v, el algoritmos exploracionGenerica en ExploracionDFS:

i. Asigna valor a la marca desc[v] en el momento en que se descubre al vértice, incrementando previamente el valor del contador reloj – Listado 5.6, línea [19].

ii. Asigna valor a la marca fin[v.getPos()] en el momento en que se termina de explorar al vértice, previo incremento del contador reloj – Listado 5.6, línea [25].

iii. Al descubrir al vértice se le pinta de GRIS y al terminar de explorarlo se le pinta de NEGRO.

iv. Por el Lema 3.4, primero se pinta al vértice de GRIS y en un momento posterior de la ejecución se le pinta de NEGRO.

$\therefore$ el contador reloj tiene un valor menor cuando se descubre al vértice que cuando se le da por explorado. ❑

De los lemas 5.2 y 5.7 se sigue la propiedad expresada en el Lema 5.8.

Lema 5.8 $\forall$v $\in V$ *tal que* s⇝v, *el vértice* v *permanece en* frontera *durante el intervalo* [desc[v], $finv$].

Demostración:

El valor de desc[v] se asigna en el momento en que v es descubierto. En ese momento se le ingresa a la pila. El valor fin[v] se asigna en el momento en que se termina de explorar al vértice, esto es cuando se le pinta de NEGRO, y esto sucede cuando se le saca de la pila. Una vez que el vértice ingresa a la pila se le pinta de GRIS; por el Lema 3.4 el vértice puede cambiar únicamente de GRIS a NEGRO, y únicamente una vez. Por el Lema 3.2, mientras está pintado de GRIS permanece en la pila.

$\therefore$ desde el momento en que se ingresa a v a la pila, marcado con desc[v], hasta el momento en que se le saca de la pila, marcado con fin[v.getPos()], v permanece en la pila. ❑

Estructura de la digráfica

Para determinar la estructura de una digráfica debemos determinar cuándo dos vértices u, v $\in V$, u $\neq$ v, satisfacen u$\prec_\pi$v – v es descendiente de u. No es sorprendente que las marcas de tiempo desc y fin nos sirvan para determinar esta relación – en lo que sigue restringiremos la relación $\prec$ a u $\prec$ v, únicamente en el caso de que u $\neq$ v. La relación "descendiente propio de" es una relación de orden parcial. Podemos resumir la manera en que

las marcas de tiempo determinan la relación entre dos vértices de una gráfica sobre la que se ejecutó exploracionGenerica en ExploracionDFS en el Teorema 5.9, considerando la notación para intervalos que dimos al inicio de esta sección, con [i, j] denotando un segmento cerrado de ejecución, esto es, que incluye a los momentos i y j. Asimismo, decimos que $[\mathsf{i},\mathsf{j}] \subset [\mathsf{k},\mathsf{m}]$ si se cumple $k < i < j < m$.

Teorema 5.9 (Teorema de paréntesis) *Entre cualesquiera dos vértices explorados de la digráfica se cumple exactamente una de las tres condiciones que siguen:*

i. $\big[\mathsf{desc[u]},\ \mathsf{fin[u]}\big] \cap \big[\mathsf{desc[v]},\ \mathsf{fin[v]}\big] = \varnothing$.

ii. $\big[\mathsf{desc[v]},\ \mathsf{fin[v]}\big] \subset \big[\mathsf{desc[u]},\ \mathsf{fin[u]}\big]$ *y* $\mathsf{u} \prec_\pi \mathsf{v}$.

iii. $\big[\mathsf{desc[u]},\ \mathsf{fin[u]}\big] \subset \big[\mathsf{desc[v]},\ \mathsf{fin[v]}\big]$ *y* $\mathsf{v} \prec_\pi \mathsf{u}$.

Demostración:

Antes que nada, es obvio que los casos ii y iii son simétricos, por lo que únicamente revisaremos uno de ellos.

Lo primero que conviene establecer es que los casos dados en el teorema son los únicos, esto es, que no podemos tener un caso como

$$\mathsf{desc[v]} < \mathsf{desc[u]} < \mathsf{fin[v]} < \mathsf{fin[u]}$$

Pero esto es bastante claro.

a. La relación

$$\mathsf{desc[v]} < \mathsf{desc[u]}$$

nos indica que v fue descubierto antes que u (lo que colocó a v en la pila antes que a u);

b. La relación

$$\mathsf{fin[v]} < \mathsf{fin[u]}$$

indica que v fue pintado de NEGRO antes que u;

c. $\mathsf{desc[u]} < \mathsf{fin[v]}$

implica que u fue colocado en la pila antes de que se sacara a v, de donde $\mathsf{v} \prec_\pi \mathsf{u}$ (Lema 5.2).

d. Sin embargo, como

$$\mathsf{fin[v]} < \mathsf{fin[u]}$$

v fue pintado de NEGRO antes que u.

e. Pero por el Lema 5.4, si $\mathsf{v} \prec_\pi \mathsf{u}$, u debe ser pintado de NEGRO antes que v, lo que contradice el inciso d.

De esto, nos quedan solamente los casos en que un intervalo esté contenido dentro del otro, o que sean ajenos.

i. Veamos primero el caso en que sean ajenos, esto es,

$$\mathsf{desc[u]} < \mathsf{fin[u]} < \mathsf{desc[v]} < \mathsf{fin[v]}$$

Esta relación nos indica que:

a. El vértice u fue pintado de NEGRO antes de que el vértice v fuera descubierto.

b. Pero para pintar a un vértice de NEGRO el algoritmo requiere que todos los vértices descendientes de él en G_π hayan sido ya pintados de NEGRO (Lema 5.4).

$\therefore$ u $\not\prec_\pi$ v.

c. Por otro lado, tampoco v $\prec_\pi$ u, ya que si así fuera, existiría un camino v $\leadsto$ u en G_π. Por el Lema 5.4, en el momento en que se agrega a u a la pila, v debe encontrarse en ella; pero fin[u] < desc[v] indica que en el momento en que v ingresó a la pila, u ya había entrado y salido, y por el Lema 3.8 esto último sucede una sola vez.

$\therefore$ v $\not\prec_\pi$ u.

Para el caso

$$\mathsf{desc[v]} < \mathsf{fin[v]} < \mathsf{desc[u]} < \mathsf{fin[u]}$$

se argumenta exactamente de la misma manera.

ii. Supongamos ahora que tenemos alguno de los casos ii o iii, y revisemos en detalle el caso

$$\big[\mathsf{desc[v]}, \mathsf{fin[v]}\big] \subset \big[\mathsf{desc[u]}, \mathsf{fin[u]}\big],$$

donde

$$\mathsf{desc[u]} < \mathsf{desc[v]} < \mathsf{fin[v]} < \mathsf{fin[u]}.$$

Como desc[v] < fin[u] y por el Lema 5.8. sabemos que v fue colocado en la pila antes de que se sacara a u, más cerca del tope de la pila de lo que se encontraba u (porque u fue descubierto antes que v). Esto indica, por el Lema 5.3 que v es descendiente propio de u. ❑

El Teorema 5.9 implica directamente el siguiente corolario:

Corolario 5.10 (Anidamiento de intervalos de descendientes)

$$\mathsf{u} \prec_\pi \mathsf{v} \iff \big[\mathsf{desc[v]},\ \mathsf{fin[v]}\big] \subset \big[\mathsf{desc[u]},\ \mathsf{fin[u]}\big]$$

Los resultados anteriores nos sirven para determinar exactamente cuáles deben ser los valores de las marcas de tiempo para un vértice dado. Notemos en la digráfica de la Figura 5.9 que el vértice origen de la exploración queda marcado con los tiempos $1 : 2|V|$. Esta observación la formalizamos en el Lema 5.11.

Lema 5.11 $\forall \mathsf{v} \in V, \mathsf{desc[v]} \neq 0, \implies \mathsf{fin[v]} = \mathsf{desc[v]} + 2k + 1,$ *donde* k *es el número de vértices descendientes propios de* v *en* G_π.

Demostración:

Supongamos que al descubrirse a v, el contador reloj tiene el valor i. La variable reloj se incrementa en 1 por cada vértice que es descubierto, y 1 por cada vértice que es pintado de NEGRO, entre el momento en que se descubre a v y el momento en que se le pinta de NEGRO. Una vez que se coloca a v en el tope de la pila, sabemos, por el Corolario 5.10, que $\forall u \neq v$ tal que $v \prec_\pi u$,

$$[\mathsf{desc}[u],\ \mathsf{fin}[u]] \subset [\mathsf{desc}[v],\ \mathsf{fin}[v]].$$

De esto, los k vértices descendientes de v en G_π deberán ser colocados en frontera y sacados de frontera antes de que v pueda ser pintado de NEGRO. Por lo tanto el contador reloj deberá ser incrementado en $2k$, más una unidad al pintar de NEGRO a v:

$$i + 2k + 1 < \mathsf{fin}[v]$$

Bastaría demostrar que entre el momento en que se descubre a v y el momento en que se le pinta de NEGRO, no se descubre (ni se pinta de NEGRO) a ningún vértice que no sea descendiente de v en G_π. Esto se sigue del Corolario 5.10, que implica que

$$u \not\prec_\pi v, \Longrightarrow [\mathsf{desc}[v], \mathsf{fin}[v]] \not\subset [\mathsf{desc}[u], \mathsf{fin}[u]].$$

Entonces, por el Teorema 5.9, los intervalos satisfacen i o iii; en el caso de que satisfaga el inciso i, son ajenos. En el caso del inciso iii, se pinta de NEGRO a u después que a v, por lo que el contador reloj no será incrementado.

$\therefore$ exactamente k vértices son descubiertos en el intervalo $[\mathsf{desc}[v],\ \mathsf{fin}[v]]$, y exactamente esos mismos k vértices son pintados de NEGRO antes de poder pintar de NEGRO a v, por lo que $\mathsf{fin}[v] = \mathsf{desc}[v] + 2k + 1$. ❑

Dada esta propiedad, sabemos que todas las marcas de tiempo van a estar en el intervalo $[1, 2|V|]$, que son las marcas que corresponden al vértice origen s, ya que todos los vértices explorados de la gráfica son descendientes del vértice origen en el árbol G_π y cumplen la siguiente desigualdad:

$$1 < \mathsf{desc}[v] < \mathsf{fin}[v.getPos()] < 2|V| \qquad \forall v \in V \text{ tal que v.color} = \text{NEGRO, con } v \neq s$$

Observando las marcas de tiempo que corresponden a exploracionGenerica de ExploracionDFS de la Figura 5.9, vemos:

i. El vértice origen v_1, tiene dos descendientes directos, v_3 y v_5, con las marcas de tiempo $2 : 11$ y $12 : 19$ respectivamente, por lo que se cumple

Para v_3 $\quad 1 < 2 < 11 < 20$
Para v_5 $\quad 1 < 12 < 19 < 20$

ii. Los vértices descendientes de v_3 son v_6, v_9, v_7 y v_{10} con marcas respectivas $3 : 10$, $4 : 5$, $6 : 9$ y $7 : 8$. Nótese que la relación que acabamos de dar, pero restringida ahora al

subárbol cuya raíz es v_3, se sigue cumpliendo, a saber:

$$\begin{array}{ccccccccl}
\mathsf{v}_3.\mathsf{desc} & < & \mathsf{v}_i.\mathsf{desc} & < & \mathsf{v}_i.\mathsf{fin} & < & \mathsf{v}_3.\mathsf{fin}, & & \text{para } i = 6,7,9,10.\\
2 & < & 3 & < & 10 & < & 11 & & \text{para } \mathsf{v}_6.\\
2 & < & 4 & < & 5 & < & 11 & & \text{para } \mathsf{v}_9.\\
2 & < & 6 & < & 9 & < & 11 & & \text{para } \mathsf{v}_7.\\
2 & < & 7 & < & 8 & < & 11 & & \text{para } \mathsf{v}_{10}.
\end{array}$$

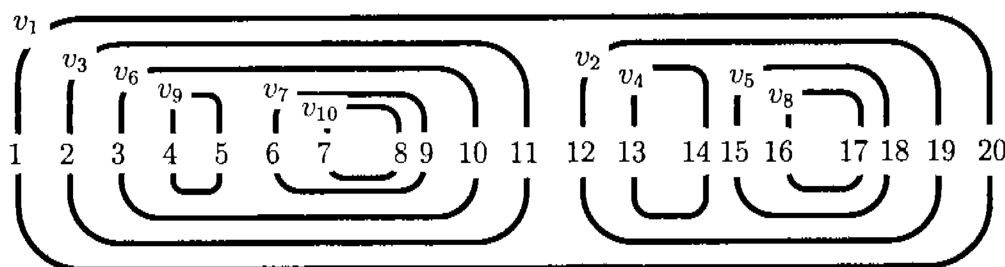
iii. Esta relación se vuelve a cumplir si consideramos el subárbol con raíz en v_6.

iv. En cambio, esta relación no se cumple entre dos vértices que no están relacionados entre sí, o sea que ninguno es descendiente del otro.

En la Figura 5.10 podemos ver gráficamente el anidamiento de intervalos que se presenta en la gráfica de la Figura 5.9. En ella se muestra el intervalo de tiempo, denotada por un rectángulo, en el que cada uno de los vértices permanece en la pila durante la ejecución de exploracionGenerica en ExploracionDFS. Se puede observar cómo el intervalo que corresponde a un vértice dado estará contenido en el intervalo de su antecesor, mientras que si no hay relación entre dos vértices, sus intervalos serán ajenos. El anidamiento de los intervalos siguen la estructura del árbol G_π, con

$$\big[\mathsf{desc}[\mathsf{v}_i.\mathsf{getPos}()],\ \mathsf{fin}[\mathsf{v}_i.\mathsf{getPos}()]\big] \subset \big[\mathsf{desc}[\mathsf{v}_j.\mathsf{getPos}()],\ \mathsf{fin}[\mathsf{v}_j.\mathsf{getPos}()]\big] \iff \mathsf{v}_j \prec_\pi \mathsf{v}_i.$$

Figura 5.10: Anidamiento de intervalos en DFS



La relación $\prec_\pi$ proporciona información importante respecto a la estructura de la gráfica, entre otras, el si existe o no un camino simple dirigido de u a v. Dimos ya una caracterización importante que define esta relación, cuando ambos vértices han sido alcanzados (Lema 5.2). También nos interesa saber si para algún vértice v que todavía no ha sido alcanzado, se cumplirá $\mathsf{u} \prec_\pi \mathsf{v}$. Para ello tenemos el siguiente teorema:

Teorema 5.12 (Teorema del camino blanco) $\mathsf{u} \prec_\pi \mathsf{v} \iff$ *en el momento en que el algoritmo descubre a* u*, el vértice* v *puede ser alcanzado desde* u *siguiendo un camino que contiene únicamente vértices pintados de* BLANCO.

Demostración:

$\Longrightarrow$ Supongamos que u $\prec_\pi$ v y que estamos en el momento en que u es descubierto, esto es, se le asigna valor $\neq 0$ a desc[u]. Como u $\prec_\pi$ v, existe un camino simple u $\rightsquigarrow$ v en G_π. Probaremos que todo vértice en este camino, incluyendo a v, está pintado de BLANCO. Supongamos que w es un vértice en ese camino, u $\rightsquigarrow$ w $\rightsquigarrow$ v. Observemos lo siguiente:

i. Como este camino es en G_π, el subcamino de u $\rightsquigarrow$ w también es un camino en G_π, por lo que u $\prec_\pi$ w.

ii. Por el Corolario 5.10, desc[u] < desc[w] al terminar el algoritmo.

iii. Como u acaba de ser descubierto, desc[u] tiene el valor máximo que ha tomado reloj hasta el momento en que se descubre a u.

iv. En ese momento desc[w] $= 0$, lo que indica que no ha sido descubierto, por lo que no puede tener un valor superior al de desc[u].

$\therefore$ w está pintado de BLANCO.

$\Longleftarrow$ Supongamos por contradicción que en el momento desc[u], existe un camino u$\rightsquigarrow$v en el que únicamente hay vértices pintados de BLANCO (que no han sido descubiertos) pero en el que **no todos** los vértices en ese camino son descendientes de u en G_π. Sin pérdida de generalidad podemos suponer que v es el primer vértice en ese camino desde u que no se convierte en descendiente de u en G_π. Sea w $\neq$ u el predecesor de v en ese camino.

i. Como v es el primer vértice en ese camino que no se va a convertir en descendiente de u en G_π, u $\prec_\pi$ w.

ii. u $\prec_\pi$ w $\Longrightarrow$ que w va a ingresar a la pila cuando u aún se encuentre en ella, más cerca del tope que u.

iii. Como estamos examinando el momento en que se descubre a u y en ese momento v está pintado de BLANCO, entonces v va a ser pintado de GRIS en un momento posterior, por lo que va a ser colocado en la pila después de u.

iv. u $\not\prec_\pi$ v $\Longrightarrow$ que u no se encuentra en la pila cuando ingresa v a frontera.

v. Como u $\prec_\pi$ w, para que ni u ni w estuvieran en la pila al momento de ingresar v, tuvieron que ser metidos y sacados de la misma antes de que entre v.

vi. Pero tenemos el arco w$\rightarrow$v, que no fue recorrido cuando w se encontraba en el tope de la pila, ya que de haberse recorrido se hubiera descubierto a v en ese momento.

vii. No se pudo sacar a w de la pila, ya que tenía todavía arcos por recorrer.

$\therefore$ u (y w) estaban en la pila en el momento de meter a v, lo que contradice el que u $\not\prec_\pi$ v. ❑

Una vez que se ejecutó exploracionGenerica de ExploracionDFS sobre una digráfica $G = (V, E)$ los arcos de la digráfica quedan clasificados, en un primer nivel, en dos conjuntos ajenos: los que pertenecen a G_π y los que no pertenecen a G_π. Podemos caracterizar de manera más precisa a estos últimos. Vale la pena observar que los arcos de la gráfica original se pueden clasificar en cuatro grupos, es decir, sea u$\rightarrow$v $\in E$:

i. **u$\rightarrow$v $\in$ G_π**. Los arcos que pertenecen al árbol G_π son los que van de v.π a v (e = v.$\pi \rightarrow$ v). Como v.π fue descubierto antes que v, tienen la característica de que

$$[\text{desc[v]},\ \text{fin[v]}] \subset [\text{desc[v.}\pi\text{]},\ \text{fin[v.}\pi\text{]}].$$

ii. $\mathsf{u}\rightarrow\mathsf{v} \not\in G_\pi$ pero $\mathsf{u} \prec_\pi \mathsf{v}$ (hacia adelante). Si bien los vértice u y v cumplen, al igual que con los arcos $\mathsf{u}\rightarrow\mathsf{v} \in G_\pi$,

$$[\mathsf{desc}[\mathsf{v}],\ \mathsf{fin}[\mathsf{v}]] \subset [\mathsf{desc}[\mathsf{v}.\pi],\ \mathsf{fin}[\mathsf{v}.\pi]],$$

en el momento de recorrer estos arcos, el vértice en el destino ya tiene valor desc[v] asignado.

iii. $\mathsf{u}\rightarrow\mathsf{v} \not\in G_\pi, \mathsf{v} \prec_\pi \mathsf{u}$ (hacia atrás). Éstos se caracterizan porque a diferencia de los arcos de G_π y los arcos hacia adelante,

$$[\mathsf{desc}[\mathsf{u}],\ \mathsf{fin}[\mathsf{u}]] \subset [\mathsf{desc}[\mathsf{v}], \mathsf{fin}[\mathsf{v}]].$$

iv. $\mathsf{u}\rightarrow\mathsf{v},\ \mathsf{u} \not\prec_\pi \mathsf{v}$ y $\mathsf{v} \not\prec_\pi \mathsf{u}$ (de cruce). En este caso

$$[\mathsf{desc}[\mathsf{u}], \mathsf{fin}[\mathsf{u}]] \cap [\mathsf{desc}[\mathsf{v}], \mathsf{fin}[\mathsf{v}]] = \varnothing.$$

Podemos decir el tipo del arco en términos del color del vértice destino, de la siguiente manera. Sea $\mathsf{u}\rightarrow\mathsf{v}$ el arco que se está recorriendo:

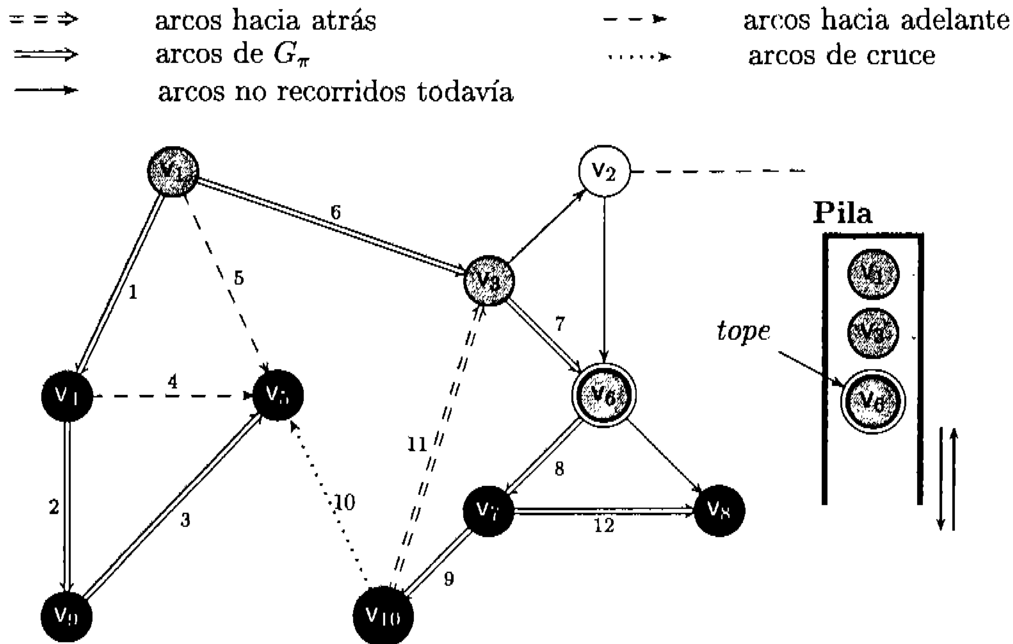
- Si v.color = NEGRO, $\mathsf{u}\rightarrow\mathsf{v}$ es de cruce o hacia adelante. El que v.color = NEGRO indica que ya fue explorado completamente. Esto se da si $\mathsf{u} \prec_\pi \mathsf{v}$ (arco hacia adelante) o si $[\mathsf{desc}[\mathsf{u}], \mathsf{fin}[\mathsf{u}]] \cap [\mathsf{desc}[\mathsf{v}], \mathsf{fin}[\mathsf{v}]] = \varnothing$, lo que indica que se encuentra en un subárbol que ya fue totalmente explorado. En este caso se trata de un arco de cruce.

- Si v.color = BLANCO, el arco es un arco de G_π, ya que al procesar al vértice que se está descubriendo se incluye al arco en G_π.

- Si v.color = GRIS, el arco es un arco hacia atrás (hacia un antecesor de u), ya que v se encuentra todavía en la pila, y por el Lema 5.2, $\mathsf{v} \prec_\pi \mathsf{u}$.

Para terminar esta sección veamos un ejemplo en la Figura 5.11, donde se muestra una foto instantánea de la ejecución de exploracionGenerica en ExploracionDFS en el punto en que v_6 es el centro de acción y se acaba de recorrer el arco etiquetado con 8. En esta Figura el centro de acción está marcado con doble círculo, y todavía no se concluye la exploración. Se muestran los distintos tipos de arcos, y dado que no se ha terminado la exploración, hay todavía arcos sin recorrer y vértices sin visitar.

Figura 5.11: Distintos tipos de arcos con DFS



De acá en adelante nos referiremos a ExploracionDFS con el agregado de las marcas de tiempo. Es claro que preserva las propiedades dadas para ExploracionBasica. Revisaremos, sin embargo, la complejidad de la especialización de ExploracionDFS dada, en la que se incluyen las marcas de tiempo.

5.3.2. Complejidad de ExploracionDFS extendido con las marcas de tiempo

En la sección 5.2.3 justificamos el que la complejidad de ExploracionDFS antes de extenderlo con las marcas de tiempo es la misma que la de ExploracionBasica, $O(|V| + |E|)$, revisando aquellos métodos que se pudieran ver afectados por el cambio de disciplina en la frontera. Con las marcas de tiempo los métodos que se van a ver afectados en la fórmula general para la complejidad de ExploracionAbstracta, una vez tomados en cuenta los cambios en la frontera, se encuentran en la Tabla 5.1.

De esta tabla, vemos que lo agregado únicamente afecta al coeficiente para el término $|V|$, por lo que la complejidad sigue siendo $O(|V| + |E|)$, lo que enunciamos en el Teorema 5.13.

Tabla 5.1: Complejidad agregada por las marcas de tiempo

Método	Extensión	Complejidad (*adicional*)
$f_{constr}(n, m)$	Hay que agregar la construcción de la estructura con las marcas de tiempo, lo que agrega a la complejidad de este método un factor de $\|V\| = n$ operaciones.	$c_t \cdot n$
$f_{marcaS}(n)$	Registra en el vértice origen el valor del reloj, por lo que incrementa el tiempo original en una unidad[3].	1
$f_{descubriendo}(n)$	Agrega una asignación, por lo que incrementa el costo en una unidad.	1
$f_{cierra}(n)$	Agrega una asignación, por lo que incrementa el costo en una unidad.	1

Teorema 5.13 (Complejidad de ExploracionDFS**)** *La especialización de ExploracionBasica en ExploracionDFS, incluyendo a las marcas de tiempo, tiene una complejidad de $O(|V| + |E|)$.*

Demostración:

Está dada en la discusión de la Tabla 5.1. ❑

5.4. Aplicaciones en digráficas

Veremos, por lo pronto, dos aplicaciones directas de ExploracionDFS en gráficas dirigidas, detección de ciclos y ordenamiento topológico. Éstas se obtienen como resultado de la especialización de la frontera de ExploracionBasica a una pila y la extensión dada por las marcas de tiempo. Como mencionamos en la sección anterior, al referirnos al algoritmo exploracionGenerica de ExploracionDFS suponemos que se alcanza a todos los vértices desde el origen, recordando que el caso general se puede resolver ejecutando exploracionGenerica repetidas veces, a partir siempre de algún vértice que todavía no haya sido descubierto.

5.4.1. Determinación de ciclos

Mencionamos en la sección anterior que DFS puede determinar si una digráfica tiene o no ciclos. En general, si la digráfica tiene algún arco hacia atrás entonces tiene un ciclo. Como ya vimos, este tipo de arco se detecta durante la ejecución de exploracionGenerica de ExploracionDFS porque se recorre un arco que va de un vértice GRIS a otro vértice GRIS. Formalicemos esto en el Lema 5.14.

Lema 5.14 *Si durante la ejecución de exploracionGenerica en ExploracionDFS se recorre un arco cuyo destino es un vértice* GRIS, *entonces la digráfica tiene al menos un ciclo.*

[3]Estamos asumiendo el autoincremento y autodecremento con costo 1.

Demostración:

Sea u el centro de acción en el momento en que se recorre un arco cuyo destino es un vértice GRIS, y sea v este vértice. Como u es centro de acción, está pintado de GRIS y se encuentra en el tope de la pila; como v está pintado de GRIS también se encuentra en la pila – Lema 3.2 – más cerca del fondo de la pila que u. Entonces existe una sucesión de vértices $x_1, x_2, \ldots, x_k$, $k \geq 0$, que se encuentran en la pila entre v y u en posiciones consecutivas; por el Lema 5.2, existe un camino dirigido $v \rightsquigarrow u = v \to x_1 \to \ldots \to x_k \to u$ en G_π (y por lo tanto en G), y si le pegamos el arco $u \to v$, tenemos un ciclo. ❏

Y su contra parte:

Lema 5.15 *Si la digráfica $G = (V, E)$ contiene algún ciclo, durante la ejecución de exploracionGenerica en ExploracionDFS se recorrerá algún arco que vaya de un vértice* GRIS *a otro vértice* GRIS.

Demostración:

Supongamos que G tiene un ciclo $C = v \to x_1 \to \ldots \to x_k \to u \to v$. Sin pérdida de generalidad, supongamos que v es el primer vértice que se descubre en ese ciclo, esto es, el primero que se pinta de GRIS y, por lo tanto, que se coloca en la pila. Como hay un camino dirigido desde v a u, en el momento en que se descubre a v este camino consiste únicamente de vértices BLANCOs; por el Teorema 5.12, u es descendiente de v en G_π; por el Lema 5.4, en el momento en que se descubre a u todos los vértices en el camino de v a u, v inclusive, se encuentran en la pila, y por lo tanto pintados de GRIS.

Para poder sacar a u de la pila deberán recorrerse todos los arcos incidentes desde u. En particular, al recorrer el arco $u \to v$, los vértices u y v ambos están pintados de GRIS, lo que termina la demostración. ❏

Los Lemas 5.14 y 5.15 se combinan en el Corolario 5.16.

Corolario 5.16 *Una digráfica $G = (V, E)$ es acíclica $\iff$ en exploracionGenerica de ExploracionDFS no se detecta ningún arco hacia atrás.*

De lo anterior, contamos con un algoritmo de $O(|V| + |E|)$ que determina si una gráfica tiene o no ciclos. Modifiquemos la especialización ExploracionDFS de la siguiente manera:

i. Si el algoritmo detecta un arco desde un vértice GRIS hacia otro vértice GRIS, determina que la gráfica no es acíclica y suspende su funcionamiento con un mensaje adecuado.

ii. Si el algoritmo termina su exploración, emite un mensaje aseverando que la digráfica es acíclica.

El inciso i se satisface dando una implementación al método abstracto yaDescubierto(Nodos centroAccion, Aristas e, Nodos u), que hasta el momento ha permanecido vacío. Utilizamos una excepción para abortar el proceso en caso de que el nodo en el otro extremo del arco esté en la frontera.

Para satisfacer el inciso ii, y por el Corolario 5.16, si el algoritmo no suspende su funcionamiento y llega al final de la ejecución, sabemos que el algoritmo no detectó ningún ciclo. Por lo que si el algoritmo termina emitimos el mensaje correspondiente. Estos cambios se pueden observar en la definición de la clase ExploracionDFSCiclos que se muestra en el Listado 5.8.

Listado 5.8: Especialización de ExploracionDFS que determina si hay ciclos

```
1 public class ExploracionDFSCiclos extends ExploracionDFS    {
2         /* Al detectar un ciclo, el proceso se aborta, pero hay     *
3          * que verificar que el vértice esté pintado de GRIS y      *
4          * no de NEGRO.                                             */
5     public void yaDescubierto(Nodos u) {
6         if (u.estaEnLaFrontera()) {
7             throw new IllegalArgumentException
8                 ("La digráfica contiene AL MENOS un ciclo.");
9         }
10    }
11
12      /* Ata todos los cabos sueltos del proceso y lo cierra.     *
13       * Emite el mensaje de que no hubo ciclos.                  */
14    protected void cierraExploracion() {
15        super.cierraExploracion();
16        mensaje("La gráfica NO contiene ciclos.");
17    }
18 }
```

Como lo hemos hecho hasta ahora, es importante demostrar la correctez de este algoritmo y determinar la clase de complejidad en la que se encuentra. Esto lo hacemos en el teorema de la correctez para ExploracionDFSCiclos:

Teorema 5.17 (Correctez de ExploracionDFSCiclos) *El algoritmo ExploracionDFSCiclos determina si una digráfica $G = (V, E)$ es acíclica, y su complejidad es $O(|V| + |E|)$.*

Demostración:

Que el algoritmo determina si una digráfica es o no acíclica se sigue del Corolario 5.16. Por otro lado, si no suspende su funcionamiento con este mensaje, quiere decir que la condición en la línea [6] del Listado 5.8 no se cumple, por lo que no se detectó nunca un arco hacia atrás que cierre un ciclo. En este caso, antes de terminar la ejecución del algoritmo, el mismo emite el mensaje de que no se encontraron ciclos.

Es claro, asimismo, que la especialización ExploracionDFSCiclos está en la misma clase de complejidad que ExploracionDFS. Para mostrar que así es, hay que notar que el código que se agregó, para emitir alguno de estos dos mensajes, es de orden constante, por lo que ambas especializaciones están en la misma clase complejidad, que es $O(|V| + |E|)$. ❑

5.4.2. Ordenamiento topológico[4]

Es común tener un orden parcial entre eventos (o elementos) donde si bien un elemento es definitivamente menor que otro (un evento debe realizarse previo al otro), hay varios elementos que no tienen ninguna relación entre sí (no importa el orden relativo en el que se lleven a cabo). Como ya mencionamos, este tipo de relación se puede representar de manera adecuada con una digráfica, donde hay un arco desde el vértice u al vértice v si es que el evento representado por u se debe realizar antes que el representado por v (seguimos suponiendo que el peso del arco es unitario). Claramente la digráfica es acíclica.

Para obtener un orden topológico en una digráfica, basta ejecutar exploracionGenerica de ExploracionDFS sobre la gráfica, marcando los tiempos en los vértices. El orden topológico estará dado por los tiempos fin[v], ordenados éstos de manera descendente: esto es, el último vértice que se termina de explorar es la primera actividad que se debe realizar.

Es razonable que ExploracionDFS entregue, en orden inverso, el orden en que se tienen que llevar a cabo las distintas actividades. Por ejemplo, las raíces de los distintos árboles G_π que se construyen, dado que son las últimas en terminarse de explorar (respecto a su propio subárbol), tendrán un tiempo fin mayor que todos los vértices alcanzables desde ellas (todas las actividades para las cuales la actividad en la raíz ya tuvo que ser terminada, antes de que ellas se puedan llevar a cabo); por ello, aparecerán antes en la lista descendente de estos tiempos, lo que indica que se deben realizar antes.

Sin embargo, si procedemos a ejecutar ExploracionDFS y después ordenamos los tiempos, la complejidad del algoritmos se va a incrementar en términos de lo que nos lleva el ordenamiento (que no se puede hacer en menos de $\Omega(n \log_2 n)$), por lo que debemos embeber este ordenamiento en ExploracionDFS sin incrementar la complejidad del algoritmo. Esto es sencillo, ya que podemos meter a una pila a los vértices conforme se les pinta de NEGRO, logrando, para obtener un orden descendente, que el último vértice en pintarse de NEGRO (la raíz) sea el primero en ser reportado. De donde agregamos una pila en la especialización ExploracionDFSTopologica de ExploracionDFS. En el método que cierra la exploración simplemente tenemos que vaciar el contenido de la pila. La definición de esta clase se encuentra en el Listado 5.9.

Listado 5.9: Especialización para la clase ExploracionDFSTopologica 1/2

```
1 public class ExploracionDFSTopologica
2                         extends ExploracionDFS    {
3        /* Para guardar los vértices conforme se van cerrando.      */
4     private MiListaLigada orden = null;
5        /* Iterador que impone la disciplina de pila.               */
6     private IteradorPila itOrden;
7        /* Construye una instancia del ordenamiento topológico.     */
8     public ExploracionDFSTopologica(Digrafica g) {
9        super(g);
```

[4] Una presentación más detallada de lo que es un ordenamiento topológico se puede consultar en el Apéndice C, o en [CLRS01].

Listado 5.9: Especialización para la clase ExploracionDFSTopologica 2/2

```
10          if (g instanceof Grafica)
11              throw new IllegalArgumentException("No puede ser"
12                          + " ejecutado en gráficas no dirigidas.");
13          orden = new MiListaLigada();
14          itOrden = new IteradorPila(orden);
15      }
16        /* Detecta la existencia de un ciclo en la digráfica, en      *
17         * cuyo caso no hay orden topológico posible.                */
18      public void yaDescubierto(Nodos u) {
19          if (u.estaEnLaFrontera()) {
20              throw new IllegalArgumentException
21                  ("No hay orden topológico posible ya que la"
22                   + " digráfica contiene AL MENOS un ciclo.");
23          }
24      }
25        /* Procesa el nodo cuando lo termina de explorar. Lo mete     *
26         * a la pila en donde los está ordenando.                    */
27      public void cierraNodo(Nodos centroAccion) {
28          super.cierraNodo(centroAccion);
29          itOrden.agrega(centroAccion);
30      }
31        /* Al terminar la exploración, reporta el contenido de la     *
32         * pila que usó para ordenar.                                */
33      public void cierraExploracion() {
34          super.cierraExploracion();
35          orden.reporta();
36      }
37 }
```

Se agrega un vértice a esta pila cuando se le pinta de NEGRO, lo que sucede en el método cierraNodo, método que se muestra en este mismo código.

Claramente, exploracionGenerica de ExploracionDFSTopologica está en l misma clase de complejidad que ExploracionDFS, $O(|V| + |E|)$. Estamos agregando una operación de costo 1 para cada vértice cuando éste se cierra en el método cierraNodo; el costo de vaciar la pila orden, que se ejecuta una única vez al terminar la ejecución, es de $O(n)$; de esto, la complejidad de exploracionGenerica en ExploracionDFSTopologica aumenta respecto a la que presenta en ExploracionDFS en $c \cdot n$ operaciones, lo que no constituye un cambio en la clase de complejidad respecto a ExploracionDFSTopologica.

Pero falta demostrar que, en efecto, este algoritmo nos entrega un orden topológico de los vértices:

Teorema 5.18 (Correctez de ExploracionDFSTopologica**)** *Sea $G = (V, E)$ una digráfica acíclica. Entonces, exploracionGenerica de ExploracionDFSTopologica produce un ordenamiento topológico de los vértices de G.*

Demostración:

Dado que la ejecución de exploracionGenerica en ExploracionDFSTopologica con las marcas fin es la que produce el ordenamiento, debemos demostrar que dado un arco u→v en la gráfica, fin[v] < fin[u], ya que este arco nos dice que para que se pueda realizar la actividad en v debe haberse realizado ya la de u; esto es, u deberá aparecer antes en el ordenamiento topológico. Veamos el porqué esto se cumple.

Supongamos que el centro de acción es, en un momento dado, el vértice u, y estamos por recorrer el arco u→v. El vértice v no puede estar pintado de GRIS, porque eso querría decir que es un antecesor de u (Lema 5.2 y 3.2) y que u→v es una arco hacia atrás. Pero como G es acíclica, por el Lema 5.16 no puede tener arcos hacia atrás. De esto, v está pintado de BLANCOo de NEGRO.

Si v es un vértice BLANCO, será introducido a la frontera inmediatamente encima de u, y tendrá que ser terminado de explorar antes que u. Como el atributo fin es asignado al terminarse de explorar y se incrementa cada vez que se asigna, v tendrá una marca menor que la que se le asignará a u, con lo que la relación fin[v] < fin[u] se mantiene.

Si v es un vértice NEGRO quiere decir que el valor del atributo fin ya está asignado, y tiene un valor menor o igual que el valor del reloj. La asignación de valor al atributo fin[u] se hará en un momento posterior, por lo que se le asignará un valor estrictamente mayor al que tiene el reloj cuando se cerró v. De esto, nuevamente vemos que se mantiene la relación fin[v] < fin[u]. ❏

Es importante notar que si la digráfica no es acíclica no podemos eliminar el caso de que el vértice destino sea un vértice GRIS. Lo que sucede es que si tenemos un ciclo en la gráfica, el ciclo indica una dependencia mutua entre actividades: a tiene que realizarse antes que b, pero el ciclo indica que b tiene que realizarse antes que a, y, por supuesto, no es posible satisfacer estas dos condiciones.

5.5. ExploracionDFS en gráficas no dirigidas

Al igual que en ExploracionBasica, el manejo de gráficas no dirigidas (o simplemente gráficas, en adelante) no agrega complejidad al algoritmo y lo único que hay que hacer, que se hereda junto con ExploracionAbstracta para gráficas no dirigidas, es verificar que la arista que se pretende usar en una dirección no ha sido utilizada antes en la otra dirección. Pero esto se implementa a nivel de la clase para gráficas no dirigidas, que redefine el método que entrega la siguiente arista.

El comportamiento de la exploración procede exactamente igual, excepto que un vértice no se va a poder cerrar hasta en tanto no se hayan usado todas las aristas incidentes en él, a diferencia de cuando se trabaja con digráficas, que un vértice puede tener arcos disponibles llegando a él aunque ya no tenga ningún arco disponible saliendo de él. Esto cambia la clasificación de las aristas respecto a la exploración. Esta característica la enunciamos en el Teorema 5.19.

Lema 5.19 *La ejecución de exploracionGenerica en ExploracionDFS cuando es ejecutado sobre una gráfica no dirigida, clasifica a las aristas de una gráfica en dos grupos:*

i. Aristas de G_π.
ii. Aristas hacia atrás.

Demostración:

Debemos demostrar que no va a ser posible que exploracionGenerica de ExploracionDFS, cuando se ejecuta sobre una gráfica no dirigida, encuentre una arista hacia adelante o una arista de cruce. Veamos el primer caso.

Arista hacia adelante: Por contradicción, supongamos que tenemos una arista $e = v_i - v_j$ hacia adelante – como se muestra en la Figura 5.13(a), donde posibles candidatos son las aristas marcadas con (a) y (b). Como es una arista hacia adelante y no pertenece a G_π, en el momento en que e es elegida para ser recorrida, v_j no puede estar pintado de BLANCO. Como es arista hacia adelante debe ir de un vértice a algún descendiente de él, de donde v_j es descendiente de v_i en G_π. Entonces,

i. Por el Lema 5.4, al momento de agregar a v_j a la pila, v_i se encuentra en la pila, más cerca del fondo que v_j.
ii. Para recorrer la arista $e = v_i - v_j$, se hace desde v_i (para que sea "hacia adelante").
iii. Para que v_i sea centro de acción en un momento posterior al que se descubrió a v_j, se requiere que se haya sacado a v_j de la pila.
iv. Al sacar a v_j de la pila se le pintó de NEGRO.
v. Para que v_j sea pintado de NEGRO se requiere que todas las aristas incidentes en él ya hayan sido recorridas.
vi. ¡Cuando se pintó a v_j de NEGRO, la arista $v_i - v_j = v_j - v_i$ no había sido recorrida, puesto que se intenta recorrer desde v_i!

$\therefore$ La arista $e = v_i - v_j$ no puede tener dirección asignada desde v_i hacia v_j (hacia adelante).

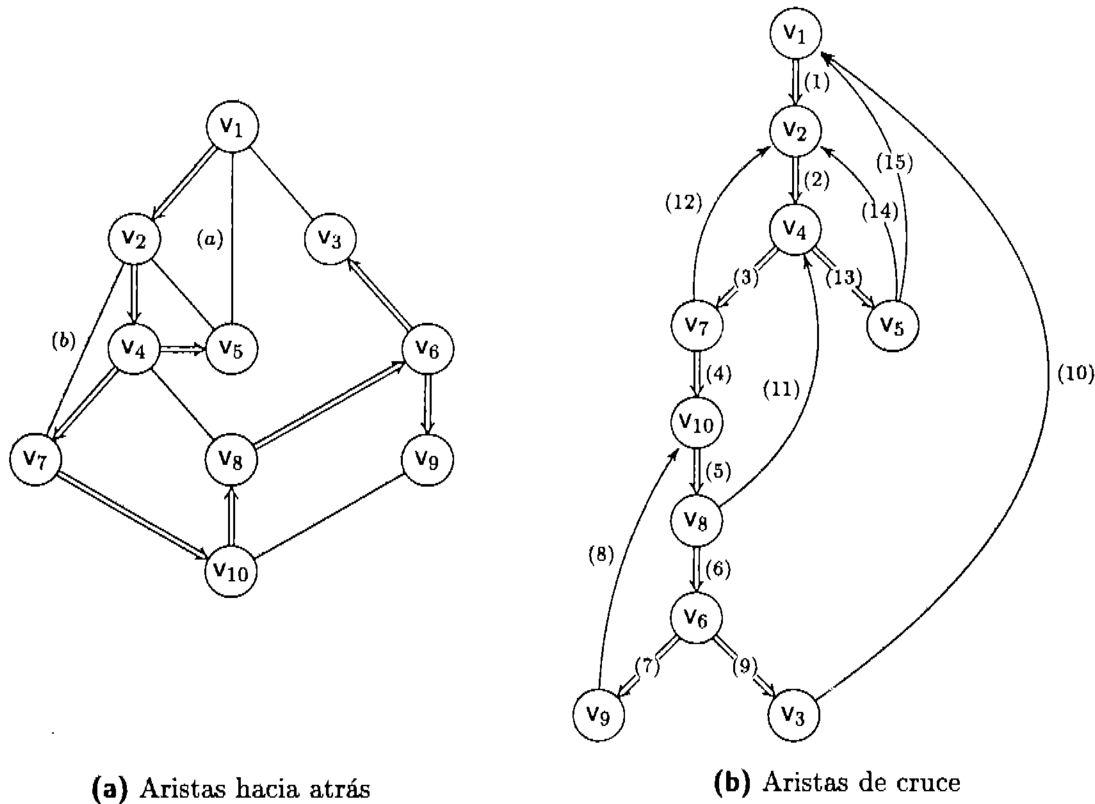
La demostración de que tampoco pueden existir aristas de cruce es similar, por lo que se omite. ❑

En la Figura 5.12 se muestran la gráfica original, y a su lado, al árbol G_π. En este último se ve claramente la dirección de las aristas que no quedan en G_π.

La ejecución de exploracionGenerica de ExploracionDFS, como en el caso de ExploracionBFS en gráficas no dirigidas, orienta a las aristas. Para aquellas aristas que finalmente son incluidas en G_π esta dirección es desde la raíz hacia las hojas, y para las aristas hacia atrás, es desde un descendiente hacia un ancestro en G_π. En ambos casos, esta dirección indica el vértice origen y destino en el momento en que la arista fue recorrida.

Con la gráfica redibujada en la parte derecha de la Figura 5.12, queda más claro que las aristas que no pertenecen a G_π van siempre hacia atrás. Sobre cada arista aparece un número entre paréntesis, que corresponde al orden en que la arista fue recorrida.

Figura 5.12: Imposibilidad de la existencia de aristas hacia adelante o de cruce



(a) Aristas hacia atrás

(b) Aristas de cruce

Dado que exploracionGenerica de ExploracionDFS orienta a las aristas, resulta conveniente definir E_π en términos de E, para poder comparar estos conjuntos:

$$E_\pi = \{\mathsf{e} = \mathsf{u}{-}\mathsf{v} \mid \mathsf{u}{-}\mathsf{v} = \mathsf{v}{-}\mathsf{u} \in E, \mathsf{v}.\pi = \mathsf{u}\}$$

El Lema 5.19 nos dice que en una gráfica, si al ejecutar exploracionGenerica de ExploracionDFS queda alguna arista no incluida en G_π, esa arista queda con dirección asignada hacia atrás, lo que indica que la gráfica original tiene ciclos, como lo demostramos en el Teorema 3.19.

5.6. Aplicaciones de ExploracionDFS en gráficas no dirigidas

En esta sección veremos dos aplicaciones importantes de ExploracionDFS con gráficas no dirigidas. La primera de ellas determina las componentes no separables de una gráfica, mientras que la segunda aprovecha la orientación asignada a las aristas por el algoritmo para encontrar caminos en la gráfica.

5.6.1. Especialización para componentes no separables

Una de las aplicaciones más interesantes de DFS es la de determinar las componentes no separables[5] en una gráfica. Podemos especializar a ExploracionDFS para que determine los componentes no separables de una gráfica aprovechando lo siguiente:

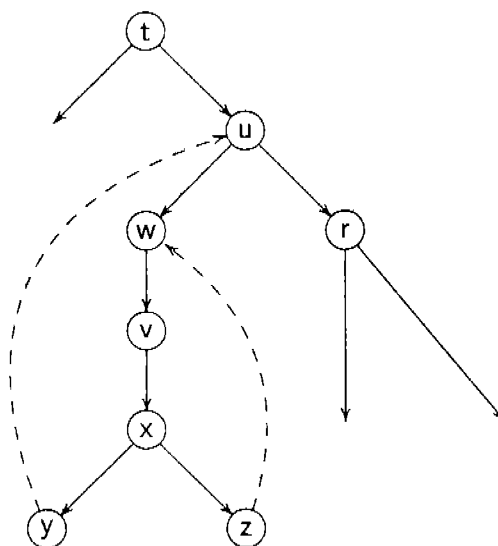
i. En el árbol G_π todos los vértices son de corte, ya que en un árbol existe un único camino entre cualesquiera dos vértices; dado esto, cualquier vértice en ese camino que se quite desconecta a los descendientes de ese vértice de la raíz, separándolo.

ii. Toda arista que no quedó en G_π es una arista hacia atrás – Lema 5.19 – y por lo tanto cierra un ciclo. Como todo ciclo debe estar contenido en una componente, sabremos que todos los vértices que componen el ciclo deberán quedar en la misma componente. Esto tiene sentido ya que entre cualesquiera dos vértices de un ciclo hay al menos 2 caminos, por lo que ningún vértice del ciclo es de corte con respecto a los demás, excepto por aquél que es el más cercano a la raíz al que regresa alguna arista.

iii. Habrá vértices que pertenezcan a más de un ciclo. Entonces, una vez detectado un ciclo y los vértices que lo componen, cualquier otro ciclo que pase por alguno de esos vértices deberá estar en la misma componente, ya que por el Lema C.7, un vértice que no es de corte no puede estar en más de una componente.

iv. Cuando un vértice no puede regresar hacia algún antecesor de él – que no haya ningún camino que termine con una arista hacia atrás entre el vértice y algún vértice antecesor de él – el vértice es un vértice de corte, ya que si se le quita de la gráfica, no habrá forma de alcanzar a ninguno de sus descendientes.

Figura 5.13: Detección de componentes no separables



[5]En el Apéndice C se pueden ver con detalle las definiciones y propiedades básicas relacionadas con componentes no separables en una gráfica.

Dadas estas observaciones, la idea general de la especialización de ExploracionDFS es la de detectar ciclos, buscando cuál es el ciclo que llega a un vértice más cerca de la raíz, más "abajo"[6] en el árbol, con la raíz hacia abajo (¡aunque esto a veces causa confusión, acostumbrados como estamos a pintar los árboles de cabeza!). Marcaremos a cada ciclo con una etiqueta que lo relacione con este vértice, y usaremos para ello el atributo desc (o cualquier contador que vaya marcando a cada vértice conforme se le va descubriendo). Esta es una manera de denotar el nivel al que está regresando el ciclo mediante su arista hacia atrás. Para explicar de mejor manera el mecanismo observemos la Figura 5.13.

En la gráfica de la Figura 5.13 suponemos que:

a. No hay ninguna arista hacia atrás desde ningún descendiente de t, por lo que el vértice t es un vértice de corte para todos los descendientes de él.

b. El ciclo u—w—v—x—y—u debe quedar contenido en una misma componente. En esa misma componente deberá quedar el ciclo w—v—x—z—w, ya que tiene vértices en común con el ciclo anterior.

c. Como ninguno de los vértices descendientes de u tiene aristas hacia atrás que logren ir más cerca de la raíz que el vértice u, u es vértice de corte, ya que si quitamos a u se corta cualquier camino entre vértices fuera de esta componente y los que se encuentran en la misma.

d. De formarse otro ciclo que incluyera al vértice r, pero que tampoco fuera más cerca de s que u, entonces tendríamos otra componente no separable. Estas dos componentes tendrían como vértice en común a u, que como es de corte puede estar en más de una componente.

Veamos ahora cómo tenemos que especializar (o aplicar) ExploracionDFS para que nos entregue las componentes no separables de la gráfica. Usaremos el contador reloj de ExploracionDFS para obtener el orden en que los vértices mediante el atributo desc. Como ya demostramos en este mismo capítulo – Lema 5.9 – si u $\prec_\pi$ v, entonces desc[u] $<$ desc[v]. Veamos algunas definiciones, utilizando el atributo desc que nos auxilian para detectar ciclos y el alcance de las aristas hacia atrás.

Definición 5.1 (punto bajo) El *punto bajo* de v, denotado por v.pBajo, es el menor desc[u] tal que se puede llegar de v a u por un camino dirigido de aristas del árbol, seguidas de **a lo más una arista hacia atrás**.

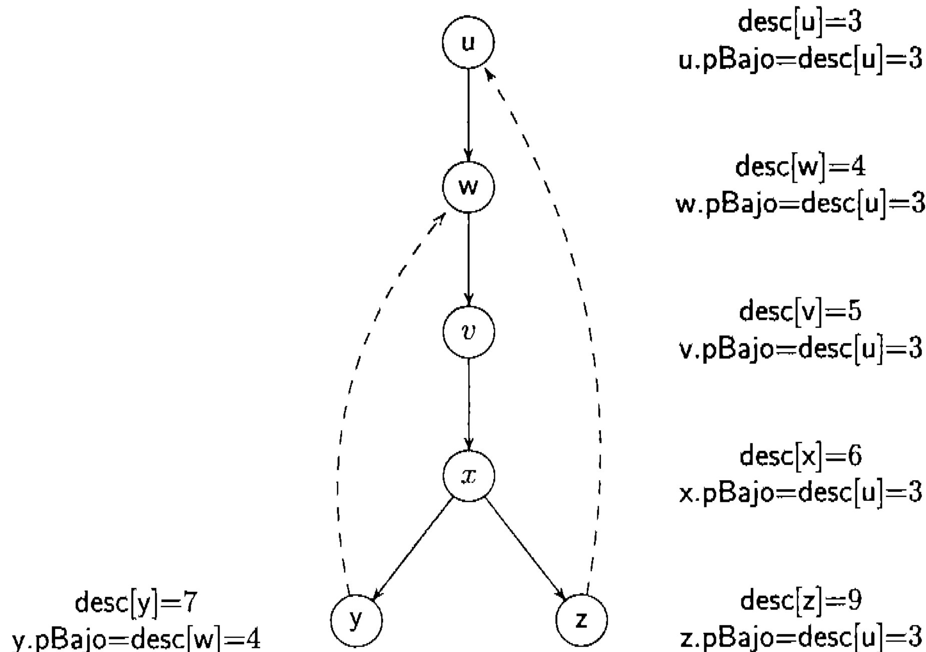
El punto bajo es, entonces, aquel vértice que es el destino de una arista hacia atrás. Nótese que en un recorrido de exploracionGenerica en ExploracionDFS éste es el primer vértice que se descubre, el primer vértice del ciclo que se está cerrando, que fue alcanzado por exploracionGenerica.

En la Figura 5.14 el punto bajo del vértice v es 3, el valor de desc[u], siguiendo el camino v→x→z→u, donde el arco z→u es el último del camino y el único que no está en G_π. De

[6] Visto el árbol como se presenta en la naturaleza.

hecho, para todos los vértices de esta subgráfica – excepto y – se puede regresar siempre al vértice u siguiendo un camino en G_π hasta z y regresando de ahí (mediante un único arco hacia atrás z → u) a u, por lo que el punto bajo de u será el de todos los vértices menos y. El punto bajo de y es el valor de desc[w], ya que con la restricción de utilizar únicamente un arco hacia atrás en el camino "de regreso", y que éste sea el último en ese camino, desde y únicamente se puede regresar a w.

Figura 5.14: Puntos bajos en una gráfica



Especialicemos a ExploracionDFS para que agregue a cada nodo su punto bajo[7]. De esto, la especialización ExploracionDFSPB agrega la declaración de una estructura paralela a la de los vértices de la gráfica, con acceso directo, como se ve en el Listado 5.10. Asimismo agregamos al constructor la construcción de esta estructura.

Listado 5.10: Extensión de ExploracionDFSPB: declaraciones

```
1 public class ExploracionDFSPB  extends ExploracionDFS {
2       /* Estructura para registrar el punto bajo.                 */
3     protected int[] pBajo;
4       /* Se agrega al constructor la construcción de la estruc-   *
5        * tura para los puntos bajos.                              */
6     public ExploracionDFSPB(Digrafica g) {
7         super(g);
8         pBajo = new int[N+1];
9     }
```

[7] Lo haremos de manera similar a como agregamos las marcas de tiempo, para no tener que recurrir a extender la clase Nodos y por ende la clase Grafica para que contenga nodos con este atributo. Lo denotaremos con pBajo[v].

Listado 5.11: Extensión de descubriendo para componentes conexas

```
10      /* Registra el valor de reloj en el momento en que el        *
11      * vértice es descubierto, para el punto bajo.                */
12      protected void descubriendo(Nodos centroAccion, Aristas e,
13      Nodos u) {
14        super.descubriendo(centroAccion, e, u);
15        int iu = u.getPos();
16        pBajo[iu] = desc[iu];
17      }
```

Ilustraremos los puntos a tratar con la Figura 5.15, en la que colocamos los valores desc y pBajo con el formato desc:pBajo abajo de cada vértice. La idea de cómo trabaja esta especialización es la siguiente:

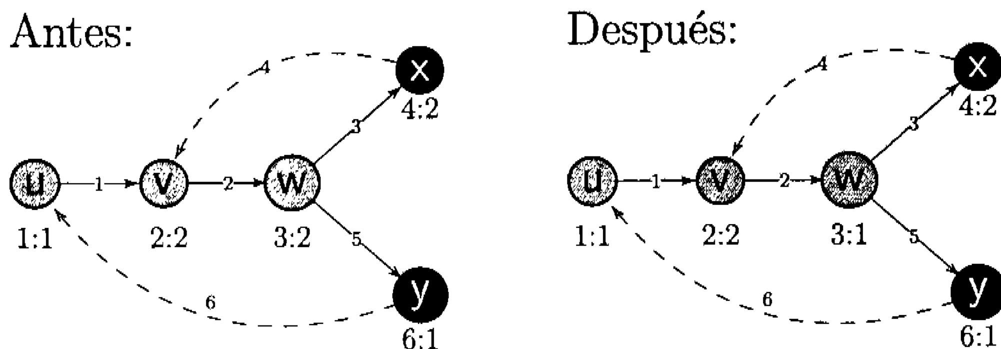
- Construimos el árbol G_π, y conforme vamos descubriendo a los vértices les anotamos que su punto más bajo es su propio desc, ya que en ese momento no sabemos todavía si ese vértice, o alguno de sus descendientes, tiene una arista hacia atrás que termine en un vértice antecesor al vértice recién descubierto pues no se ha recorrido ninguna arista incidente en él. Esto se hace en el método descubriendo – ver Listado 5.11.

- Al estar explorando desde el centro de acción v, supongamos que encontramos una arista hacia atrás. En ese momento modificamos el valor de pBajo[v], ya que siguiendo esa arista se puede llegar a un vértice antecesor de él. Esto lo hacemos en el centro de acción del algoritmo, al recorrer una arista hacia atrás, por lo que se presenta en el método yaDescubierto. Sin embargo, debemos verificar que el centro de acción v no pertenezca ya a algún ciclo que llegue más abajo del que acabamos de encontrar. Si esto último sucede, pBajo[v] tendrá ya un valor inferior al que intentamos asignarle, por lo que ya no deberá modificarse. Por ello, el valor que se asigna al atributo pBajo del vértice es el mínimo entre el que pBajo[v] ya tiene y el que se acaba de encontrar. En el caso de la Figura 5.15-(a), en el vértice x tenemos la arista marcada con 4 que llega al vértice v, por lo que pBajo[x] toma el valor de pBajo[v].

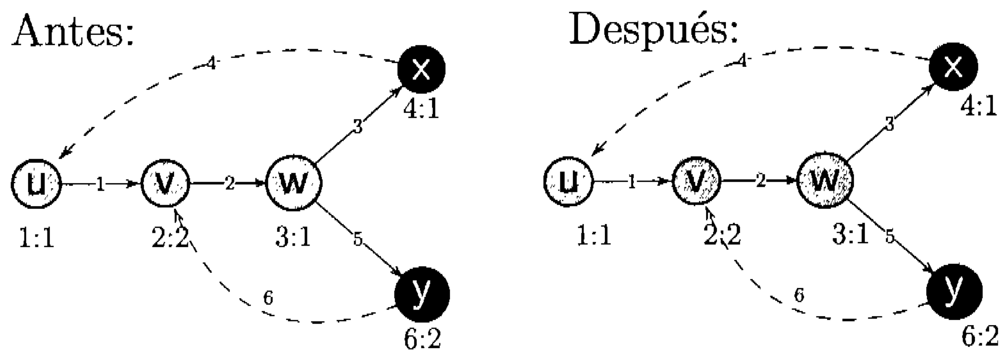
- Supongamos ahora que terminamos de explorar al vértice v, y que este vértice tiene al menos una arista hacia atrás – pBajo[v] < desc[v]. Debemos pasar esta información al padre de ese vértice, pues el padre puede usar esa misma arista para bajar hacia un antecesor de él, si este es el caso y si es que no tiene ya una mejor. Esto lo hacemos en el momento de terminar de explorar a v, modificando al padre del vértice a que su punto bajo sea el mínimo entre el que tenía y el del vértice que se está pintando de NEGRO. Si el padre tenía ya una opción mejor, el punto bajo registrado previamente va a ser menor y el atributo no se modificará. También, si es que la arista hacia atrás de v no es hacia algún antecesor de su padre, el valor de pBajo[v.π] = desc[v.π] será menor que el de pBajo[v], por lo que tampoco será modificado. En la Figura 5.15 mostramos dos casos posibles. En la subfigura (a), w tiene dos descendientes, x y y. Al explorar a los descendientes de w, se descubre primero a x, por lo que la arista etiquetada con 4 se recorre antes que la arista etiquetada con 6 – las etiquetas de las aristas marcan el orden en que se les usa. Como la arista 4 regresa menos que la arista 6, y ambas aristas

son incidentes desde descendientes del vértice w, al terminar de explorar al vértice y, en el caso (a) se modifica el punto bajo del padre de y, w, mientras que en el caso (b), al terminar de explorar al vértice y no se modifica el punto bajo de su padre w, ya que pBajo[w], dado por pBajo[x], es menor que pBajo[y]. Esto se hace en el método cierraNodo.

Figura 5.15: Acciones al recorrer la arista 6 (hacia atrás)



(a)



(b)

- Supongamos que ya terminamos de revisar a un vértice dado v (que ya no tiene aristas sin usar). Si ese es el caso, ya no va a haber, en el futuro, ninguna otra arista hacia atrás que se origine en ese vértice, pues ya usamos todas. Entonces, el valor de pBajo[v] para ese vértice ya quedó determinado, como el mejor de entre todas sus aristas hacia atrás, o la asignada por alguno de sus descendientes que ya están también pintados de NEGRO – Lema 5.5.

La clase que corresponde a la especialización que determina los puntos bajos de una gráfica es, como ya vimos, ExploracionDFSPB. Las extensiones que nos faltan por hacer a los

métodos yaDescubierto y cierraNodo se presentan en el Listado 5.12.

Listado 5.12: Extensiones para registrar el punto bajo en ExploracionDFSPB

```
18        /* Registra el punto bajo como el mínimo entre el que       *
19         * tiene y el recién encontrado.                              */
20      protected void  yaDescubierto(Nodos centroAccion, Aristas e,
21                                     Nodos u) {
22          super.yaDescubierto(centroAccion, e, u);
23          int iv = centroAccion.getPos();
24          int iu = u.getPos();
25          if (pBajo[iv] > pBajo[iu])  {
26              pBajo[iv] = pBajo[iu]);
27          }
28      }
29        /* Al cerrar un vértice verifica si su padre puede usar     *
30         * su punto bajo.                                            */
31      protected void cierraNodo(Nodos centroAccion) {
32          super.cierraNodo(centroAccion);
33          int ivPi = centroAccion.getPi().getPos();
34          int ica  = centroAccion.getPos();
35          if (pBajo[ivPi] > pBajo[ica])  {
36               pBajo[ivPi] = pBajo[ica];
37          }
38      }
39 }
```

5.6.2. Propiedades de ExploracionDFSPB

Por cómo se calcula el punto bajo de cada vértice, las siguientes son propiedades del punto bajo:

Lema 5.20 pBajo[v] $\leq$ desc[v]*; si se usa un camino vacío de longitud 0, entonces* pBajo[v] $=$ desc[v].

Demostración:

- Cuando se descubre el vértice se inicia pBajo[v] con desc[v] – línea [16] en Listado 5.11 – por lo que se cumple pBajo[v] $\leq$ desc[v].
- Mientras no se cierre el vértice, el único punto en que podemos cambiar el valor de pBajo[v] es en la línea [27] en el método yaDescubierto que se muestra en el Listado 5.12 (pBajo[v] $\leftarrow$ mín{pBajo[v], desc[u]}), cuando desde el centro de acción v sale una arista hacia atrás con el otro extremo en el vértice u. Si llamamos p′ al valor del mín{pBajo[v], desc[u]}, tenemos lo siguiente:
  - desc[u] $<$ desc[v], ya que como v—u es una arista hacia atrás, u fue descubierto antes que v.

– $p' \leq$ pBajo[v] $\leq$ desc[u] $<$ desc[v] si p′ tomó el valor de pBajo[v], o sea que hay ya, entre los descendientes de v, alguna arista hacia atrás que llega "más abajo" de lo que está el vértice u, y

– $p' \leq$ desc[u] $\leq$ pBajo[v] si p′ tomó el valor desc[u], o sea que ésta es la arista hacia atrás que sale de v, de entre las encontradas hasta el momento que salen de v o de algún descendiente de v, que llega más abajo. Como desc[u] $<$ desc[v], tenemos que $p' <$ desc[v].

De cualquier manera, pBajo[v] empieza valiendo desc[v] y en la línea [26] del método yaDescubierto (Listado 5.12) lo único que podemos hacer es que pBajo[v] se haga más pequeño, por lo que si pBajo[v] únicamente ha sido modificado en este método se cumple que pBajo[v] $\leq$ desc[v].

- Al pintarse de NEGRO el vértice v, el atributo pBajo del vértice v.π se modifica (línea [36] del método cierraNodo en el Listado 5.12). Tenemos que ver que esta modificación no hace que la relación de orden pBajo[v] $\leq$ desc[v] deje de cumplirse. Pero si sustituimos a pBajo[v.π], lo haremos por algún valor menor a mín{pBajo[v.π], pBajo[v]}; por lo que también esta modificación al valor de pBajo[v], que empieza valiendo desc[v], como en el caso anterior, es sustituido por un valor menor, por lo que la relación pBajo[v] $\leq$ desc[v] se mantiene también bajo esta asignación.

- Si el camino es vacío, de longitud 0, quiere decir que no hay ninguna arista hacia atrás que lleva a algún vértice más abajo que el de v, ni saliendo de v ni de ninguno de sus descendientes. Entonces, lo más bajo que se puede ir desde v es v mismo, por lo que pBajo[v] mantuvo el valor dado al inicio, que es desc[v], ya que nunca se modifica una vez asignado cuando es descubierto. ❑

Lema 5.21 *Si se usa un camino no vacío, la última arista es hacia atrás.*

Demostración:

Lo que estamos tratando de hacer es de "regresar" lo más abajo posible. Si la última arista no es hacia atrás, entonces lo último que vamos a hacer es volver a subir por el árbol, ya que todas las aristas del árbol suben. ❑

Cuando un descendiente directo v de un vértice u en G_π tiene un valor en pBajo[v] mayor o igual a desc[u], lo que implica esta relación es que desde v no se puede bajar hacia la raíz del árbol más allá del vértice u. De esta manera sabemos que u es un vértice de corte, y por lo tanto, cualquier camino desde predecesores de u hacia descendientes de u tiene que pasar por u. Tenemos entonces el siguiente lema:

Lema 5.22 *Si* u–v *es una arista en G_π, con*

$$\text{desc[u]} > 1 \quad y \quad \text{pBajo[v]} \geq \text{desc[u]},$$

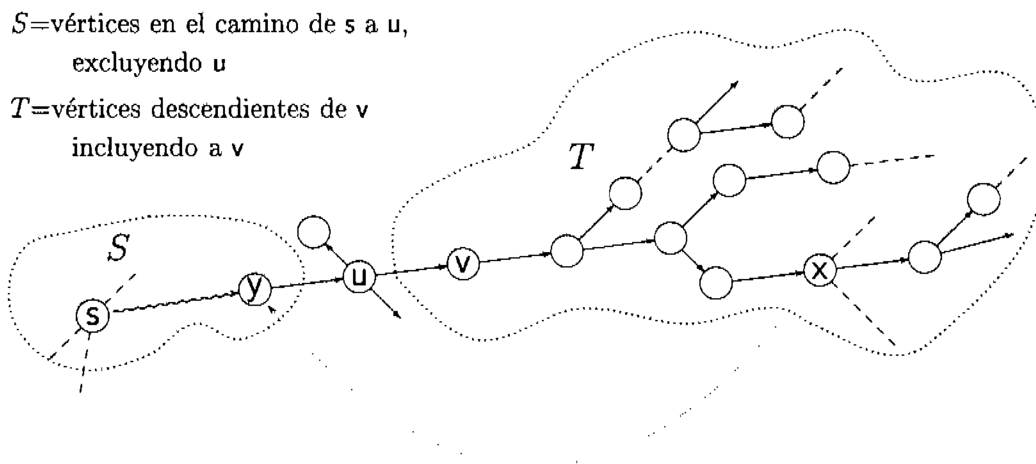
entonces u *es un vértice de corte de G.*

Demostración:

Ejemplificaremos con la gráfica de la Figura 5.16. Por contradicción supongamos que tenemos un vértice u que no es raíz (desc[u] > 1) y que no es vértice de corte, y la arista u–v $\in G_\pi$; demostraremos que en este caso pBajo[v] $<$ desc[u]. Sea S el conjunto de vértices

en el camino de s a u en G_π, pero sin incluir a u (el conjunto de antecesores propios de u en G_π), y sea T el conjunto de descendientes de v en G_π, incluyendo a v. Como u no es vértice de corte existe otro camino desde algún vértice x $\in T$ hacia algún vértice y $\in S$ (se muestra en la figura con arco punteado), de tal manera que entre y y v hay algún camino que no pasa por u (u—y $\rightsquigarrow$ x $\rightsquigarrow$ v). Pero entonces pBajo[v] sería tan pequeño como desc[y], ya que para bajar (o regresar) por el árbol, se puede salir de v, llegar por las aristas de G_π a x, y después usar la arista x—y para regresar. Como y fue descubierto antes que u (u es descendiente de y), desc[y] < desc[u]. Entonces tenemos pBajo[v] $\leq$ desc[y] < desc[u], que es lo que queríamos demostrar.

Figura 5.16: Lema 5.22



❑

Con el Lema 5.22 decidimos si un vértice v $\neq$ s es o no vértice de corte. Para decidir sobre s tenemos el Lema 5.23.

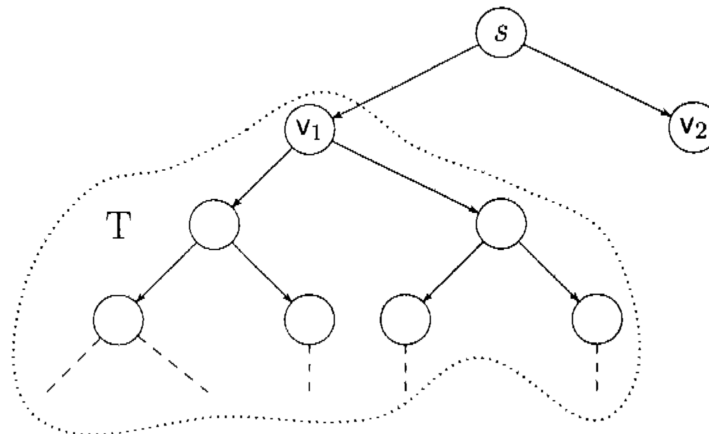
Lema 5.23 *El vértice origen* s *es vértice de corte* $\iff$ *existen al menos dos aristas del árbol* G_π *incidentes en* s.

Demostración:

$\Longrightarrow$ Supongamos que s es vértice de corte y sean $V_1, V_2, \ldots V_k$ la partición de $V - \{s\}$. Ningún camino del árbol G_π que sale de s y lleva a $\mathsf{v}_i \in V_i$ puede llegar a algún $\mathsf{v}_j \in V_j$, $i \neq j$ ya que todos los caminos entre cualesquiera u $\in V_i$ y v $\in V_j$ pasan por s. De donde de s cuelgan al menos dos subárboles, y en uno de ellos está u y en el otro v, por lo que el número de aristas que salen de s en G_π es al menos 2.

$\Longleftarrow$ Supongamos que en G_π salen dos aristas de s, s $\rightarrow \mathsf{v}_1$ y s $\rightarrow \mathsf{v}_2$. Sea T el subárbol que cuelga de v_1 con todas sus aristas hacia atrás. Entonces no existen aristas que vayan de algún vértice en T a algún vértice en $V - (T \cup \{s\})$, porque DFS sobre una gráfica no dirigida no permite aristas que crucen. De donde s separa a T del resto de la gráfica, que no es vacía ya que contiene al menos a v_2, como se puede ver en la Figura 5.17.

Figura 5.17: Lema 5.23



5.6.3. La especialización que determina componentes no separables

Para poder reportar a los componentes no separables, vamos a construir sucesivamente al conjunto S que contendrá a una componente no separable. A S se agregan sucesivamente vértices, hasta que al ir pintando de NEGRO a los vértices, encontramos uno que sea de corte. En cuanto encontramos uno que es de corte, reportamos a ese vértice, junto con todos sus descendientes como una componente no separable. Finalmente podemos determinar, usando el Lema 5.23, si s es o no vértice de corte.

En el Listado 5.13 se encuentra la clase que reporta a los componentes no separables, heredando de la clase que calcula los puntos bajos de una gráfica. Identificamos a esta clase como ExploracionDFSCNS. La única modificación que debemos hacer es la de definir a la estructura S, que deberá tener una disciplina de pila, para poder reportar a un vértice junto con todos los vértices descendientes de él en el árbol G_π, que son los que se encuentren en esta pila encima del vértice de corte. No podemos utilizar para ello a la frontera (con disciplina de pila), porque los vértices que se terminan de explorar se sacan de frontera, mientras que para los componentes no separables debemos mantenerlos hasta que encontremos el punto de corte del que descienden. La declaración y definición de la pila se encuentra en el Listado 5.13, en las líneas [4] y [6].

Cada vez que se inicia el recorrido de una componente conexa de la gráfica, al elegir un nuevo origen para la exploración, la pila S debe vaciarse y se debe colocar al origen de la exploración en ella. Esto se lleva a cabo en ponComoOrigen. La extensión se puede ver en las líneas [17–20] en el Listado 5.13.

Al descubrir a un vértice, éste debe colocarse en el tope de la pila S, lo que se puede ver en la línea [27] del Listado 5.13.

Listado 5.13: Especialización para encontrar componentes no separables

```
1  public class ExploracionDFSCNS extends ExploracionDFSPB {
2          /* Una pila para colocar ahí los vértices que se van      *
3           * cerrando en orden inverso al que se descubrieron.       */
4      protected MiListaLigada S = new MiListaLigada();
5          /* Iterador que impone la disciplina de pila a S.          */
6      protected IteradorDePila itPila = new IteradorDePila(S);
7          /* Constructor de la clase.                                */
8      public ExploracionDFSCNS(Grafica g) {
9          super(g);
10     }
11         /* Además de meter a s a la frontera, lo metemos a la      *
12          * pila de componentes conexas, después de vaciar la       *
13          * pila, si es que contiene algo de la ejecución           *
14          * anterior.                                               */
15     public void ponComoOrigen(Nodos s) {
16         super.ponComoOrigen(s);
17         if (S.notVacia()) {
18             S.reportaPila();
19         }
20         S.agrega(s);
21     }
22         /* Cuando se descubre un vértice se le agrega a S.
23         */
24     protected void descubriendo(Nodos centroAccion, Aristas e,
25                                 Nodos u) {
26         super.descubriendo(centroAccion, e, u);
27         S.agrega(u);
28     }
```

Finalmente, al terminar de explorar un vértice es cuando se puede determinar si el vértice padre del mismo es vértice de corte. Si lo es, se procede a reportar a ese vértice, junto con todos sus descendientes (que se encuentran en la pila S) como una componente no separable. Se sacan de la pila S a todos los vértices de esa componente no separable, excepto al vértice de corte, ya que éste puede formar parte de alguna otra componente no separable y aún no ha sido terminado de explorar. El código correspondiente se muestra en las líneas [34–45] del Listado 5.14.

Listado 5.14: Modificación al terminar de explorar. 1/2

```
29         /* Al terminarse de explorar un vértice, se determina si   *
30          * su padre es punto de corte, y si lo es, se sacan de la  *
31          * pila a todos los vértices introducidos después que él.  */
32     protected void cierraNodo(Nodos centroAccion) {
33         super.cierraNodo(centroAccion);
34         Nodos u = centroAccion.getPi();
```

Listado 5.14: Modificación al terminar de explorar. 2/2

```
35          int iv = centroAccion.getPos();
36          int iu = u.getPos();
37          Nodos v;
38          reporta("Ésta es una componente no separable:");
39          if (pBajo[iv] >= desc[iu]) {
40              while (u != (v = S.elige())) {
41                  S.quitaNodo(v);
42                  reporta(v);
43              }
44              reporta(u);
45          }
46      }
```

Correctez de especialización para componentes no separables

Mediante los lemas y teoremas de esta sección, establecimos ya que la especialización ExploracionDFSCNS trabaja correctamente y entrega las componentes no separables de una gráfica. La complejidad está dada en el Teorema 5.24

Teorema 5.24 *La complejidad de exploracionGenerica cuando se ejecuta desde un objeto de la clase ExploracionDFSCNS es $O(|V| + |E|)$.*

Demostración:

Tenemos que ver de qué manera, si es que, cambia la complejidad de exploracionGenerica en ExploracionDFSPB cuando es llamado desde un objeto de la clase ExploracionDFSCNS. Examinemos los cambios que hicimos:

i. Agregar a s a la pila S. Esto provoca que la constante para ponComoOrigen suba en a lo más tres unidades.

ii. Vaciar la pila S cuando se elige un nuevo origen. Agregadas todas las veces que se vacía la pila, nunca se pueden sacar más de n vértices de la pila, puesto que cada vértice se agrega a lo más una vez. Esto hace que el coeficiente para el término en n crezca en una unidad.

iii. Agregar a un vértice a la pila S cuando se le descubre. Nuevamente el coeficiente para el término en n crece en tres unidades.

iv. Determinar si un vértice es punto de corte. Se hace para cada vértice que se cierra, por lo que el coeficiente para el término en n crece en tres unidades, por los accesos a los atributos pBajo y desc, y la comparación entre ellos.

v. Vaciar la pila S cada vez que se cierra un vértice cuyo padre es punto de corte. Esto incrementa el coeficiente del término en n en 3 unidades.

vi. Reportar al vértice de corte sin sacarlo de la pila. Esto se va a hacer tantas veces como el número de intersecciones entre componentes no separables, pero nuevamente está acotado por n en toda la ejecución del algoritmo.

Resumiendo, la clase de complejidad a la que pertenece exploracionGenerica de ExploracionDFSCNS es la misma que la clase a la que pertenece exploracionGenerica de ExploracionDFSPB, $O(|V| + |E|)$. ❑

5.6.4. Direccionamiento de calles

Representemos a una cierta sección de una ciudad mediante una gráfica de la siguiente manera:

a. Las esquinas de la sección están representadas por vértices.
b. Las calles están representadas por aristas.

El problema consiste en asignar sentido a cada una de las calles de tal manera que se pueda llegar de cualquier punto de la sección a cualquier otro punto. Como las calles de esta sección son angostas, a cada calle se le puede asignar únicamente un sentido.

Si vemos la definición de arista *puente* que damos en el Apéndice C – Definición C.10 – notaremos que los vértices extremos de una arista puente son vértices de corte en la gráfica correspondiente.

Con esta definición, formalizamos el problema de la siguiente manera:

Sea $G = (V, E)$ una gráfica no dirigida conexa sin *puentes*.

Problema: Elegir una dirección para cada arista en G, de tal manera que la gráfica resultante $\vec{G} = (V, \vec{E})$ sea fuertemente conexa, esto es que exista un camino dirigido entre cualesquiera dos vértices.

Notar: Si G tiene puentes, entonces no es posible resolver el problema pues para pasar de una componente a otra se asigna dirección al puente, que no se puede invertir para regresar.

Solución: Correr exploracionGenerica de ExploracionDFS sobre G a partir de cualquier vértice s, dejando que el algoritmo asigne dirección a las mismas, hacia "adelante" en el árbol para aquéllas que pertenezcan a G_π, y hacia "atrás" para el resto de las calles.

Complejidad: $O(|V| + |E|)$

Correctez: La correctez de esta especialización xse apoya en que la gráfica $\vec{G} = (V, \vec{E})$ es fuertemente conexa si y sólo si $G = (V, E)$ es conexa y no tiene puentes. La demostración de esta propiedad se encuentra en el Apéndice C.

Sabemos que si damos a las calles las direcciones asignadas por la ejecución de ExploracionDFS, tendremos garantizado que siempre habrá forma de llegar de cualquier punto de la ciudad a cualquier otro.

Aún cuando el algoritmo DFS tiene muchas más aplicaciones, con esto damos por terminado este capítulo.

5.7. Conclusiones

El algoritmo DFS es uno de los más versátiles y útiles que se conocen. A lo largo de este capítulo revisamos varias aplicaciones del mismo y vimos como todas ellas se pueden expresar como especializaciones sucesivas del algoritmo genérico de exploración. Asimismo contamos con las demostraciones de este último como punto de partida para la demostración de correctez de cada una de las especializaciones.

Capítulo 6

Especialización para circuitos eulerianos

En este capítulo se presenta una especialización para el problema clásico de los circuitos o trayectorias eulerianas. Esta especialización se aleja de la forma tradicional en que se presenta el algoritmo clásico de Euler, pero precisa de mejor manera la forma de elegir el siguiente vértice como origen de un subcircuito o subtrayectoria. El proceso de cada vértice es, en todo momento, local al vértice y no requiere de información adicional más allá de los arcos (las aristas) incidentes en el vértice que no han sido usados (usadas) para el circuito, para tomar la decisión de cómo proseguir. Es en este sentido en que encaja de manera natural como una especialización de la exploración genérica.

Al final del capítulo se presentan aplicaciones de este tipo de exploración.

6.1. Trayectorias y circuitos eulerianos

Una *trayectoria euleriana* de una digráfica $G = (V, E)$ es un camino $\langle e_{i_1}, e_{i_2}, \ldots, e_{i_m} \rangle$ tal que cada arista de G aparece exactamente una vez. De donde $m = |E|$ es la longitud del camino, mientras que un *circuito euleriano* de $G = (V, E)$ es una trayectoria euleriana de G tal que empieza y termina en el mismo vértice.

Entre otros, en [CLRS01, Har72, Eve79] y en el Apéndice D se presentan la caracterización y propiedades de las gráficas en cuanto a trayectorias y circuitos eulerianos.

Dado que en este caso también se trata de explorar a toda la gráfica, proponemos que las trayectorias y circuitos eulerianos se pueden determinar mediante una especialización del algoritmo genérico.

6.2. Especialización de ExploracionBasica para circuitos eulerianos

Como en los capítulos anteriores, la especialización la daremos primero para digráficas, para después dar lo necesario para que esta especialización trabaje con gráficas no dirigidas.

El algoritmo que construye el circuito euleriano dirigido se puede ver como un caso particular del algoritmo genérico de exploración (ExploracionBasica). Habrá que decidir, entre otras cosas, cuál es la concretización de la frontera, ya que insistimos mucho en la presentación del algoritmo genérico que de eso dependía la complejidad del algoritmo. Empecemos por enunciar el problema:

Problema: Dada una digráfica euleriana $G = (V, E)$ construir un circuito euleriano dirigido a partir de un vértice origen s (la gráfica cumple con el D.3).

Entrada: Una digráfica $G = (V, E)$ euleriana y un vértice origen s$\in V$. Como ExploracionBasica no sabe si la digráfica es o no euleriana, esta especialización deberá agregar un método que verifique esta condición, suponiendo para ello que cada vértice en la digráfica conoce su exgrado y su ingrado, y los puede proporcionar.

Salida: Un circuito euleriano dirigido que empieza y termina en s.

Estructuras de datos: Ésta es la parte donde hay que hacer la mayor cantidad de trabajo, ya que la especialización de ExploracionBasica dependerá de las estructuras de datos que usemos. Pasaremos a revisar cada una de ellas.

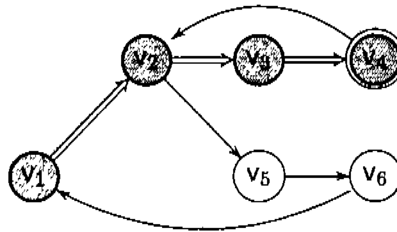
6.2.1. La frontera para la especialización de circuitos eulerianos

Podemos observar una similitud entre el objetivo de encontrar un circuito euleriano y la manera como ExploracionDFS hace su recorrido: construyendo un camino tan largo como sea posible. La diferencia fundamental es que mientras ExploracionDFS no continúa el camino que está construyendo cuando encuentra en ese camino un vértice GRIS, en el caso de la especialización para circuitos eulerianos, el arco que va a un vértice ya descubierto debe agregarse al camino y continuar la exploración en ese vértice.

La caracterización que tenemos de una gráfica euleriana en el Teorema D.3 nos dice que el siguiente vértice a elegir en la frontera debe ser siempre el último vértice incluido en el camino. Queremos que la especialización para circuitos eulerianos (en adelante ExploracionEuleriana), como lo hace ExploracionDFS, vaya metiendo a la frontera a los vértices conforme los va descubriendo, y que en cada momento, el nuevo centro de acción sea el último vértice visitado. Supongamos que tenemos como centro de acción al vértice u y elegimos para recorrer al arco e = u$\rightarrow$v. De acuerdo a la disciplina en ExploracionDFS, u se encuentra en el tope de la pila, y si v es descubierto en esta iteración, se le coloca en el tope de la pila y pasa a ser el siguiente centro de acción; si v ya se encuentra en la frontera, el vértice simplemente es ignorado y el algoritmo no cambia de centro de acción para la siguiente iteración. Pero en el caso de ExploracionEuleriana debemos colocar a v como siguiente centro de acción, independientemente

de que ya se encuentre o no en la frontera. Sin embargo, para preservar las propiedades de ExploracionBasica no debemos volver a meter a v a la frontera, y debemos cuidar que elegir a v sea una operación de complejidad constante. Para ello debemos poder localizar a cualquier vértice dentro de la frontera en tiempo constante, independientemente de su posición dentro de ella.

Figura 6.1: Construcción de una trayectoria euleriana



Por ejemplo, supongamos que tenemos la gráfica de la Figura 6.1. En esta figura, una vez que tenemos el camino $P = v_1 \to v_2 \to v_3 \to v_4$, cuando tratamos de agregar al camino el arco $v_4 \to v_2$ nos enfrentamos a que v_2 ya está en frontera, por lo que no lo podemos volver a meter; pero la construcción del camino nos fuerza a que v_2 debe ser el siguiente vértice que sea elegido como centro de acción. Esto es sencillo ya que:

a. Cuando en el camino aparece un vértice v_i tal que v_i.color = GRIS, lo agregamos al camino sin agregarlo a la frontera. Debemos recordar que tanto lo que tenemos en el camino, como en frontera, son referencias al vértice en la estructura de datos de la digráfica. Trabajar con el vértice en la frontera (o en el camino) es, de hecho, trabajar con el de la digráfica.

b. El siguiente vértice a ser elegido es el último que se agregó al camino. Éste se localiza directamente en la digráfica, desde su referencia en el camino.

c. La siguiente vez que invoquemos a frontera.elige() nos va a regresar precisamente el vértice que deseamos, el que se encuentre al final del camino.

En esta solución estamos suponiendo que todo vértice que entra al camino será metido a la frontera o bien se encuentra ya en ella. En el Lema 6.1 demostramos que así es.

Lema 6.1 *Cuando se está construyendo el circuito euleriano en G, una gráfica euleriana, los vértices que se van alcanzando no pueden estar pintados de* NEGRO.

Demostración:

Supongamos que la gráfica es euleriana – ver Definición D.3 – y por contradicción, supongamos que al recorrer el arco $e = v_a \to u$, u está pintado de NEGRO. Para que u haya sido pintado de NEGRO, se requiere que haya sido centro de acción al menos una vez más que $exgrado$(u). Como la gráfica es euleriana cumple $exgrado(u) = ingrado(u)$, por lo que tuvimos que llegar a u, por arcos distintos, al menos $ingrado$(u) veces. Pero estamos llegando a u una vez más, y esto no es posible ya que las propiedades de ExploracionBasica nos garantizan que los arcos se recorren a lo más una vez, y ya no hay arcos sin recorrer que salgan de u. Por lo tanto, no puede haber un arco sin recorrer cuyo destino sea un vértice NEGRO. ❑

Aseguramos, por lo tanto, que todo vértice que aparezca en el camino en construcción tendrá color BLANCO, en cuyo caso lo meteremos a **frontera** y lo podemos elegir, o bien está pintado de GRIS, en cuyo caso ya se encuentra en **frontera** y también lo podemos elegir. Por lo tanto, no será ningún problema agregar vértices a la frontera y elegirlo, o elegir alguno que ya se encuentre en ella, todo ello en tiempo constante, debido al manejo de referencias que expusimos. El problema puede radicar en eliminar a un vértice de la frontera.

Supongamos que debemos pintar de NEGRO al vértice que es el último en el camino. Su posición dentro de la frontera depende de cuándo fue incluido en el camino por primera vez. Por lo tanto, para localizarlo dentro de la frontera y poder eliminarlo, deberíamos hacer una búsqueda lineal de la frontera. El problema con esta solución es que incrementamos innecesariamente la complejidad del algoritmo, pues nos costaría $O(n)$ sacar a un vértice de la frontera, y no $O(1)$ como está presupuestado. Una estrategia adecuada que no modifica las propiedades de ExploracionBasica es la siguiente: según vayamos descubriendo vértices, al agregarlos a la frontera registramos su posición dentro de ella en el vértice mismo. Esto se logra anotando una referencia a una posición en la frontera, como se muestra en el Listado 6.1. Esta modificación la hacemos en la definición original para la clase Nodos, ya que puede ser útil para otras aplicaciones. Esta referencia será nula si el vértice no está en la frontera, y se referirá a la posición en la frontera en caso de que el vértice se encuentre en la misma.

Listado 6.1: Especialización de los vértices para circuitos eulerianos

```
 1     /* Referencia a la posición en frontera.                      */
 2   protected Object posFrontera;
 3
 4     /* Métodos agregados para este atributo.                      */
 5     /* Al entrar o salir de frontera, actualiza la referencia a   *
 6      * la posición en ella.                                       */
 7   public void setPosFrontera(Object el) {
 8       posFrontera = el;
 9   }
10
11     /* Regresa la referencia a frontera.                           */
12   public Object getPosFrontera() {
13       return posFrontera;
14   }
```

Agregamos a la frontera una pila simple, donde iremos construyendo el camino. De esto, **frontera** contiene dos estructuras, la que registra cuáles vértices han sido descubiertos y se encuentran en proceso, y la pila donde se va construyendo el camino. Tenemos que redefinir a los métodos que meten elementos a **frontera**, ya que ahora, también en el caso de que ya hayan sido descubiertos, deben ingresar al camino. También, por supuesto, tenemos que redefinir al método que selecciona al siguiente centro de acción, utilizando para ello el camino y la estructura original de ExploracionBasica. En el Listado 6.2 mostramos los cambios necesarios en la frontera – omitimos los métodos usados para depuración.

Listado 6.2: Especialización de la frontera. 1/2

```
1  public class FronteraConCamino extends FronteraBasica {
2        /* Guarda el camino conforme lo va construyendo          */
3     private MiListaLigada camino = null;
4        /* Iterador asociado con camino.                          */
5     private IteradorDePila caminoIter;
6        /* Guarda el camino conforme cierra circuitos.           */
7     private MiListaLigada caminoDefinitivo = null;
8        /* Iterador asociado con caminoDefinitivo.               */
9     private IteradorDePila caminoIterDef;
10       /* Constructor que inicia a la frontera y a los caminos. */
11    public FronteraConCamino() {
12        camino = new MiListaLigada();
13        caminoIter = new IteradorDePila(camino);
14        caminoDefinitivo = new MiListaLigada();
15        caminoDefIter = new IteradorDePila(caminoDefinitivo);
16    }
17        /* Agrega un nodo en el tope de camino de trabajo.        */
18    public void pushCamino(Object v) {
19        caminoIter.push(v);
20    }
21        /* Dice si la frontera está o no vacía.                   */
22    public boolean esNoVacia() {
23        Nodos vHacia = null;
24        Aristas e - null;
25        if (caminoIter.esNoVacia())
26            vHacia = (Nodos) caminoIter.peek();
27        while (caminoIter.esNoVacia()
28                && vHacia.yaExplorado()) {
29            /* Quitamos los nodos NEGROs hasta encontrar uno GRIS. */
30            vHacia = (Nodos) caminoIter.pop();
31            if (caminoIter.esNoVacia()) {
32                e = (Aristas) caminoIter.pop();
33                defCaminoIter.push(e);
34            }
35            if (caminoIter.esNoVacia())
36                vHacia = (Nodos) caminoIter.peek();
37        }
38        return caminoIter.esNoVacia();
39    }
40       /* Selecciona al siguiente centro de acción. Si el          *
41        * vértice "al final" del camino tiene aristas disponibles  *
42        * (es GRIS), ése es el siguiente centro de acción. Si es   *
43        * NEGRO, recorre el camino hacia atrás buscando el primer  *
44        * vértice GRIS en el mismo, para ponerlo como centro de    *
45        * acción. Conforme regresa por el camino, va colocando a   *
46        * los vértices NEGROs en el camino definitivo. La frontera *
47        * se llena con parejas (arista, vértice destino), el último*
48        * encima del primero, excepto por el vértice origen que se *
```

Listado 6.2: Especialización de la frontera. 2/2

```
49         * coloca solo en el camino. El vértice se usa para el     *
50         * siguiente centro de acción, la arista para construir el *
51         * camino.                                                 */
52     public Nodos elige() {
53         if (!caminoIter.esNoVacia())
54             throw new EmptyCollectionException
55                 ("¡Tratando de elegir de un camino vacío!");
56         Aristas e = null;
57         Nodos vHacia = null;
58         if (caminoIter.esNoVacia()) // El del tope del camino
59             vHacia = (Nodos) caminoIter.peek();
60         while (caminoIter.esNoVacia()
61                 && vHacia.yaExplorado()) {
62             /* Debemos sacar del camino a todos los vértices       *
63              * NEGROs que están encima de uno GRIS.                */
64             vHacia = (Nodos) caminoIter.pop();
65             if (caminoIter.esNoVacia()) {
66                 e = (Aristas) caminoIter.pop();
67                 defCaminoIter.push(e);
68             }
69             if (caminoIter.esNoVacia())
70                 vHacia = (Nodos) caminoIter.peek();
71         }
72         if (!caminoIter.esNoVacia())
73             return null;
74         return vHacia;
75     }
76 }
```

Notarán en el Listado 6.2, línea [9] que se declara una segunda pila que también tiene que ver con el camino que se está construyendo. Este camino es para registrar losa subcaminos por los que se tiene que retroceder cuando algún vértice ya no tiene salidas. Mostraremos más adelante el papel que juega esta segunda pila, que corresponde a un recorrido inverso del circuito euleriano.

Cada vez que se agrega un vértice a frontera, este vértice se coloca en el tope de la pila que estamos usando para registrar el camino, precedido por la arista que se recorrió para alcanzarlo. Conforme el método elige de la frontera que diseñamos, selecciona al siguiente centro de acción, el siguiente vértice elegido será el que se encuentra en el tope del camino. Para eliminar a un vértice de la frontera, como ésta corresponde a un conjunto implementado con listas doblemente ligadas, simplemente ubicamos el eslabón de la lista en la que se encuentra el vértice que se desea eliminar. Esto se consigue a través de la referencia que anotamos en posFrontera. Una vez localizado este eslabón, se requiere únicamente de dos operaciones para actualizar a la lista.

Un vértice no se puede colocar en el camino cuando ya está pintado de NEGRO, pues no habría tenido aristas a través de las cuales se pudiera llegar a él. Por ello, cuando un vértice se coloca en el camino es GRIS. Como en el caso de las especializaciones que ya vimos, para pintar de NEGRO a un vértice se le tiene que elegir por última vez, y se le saca de la frontera y del camino. En este punto, en el camino pudieran quedar una o más referencias a vértices que ya estuvieran pintados de NEGRO. Veamos por qué.

En la estructura que representa al camino, podemos tener más de una referencia a un mismo vértice. Esto se debe a que se le coloca en el camino cada vez que se llega a él. Pudiera suceder, entonces, que se llega a él por última vez y se le pinta de NEGRO. Sin embargo, en la pila del camino se encuentra una referencia a ese vértice por cada vez que se pasó por él, anterior a la actual. Si intentamos quitar todas las apariciones de un vértice en el camino cuando al vértice se le pinta de NEGRO, por cada vértice que se cierra nos veríamos obligados a recorrer el camino, además de que perderíamos el registro del camino recorrido hasta ese momento. Para evitar esto, y no aumentar la complejidad del método que cierra a los vértices, dejamos a los vértices NEGROs en el camino, y los eliminamos cuando se va a elegir el siguiente centro de acción. De esto, el número de vértices NEGROs que nos podremos encontrar en este proceso es lineal con el número de aristas. Antes de elegir al siguiente centro de acción se verifica que el vértice en el tope del camino no esté pintado de NEGRO. Se tiene acceso al vértice en la frontera (una referencia) pues colocamos en camino una referencia al vértice original, en el que apuntamos la referencia directa a su posición en la frontera. Por lo tanto, dado un vértice en el camino, ubicarlo en la frontera toma dos direccionamientos nada más. Cada vez que el otro extremo del arco sea un vértice GRIS simplemente lo colocamos en el tope de la pila que estamos usando para el camino, y eso lo hará el siguiente vértice a ser elegido, esto sin tocar a la estructura de datos que heredamos de FronteraBasica, frontera. Requerimos, sin embargo, poder agregar vértices al camino, aún cuando no se le esté agregando a la frontera, lo que hacemos en el método pushCamino. Al extender a ExploracionBasica en la clase ExploracionEuleriana lo único que tenemos que hacer es extender el método descubriendo para que agregue al camino el vértice alcanzado, precedido de su arista, y redefinir a yaDescubierto también para que agregue a la arista y al vértice al camino. Ambos métodos deben no tocar frontera, la estructura original, que es la que preserva las propiedades dadas en ExploracionBasica. En el Listado 6.3 mostramos la especialización para ExploracionEuleriana.

Listado 6.3: Especialización para trayectorias eulerianas 1/2

```
 1 public class ExploracionEuleriana extends ExploracionBasica {
 2         /* Constructor. Verifica que la gráfica sea euleriana.      *
 3          * Construye la frontera vacía.                             */
 4     public ExploracionEuleriana(Digrafica g) {
 5         super(g);
 6         if (!euleriana(G)) {
 7             throw new IllegalArgumentException("No se procesa"
 8                         + " la gráfica ya que no es euleriana.");
 9         }
10         frontera = new FronteraConCamino();
11     }
```

Listado 6.3: Especialización para trayectorias eulerianas 2/2

```
1         /* Coloca a s en el fondo del camino.                    */
2     protected void ponComoOrigen(Nodos s)   {
3         super.ponComoOrigen(s);
4         ((FronteraConCamino) frontera).pushCamino(s);
5     }
6         /* Agrega a un vértice GRIS al camino.                     */
7     protected void yaDescubierto(Nodos centroAccion, Aristas e,
8                                                  Nodos u) {
9         super.yaDescubierto(centroAccion, e, u);
10        ((FronteraConCamino) frontera).pushCamino(e);
11        ((FronteraConCamino) frontera).pushCamino(u);
12    }
13        /*  Además de meter al vértice recién descubierto a       *
14         * frontera, lo coloca también en el tope del camino.     */
15    protected void descubriendo(Nodos centroAccion, Aristas e,
16                                                 Nodos u) {
17        super.descubriendo(centroAccion, e, u);
18        ((FronteraConCamino) frontera).pushCamino(e);
19        ((FronteraConCamino) frontera).pushCamino(u);
20    }
21        /* Al terminar la exploración, reporta el camino          *
22         * construido.                                            */
23    protected void cierraExploracion() {
24        super.cierraExploracion();
25        reportaCamino();
26    }
27 }
```

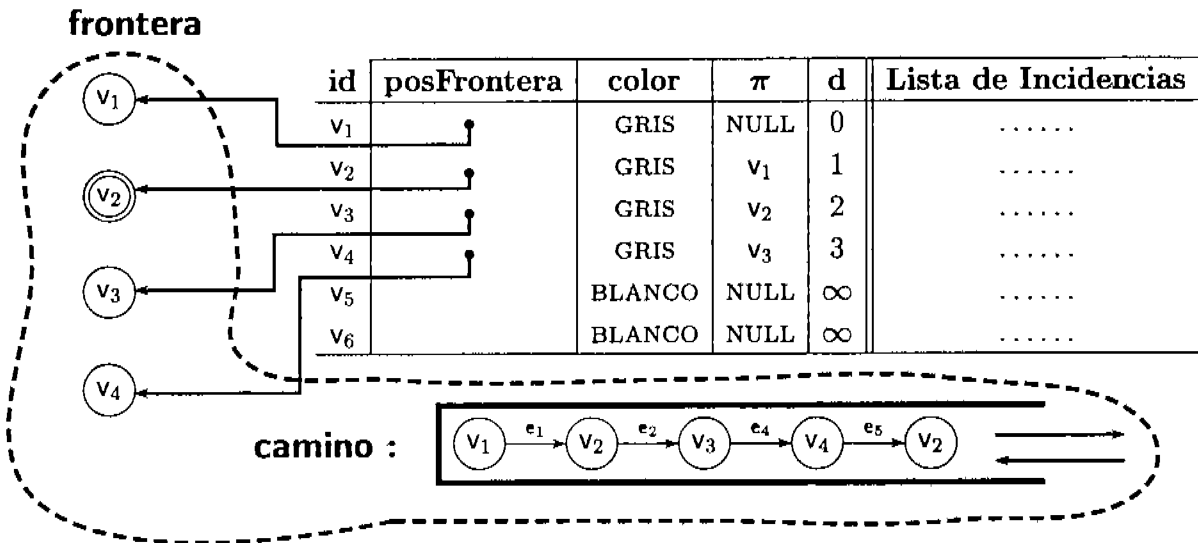
Se extiende, entonces, a los métodos descubriendo y yaDescubierto agregando al camino primero la arista recorrida y encima de ella el vértice destino que se visita – líneas [10–11] y [18–19] en el Listado 6.3. Al cerrar el proceso se reporta el camino definitivo que se haya construido – líneas [23–24] del mismo listado. Al agregar estas operaciones estamos incrementando con una constante la complejidad de cada método, por lo que no cambia la clase de complejidad de esta especialización con respecto a ExploracionBasica.

Para que frontera no esté vacía al iniciar el algoritmo, debemos, como en las otras especializaciones, meter a s a frontera y ponerlo en el camino. s va a ser el único vértice que entre al camino sin su correspondiente arista, ya que no se recorrió a ninguna para llegar a s. El método elige verifica siempre que el vértice seleccionado no sea el primero que se metió, para no intentar sacar de una pila vacía. Para esto se debe extender también el método ponComoOrigen, para que, además de meter a s a la frontera, lo coloque en el fondo del camino. La extensión para este método se encuentra en el Listado 6.3, líneas [3–4].

Veamos en la Figura 6.2, esquemáticamente, cómo quedan las estructuras de datos que acabamos de describir. La figura corresponde al estado de la exploración en la gráfica de la Figura 6.3. Suponemos que ya se han recorrido los arcos marcados con línea doble y que el

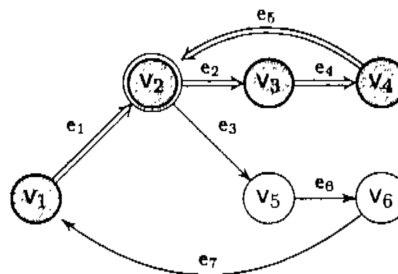
centro de acción se encuentra en ese momento en el vértice v_2, que es el que se encuentra en el tope del camino. Al elemento activo de la frontera lo encerramos con doble círculo. Como se puede observar, la estructura de datos de la frontera puede muy bien ser la de un conjunto (siempre y cuando eliminar a un vértice, una vez seleccionado, o agregarlo tome tiempo constante, lo que se logra con una lista doblemente ligada y referencias desde cada vértice a la frontera). Como ya mencionamos, para el camino tenemos una pila.

Figura 6.2: Estructuras de datos para la especialización de Euler



id	posFrontera	color	π	d	Lista de Incidencias
v_1		GRIS	NULL	0	......
v_2		GRIS	v_1	1	......
v_3		GRIS	v_2	2	......
v_4		GRIS	v_3	3	......
v_5		BLANCO	NULL	∞	......
v_6		BLANCO	NULL	∞	......

Figura 6.3: Trayectoria euleriana representado en las estructuras de la Figura 6.2



El camino definitivo se arma, similar al camino de trabajo, conforme se van sacando vértices NEGROs del camino, en la búsqueda de un vértice GRIS. De hecho, lo que se hace acá es retroceder sobre el camino hasta encontrar un vértice desde el cual posiblemente se puede salir. En lugar de reportar el camino, como lo haría el algoritmo original, lo guardamos en otra pila que refleja el orden en que se regresa sobre el mismo.

Estamos suponiendo que las únicas dos posibilidades para los vértices encontrados al construir un circuito en una gráfica euleriana es que sea vértice BLANCO (se le está descubriendo

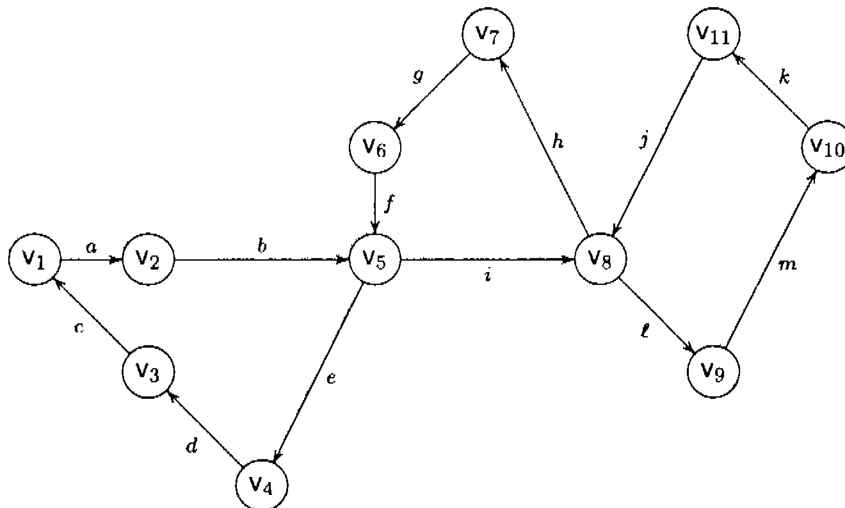
en este momento) o que sea GRIS, esto es que se encuentre en frontera. La imposibilidad de que sea un vértice NEGRO se demostró en el Lema 6.1.

Lo que llevamos hecho hasta ahora establece bien la disciplina para ir visitando los vértices en un orden tal en que, de hecho, se va recorriendo el circuito, hasta el momento en que se llegue a un vértice desde el cual ya no se pueda salir. En términos de ExploracionBasica esto implica que hay que sacar al vértice de frontera, pues ya no tiene arcos disponibles de salida, y pintarlo de NEGRO. Como en camino tenemos en realidad referencias a la estructura de los vértices, al pintar a un nodo de NEGRO todas las apariciones de ese nodo en camino se refieren al nodo pintado de NEGRO. El método que selecciona el siguiente centro de acción tendrá que retroceder a través de estas apariciones de nodos NEGROs, hasta encontrar un vértice GRIS del que salga algún subcircuito, para integrarlo al circuito principal.

Siempre que se regresa por el camino buscando un vértice GRIS, se debe reportar (agregar al camino definitivo) a los arcos que se recorren en ese retroceso.

Finalmente, el proceso de desandar el camino se muestra en el método **elige()**, ya que este método es el encargado de "brincar" los vértices en el camino que ya estén pintados de NEGRO– ver líneas [52–75] del Listado 6.2.

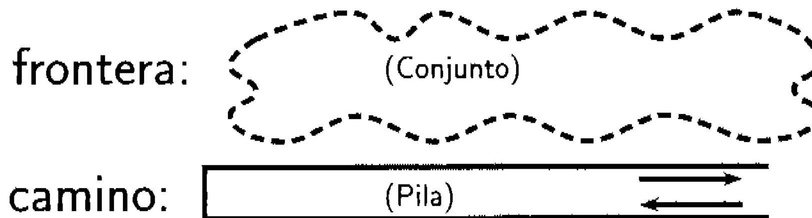
Figura 6.4: Ejemplo de digráfica para la construcción de un circuito



Las estructuras de datos correspondientes se muestran en la Figura 6.5 como se encuentran al iniciarse el algoritmo, una vez construida la gráfica con sus listas de incidencia. Tanto frontera como camino se encuentran vacíos, y no se ha reportado ninguno de los arcos del camino.

Figura 6.5: Estructuras de datos en la construcción de un camino

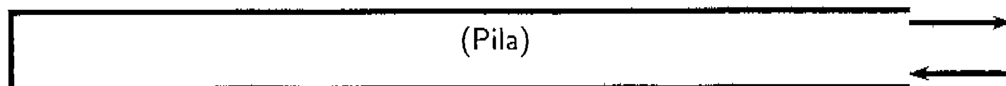
v	color	π	d	lista de incidencia	enFrontera
v_1	BLANCO	NULL	0	→ a → NULL	NULL
v_2	BLANCO	NULL	∞	→ b → NULL	NULL
v_3	BLANCO	NULL	∞	→ c → NULL	NULL
v_4	BLANCO	NULL	∞	→ d → NULL	NULL
v_5	BLANCO	NULL	∞	→ i → e → NULL	NULL
v_6	BLANCO	NULL	∞	→ f → NULL	NULL
v_7	BLANCO	NULL	∞	→ g → NULL	NULL
v_8	BLANCO	NULL	∞	→ h → l → NULL	NULL
v_9	BLANCO	NULL	∞	→ m → NULL	NULL
v_{10}	BLANCO	NULL	∞	→ k → NULL	NULL
v_{11}	BLANCO	NULL	∞	→ j → NULL	NULL



e	recorrido	u	v	peso
a	NO	v_1	v_2	1
b	NO	v_2	v_5	1
c	NO	v_3	v_1	1
d	NO	v_4	v_3	1
e	NO	v_5	v_4	1
f	NO	v_6	v_5	1
g	NO	v_7	v_6	1

e	recorrido	u	v	peso
h	NO	v_8	v_7	1
i	NO	v_5	v_8	1
j	NO	v_{11}	v_8	1
k	NO	v_{10}	v_{11}	1
ℓ	NO	v_8	v_9	1
m	NO	v_9	v_{10}	1

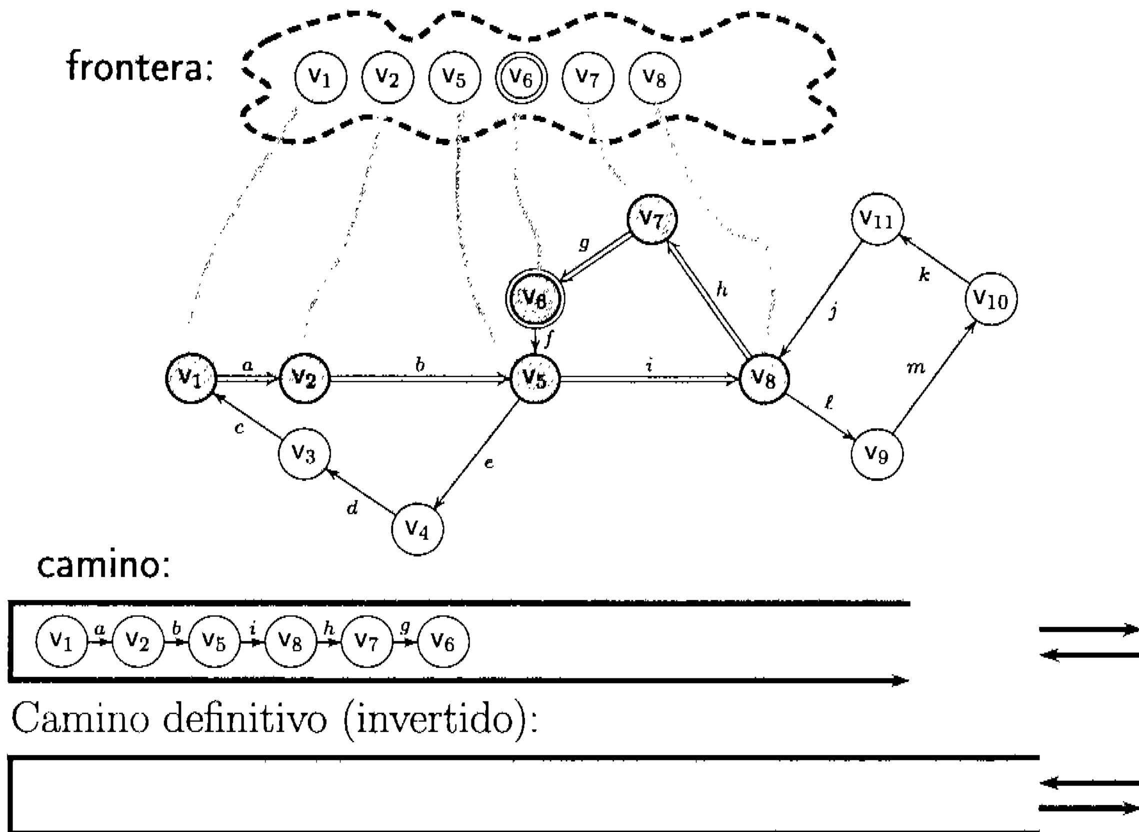
Camino definitivo (invertido):



Apliquemos nuestro algoritmos a la digráfica de la Figura 6.4, que es una digráfica euleriana, eligiendo como vértice origen a v_1.

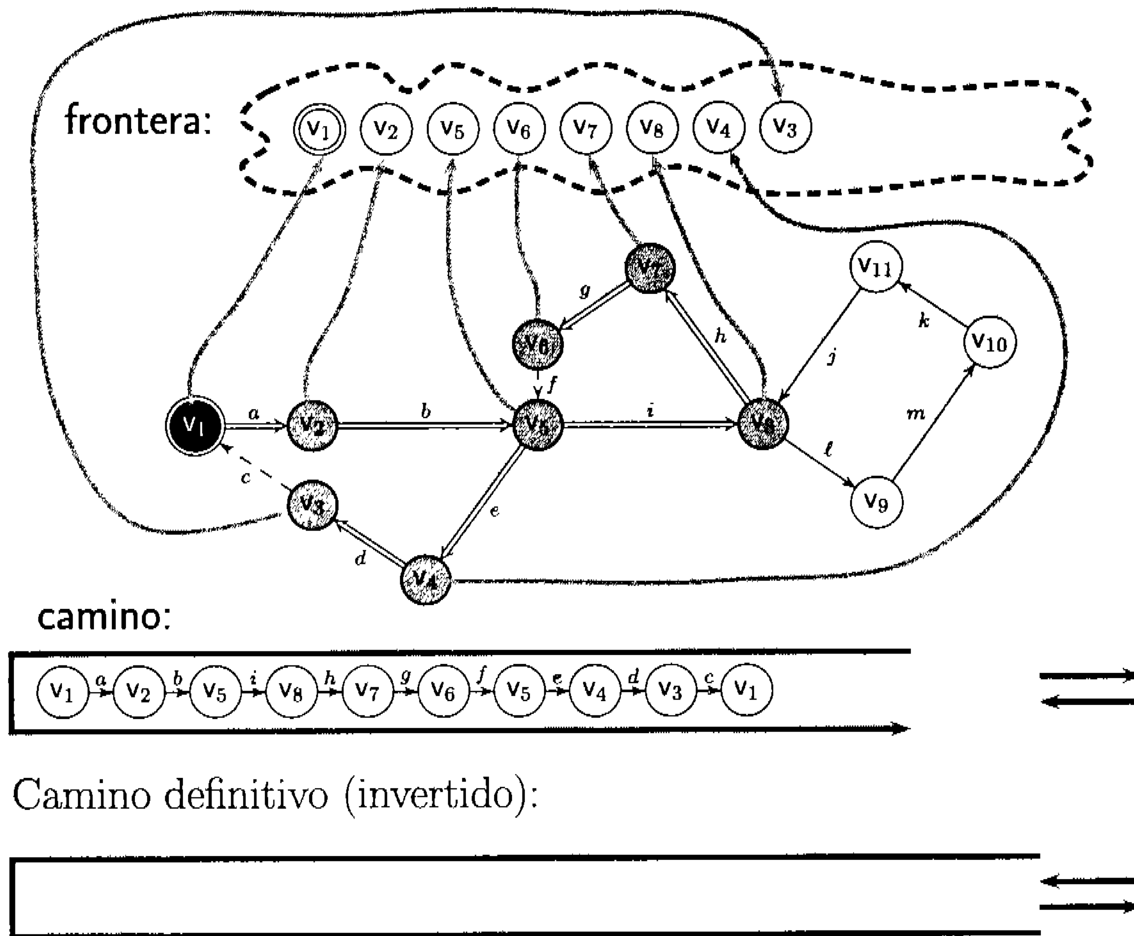
Después de ejecutar la construcción de la clase, lo primero que se hace en exploracionGenerica es colocar al vértice origen en la frontera y en el camino, para elegirlo como primer centro de acción. Conforme se descubren los vértices v_1, v_2, v_5, v_8, v_7 y v_6, en ese orden (de acuerdo a las listas de incidencias mostradas), se les va colocando en frontera registrando en el vértice una referencia a la posición de cada vértice en la frontera (posFrontera). Asimismo, conforme se van recorriendo los arcos, se van colocando en camino, alternándolos con los vértices destino de cada arco. La pila camino fue definida con el objetivo de ir recordando el orden en que se van recorriendo los arcos. El estado de las estructuras de datos en este momento se muestra en la Figura 6.6. En esta figura no mostramos las estructuras de datos tal cual, sino que presentamos la gráfica para hacer más clara la exposición, suponiendo en todo momento que el contenido de estas estructuras es como se mostró en la Figura 6.4. Usamos nuevamente arcos dobles para indicar a aquellos arcos que pertenecen a G_π. Los arcos que no han sido recorridos se muestran con línea sencilla.

Figura 6.6: Estructuras de datos al descubrir a v_6 y tenerlo como centro de acción



Cuando el algoritmo llega por segunda vez a v_5 simplemente utiliza la referencia a frontera para activar a v_5. Como logra salir de él usando el arco e, sigue construyendo el camino descubriendo a v_4. En este proceso mete a los arcos f, e, d y c a la pila donde va construyendo el camino. Observemos nuevamente el estado de las estructuras de datos cuando el algoritmo regresa al vértice v_1, en el momento inmediato posterior a que pinta de NEGRO a dicho vértice al ya no haber arcos disponibles – ver Figura 6.7.

Figura 6.7: Estructuras de datos al llegar nuevamente a v_1



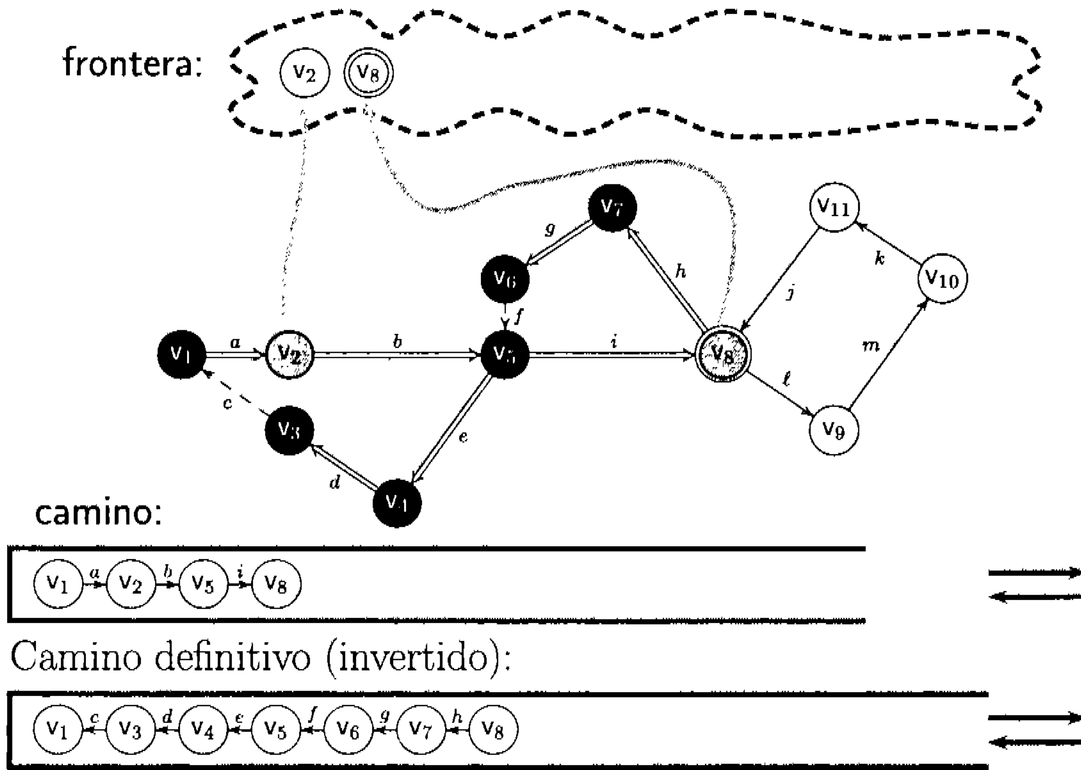
Al sacar a v_1 de frontera procedemos a reportar el arco que se encuentra en el tope de camino, que es el arco c, y se pasa a tener a v_3 como centro de acción, ya que es el vértice origen del arco c. Pero como v_3 no tiene ya tampoco arcos disponibles, es pintado de NEGRO, y se procede nuevamente a reportar al arco d y sacarlo de camino, colocando a v_4 como siguiente centro de acción, ya que es el vértice origen del arco c y se encuentra en frontera. Como v_4 no tiene arcos disponibles, se le pinta de NEGRO. Se procede de la misma manera reportando a los arcos e, f, g y h, pintando de NEGRO, en este orden, a los vértices v_5, v_6 y v_7, dejando como centro de acción a v_8. El estado de las estructuras de datos en este momento se muestra en la Figura 6.8.

Una vez que se tiene a v_8 como centro de acción, de él sí se puede salir, por el arco ℓ, por lo que se procede nuevamente a extender el camino, descubriendo sucesivamente los vértices v_9, v_{10} y v_{11} y metiendo a la pila de camino los arcos ℓ, m, k y j, en ese orden – que es en el que se les fue recorriendo – quedando nuevamente el centro de acción en el vértice v_8 – ver Figura 6.9.

Cuando se tiene nuevamente a v_8 como centro de acción, ya no hay arcos disponibles para

salir de él, por lo que hay que pintarlo de NEGRO y regresar por el camino construido hasta encontrar un vértice GRIS en ese camino. Se reporta al arco j y se tiene a v_{11} como centro de acción, repitiéndose la situación de que hay que pintarlo de NEGRO, por lo que volvemos a retroceder por ese camino. Estas iteraciones se van a realizar hasta que lleguemos a v_8 y lo pintemos de NEGRO, con la situación que se muestra en la Figura 6.10.

Figura 6.8: Estructuras de datos al regresar buscando un vértice GRIS



Como ya no se puede salir de v_8 se le pinta de NEGROy se retrocede por el camino hasta encontrar el primer vértice GRIS, que en este caso es v_2. Se le usa como centro de acción, pero como ya no se puede salir de él, se procede a regresar por el camino, hasta que ya no quedan arcos por retroceder, quedando las estructuras de datos como se ve en la Figura 6.11. Con línea punteada mostramos el recorrido de la gráfica y con línea intermitente el retroceso, para que se pueda apreciar el momento de la construcción del camino y el momento en que se reporta. Se ve de seguir el camino punteado que el circuito está definido siguiendo los arcos en sentido inverso.

Es un poco molesto el que el camino se reporte en sentido inverso a como debe ser recorrido. Para obtener el camino exactamente en la dirección que tienen los arcos basta contar con una segunda pila. En lugar de "reportar" el arco cuando lo recorremos de regreso, lo podemos colocar en la pila, y al final del proceso imprimimos el contenido de la segunda pila conforme la vamos vaciando. Es claro que los arcos van a entrar a la segunda pila exactamente en el mismo orden en que se reportan en la primera versión del algoritmo, y van a salir de ella en orden inverso a como entraron, consiguiendo reportar el camino de manera más adecuada.

Figura 6.9: Estructuras de datos cuando el camino llega a v_8 desde v_{11}

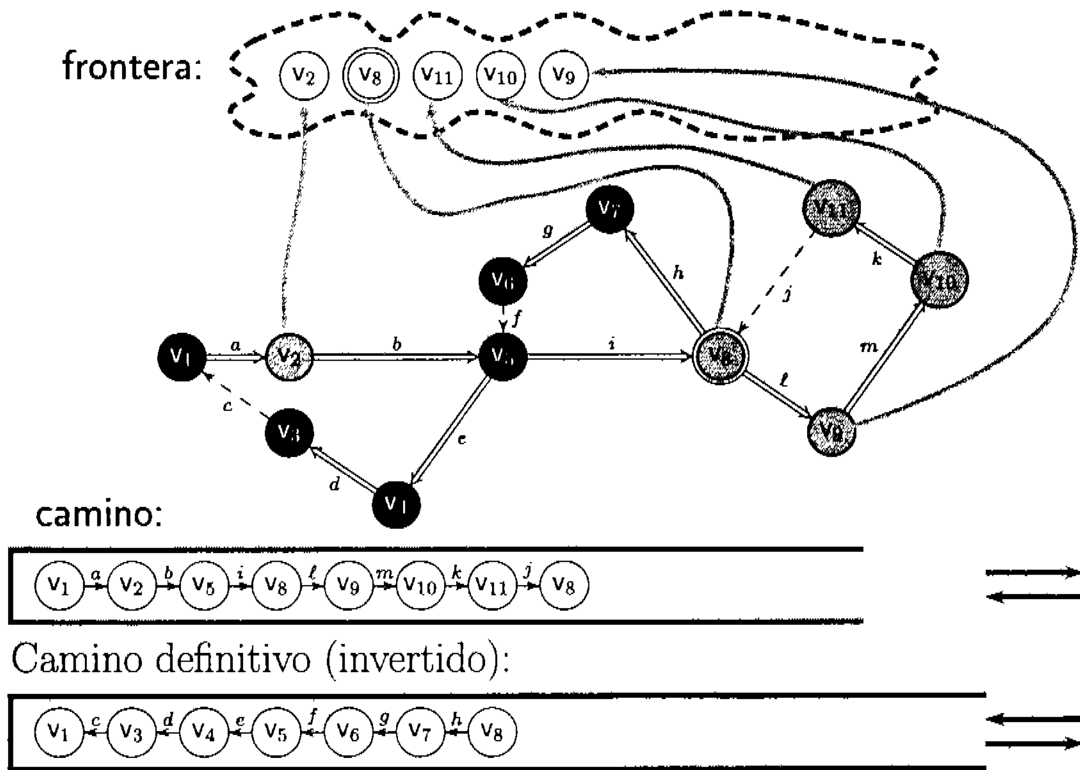


Figura 6.10: Estructuras de datos al retroceder hasta v_8 para pintarlo de NEGRO

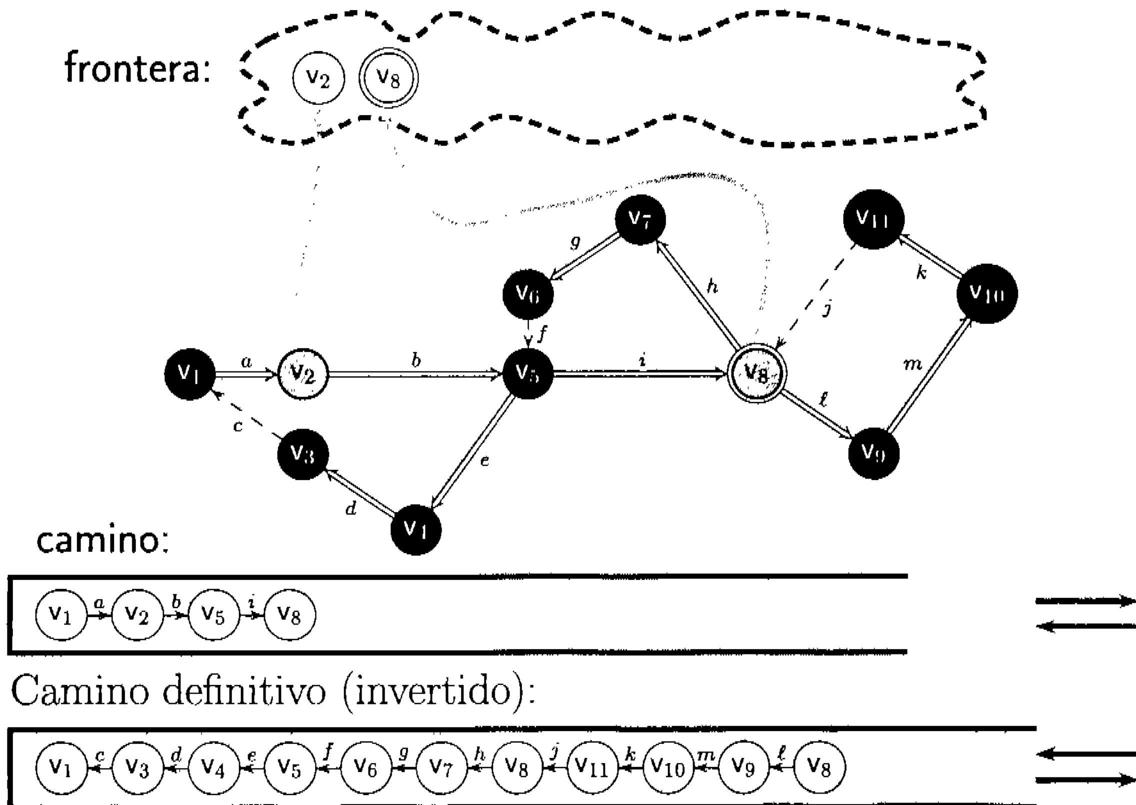
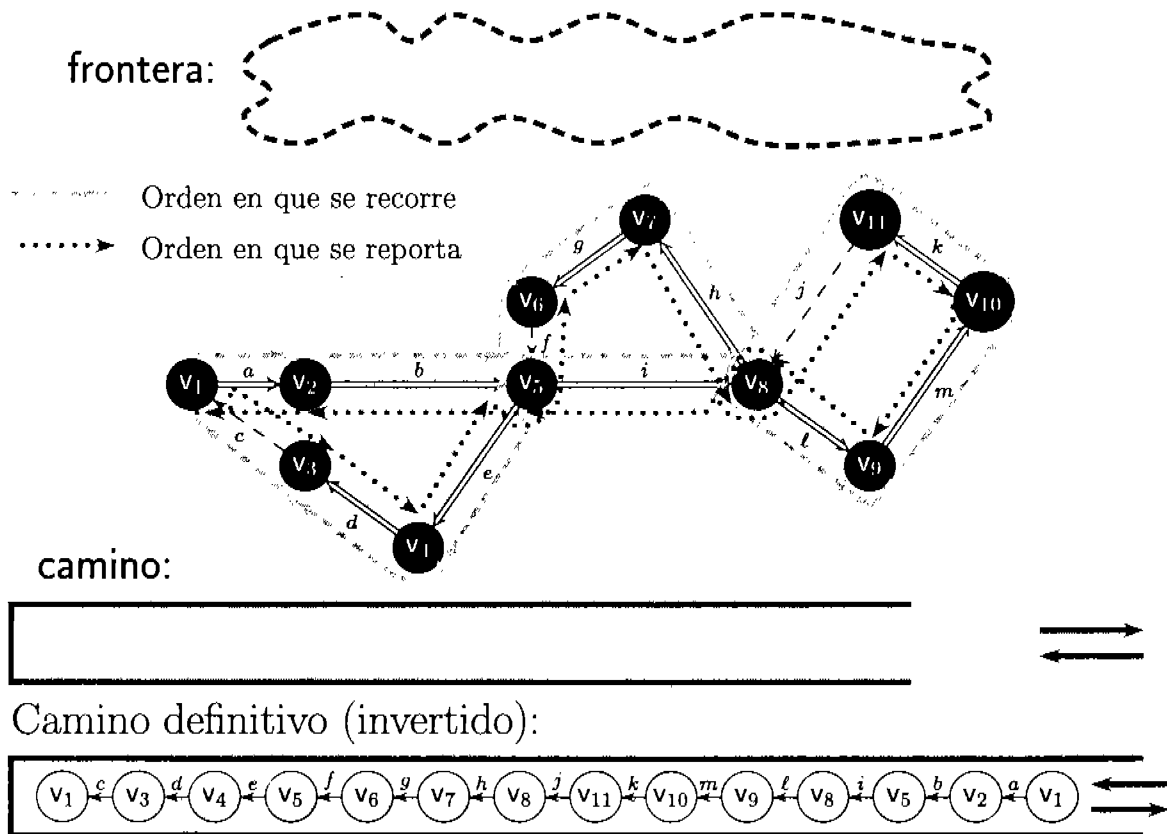


Figura 6.11: Recorrido realizado para la construcción del circuito euleriano



6.2.2. Modificaciones para trayectorias eulerianas

En el caso de que la gráfica tenga un circuito euleriano no va a importar en cuál vértice iniciamos la exploración de la gráfica, pues dado que el circuito existe, éste va a ser construido de manera adecuada (como estableceremos en la siguiente sección). Sin embargo, si lo que tiene la gráfica es uns *trayectoria* euleriana no se puede elegir arbitrariamente el vértice origen de la exploración, sino que éste tiene que ser aquél que cumpla $exgrado(\mathsf{v}_i) = ingrado(\mathsf{v}_i) + 1$, y el camino terminará en aquel vértice que cumpla $ingrado(\mathsf{v}_i) = exgrado(\mathsf{v}_i) + 1$. Esto se debe a que si no se utiliza el arco "excedente" para iniciar el camino, siempre quedará un arco sin recorrer, pues no habrá manera de llegar a ese vértice una vez que se han utilizado todas las parejas de llegada y salida. El camino terminará, en efecto, en el vértice que cumpla tener más arcos de llegada que de salida, pero sabemos que al menos quedó un arco sin recorrer.

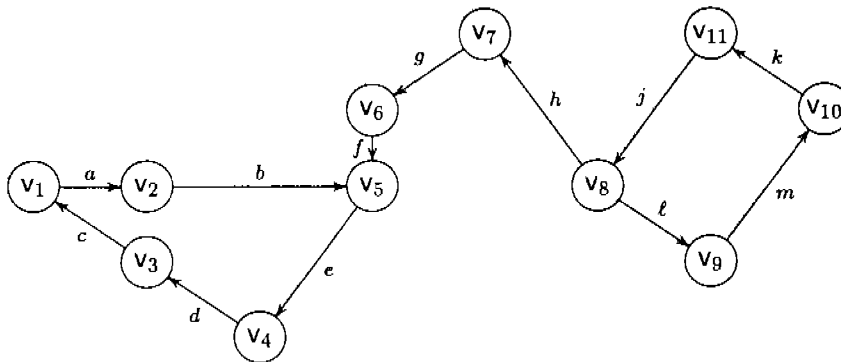
Por ejemplo, en la gráfica de la Figura 6.5, si quitamos el arco etiquetado con i – como se muestra en la Figura 6.12 – debemos empezar el recorrido en v_8. La trayectoria euleriana está dada por los arcos $\langle \ell$, m, k, j, h, g, f, e, d, c, a, b $\rangle$, en ese orden, y no hay ninguna otra posibilidad. Se puede verificar que el algoritmo que dimos construye esta trayectoria con v_8 como origen de la exploración.

Si empezáramos el recorrido en alguno de v_1 a v_5, no exploraríamos ninguno de los vértices v_6 a v_{11}, y si lo iniciamos en alguno de los vértices v_6 a v_{11} que no sea v_8 no exploraremos los

vértices que estén en el camino de v_8 al vértice origen.

El método obtenOrigen debe redefinirse de tal manera que el vértice seleccionado como origen sea el que cumpla con tener exgrado uno mayor que su ingrado. Podemos elegir este vértice en tiempo lineal con respecto a $|V|$ si al construir la gráfica anotamos en cada vértice su exgrado y su ingrado, lo que también requerimos para poder decidir si una gráfica es o no euleriana. Si la gráfica cuenta con estos atributos, en obtenOrigen simplemente se revisa la lista de vértices para ver cuál cumple con poder ser el origen del camino – si la gráfica es euleriana, esto será cierto para un único vértice. Extendemos a la clase ExploracionEuleriana a la clase ExploracionCaminosEulerianos, en la que únicamente se redefine el método obtenOrigen.

Figura 6.12: Ejemplo de digráfica para la construcción de un camino euleriano



6.3. Correctez de ExploracionEuleriana

¿Qué queremos decir con demostrar que ExploracionEuleriana es correcta? Lo principal que queremos demostrar es que dada una gráfica euleriana $G = (V, E)$, exploracionGenerica, invocado desde un objeto de la clase ExploracionEuleriana en efecto construye el circuito euleriano de la gráfica. Como lo hemos hecho hasta ahora, estableceremos primero algunas propiedades de esta especialización, para que la demostración del teorema de correctez sea más tersa.

Como G es euleriana, y ExploracionBasica nos garantiza que recorre a todos los arcos incidentes desde los vértices alcanzables desde s, tenemos, por el Lema D.5 en el Apéndice D, que todos los vértices de G son alcanzables desde s.

Es claro que seguimos cumpliendo el que cada vértice entre y salga de la frontera exactamente una vez. Asimismo, sabemos que cada arco aparecerá exactamente una vez. Ahora deberemos demostrar que lo que construye exploracionGenerica desde ExploracionEuleriana es un circuito.

Lema 6.2 *exploracionGenerica desde* ExploracionEuleriana *construye un circuito.*

Demostración:

Debemos establecer que exploracionGenerica desde ExploracionEuleriana construye un camino que empieza en s y termina en s. Primero mostraremos que es un camino desde s, lo que haremos por inducción sobre el contenido de camino.

Base: Lo primero que hace exploracionGenerica es meter al vértice s a camino y a frontera. La base de la inducción será, entonces, que la primera vez que incluye a una arista en camino, el contenido de camino representa a un camino. Como el método elige toma al vértice que está en el tope de camino para siguiente centro de acción, y al iniciar éste es s, el primer arco a incluir en el camino debe ser un arco $s \to v$. El arco es incluido en camino – seguido de su vértice destino – porque es un arco que existe y que representa a un camino $s \leadsto v$ de longitud 1.

Inducción: Supongamos que los arcos que están incluidos en camino hasta el momento forman un camino bien construido en G y observemos qué sucede en la siguiente iteración del método. Tenemos dos casos:

i. Que sigamos extendiendo el camino. Por la disciplina de la frontera de ExploracionEuleriana, el vértice que es el centro de acción en esta iteración es precisamente u, el último vértice incluido en el camino. Existen dos posibilidades para u: que tenga todavía arcos incidentes desde él sin recorrer o que ya no los tenga. Si todavía tiene al menos un arco sin recorrer que salga de u, digamos $u \to v$, se sale por él y se agrega éste a camino seguido de v, y seguimos teniendo un camino dado por $s \leadsto u \to v$.

ii. Que ya no podamos salir del vértice que es centro de acción. Si ya no se puede salir de u, como G es euleriana quiere decir que ya se ha regresado a u tantas veces como se ha salido, por lo que hay un circuito que empieza y termina en u. Pintaremos a u de NEGRO, y retrocederemos por camino hasta encontrar algún vértice GRIS, digamos x. Por la hipótesis de inducción, hay un camino bien formado $s \leadsto x$, y por un razonamiento idéntico al que dimos en el párrafo anterior, simplemente extiende un camino bien formado o procede a retroceder.

∴ en camino siempre tenemos un camino bien formado.

Ahora deberemos establecer que ese camino empieza y termina en s. Que empieza en s es claro, puesto que, por construcción, el primer arco que se incluye es uno cuyo origen es s. Como G es euleriana, $ingrado(s) = exgrado(s)$, deberemos regresar a s una vez por cada arco que salga de s. Por ello, s deberá ser alcanzado y aparecer como destino de un arco en camino, y volverá a ser centro de acción, tantas veces como se le alcance. Cuando esto suceda por última vez, se le pintará de NEGRO y se reportará este último arco, procediendo a retroceder por el camino. Que s debe ser el destino del primer arco por el que se retroceda y se le pinte de NEGRO es claro, porque cualquier otro vértice en el camino, primero se llega a él y después se sale. Entonces, el primer arco incluido en el circuito definitivo será uno que tiene como destino a s.

Por otro lado, el primer arco incluido en camino fue uno que tiene como origen a s. De esto, el último arco que va a ser transferido al circuito es el primero que se incluyó en camino, que tiene como origen a s (camino se vaciará cuando se retroceda por todos los arcos en el camino y ya no se encuentre ningún vértice GRIS en el mismo). De esto, el circuito que se construyó reportará como primer arco (el que quede en el tope de la pila correspondiente) a un arco con origen en s y como último arco del circuito a uno con destino en s.

∴ Lo que se construye en ExploracionEuleriana es un circuito. ❑

Podemos ahora proceder a enunciar y demostrar el teorema de correctez de ExploracionEuleriana de la siguiente manera:

Teorema 6.3 (Correctez de ExploracionEuleriana) *Sea $G = (V, E)$ una digráfica euleriana. Entonces,* ExploracionEuleriana *construye un circuito euleriano en la digráfica.*

Demostración:

Por el Lema 6.2 sabemos que construye un circuito que empieza y termina en s, y por el Lema 3.12(B) sabemos que este circuito incluye a todos los arcos de E exactamente una vez. ❑

Teorema 6.4 (Complejidad de ExploracionEuleriana) *La complejidad de* ExploracionEuleriana *es* $\Theta(|E| + |V|)$.

Demostración:

Veamos cómo se conforma la complejidad del algoritmo:

i. Como cada arco aparece en el circuito exactamente una vez, la construcción del camino deberá llevar $\Theta(|E|)$, ya que metemos al arco al circuito únicamente en el caso de que no haya sido recorrido previamente. El control de que esto suceda así reside en el método proceArco, controlado desde ExploracionBasica, que garantiza que se lleva a cabo exactamente una vez.

ii. Donde pudiéramos tener problemas es al retroceder por el circuito buscando un nuevo vértice de donde volver a salir. Dado que esto aparece en una iteración, cada vez que se retrocede se puede recorrer más de un arco. Pero como cada arco aparece en el circuito una sola vez, será metido a camino también una única vez, por lo que a todo lo largo de la ejecución del algoritmo no se le podrá sacar de camino más de una vez. En resumen, a lo largo de la ejecución ExploracionEuleriana el número de arcos por los que se puede retroceder no puede ser más que el número de arcos que se incluyeron en camino, y como este número es $|E|$, a lo largo de *toda* la ejecución, la contribución de retroceder por camino será a lo más de $|E|$, no alterándose la complejidad que se hereda de ExploracionBasica.

iii. La contribución de $|V|$ está dado por el algoritmo original y cuenta el número de veces que se mete y saca a un vértice de la frontera. Esta aportación sigue siendo válida en ExploracionEuleriana, aunque la conducción del algoritmo esté dada por la construcción del circuito, como se demuestra en el inciso i de este mismo teorema. ❑

Vale la pena notar que para la demostración para la clase de complejidad a la que pertenece este algoritmo utilizamos complejidad amortizada. No es ésta la primera vez que utilizamos este tipo de cálculo de complejidad.

6.4. ExploracionEuleriana en gráficas no dirigidas

A simple vista parece trivial la extensión de ExploracionEuleriana a gráficas no dirigidas, ya que la restricción de que cada arista sea utilizada en el circuito euleriano únicamente una

vez está vigilada directamente en la clase de los arcos, que marca como recorrida una arista y no la vuelve a elegir. Sin embargo, el retroceso por los vértices buscando un nuevo origen para un subcircuito depende enteramente de que ExploracionEuleriana trabajó con arcos, por lo que siempre sabe cuál es el destino y cuál el origen de cada arco. Esto se conoce porque al ingresar una arista a camino, inmediatamente se coloca sobre ella al vértice destino de la misma. De esa manera, se puede usar ese vértice para el retroceso, mientras que se usarán las aristas para reportar el camino. De esto, al reportar el camino, deberemos orientar las aristas para que tengan el formato adecuado. No se requiere de ninguna otra extensión o redefinición para manejar gráficas no dirigidas.

6.5. Aplicaciones de circuitos y trayectorias eulerianas

Esta especialización tiene aplicaciones importantes como las sucesiones de de Bruijn, que tienen que ver con la codificación de subcadenas dentro de una cadena, de la manera más económicamente posible, y tal que la cadena contenga exactamente una vez a cada yuna de las subcadenas. En el Apéndice D damos una presentación con más detalle de este tema.

Otro problema interesante en el que se utiliza esta especialización es en la *secuenciación por hibridación* (SBH en inglés), relacionado con la estructura del ADN, y consiste en lograr encontrar, dada una subcadena, la supercadena que la contiene. En este caso se trabaja a partir de una "muestra" del material genético y se busca reconocer a la estructura a la que representa esta muestra.

P. Pevzner, en [Pev89], redujo el problema SBH al de encontrar caminos eulerianos en una digráfica. También en el Apéndice D se presenta la manera en que Pevzner desarrolló este tema.

6.6. Conclusiones

En este capítulo revisamos el problema de circuitos o trayectorias eulerianas, sometiéndolo a la estructura del algoritmo genérico de exploración. Estamos convencidos que este enfoque, sobre todo después de otras especializaciones más directas del algoritmo genérico de exploración, pone de manifiesto el carácter local del proceso de los vértices y las aristas para encontrar el circuito euleriano (la trayectoria euleriana). Al haber heredado la propiedad de que la super clase explora a todas las aristas, únicamente hubo que demostrarse que efectivamente construye el camino correctamente. Nuevamente el cálculo de la complejidad se dio en términos dce los métodos especializados, o de las estructuras de datos implementadas, lo que fue prácticamente directo.

Para todas las aplicaciones que mencionamos se puede usar directamente la especialización dada, simplemente agregando algún tipo de preproceso a la gráfica, y asociando cierta información conforme se va construyendo el camino.

Capítulo 7

Caminos más cortos

En lo que va de este trabajo hemos explorado gráficas y digráficas, encontrando en esas exploraciones características propias de la especialización particular de la exploración. En todas estas especializaciones el objeto a explorar fue una gráfica o digráfica, que si bien tenía como parte de su definición la posibilidad de asignar pesos arbitrarios a las aristas (arcos), las especializaciones eran correctas trabajando con gráficas con peso homogéneo asignado a las aristas (arcos). Este peso se tomó uniformemente como unitario, aunque de no ser unitario pero permanecer uniforme, las especializaciones siguen funcionando de manera correcta, aún BFS que utiliza el hecho de que el peso de las aristas (arcos) es unitario (uniforme) para entregar su resultado.

En este capítulo hacemos una ligera modificación a la definición de las gráficas, permitiendo que los pesos en las aristas o arcos sean números arbitrarios, aunque en esta primera especialización pediremos que sean números positivos. Como lo dice el nombre del capítulo, exploraremos el problema de caminos más cortos en una gráfica, que si bien resolvimos en el caso de pesos uniformes con la especialización BFS, ésta no es una solución correcta para el caso que nos ocupa.

Daremos una especialización para el algoritmo de Dijkstra para caminos más cortos, que maneja de manera un poco distinta a la tradicional la cola de prioridades para los vértices a los que se les va designando una distancia provisional. Empezaremos primero, como lo hemos hecho hasta ahora, revisando el caso de gráficas dirigidas, para pasar posteriormente al caso de gráficas no dirigidas.

7.1. Descripción general del problema

En un problema de *caminos más cortos* el objetivo principal es encontrar un camino tan corto como sea posible entre cualesquiera dos vértices de una digráfica. Para el desarrollo de este tema utilizaremos las definiciones de *distancia*, *función de peso* y *camino más corto* – definiciones C.11 a C.13 en el Apéndice C.

De la definición de camino más corto debemos tener presente la posibilidad de la existencia de más de un camino entre dos vértices, con peso que corresponde a la distancia entre esos dos vértices.

Este problema, por supuesto, tiene muchas áreas de aplicación. El más común es el de caminos más cortos en un mapa de carreteras, donde las ciudades corresponden a los vértices, las aristas a las carreteras y los pesos al número de kilómetros de carretera entre dos ciudades que son adyacentes. Si el interés es en el tiempo que lleva recorrer el camino entre dos ciudades, el peso de las aristas estaría dado, entonces, por el tiempo que lleva recorrer cada segmento (arista). Por último, a lo mejor estamos interesados en el precio (o costo) en cuyo caso cada arista tendría asociado un costo, como por ejemplo, si hay que parar en un hotel o a comer, o si hay que pagar peaje, etc. En resumen, el peso de las aristas puede ser cualquier medida, siempre y cuando se "acumulen" a lo largo del camino, esto es, que sean en efecto, una distancia y cumplan con la siguiente propiedad:

Propiedad 7.1 *Sea* $P = \mathsf{v}_0 \overset{P_1}{\rightsquigarrow} \mathsf{v}_i \overset{P_2}{\rightsquigarrow} \mathsf{v}_j$ *un camino de* v_0 *a* v_j *que pasa por* v_i. *Entonces,* $w(P) = w(P_1) + w(P_2)$.

El problema central consiste en tratar de optimizar este peso.

Variantes: Tenemos algunas variantes de este problema, que mencionamos a continuación:

Camino más corto con origen único: Dada una gráfica $G = (V, E)$ queremos encontrar el camino más corto dado el vértice *origen* s a todos y cada uno de los vértice $\mathsf{v} \in V$, con $\mathsf{v} \neq \mathsf{s}$. Muchas otras variantes se pueden plantear como un caso particular de ésta.

Camino más corto con destino único: Consiste en encontrar el camino más corto al vértice t desde cualquier vértice $\mathsf{v} \in V$, con $\mathsf{v} \neq \mathsf{t}$. Si invertimos la dirección de las aristas de la gráfica, lo podemos plantear como un problema de camino más corto de origen único.

Camino más corto entre una pareja determinada de vértices: Dados los vértices $\mathsf{u}, \mathsf{v} \in V$, encontrar el camino más corto entre ellos. También en este caso, si resolvemos el problema de camino más corto con origen único, tomando a u como origen, este problema queda resuelto. Más aún, no se conoce ningún algoritmo que resuelva este caso particular asintóticamente más rápido que el mejor algoritmo para camino más corto con origen único, en el peor caso.

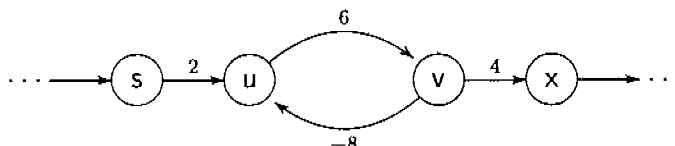
Camino más corto entre cualesquiera dos vértices: Este problema se resuelve también si se ejecuta el de camino más corto con origen único, poniendo el origen en cada uno de los vértices de V.

Como se ve, podemos concentrarnos en el problema de camino más corto con origen único, ya que si tenemos un buen algoritmo para resolver éste, podremos usarlo para resolver cualquiera de las otras variantes.

Pesos negativos

Si todos los pesos de la gráfica son positivos (o cero) no tenemos ningún problema. Pero veamos qué pasa si se presenta una situación como la de la Figura 7.1, y supongamos que queremos obtener la distancia entre s y x.

Figura 7.1: Gráfica con pesos negativos.



La distancia de s a x es 10 para el camino $s \to u \to v \to x$. Sin embargo, para el camino $s \to u \to v \to u \to v \to x$ la distancia es 4. Podemos seguir disminuyendo el peso del camino de manera infinita y no encontrar el mínimo, pues para cualquier camino que nos den, podemos dar una vuelta más al ciclo $u \to v \to u$ y se disminuye en 3 la distancia de s a x. En estos casos definimos a $\delta(s, x) = -\infty$, que para obtener caminos más cortos no es manejable. Por ello, vamos a exigir que no haya ciclos negativos, donde un ciclo negativo es aquél que cuando se recorre el ciclo una vez sumando los pesos de sus aristas, el resultado de esta suma es un valor negativo.

Otro problema que se puede presentar con los algoritmos es que éstos supongan que cada vez que agregan una arista a un camino, la distancia recorrida no debe disminuir. De hecho el algoritmo está suponiendo $w(u \to v) \geq 0$. Si esta desigualdad no se cumple, algunas invariantes del algoritmo no van a poderse presuponer, lo que impedirá que el algoritmo dé, en algunos casos, un resultado correcto.

Otros algoritmos aceptan pesos negativos, pero si hay un ciclo con peso negativo en la gráfica, no podrán asignar valores correctos a las distancias. Estos algoritmos, como el de Ford, que no veremos en este trabajo por no compartir la estrategia genérica dada, son capaces de detectar la presencia de ciclos con peso negativo, en cuyo caso reportarán la imposibilidad de calcular las distancias.

Una de las técnicas más eficientes para encontrar distancias es el *algoritmo de Dijkstra* que veremos a continuación. Recordemos la propiedad de caminos más cortos que está demostrada en el Apéndice E y que nos dice que los subcaminos de un camino más corto son caminos más cortos. Usaremos esta propiedad para establecer la correctez de este algoritmo.

7.2. Especialización para caminos más cortos de Dijkstra

El objetivo principal de la especialización de Dijkstra es la de construir un árbol de caminos más cortos, G_π, muy parecido al que construye BFS. De manera similar a como lo

hicimos en BFS para caminos más cortos con peso uniforme en los arcos, en la especialización de Dijkstra registramos, durante la construcción del árbol y cada vez que evaluamos un arco incidente a un vértice, el camino más corto a ese vértice. Cuando se alcanza al vértice por primera vez, el camino más corto hasta ese momento es el que se acaba de construir, ya que es la primera vez que se llega a ese vértice. Si posteriormente se vuelve a alcanzar ese vértice, habrá la posibilidad de haber llegado a él por algún otro camino que sea, en efecto, de peso menor. Como podemos ver, la estrategia general de exploración es exactamente la misma.

De la estrategia general delineada en exploracionGenerica, los únicos métodos que deben ser especializados son aquéllos que se encargan de revisar que la estimación anterior, si es que la hubo, sea correcta, y esto se hace en los métodos descubriendo y yaDescubierto, donde deberemos cotejar que las distancias que tenemos almacenadas en los vértices sean correctas. En el caso de descubriendo, ExploracionBasica ya se encarga de registrar una estimación inicial, que resulta de sumar la distancia acumulada del padre al peso de la arista, por lo que no tendremos que hacer nada en ese método, y lo heredamos tal cual. En el caso de yaDescubierto, tendremos que actualizar la distancia, en el caso de que el cálculo de la longitud del camino actual sea menor que la longitud registrada. También deberemos modificar la referencia al nuevo padre del nodo. El código necesario para esto se encuentra en el Listado 7.1. Debemos recordar que en la construcción de la gráfica se inicia al atributo d con ∞, ya que no sabemos todavía si existe o no un camino desde s a ese vértice.

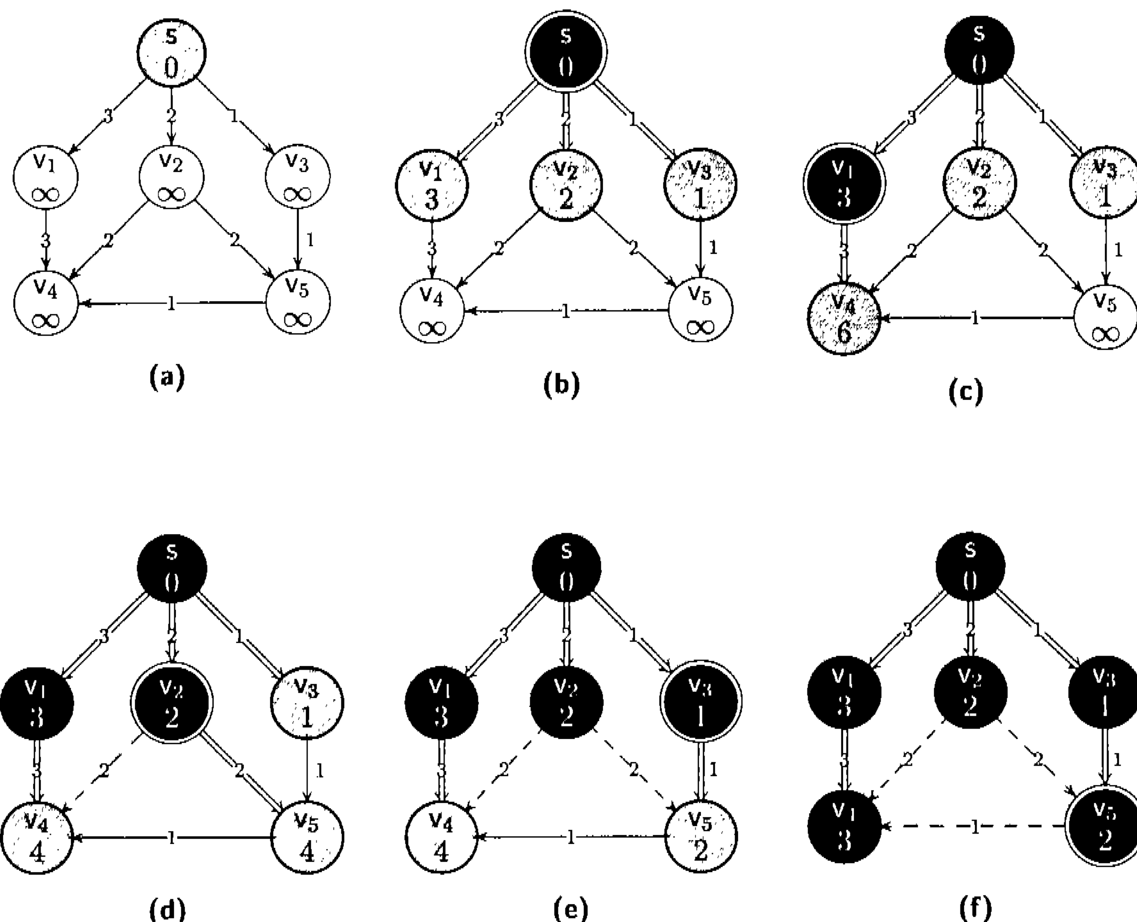
Listado 7.1: Extensión cuando se descubre un vértice

```
1  public class ExploracionDijkstra  extends ExploracionBasica     {
2           /* Construye un objeto para exploración.                 */
3      public ExploracionDijkstra(Digrafica g) {
4           super(g);
5           frontera = new FronteraEnCola();
6      }
7            /* Verifica que el camino registrado hasta ahora sea de  *
8             * menor longitud que el que se acaba de determinar.     */
9      protected void yaDescubierto(Nodos centroAccion, Aristas e,
10                                                     Nodos u) {
11          if (centroAccion.getD() + e.getPeso() < u.getD()) {
12              u.setD(centroAccion.getD() + e.getPeso());
13              u.setPi(centroAccion);
14          }
15     }
```

Si la frontera sigue la disciplina de una cola, como en el caso de BFS, y si corregimos el camino más corto al origen cuando encontramos uno mejor, las únicas modificaciones que habría que hacerle a BFS son las que acabamos de dar.

Veamos un pequeño ejemplo en la Figura 7.2, donde de la manera usual, pintamos de GRIS a los vértices que vamos descubriendo, y de NEGRO a los que damos por explorados, mientras que los arcos que van siendo procesados o recorridos se marcan con línea doble. El orden en que los vértices van siendo centro de acción es exactamente el mismo que en BFS, mediante una cola en la que el primer vértice descubierto es el primero en ser centro de acción, y permanecerá como centro de acción hasta que se dé por explorado.

Figura 7.2: Ejecución de BFS especializado con peso heterogéneo en los arcos

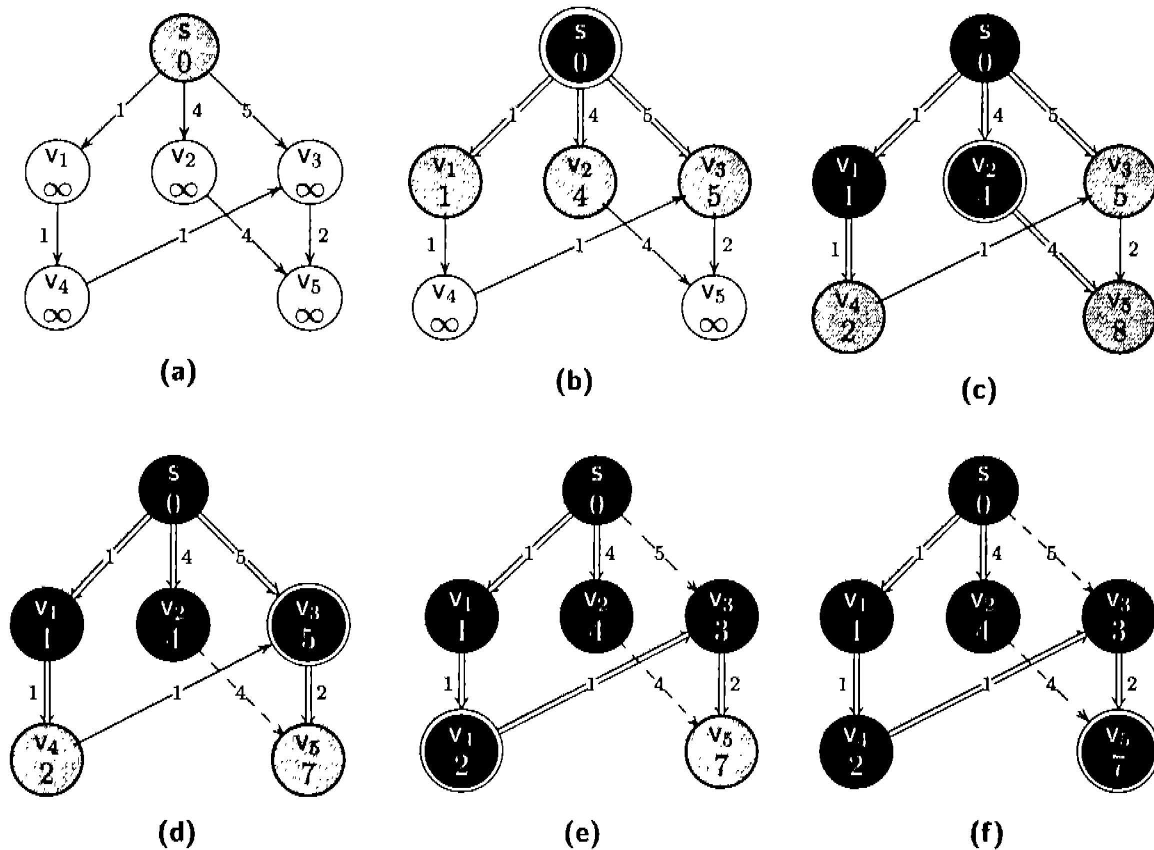


Podemos verificar en este caso que, en efecto, en cada vértice tenemos su distancia a s.

Desafortunadamente, la solución no es tan sencilla como parece. Veamos un ejemplo ligeramente distinto en la Figura 7.3, en donde elegir los centros de acción al estilo de BFS no va a funcionar.

Podemos verificar en esta digráfica que el atributo que registra la distancia al origen no está correctamente calculado para el vértice v_5. Este vértice recibe una primera estimación de 8 en la digráfica 7.3(c), que es corregido en la siguiente digráfica a 7. Sin embargo, al corregirse la distancia del vértice v_3 de 5 a 3 en la digráfica 7.3(e), también se debería haber corregido la distancia de v_5 a s, ya que un camino más corto de s a v_5 está dado por $s \overset{1}{\rightsquigarrow} v_1 \overset{1}{\rightsquigarrow} v_2 \overset{1}{\rightsquigarrow} v_3 \overset{2}{\rightsquigarrow} v_5$, que tiene un peso total de 5, menor que el de 7 que se calculó. El problema fundamental es que no hubo manera de que la disminución en la estimación del vértice v_3 se reflejara en la estimación de v_5, pues v_3 ya había sido totalmente explorado.

Figura 7.3: Ejemplo donde no funciona la estrategiaespecializada BFS.



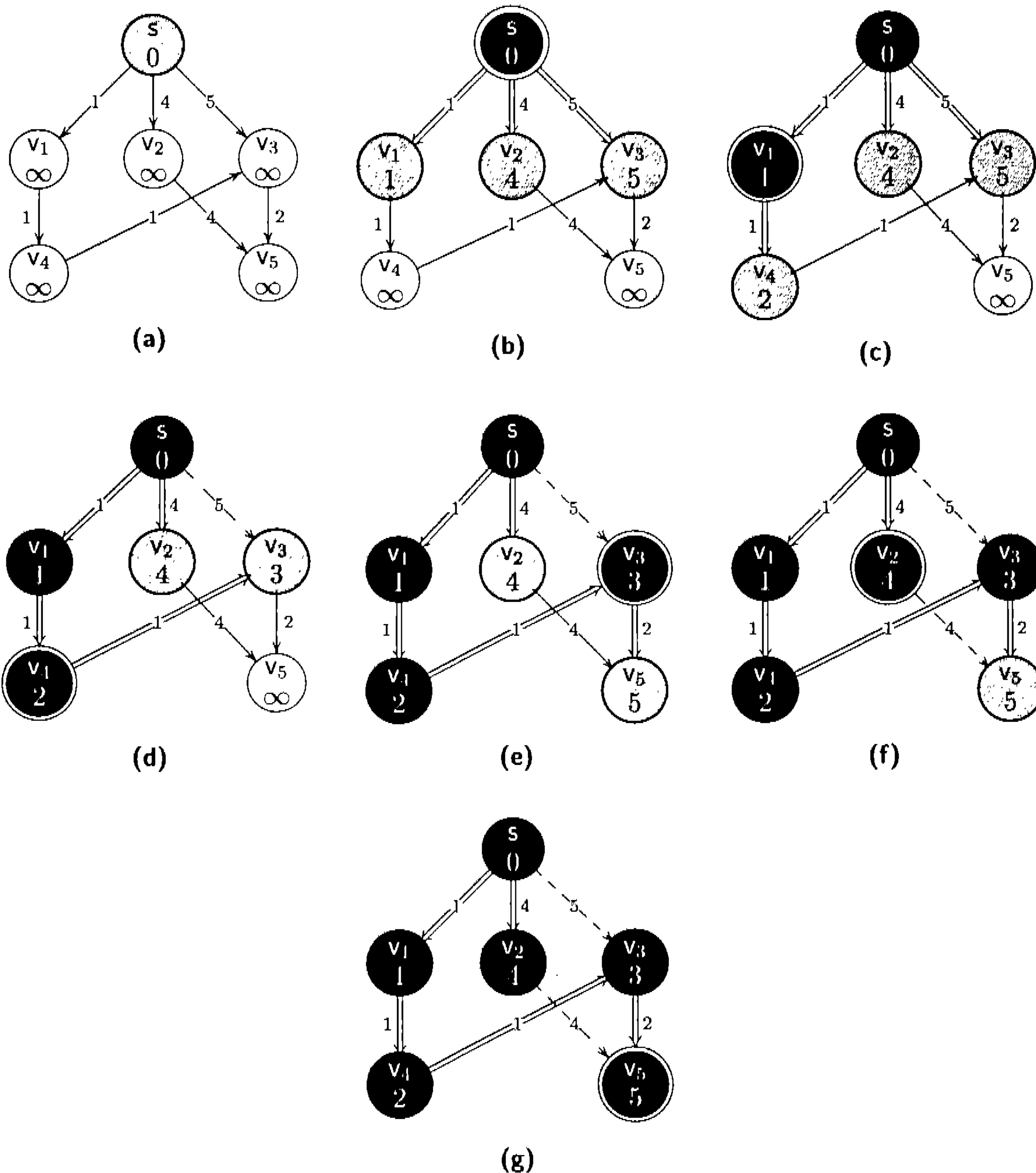
La estrategia de BFS funciona bien porque va descubriendo los vértices por capas, donde el orden en que se convierten en centro de acción tiene que ver con su distancia al origen. Si interpretamos esto en términos de arcos con pesos heterogéneos, lo que tendríamos que hacer es que el siguiente centro de acción fuera el vértice con la menor estimación de entre los que se encuentran en la frontera. En la Figura 7.4 aplicamos esta estrategia a la digráfica de la Figura 7.3.

Como se puede observar, imponiendo a la frontera una disciplina de cola de prioridades se consigue que los vértices se conviertan en centro de acción similar a como lo hacen en BFS, por capas de distancia al origen. En el caso de BFS cada capa contiene a todos los vértices de ese nivel del árbol, ya que la distancia es, precisamente, el nivel del árbol G_π en el que se encuentra el vértice. En cambio, en el algoritmo de Dijkstra, los vértices en una misma capa no coinciden con los niveles del árbol que se construye.

Es claro que la complejidad de esta especialización es mayor que la de las que hemos revisado hasta ahora, ya que no podemos seguir garantizando que la elección de un vértice en la frontera, para convertirse en el siguiente centro de acción, tome un tiempo constante. Si bien las iteraciones externas siguen siendo $O(|V| + |E|)$, para cada vértice que se elige como centro de acción se deberá contabilizar el costo de elegirlo o, en su caso, de mantener la cola

de prioridades ordenada. Dejamos este aspecto de la correctez del algoritmo para después.

Figura 7.4: Ejemplo con la estrategia BFS ajustada.



Establecemos la correctez de esta especialización de ExploracionBasica en el Teorema 7.1.

Teorema 7.1 (Correctez del Algoritmo de Dijkstra) *Al terminar de ejecutarse la especialización de Dijkstra,* $G_\pi = (V_\pi, E_\pi)$ *es un* árbol de caminos más cortos *que cumple con:*

i. $V_\pi \subseteq V$ *son los vértices a los cuales se puede llegar desde* s.

ii. *G_π es un árbol dirigido con raíz en* s.

iii. $\forall$ v $\in V_\pi$ *el único camino de* s *a* v *en el árbol es un camino más corto en G, en el que* v.d $= \delta($s, v$)$.

Demostración:

El inciso *i* se demuestra fácilmente, ya que los vértices incluidos en V_π son, precisamente, aquellos que ingresan a la frontera, y por lo tanto, son a los que se puede llegar desde s. Por otro lado, ningún vértice al que no se pueda llegar desde s será descubierto e incluido en la frontera, por lo que no será considerado para V_π.

El inciso *ii*, aunque parecería también directo de la especialización de Dijkstra, no es así, ya que, a diferencia de ExploracionBasica, el algoritmo de Dijkstra modifica la asignación inicial del atributo π al modificar la estimación de la distancia. Si la asignación inicial que se hace al atributo π en descubriendo no se modifica, sabemos que esa asignación va construyendo bien el árbol G_π. Debemos establecer, entonces, que si durante la ejecución del algoritmo se cambia esta asignación, la propiedad de árbol para G_π no se deja de cumplir. Dividiremos la demostración de este inciso, junto con el inciso *iii*, en varios lemas, por lo que por el momento queda pospuesta. ❑

Trabajaremos primero con la validez del atributo d y varias de sus propiedades, para este caso en el que los pesos de los arcos no son homogéneos, aunque sí positivos.

Lema 7.2 *Para todo arco* e = u$\rightarrow$v *que se procesa, el vértice* u *tiene asignado un valor finito para el atributo* d.

Demostración:

Haremos la demostración por inducción en el número de arcos que se han procesado hasta el momento en que un vértice u es centro de acción.

Base: Cuando se va a procesar el primer arco el centro de acción es s. Antes de procesar el primer arco se ejecutó el método ponComoOrigen, en el que se asignó 0 a s.d, por lo que se cumple que el centro de acción tiene una asignación finita.

Inducción: Supongamos como hipótesis de inducción que todo vértice que es origen de alguno de los primeros k arcos que se han procesado tiene una estimación finita. Veamos qué pasa cuando se procesa el arco $k+1$.

Sea u el vértice origen del arco que se está procesando. Como es el vértice origen, es centro de acción. Para ser centro de acción, tuvo que estar en la frontera y haber sido elegido de allí. u ingresó a la frontera cuando fue vértice destino de un arco procesado antes que el actual.

$\therefore$ por la hipótesis de inducción el vértice origen de ese arco tiene una asignación finita, y la tenía en el momento en que desde él se descubrió a u. Cuando se asignó valor a u.d, se hizo sumándole el peso del arco que se estaba procesando, que es un valor finito, al de v_a.d, por lo que el valor asignado a u también es finito. ❑

Lema 7.3 $\forall$ v $\in V$, v.d *tiene un valor finito* $\Longleftrightarrow$ v.color $\neq$ BLANCO.

Demostración:

$\Longrightarrow$ Supongamos que un vértice tiene un valor finito en el atributo d. Como todos los vértices empiezan con ∞ en ese atributo, únicamente se pudo cambiar este valor por uno finito en uno de tres lugares:

ponComoOrigen: Al elegir a s como origen, en exploracionGenerica, se le pinta de GRIS y se le pone 0 como distancia. Este comportamiento se hereda tal cual, ya que no hemos modificado a ponComoOrigen.

descubriendo: Cuando se descubre al vértice, método que estamos heredando tal cual de ExploracionBasica, se le pinta de GRIS y se le pone una primera estimación de la distancia, que es la distancia de su padre sumada con el peso del arco con el que se le está descubriendo, que es un valor finito. Por las propiedades del algoritmo genérico, el color del vértice nunca regresa a ser BLANCO otra vez, por lo que nuevamente se cumple esta implicación.

yaDescubierto: Nuevamente, la asignación en la línea [12] del Listado 7.1 es de un valor finito, ya que por el Lema 7.2, el valor de v_a.d es finito, como también lo es el peso del arco; por otro lado, el color del vértice que está siendo actualizado es GRIS, y éste no cambia, por lo que nuevamente se cumple esta implicación.

$\Longleftarrow$: Supongamos ahora que el color de un vértice no es BLANCO. Sabemos por las propiedades de ExploracionBasica que el color de un vértice, una vez que dejó de ser BLANCO, ya no vuelve a ser BLANCO. Como todos los vértices empiezan de color BLANCO, el color cambió de BLANCO a GRIS en uno de dos métodos:

ponComoOrigen: Al elegir al origen de la exploración, heredado de ExploracionBasica.

descubriendo: En el método que se hereda de ExploracionBasica.

Como la asignación de valor finito al atributo d se realiza o bien ligado a la asignación de color GRIS al vértice, o bien cuando el vértice ya tiene color distinto de BLANCO, estas dos características van a estar siempre unidas. ❏

Lema 7.4 *Sea* e = x $\rightarrow$ y $\in E$. *En cualquier momento después de recorrer a este arco se tiene que* y.d $\leq$ x.d + e.peso.

Demostración:

Sea e = x$\rightarrow$y el arco que se acaba de recorrer. Tenemos los siguientes casos posibles:

- Ambos vértices tenían ya una estimación finita y la desigualdad ya se cumplía. Por el Lema 7.3, y.color $\neq$ BLANCO por lo que se invocó a yaDescubierto. Y como la condición en la línea [11] del Listado 7.1 no se cumplió, no se modificó ningún valor y la desigualdad se sigue cumpliendo.
- Ambos vértices tenían ya una estimación finita, pero la desigualdad no se cumple. Como en el inciso anterior, por el Lema 7.3, y.color $\neq$ BLANCO, por lo que nuevamente se va a invocar al método yaDescubierto. Como la desigualdad no se cumple, la condición en la línea [11] se va a evaluar a verdadera, lo que va a hacer que se lleve a cabo la asignación en la línea [12], teniendo como resultado y.d = x.d + e.peso, que es lo que queríamos demostrar.

– x tenía una estimación finita, pero y no. y.color=BLANCO, por el Lema 7.3, por lo que al procesar al arco x→y se invocará al método descubriendo, donde, como en el caso de la superclase, se hará la asignación y.d ← x.d + e.peso, lo que hace que se cumpla el lema.

Por supuesto que no puede suceder que ninguno de los vértices tenga un valor finito en d, porque eso querría decir que ninguno ha sido descubierto, y por lo tanto ninguno puede ser centro de acción y origen de un arco que se está procesando.

Tampoco puede suceder que x no tenga asignación finita, porque entonces no podría ser centro de acción, similarmente a lo que argumentamos en el párrafo anterior. ❏

Pasamos ahora a demostrar que las modificaciones que pudieran hacerse al atributo π no tienen ningún efecto sobre la estructura de árbol de G_π.

Lema 7.5 *Las modificaciones al atributo π en yaDescubierto preservan la estructura de árbol de G_π.*

Demostración:

Caracterizamos a un árbol de la siguiente manera:

G_π tiene raíz, esto es, tiene un vértice tal que su ingrado es 0 y desde el que hay un único camino a cualquiera de los otros vértices del árbol.

Si la asignación del atributo π se hace una única vez, en descubriendo, y no se vuelve a modificar, entonces no hay problema y G_π es un árbol con raíz en s, como se demostró en ExploracionBasica.

Al iniciar la ejecución del algoritmo, s es colocado como raíz de G_π. Seguirá siendo raíz siempre y cuando no se modifique su atributo π. Y esto es así. Veamos por qué:

i. El valor de s.d va a ser 0. Como los arcos tienen todos pesos positivos, no podrá haber un arco que regrese a s y que logre dar una estimación menor que cero.

ii. El valor de s.π sólo se puede cambiar en yaDescubierto, ya que s queda pintado de GRIS en ponComoOrigen.

iii. El valor del atributo π se va a modificar siempre y cuando se procese un arco e = v→s, tal que v.d + e.peso < s.peso – líneas [12–13] de yaDescubierto. Pero esta relación no se puede cumplir, ya que como el peso de los arcos es mayor o igual a 0, v.d + e.peso ≥ 0 $\forall$v $\in V$ y e $\in E$.

∴ s.π permanece con el valor NULL durante toda la ejecución del algoritmo.

Sabemos que si se hace una única asignación al atributo π, G_π es un árbol dirigido. Tenemos que demostrar, entonces, que las posibles modificaciones a este atributo preservan esta cualidad para G_π. Lo haremos por inducción en el número de modificaciones.

Base: Si no se hace ninguna modificación después de la primera asignación al atributo π de ninguno de los vértices, como ya demostramos en el ExploracionBasica que G_π es un árbol, no hay nada que demostrar acá.

Inducción: Supongamos como hipótesis de inducción que k modificaciones a atributos π de diversos vértices preservan el que G_π sea un árbol, y veamos qué pasa con la siguiente modificación.

Supongamos que se está procesando el arco $\mathsf{u} \rightarrow \mathsf{v}$ (sabemos que $\mathsf{v} \neq \mathsf{s}$, por la argumentación que dimos en la primera parte de esta demostración). Por la hipótesis de inducción, es importante notar que antes de que se lleve a cabo el cambio, tanto u como v tienen asignaciones en sus atributos π, por lo que están en el árbol G_π, esto es, $\exists$ un camino único de $\mathsf{s} \overset{P_1}{\rightsquigarrow} \mathsf{u}$ y $\mathsf{s} \overset{P_2}{\rightsquigarrow} \mathsf{v}$ en G_π. Para que se modifique el atributo $\mathsf{v}.\pi$ se requiere que la condicional en la línea [11] de yaDescubierto – Listado 7.1 – se evalúe a verdadero. Al modificar a $\mathsf{v}.\pi$, se quita a v del camino P_2 y se le coloca a continuación de u en P_1: $\mathsf{s} \overset{P_1}{\rightsquigarrow} \mathsf{u} \rightarrow \mathsf{v}$. El camino de s a u sigue siendo único, como lo es el de todos los descendientes de u, mientras que el camino de s a v también es único, ya que utiliza al único camino de s a u para llegar a v, y como hay un único apuntador π en v, $ingrado(v)$ sigue siendo 1 en G_π. Si es que en el momento de modificar a $\mathsf{v}.\pi$, v tiene descendientes en G_π (demostraremos más adelante que esto no es posible), todos ellos tienen también un camino único desde s a ellos que pasa por v, y como sigue habiendo un camino único de s a v (nada más que ahora pasa por u), también sigue habiendo un camino único de s a cualquiera de los descendientes de v, de haberlos en ese momento.

$\therefore$ si antes de esta modificación el atributo π conforma un árbol, después de esta modificación π sigue conformando un árbol. ❑

G_π es un árbol de distancias

Habiendo demostrado ya que G_π es un árbol, debemos demostrar ahora que la longitud del único camino entre s y cada uno de $\mathsf{v} \in V$ en G_π es $\delta(\mathsf{s}, \mathsf{v})$ en G.

Para ello demostraremos varias propiedades del atributo v.d, que nos da, precisamente, la longitud del camino $\mathsf{s} \rightsquigarrow \mathsf{v}$ en G_π. Esto es, debemos demostrar que $\forall \mathsf{v} \in V, \mathsf{v.d} = \delta(\mathsf{s}, \mathsf{v})$. Lo haremos primero acotando por arriba a la distancia con el atributo d, y después demostrando que no puede ser menor, en el Lema 7.6.

Lema 7.6 (La estimación es una cota superior de la distancia) *Sea* s *el vértice origen de la exploración. En cualquier momento posterior al inicio de* exploracionGenerica *(ejecución de* ponComoOrigen*),*

$$\mathsf{v.d} \geq \delta(\mathsf{s}, \mathsf{v}) \quad \forall \mathsf{v} \in V$$

y esta invariante se mantiene durante la ejecución del algoritmo. Más aún, una vez que v.d *llega a su cota inferior que es* $\delta(\mathsf{s}, \mathsf{v})$*, ya no vuelve a cambiar.*

Demostración:

Demostremos primero que el atributo d acota por arriba, en todo momento durante la ejecución del algoritmo, a la distancia.

Por el Lema 3.17, el valor asignado al atributo v.d en el método descubriendo cumple con la invariante de acotar por arriba a $\delta(\mathsf{s}, \mathsf{v})$; también como herencia de las propiedades

de ExploracionBasica sabemos que todo vértice en G – suponemos que G tiene una sola componente conexa – va a ser descubierto, podemos partir de esta base. Lo que quedaría por demostrar, entonces, es que ninguna modificación al atributo d de ningún vértice viola esta invariante. Lo haremos por inducción en el número de modificaciones que se hacen al atributo, una vez ejecutado ponComoOrigen. Argumentaremos por separado el caso de s.

Si $\mathsf{v} = \mathsf{s}$, la primera modificación a s.d se hace en ponComoOrigen, donde se le asigna 0. Por un razonamiento similar al del Lema 7.5, como la modificación a $\mathsf{s}.\pi$ se hace simultáneamente a la del atributo s.d, si el primero nunca se va a modificar después de la primera asignación, tampoco el segundo. De esto, s no puede ser el destino de ningún arco que provoque la ejecución de las líneas [12–13] de yaDescubierto.

Veamos ahora el caso para $\mathsf{v} \neq \mathsf{s}$, por inducción en el número de modificaciones que se hace al atributo d.

Base: La primera modificación se hace en descubriendo, ya que todo vértice tiene que ser pintado de GRIS. En ExploracionBasica esta asignación no se veía nunca modificada, y demostramos que cumplía con la invariante de ser una cota superior para $\delta(\mathsf{s},\mathsf{v})$ – Lema 3.17 – por lo que queda demostrado que la invariante se cumple después de la primera modificación.

Inducción: Supongamos ahora que si se han llevado a cabo menos de k modificaciones, esta invariante se sigue cumpliendo, y veamos qué pasa cuando un vértice v es el primero al que se le va a realizar la k-ésima modificación a su atributo d.

Una vez descubierto el vértice, que corresponde a la primera modificación, el único otro lugar donde se puede modificar este valor es en yaDescubierto. Digamos que estamos explorando el arco $\mathsf{e} = \mathsf{u} \to \mathsf{v}$. Para modificar a $\mathsf{v}.\pi$, la condición en la línea [11] del Listado 7.1 debe evaluarse a verdadera, ésto es, se debe cumplir

$$\mathsf{u.d} + \mathsf{e.peso} < \mathsf{v.d}.$$

Por la hipótesis de inducción, $\mathsf{u.d} \geq \delta(\mathsf{s},\mathsf{u})$, y también sabemos que antes de la modificación $\mathsf{v.d} \geq \delta(\mathsf{s},\mathsf{v})$. Por otro lado, sabemos que un subcamino de un camino más corto es un camino más corto, de donde

$$\delta(\mathsf{s},\mathsf{v}) \leq \delta(\mathsf{s},\mathsf{u}) + \mathsf{e.peso}.$$

Por la hipótesis de inducción, tenemos

$$\delta(\mathsf{s},\mathsf{u}) \leq \mathsf{u.d}.$$

Combinando

$$\begin{aligned} \delta(\mathsf{s},\mathsf{v}) &\leq \mathsf{u.d} + \mathsf{e.peso} \\ &= \mathsf{v.d}. \end{aligned}$$

ya que $\mathsf{u.d} + \mathsf{e.peso}$ fue el valor asignado a v.d.

∴ la propiedad de que el atributo d acota por arriba a la distancia es una invariante del algoritmo.

Nos falta por demostrar que una vez que v.d llega a su cota inferior $\delta(\mathsf{s},\mathsf{v})$, ya no vuelve a cambiar. Pero para ello nótese que una vez que v.d alcanza su cota inferior, ya no puede decrecer pues, como acabamos de demostrar, $\mathsf{v.d} \geq \delta(\mathsf{s},\mathsf{v})$. Por otro lado, tampoco puede incrementarse, pues el proceso de un arco no nos permite sustituir un valor de v.d por uno mayor, sino que sólo permite que el valor de v.d decrezca. ❏

Siguiendo con el tema de las características del atributo d y su relación con las distancias, hay que demostrar que si se llega a modificar una estimación en algún vértice, la estimación de los vértices que quedan "colgando" de él en G_π no han sido calculadas, esto es, que las modificaciones a la estimación d únicamente se puede hacer en vértices GRISes. Esto es importante, porque si no fuera así, y de haber sido procesados ya los arcos que inciden desde el vértice, no hay posibilidad de modificar la estimación en los descendientes de ese vértice (ver Figura 7.3). En resumen, lo que queremos demostrar es que la exploración de los vértices en efecto se hace por capas de distancia al origen.

Lema 7.7 *Si un vértice* v *es pintado de* NEGRO, *posterior al momento en que se le pinta de* NEGRO *no ingresará a la frontera ningún vértice que tenga una estimación de distancia menor a la de* v.

Demostración:

Supongamos que un vértice v ya fue pintado de NEGRO y que $\mathsf{v.d} = k$. Cuando fue seleccionado como centro de acción de la frontera, $\forall \mathsf{u} \in \mathsf{frontera},\ \mathsf{v.d} \leq \mathsf{u.d}$.

Por contradicción, supongamos que $\exists$ u tal que al ingresar a u a la frontera lo hace con un valor en $\mathsf{u.d} < \mathsf{v.d}$, y sin pérdida de generalidad, supongamos que éste es el primer vértice que viola esta condición. Veamos el momento en que ingresa u a la frontera, posterior a que se pintó a v de NEGRO. u debió ser descubierto desde x al procesar a un arco $\mathsf{e} = \mathsf{x} \rightarrow \mathsf{u}$, y x debía encontrarse en la frontera en ese momento, por lo que u.d toma el valor de $\mathsf{x.d} + \mathsf{e.peso}$. Pero como v fue elegido de la frontera antes que x, y x se encontraba en frontera en ese momento, sabemos que

$$\mathsf{v.d} \leq \mathsf{x.d}. \tag{7.1}$$

Como u está siendo descubierto, toma el valor de $\mathsf{x.d} + \mathsf{e.peso}$, de donde también sabemos

$$\begin{aligned} \mathsf{u.d} &= \mathsf{x.d} + \mathsf{e.peso} \\ \therefore \mathsf{x.d} &\leq \mathsf{u.d}. \end{aligned} \tag{7.2}$$

Pero combinando (7.1) y (7.2) con la hipótesis de que $\mathsf{u.d} < \mathsf{v.d}$ tenemos:

$$\begin{aligned} \mathsf{u.d} &< \mathsf{v.d} && \text{Hipótesis de contrapositivo} \\ &\leq \mathsf{x.d} && 7.1 \\ &\leq \mathsf{u.d} && 7.2 \\ \therefore\ & \mathsf{u.d} < \mathsf{u.d}, \end{aligned}$$

¡lo que constituye una contradicción! y esta invariante queda demostrada. ❏

Lema 7.8 *Cuando un vértice* $\mathsf{v} \in V$ *es elegido en* frontera, *se cumple que* $\mathsf{v.d} = \delta(\mathsf{s},\mathsf{v})$.

Demostración:

El primer vértice que se elige de frontera es s. La única asignación que se hace a s.d es 0 en ponComoOrigen, por lo que, en efecto, al elegirlo de frontera para explorar se cumple que

$$\mathsf{s.d} = 0 = \delta(\mathsf{s},\mathsf{d})$$

Por contradicción, sea $\mathsf{v} \neq \mathsf{s}$ el primer vértice que al elegirlo de frontera no cumple que $\mathsf{v.d} = \delta(\mathsf{s},\mathsf{v})$. Sea $S = \{\mathsf{u} \in V \mid \mathsf{u.color} = \text{NEGRO}\}$, en el momento en que se elige a v de frontera.

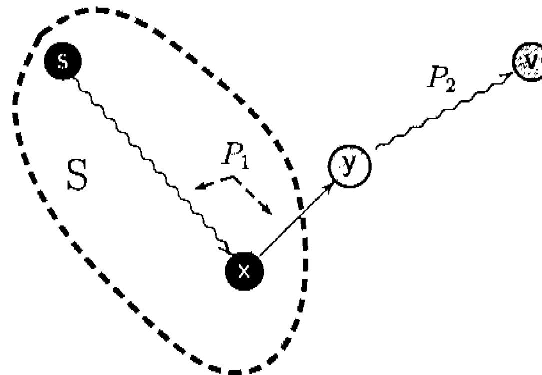
Como v es el primer vértice para el que no se cumple la igualdad, y como cuando se elige a un vértice de frontera continuará siendo el centro de acción hasta que se exploren todos sus arcos incidentes desde él, todos los vértices en S han sido sacados de frontera, por lo que cumplen $\mathsf{u.d} = \delta(\mathsf{s},\mathsf{u})$. Hacemos las siguientes observaciones:

- Por el Lema 7.6, $\mathsf{v.d} \geq \delta(\mathsf{s},\mathsf{v})$, pero como por hipótesis $\mathsf{v.d} \neq \delta(\mathsf{s},\mathsf{v})$, entonces

$$\mathsf{v.d} > \delta(\mathsf{s},\mathsf{v}). \tag{7.3}$$

- $\exists P = \mathsf{s} \rightsquigarrow \mathsf{v}$ tal que $|P| = \delta(\mathsf{s},\mathsf{v})$.
- Sea y el primer vértice que toca P y que no está en S, y sea x el antecesor de y en P, como se muestra en la Figura 7.5. Sea $P_1 = \mathsf{s} \rightsquigarrow \mathsf{x} \to \mathsf{y}$ y $P_2 = \mathsf{y} \rightsquigarrow \mathsf{v}$.

Figura 7.5: Camino más corto de s a v.



- Por hipótesis, y no está en S, por lo que $\mathsf{y.color} \neq \text{NEGRO}$. De hecho, $\mathsf{y.color} = \text{GRIS}$, porque como x ya fue explorado, el arco $\mathsf{e} = \mathsf{x} \to \mathsf{y}$ ya fue procesado, y esto implica que y ya fue descubierto y se encuentra en frontera.
- Como v está siendo elegido de frontera, sabemos que

$$\mathsf{y.d} \geq \mathsf{v.d}. \tag{7.4}$$

- Por otro lado, como P_1 es subcamino de un camino más corto, lo mismo que el camino $\mathsf{s} \rightsquigarrow \mathsf{x}$, tenemos

$$\delta(\mathsf{s},\mathsf{y}) = \delta(\mathsf{s},\mathsf{x}) + \mathsf{e.peso}. \tag{7.5}$$

- Como x$\in S$,

$$\mathsf{x.d} = \delta(\mathsf{s},\mathsf{x}). \tag{7.6}$$

- De (7.5) y (7.6) tenemos

$$\delta(\mathsf{s},\mathsf{y}) = \mathsf{x.d} + \mathsf{e.peso}. \tag{7.7}$$

- Como el arco $\mathsf{e} = \mathsf{x} \dashrightarrow \mathsf{y}$ ya fue procesado y $\mathsf{y.d} \geq \delta(\mathsf{s},\mathsf{y})$ por el Lema 7.6, al procesar a e y.d queda valiendo x.d + e.peso, que por (7.7) es precisamente $\delta(\mathsf{s},\mathsf{y})$.
- Como P_1 es subcamino de P,

$$\begin{aligned} \mathsf{y.d} &= \delta(\mathsf{s},\mathsf{y}) && \text{inciso anterior} \\ &\leq \delta(\mathsf{s},\mathsf{v}) && \text{por ser subcamino} \\ &< \mathsf{v.d} && \text{por hipótesis.} \end{aligned} \tag{7.8}$$

- De (7.4) y (7.8) tenemos

$$\mathsf{y.d} < \mathsf{v.d} \leq \mathsf{y.d}, \tag{7.9}$$

¡una contradicción!

∴ No se cumple que $\mathsf{v.d} > \delta(\mathsf{s},\mathsf{v})$ y se debe cumplir, por el Lema 7.6, $\mathsf{v.d} = \delta(\mathsf{s},\mathsf{v})$, como queríamos demostrar. ❏

Demostremos ahora que el atributo d tiene un comportamiento monótono no decreciente.

Lema 7.9 $\forall \mathsf{v} \in V$, *una vez que* v.d *toma el valor* $\delta(\mathsf{s},\mathsf{v})$, *este valor ya no vuelve a cambiar.*

Demostración:

Por el Lema 7.6, $\mathsf{v.d} \geq \delta(\mathsf{s},\mathsf{v})$, por lo que no podrá tomar nunca un valor menor a la distancia.

Tampoco va a incrementarse. Esto es porque para que se modifique el valor del atributo d, éste tiene que sustituirse por uno menor, ya que de otra manera no entra a ejecutar la línea [12] de yaDescubierto.

∴ Una vez que el atributo d alcanza el valor mínimo, ya no vuelve a cambiar. ❏

Lema 7.10 *Una vez que un vértice es pintado de* NEGRO, *ninguno de sus atributos será modificado.*

Demostración:

El atributo d: Por el Lema 7.8, cuando un vértice es elegido de frontera se cumple que $\mathsf{v.d} = \delta(\mathsf{s},\mathsf{v})$. Por el Lema 7.10, una vez que v.d toma el valor de la distancia ya no vuelve a cambiar.

El atributo color: Por las propiedades heredadas de ExploracionBasica, una vez que se pinta a un vértice de NEGRO, su color ya no volverá a cambiar.

El atributo π: En la especialización de Dijkstra, el atributo π cambia si y cuando cambia el atributo d, y como ya demostramos que éste último ya no va a cambiar, tampoco lo hará el atributo π.

∴ Una vez que un vértice es pintado de NEGRO, ninguno de sus atributos será modificado. ❑

Estamos ya listos para demostrar que G_π es un árbol de caminos más cortos. Debemos demostrar que G_π cumple con las tres propiedades del Teorema 7.1. La propiedad *i* está ya demostrada, ya que, heredado de las propiedades del algoritmo genérico, van a estar incluidos en V_π todos los vértices alcanzables desde s. Por otro lado, también ya demostramos que es un árbol dirigido con raíz con s – Lema 7.5 – por lo que ahora sólo nos falta demostrar la propiedad *iii*, lo que hacemos en el Lema 7.11.

Lema 7.11 *Al terminar el algoritmo, G_π es un árbol dirigido de caminos más cortos con raíz en* s.

Demostración:

Como G_π es un árbol dirigido con raíz en s, $\forall \mathsf{v} \in V_\pi$ $\exists$ un único camino simple dirigido de s a v. Sea $P = \mathsf{v}_0 \rightarrow \mathsf{v}_1 \rightarrow \ldots \rightarrow \mathsf{v}_k$, con $\mathsf{v}_0 = s$ y $\mathsf{v} = \mathsf{v}_k$ ese camino simple de s a v en G_π. Para $i = 1, 2, \ldots, k$, se cumple

$$\mathsf{v_i.d} \geq \mathsf{v_{i-1}.d} + (\mathsf{v}_{i-1} \rightarrow \mathsf{v}_i).\mathsf{peso} \qquad \text{Lema 7.4.}$$
$$\mathsf{v_i.d} = \delta(\mathsf{s}, \mathsf{v}_i) \qquad \text{Lema 7.8.}$$

De estas dos expresiones, podemos concluir

$$(\mathsf{v}_{i-1} \rightarrow \mathsf{v}_i).\mathsf{peso} \leq \delta(\mathsf{s}, \mathsf{v}_i) - \delta(\mathsf{s}, \mathsf{v}_{i-1})$$

Como el peso del camino está dado por el peso de los arcos, y tomando esta última desigualdad,

$$\begin{aligned} w(P) &= \sum_{i=1}^{k} (\mathsf{v}_{i-1} \rightarrow \mathsf{v}_i).\mathsf{peso} \\ &\leq \sum_{i=1}^{k} (\delta(\mathsf{s}, \mathsf{v}_i) - \delta(\mathsf{s}, \mathsf{v}_{i-1})) \\ &= \delta(\mathsf{s}, \mathsf{v}_1) - \delta(\mathsf{s}, \mathsf{v}_0) + \delta(\mathsf{s}, \mathsf{v}_2) - \delta(\mathsf{s}, \mathsf{v}_1) + \ldots + \\ &\quad + \delta(\mathsf{s}, \mathsf{v}_k) - \delta(\mathsf{s}, \mathsf{v}_{k-1}). \end{aligned}$$

Se puede observar que en esta expresión cada término aparece una vez sumando y una restando, excepto $\delta(\mathsf{s}, \mathsf{v}_0)$ que sólo aparece restando, y $\delta(\mathsf{s}, \mathsf{v}_k)$ que sólo aparece sumando, por lo que tenemos:

$$w(P) = \delta(\mathsf{s}, \mathsf{v}_k) - \delta(\mathsf{s}, \mathsf{v}_0).$$

Pero como $\mathsf{v}_0 = s$, $\mathsf{v}_k = v$ y $\delta(\mathsf{s}, \mathsf{s}) = 0$, tenemos:

$$w(P) = \delta(\mathsf{s}, \mathsf{v}).$$

De esto, el camino de s a v en G_π es tan pequeño como un camino más corto en G, con lo que terminamos la demostración. ❑

Terminamos esta sección con el enunciado del teorema de correctez para la especialización de Dijkstra.

Teorema 7.12 *Cuando se invoca a explorationGenerica desde un ejemplar de Exploracion-Dijkstra, suponiendo a la frontera como una cola de prioridades donde la llave es la distancia a* s*, al terminar la ejecución de exploracionGenerica se cumple con todas las propiedades dadas para ExploracionBasica, y además, el árbol G_π construido es un árbol de caminos más cortos.*

Demostración:

Fue hecha a lo largo de esta sección. ❏

7.3. Complejidad de ExploracionDijkstra

Como se puede ver, la complejidad del algoritmo dependerá de cómo se implemente la cola de prioridades, ya que esta implementación influye de tres maneras:

i. El costo de elegir al elemento con la menor estimación de la frontera. Este costo realmente es elevado, pues cada proceso de un arco puede modificar la estimación en el vértice destino, por lo que el ordenar a los vértices en la frontera requeriría estar ordenándolos cada vez que hay un cambio o que entra uno nuevo.

ii. Cada vez que se procesa un arco, como acabamos de mencionar, implica posiblemente un reordenamiento de la frontera, por lo que se debe tener cuidado con el costo de estas operaciones.

iii. Hay implementaciones de colas de prioridades donde resulta muy costoso el eliminar al mínimo de la estructura. Si elegimos alguna de estas implementaciones, el costo de eliminar a cada vértice que es seleccionado como el mínimo pudiera representar un incremento considerable en la complejidad.

Como sabemos – este material se presenta en el Apéndice F – tenemos varias posibles implementaciones para colas de prioridades, que permiten un costo menor para las operaciones que nos interesan, que las que nos ocasiona el intentar mantener a la frontera ordenada. Analicemos cada una de estas opciones.

7.3.1. Implementación de frontera con arreglo o lista ligada

Si mantenemos a la cola frontera simplemente como un **vector o lista ligada**:

Cada vez que se procesa una arista hay que volver a encontrar el mínimo, si es que se redujo la estimación.	$O(\|V\|)$
Número de veces que se procesa un arco.	$O(\|E\|)$

Total para la estimación		$O(\vert E\vert \cdot \vert V\vert)$
Eliminar al mínimo del heap toma:	$O(1)$	
Número de veces que se elimina al mínimo:	$O(\vert V\vert)$	
Total para vaciar a Q		$O(\vert V\vert)$
TOTAL para una lista ligada		$O((\vert E\vert \cdot \vert V\vert) + \vert V\vert)$

Pero como en toda gráfica conexa $|E| \geq |V|$, excepto los árboles donde $|E| = |V| - 1$, el término producto domina, por lo que podemos eliminar el sumando $|V|$ y entonces la complejidad de esta especialización utilizando para la cola de prioridades un arreglo o lista ligada es

$$O(|E| \cdot |V|).$$

7.3.2. Implementación de frontera con heaps binarios*

El cálculo total de la complejidad del algoritmo de Dijkstra cuando utilizamos un heap binario para almacenar a los vértices en frontera es como sigue:

Reencontrar el mínimo cuando se procesa una arco y se reduce la estimación del vértice destino	$O(\log \vert V\vert)$	
Número de veces que se procesa un arco	$O(\vert E\vert)$	
Total para el proceso de arcos		$O(\vert E\vert \cdot \log \vert V\vert)$
Eliminar al mínimo del heap toma	$O(\log \vert V\vert)$	
Número de veces que se elimina al mínimo:	$O(\vert V\vert \log \vert V\vert$	
Total para vaciar a frontera		$O(\vert V\vert \cdot \log \vert V\vert)$
TOTAL para Q un heap binario:		$O((\vert E\vert + \vert V\vert) \cdot \log \vert V\vert)$

Nuevamente, haciendo la consideración de la implementación anterior, podemos eliminar el término $|V| \cdot |V|$ y dejar como complejidad de esta implementación

$$O(|E| \cdot \log |V|).$$

* En el Apéndice F se encuentra la definición y manipulación de los heaps binarios.

7.3.3. Implementación de frontera con heaps de Fibonacci**

Como sabemos, los heaps de Fibonacci ofrecen buenas medidas de complejidad amortizada, y son ideales para algoritmos como el de Dijkstra para caminos más cortos, donde el número de veces que se va a modificar la estimación de un vértice puede llegar a ser muy alta.

Cada vez que se procesa un arco y se reduce la estimación del vértice destino.	$O(1)$	
Número de veces que se procesa un arco.	$O(\|E\|)$	
Total para la estimación		$\boldsymbol{O(\|E\|)}$
Eliminar al mínimo del heap toma	$O(\log\|V\|)$	
Número de veces que se elimina al mínimo:	$O(\|V\|)$	
Total para vaciar a Q		$\boldsymbol{O(\|V\|\cdot\log\|V\|)}$
TOTAL para frontera un heap de Fibonacci.		$\boldsymbol{O(\|E\|+\|V\|\cdot\log\|V\|)}$

Dado que $|E|$ pudiera ser mucho más grande que $|V|$, no eliminamos por lo pronto el término en $|E|$, quedando la complejidad en este caso

$$\boldsymbol{O(|V|\cdot\log|V|+|E|)}.$$

Sin embargo, no cantemos victoria todavía, ya que la eficiencia que ganemos dependerá del número de arcos que tenga la gráfica, en relación al número de vértices. Si implementamos a la lista frontera con una lista tenemos que la complejidad del algoritmo es de $O(|V|^2)$, mientras que en un heap binario es de $O(|E|\cdot\log|V|)$. Es obvio que la implementación en un heap binario resulta mucho más complicada, y el orden de complejidad esconde algunas constantes, como la que corresponde a la inserción de los vértices en el heap. Supongamos que tenemos una gráfica completa en la que $|E| = O(|V|^2)$. En este caso, el uso de heaps binarios nos va a dejar con una complejidad de

$$O(|E|\cdot\log|V|) \;=\; O(|V|^2\cdot\log|V|) \;>\; O(|V|^2)$$

resultando la misma que si hubiéramos utilizado una lista ligada, cuya implementación, como ya discutimos, es mucho más sencilla. En general, si

$$O(|E|) \;\geq\; O\left(\frac{|V|^2}{\log|V|}\right)$$

vamos a tener que la complejidad del algoritmo está dada por

$$O(|E|\cdot\log|V|) \;\geq\; O\left(\frac{|V|^2}{\log|V|}\cdot\log|V|\right) \;=\; O(|V|^2),$$

** En el Apéndice F se encuentra la definición y manipulación de los heaps de Fibonacci.

que si tomamos en cuenta las constantes y la mayor complicación para la programación de heaps binarios o heaps de Fibonacci, resulta poco ventajoso utilizar estas estructuras más complejas.

El análisis que hicimos en los dos primeros casos corresponde a un análisis de peor caso, mientras que en el caso de heaps de Fibonacci logramos bajar aún más el orden de complejidad del algoritmo, realizando un análisis de *costos amortizados*, que, como sabemos, está justificado en este caso.

7.4. Conclusiones

El algoritmo de caminos más cortos de Dijkstra nuevamente presenta un proceso local: las distancias se calculan a partir de un centro de acción que es elegido de entre el conjunto de vértices que ya han sido alcanzados y que siguen activos. En este sentido cae de manera natural bajo la estrategia genérica de exploración que estamos manejando.

En el caso de esta especialización, además, queda claro cuál es la porción del algoritmo donde se puede mejorar la clase de complejidad, y ésta es en el manejo de la frontera – la cola de prioridades. Al trabajar con especializaciones similares, como el algoritmo de Prim para árboles generadores de peso mínimo, los costos se pueden asignar directamente al manejar este nuevo caso, ya que dentro del marco de la estrategia genérica lo único que va a cambiar es el valor que acomoda al nodo en la cola de prioridades.

Capítulo 8

Exploración con revisión exhaustiva de aristas (SFS)

Uno de los problemas más interesantes en redes de computadoras es el relacionado con la "supervivencia" de la comunicación, esto es, dadas ciertas conexiones entre los distintos equipos en la red, cuántos de estos equipos o líneas de comunicación pueden descomponerse sin que se afecte la conectividad general de la red. Este problema tiene que ver con la posible redundancia que haya en las líneas de comunicación o en los equipos; dicho de otra manera, en el número de caminos entre dos equipos, y en el número de caminos que pasan por determinados nodos. Mientras menos caminos hay entre dos equipos, la desaparición de una línea en uno de esos caminos puede ser fatal; de la misma manera, mientras más caminos pasen por un cierto nodo, la descompostura de ese nodo afecta de manera más importante la conectividad en la red.

En este capítulo revisaremos otro algoritmo de exploración, que encuentra propiedades de conectividad en la gráfica, tanto en términos de líneas de comunicación como de nodos estratégicos, aquéllos por los que pasan varios caminos. Dado que cada línea de comunicación y cada equipo (nodo) tienen un valor estratégico distinto, dependiendo no tanto de ellos mismos sino de la relación que guardan con el resto de la red, se debe evaluar de alguna manera este papel, conforme se va determinando. El algoritmo que presentaremos en este capítulo recibe el nombre de SFS (**S***can* **F***irst* **S***earch* [NI92, CKT93]) ya que utiliza la estrategia general de llegar a un vértice, revisar y evaluar una única vez todas las aristas incidentes en ese vértice, y proceder a recorrer alguna de ellas, dependiendo de la evaluación.

Daremos primero la especialización para este algoritmo, para ver después una aplicación que es muy importante, la de certificados delgados de k-conexidad.

8.1. El algoritmo SFS

Como mencionamos en la presentación de este capítulo, el algoritmo SFS inicia la exploración desde un vértice origen, y cada vez que ingresa a un vértice a la frontera procede a evaluar a todas las aristas incidentes desde él que no han sido recorridas, para elegir a una

de ellas a ser la siguiente a recorrer, dependiendo de la evaluación hecha.

Cuando el vértice al que se llega ya no tenga aristas disponibles, se procederá a regresar por el camino construido, hasta que se llegue a algún vértice que sí tenga una arista disponible. El algoritmo continúa de esta manera hasta que todos los vértices hayan sido alcanzados y todas las aristas hayan sido recorridas. El nombre SFS se deriva del hecho de que se tienen que examinar primero (y evaluar) – *scan first* – a las aristas de cada vértice al que se descubre antes de decidir por dónde, si es posible, salir de él, ya sea la primera vez que es centro de acción o en visitas subsecuentes.

El algoritmo SFS es un algoritmo más para exploración de gráficas, que siguiendo la estrategia general del ExploracionBasica toma una decisión respecto a cuál de los vértices es el siguiente centro de acción, y de ese vértice cuál de las aristas va a ser la siguiente a recorrer. El criterio para tomar estas decisiones va a estar determinado por la evaluación que se haya hecho a las aristas una única vez. Esta evaluación obliga a que, cuando un vértice es descubierto, todas aquellas aristas incidentes en el vértice que no hayan sido evaluadas lo sean, aun antes de que sea centro de acción por primera vez. Cuando ese vértice se convierta en centro de acción esta evaluación determinará cuál de ellas usar para salir. La elección de un vértice como centro de acción dependerá también de la evaluación de las aristas que quedaron orientadas desde él.

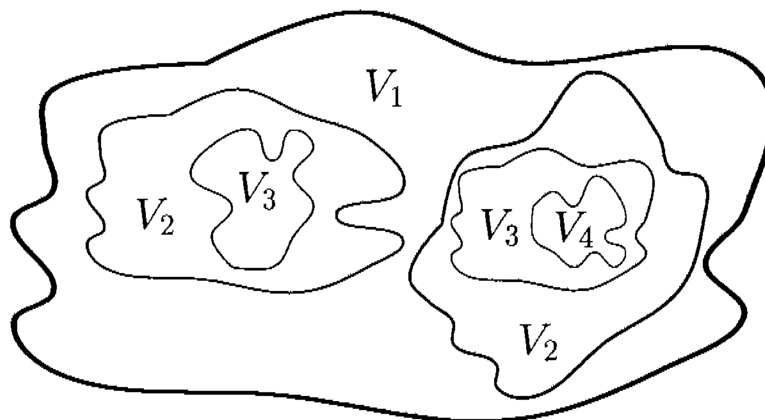
La evaluación que se hace a las aristas tiene que ver con el nivel de comunicación al que representan, esto es, si son la primera, segunda, ...que llega a un vértice dado. Llamamos a este valor el *rango* de la arista. Este valor es siempre estrictamente mayor que cero, una vez que la arista ha sido evaluada, teniendo como valor inicial 0. A cada vértice de la gráfica le asociamos, asimismo, un entero que nos dice, de cierta manera, el número de caminos que pasan por él. El valor inicial de este atributo es 0.

El algoritmo irá construyendo un conjunto de árboles, formados por las aristas de rango mayor que cero, conforme las va evaluando. Inicialmente este conjunto es vacío y el rango de todas las aristas es cero. Al evaluar una arista se le asigna rango mayor que cero. Cada vez que se evalúa una nueva arista, la incluimos en algún árbol $T_{\mathsf{y},k}$, que consiste del árbol con raíz en y y donde todas las aristas incluidas en ese árbol son de rango k. Puede haber más de un árbol de rango k, para cada k. Cada árbol se identifica mediante su raíz y su rango. Si G es conexa, habrá un único árbol de rango 1 con raíz en s, $T_{\mathsf{s},1}$.

Empezamos la ejecución construyendo $T_{\mathsf{s},1}$ y haciendo s.etiqueta $= 1$. Cuando evaluamos a una nueva arista e $=$ v $\rightarrow$ x desde v, es porque llegamos a v recorriendo una arista u $\rightarrow$ v de rango k; es decir, el algoritmo se encuentra construyendo un árbol de rango k con raíz en y. En primera instancia tratamos de extender el árbol $T_{\mathsf{y},k}$ con la arista e, pero si el vértice x ya tiene una arista incidente en él de rango k, e cerraría un ciclo, por lo que se procede a iniciar la construcción de un nuevo árbol $T_{\mathsf{v},m}$ con raíz en v y de nivel m, incluyendo a e en ese árbol. Para determinar m le asociamos a los vértices un atributo etiqueta que registra el mayor rango de entre los árboles que pasan por él. El vértice x al que llega la arista e $=$ v $\rightarrow$ x pertenece al árbol de rango m si es que al recorrerse la arista e el valor del atributo x.etiqueta tiene el valor $m-1$. Se registra en x.etiqueta que también pasa por él el árbol $T_{\mathsf{v},m}$, asignando el valor m a este atributo.

Como se inicia un árbol de rango $k+1$ cuando se detecta que una arista cierra un ciclo en un árbol de rango k, el conjunto de vértices de cada árbol de rango k va a estar contenido en el conjunto de vértices de un árbol de rango $k-1$, como se muestra en la Figura 8.1, donde V_i se refiere a los vértices que están incluidos en los árboles de rango i.

Figura 8.1: Construcción de los árboles de distintos niveles y con distintas raíces



En lo que resta de este capítulo, sin pérdida de generalidad y como lo hemos hecho en capítulos anteriores, supondremos que la gráfica sobre la que estamos trabajando consiste de una única componente conexa.

Supongamos que estamos en una iteración del algoritmo y ya está determinado el siguiente centro de acción y la arista a recorrer de ese vértice. Si el vértice destino de la arista elegida está siendo descubierto – está pintado de BLANCO – en el caso de gráficas dirigidas se evaluará a aquellos arcos que sean incidentes desde el vértice que está siendo descubierto, mientras que en gráficas no dirigidas esta evaluación tendrá como efecto el orientar a las aristas no utilizadas hasta el momento, definiendo al vértice que se está descubriendo como el vértice origen (*desde* el vértice) de estas aristas.

Cada vez que un vértice es centro de acción se deberá elegir para recorrer una arista adecuada, que sea la *mejor* de entre las aristas disponibles orientadas desde él. El criterio de cuál arista es "mejor" pudiera consistir, simplemente, en respetar el orden en que se presentaron en la construcción de la gráfica, o puede involucrar una función de evaluación tan complicada como queramos. Si el valor que se asigna a cada arista/arco es homogéneo entonces regresaremos a un modelo no determinístico para decidir cuál de las siguientes aristas/arcos utilizar; mientras que si se cuenta con una función de evaluación que asigne pesos heterogéneos, la decisión deberá estar influida de manera definitiva por esta evaluación. Nos interesa este segundo caso. Como ya mencionamos, llamaremos rango al atributo que corresponda al valor asignado por la función de evaluación a la arista, y etiqueta al valor que va registrando en cada vértice el rango máximo de las aristas que inciden en o desde él.

Cuando un vértice es descubierto e ingresado en la frontera es el momento de evaluar las aristas incidentes en él. A partir de ese momento y es pintado de GRIS, el vértice es susceptible de ser centro de acción, por lo que deberá tener evaluadas todas sus aristas para que se pueda decidir si ponerlo como centro de acción o no, y si se le pone, por cuál de las aristas salir de

él.

Como en el caso del ExploracionBasica, el vértice es ingresado a la frontera cuando se le descubre. Tendrá asociado un valor que corresponderá al rango más alto de las aristas que tiene orientadas hacia él, que se va actualizando conforme se orienta a las aristas. Este valor podrá también cambiar una vez que el vértice sea descubierto, tomando el valor máximo de los rangos de las aristas que inciden en él, ya sean orientadas desde o hacia él. Mientras no se le descubra, el vértice estará pintado de BLANCO. Al ingresar a la frontera se le pinta de GRIS, y cuando ya no tiene aristas para usar se le pinta de NEGRO.

Esta estrategia, como se puede ver, se puede utilizar tanto en gráficas dirigidas como no dirigidas. En el caso de gráficas dirigidas se evaluarán, como ya mencionamos, únicamente aquellos arcos incidentes *desde* el vértice en cuestión, respetando la dirección asignada a los arcos; en cambio, en el caso de gráficas no dirigidas el proceso de evaluación involucra también el de *orientar* a las aristas que no han sido evaluadas en el momento de descubrir al vértice – pudieron ser evaluadas desde el vértice en el otro extremo de la arista – ya que el valor asignado supone una dirección definida.

Ejemplo de ejecución de SFS

En el caso del vértice origen s, que de hecho no es descubierto en el transcurso de la ejecución del algoritmo, sino que se le coloca en la frontera como parte del inicio de la exploración, la evaluación del vértice – que se realiza evaluando a todas las aristas incidentes en s – se hará al meter a s a la frontera.

A partir de la Figura 8.2 y hasta la Figura 8.12 podemos ver una ejecución de SFS en una gráfica sencilla no dirigida. La función de evaluación que elegiremos para asignar rango a las aristas tiene que ver con el orden en que las aristas son orientadas hacia un determinado vértice: si la arista es la primera que llega al vértice destino le asignamos rango 1; si es la segunda, rango 2; y así sucesivamente. Anotaremos en el vértice, bajo su "nombre", el valor del atributo etiqueta seguido del valor del rango de la arista disponible con más alto rango de entre las orientadas desde él – etiqueta : rango máximo disponible. Un valor de 0 como segundo valor abajo del nombre del vértice indica que no tiene aristas evaluadas **disponibles** orientadas *desde* él, mientras que un valor de 0 en el primer parámetro indica, si el vértice es BLANCO, que sus aristas no han sido evaluadas, y si el vértice no es BLANCO que simplemente no quedó ninguna arista orientada *desde* él.

Para elegir el siguiente centro de acción se usará el criterio de elegir al vértice de la frontera que tenga a la arista con el rango asignado más alto de entre las aristas evaluadas y no recorridas. Cuando el conjunto de vértices que cumplan estos criterios contenga a más de un elemento, la elección del siguiente centro de acción será no determinística de entre los vértices elegibles. El criterio para elegir a la siguiente arista a recorrer, una vez seleccionado el centro de acción, será similar al de elección de centro de acción: aquella arista que tenga el rango más alto y que esté disponible. Como la elección del centro de acción se hace porque el vértice elegido cuenta con una arista de rango máximo en ese momento, de hecho la arista elegida será la que coloca al vértice como centro de acción. Por supuesto que cuando haya más de una arista evaluada con rango máximo, la elección será no determinística.

Anotaremos a las aristas con una pareja a:b, donde a representará el orden en que la arista fue elegida para recorrer y b representará el rango asociado a la misma. Un valor de 0 en la posición de a indica que la arista no ha sido recorrida, sino nada más evaluada. El valor que asignaremos al atributo rango se calculará, como acabamos de decir, contando el número de aristas que han sido orientadas, hasta el momento, hacia el vértice destino. A los vértices, como ya mencionamos, los anotaremos con una pareja $c : d$, donde c representará el valor que va tomando el atributo etiqueta y d representará al rango de la "mejor" arista. En las figuras que siguen representaremos con línea punteada a las aristas que ya estén orientadas y con doble línea a las que ya hayan sido usadas. La línea sencilla es para aquellas aristas que no han sido orientadas, y por lo tanto, no han sido usadas. Nótese que las aristas de línea sencilla no tienen dirección.

En la Figura 8.2 se muestra el estado de la gráfica una vez hecha la inicialización y con s el único vértice en la frontera y pintado de GRIS.

Figura 8.2: Exploración SFS de una gráfica con s como origen. Se ingresa a s a la frontera con sus aristas evaluadas

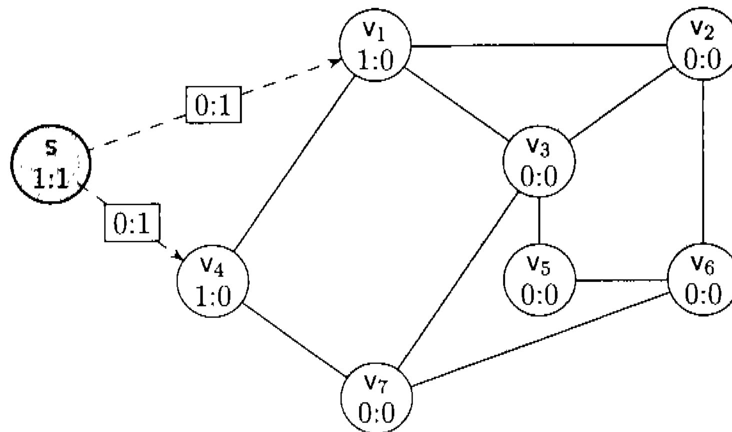
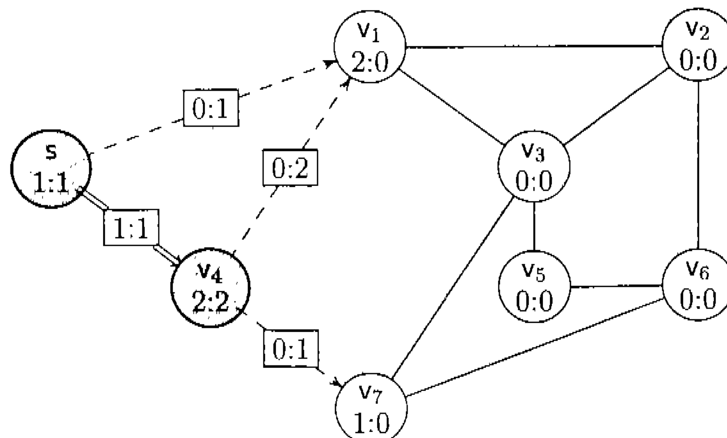


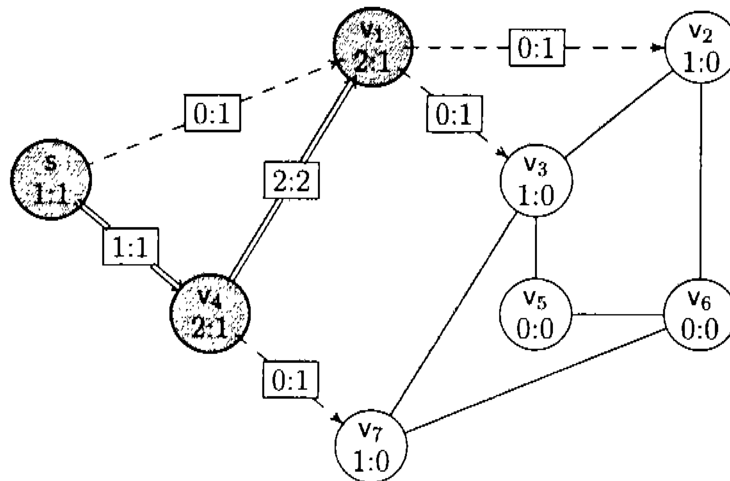
Figura 8.3: Se usa s→v4 y se ingresa a v4 a la frontera con sus aristas evaluadas



s es el único vértice en la frontera, por lo que es elegido como primer centro de acción. En la Figura 8.3 se muestra el resultado de usar no determinísticamente la arista $s \to v_4$, se introduce a v_4 en la frontera, se le pinta de GRIS y se evalúan sus aristas.

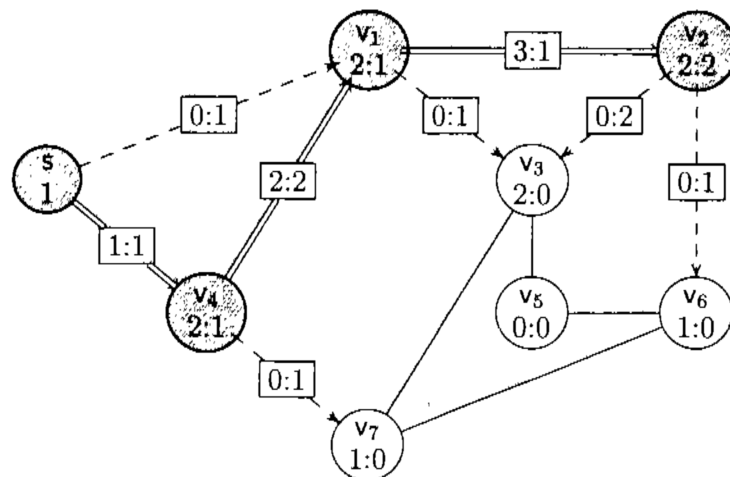
Como el vértice v_4 es el que tiene la arista con el más alto rango de entre los que se encuentran en la frontera, es el que se elige como siguiente centro de acción, y a la arista $v_4 \to v_1$ como la arista a recorrer. El efecto de esto se puede ver en la Figura 8.4.

Figura 8.4: Se usa la arista $v_4 \to v_1$, se mete a v_1 a la frontera y se evalúan sus aristas



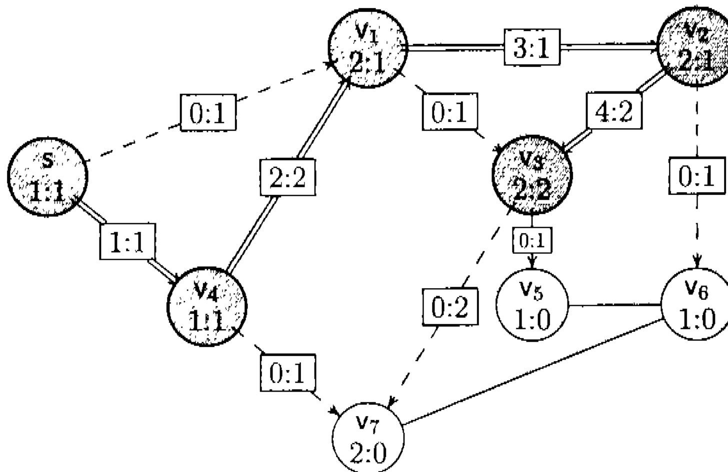
En este punto, todos los vértices en la frontera tienen aristas disponibles de rango 1, por lo que cualquiera de ellos puede ser elegido como siguiente centro de acción. Elijamos a v_1 y como arista a usar $v_1 \to v_2$. El resultado se muestra en la Figura 8.5.

Figura 8.5: Se usa la arista $v_1 \to v_2$ y se evalúa a las aristas sin evaluar de v_2, a quien se introduce a la frontera



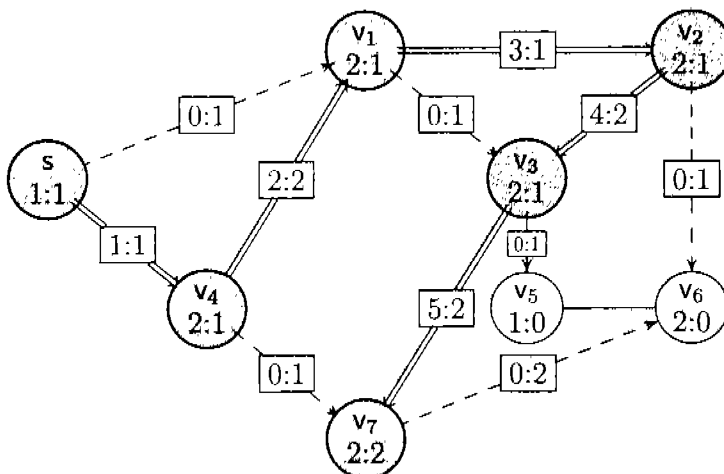
El vértice v_2 es el único en la frontera que tiene arista de mayor rango, 2, por lo que es el siguiente centro de acción, y la arista $v_2 \rightarrow v_3$, que es la que tiene rango 2, la siguiente que se recorre. El resultado se puede ver en la Figura 8.6.

Figura 8.6: Se usa a la arista $v_2 \rightarrow v_3$ y se evalúa a las aristas sin evaluar de v_3 al meterlo a la frontera



Como el vértice en la frontera con aristas de mayor rango, el único, es v_3, escogemos la arista orientada de este rango que es $v_3 \rightarrow v_7$, ingresamos a v_7 a la frontera y lo pintamos de GRIS después de evaluar a sus aristas no evaluadas. El resultado se puede ver en la Figura 8.7.

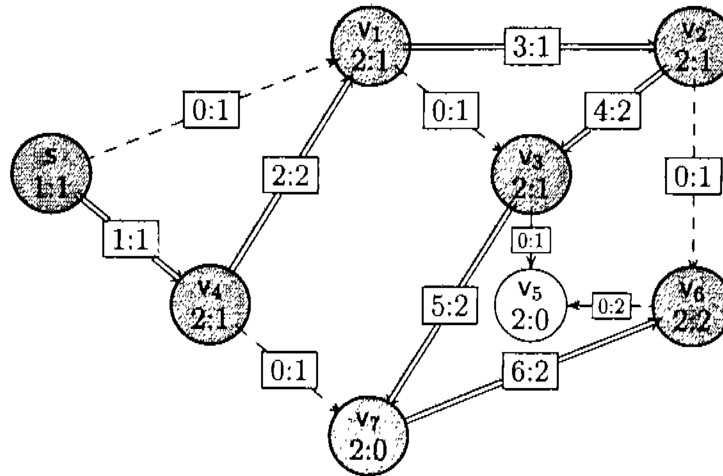
Figura 8.7: Se usa la arista $v_3 \rightarrow v_7$ y se evalúa a las aristas sin evaluar de v_7, a quien se mete a la frontera



Al ingresar a v_7 a la frontera, pintarlo de GRIS y evaluar a las aristas de v_7 que hasta el momento no han sido evaluadas, nuevamente surge una arista con rango 2, que es el mayor rango presente de entre los vértices en la frontera, y corresponde al vértice v_7, por lo que

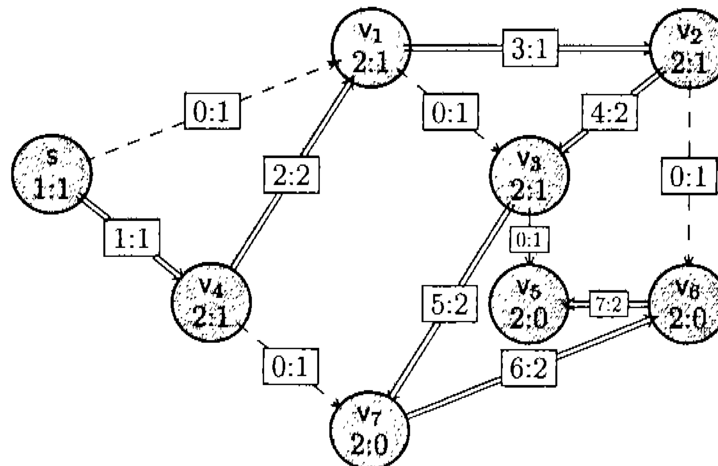
éste es el siguiente centro de acción y se recorre la arista $v_7 \rightarrow v_6$, descubriendo al vértice v_6, pintándolo de GRIS al tiempo que se le mete a la frontera, y orientando a la arista $v_6 \rightarrow v_5$, que es la única que quedaba sin orientar incidente en v_6. En la Figura 8.8 mostramos el estado de la gráfica después de ejecutar los pasos que acabamos de describir.

Figura 8.8: Se usa la arista $v_7 \rightarrow v_6$ y se evalúa a las aristas restantes de v_6, a quien se mete a la frontera



El vértice en la frontera con aristas disponibles de mayor rango es v_6, por lo que se convierte en el siguiente centro de acción y se recorre la arista $v_6 \rightarrow v_5$, descubriendo al vértice v_5. Como éste ya no tiene aristas por orientar, simplemente lo pinta de GRIS, lo mete a la frontera y le pone 0 como rango de su arista disponible de más alto rango, ya que no hay ninguna. La Figura 8.9 nos presenta el estado de la gráfica en este momento.

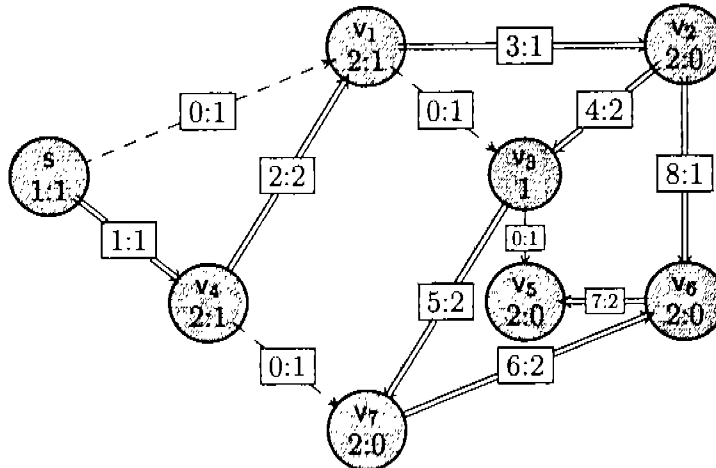
Figura 8.9: Se usa la arista $v_6 \rightarrow v_5$ y se descubre a v_5, que ya no tiene aristas por evaluar



En este momento todos los vértices en la frontera tienen rango menor o igual a 1, por

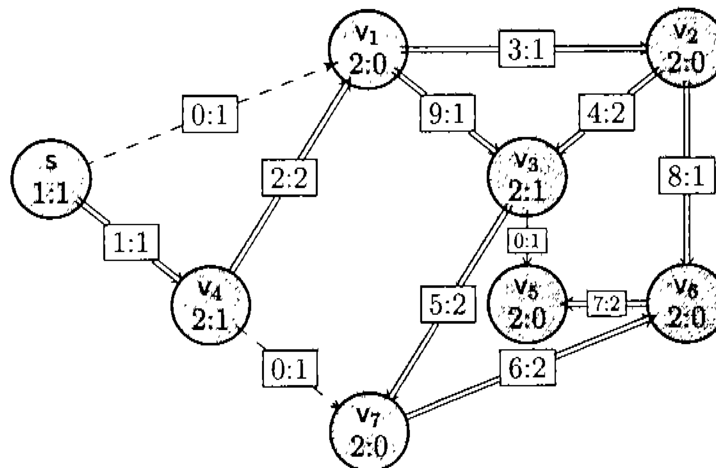
lo que escogemos no determinísticamente a uno de rango 1 – v_2 – y a la arista $v_2 \to v_6$ para recorrer. Como v_6 ya había sido alcanzado, y ya no tiene aristas disponibles, no se hace nada con él. La gráfica queda como se puede ver en la Figura 8.10.

Figura 8.10: Se usa la arista $v_2 \to v_6$



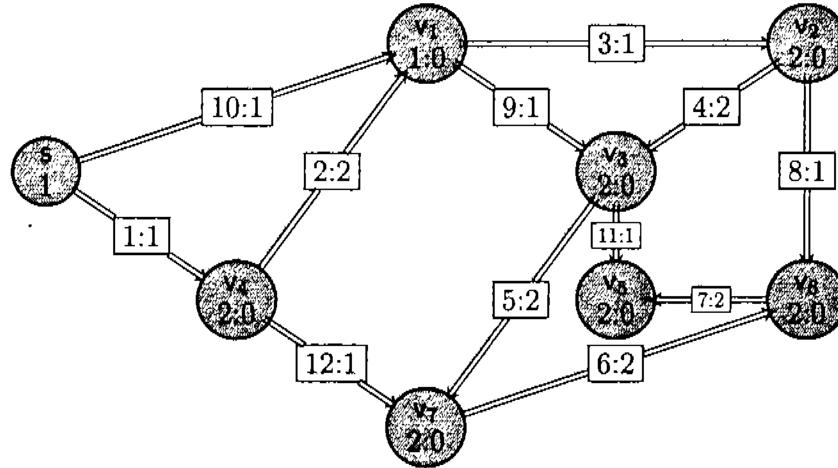
Elegimos nuevamente no determinísticamente a un vértice de la frontera que tenga a una arista con rango 1, que es el mayor rango presente – v_1 – y recorremos la arista $v_1 \to v_3$, que tiene como destino a v_3 que ya fue descubierto. Ver el resultado en la Figura 8.11.

Figura 8.11: Se usa la arista $v_1 \to v_3$



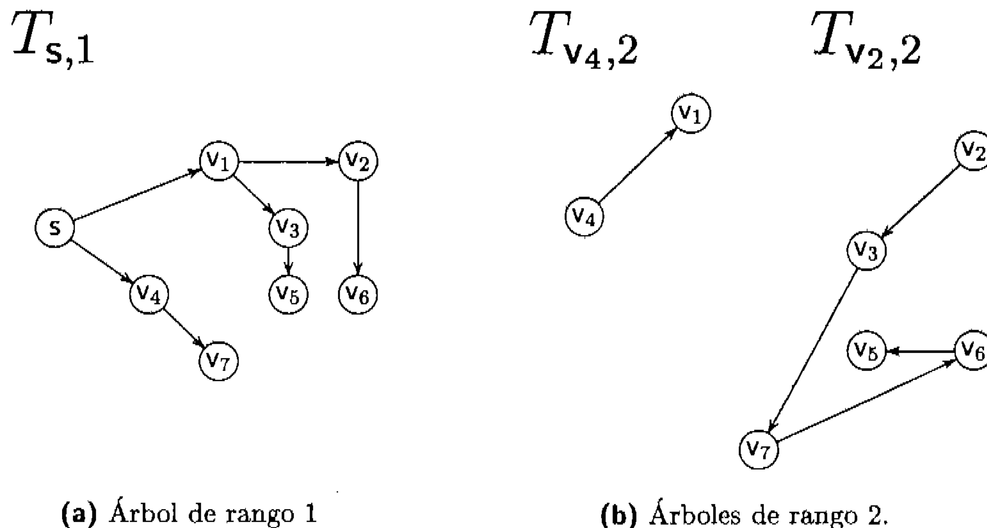
Podemos ver que ya no quedan vértices sin descubrir pintados de BLANCO, y por lo tanto tampoco hay aristas sin evaluar. Esto implica que los rangos disponibles ya no van a cambiar. Quedan en este momento tres vértices en la frontera con rango mayor que 0, s, v_3 y v_4. Podemos pensar que los escogemos en ese orden y recorremos a la arista que cada uno tiene disponible. Después de tres iteraciones, donde en cada una de ellas recorremos a una de estas aristas, la gráfica queda como se muestra en la Figura 8.12.

Figura 8.12: Se recorren las aristas $s \to v_1$, $v_3 \to v_5$ y $v_4 \to v_7$



Al terminar este proceso todos los vértices en la frontera tienen como aristas disponibles con rango máximo 0, lo que quiere decir que ya no tienen aristas disponibles. Como todos tienen el mismo valor, procedemos a escoger uno a uno, no determinísticamente, y al ser cada uno de ellos centro de acción se detecta que ya no tiene aristas disponibles, por lo que se procede a pintar a cada uno de los vértices de NEGRO y a sacarlo de la frontera. Aunque sea reiterativo, se debe notar que hasta ahora no se había detectado que ninguno de ellos ya no tuviera aristas por recorrer, pues como existía algún vértice con rango mayor que cero, ninguno con rango cero – sin aristas por recorrer – había sido elegido como centro de acción, y por lo tanto no se le había detectado. (No mostramos las figuras correspondiente porque no contribuye en nada, ya que es idéntica a la gráfica de la Figura 8.12, excepto que los vértices están pintados de NEGRO en lugar de ser GRISes.)

Figura 8.13: Árboles determinados por SFS



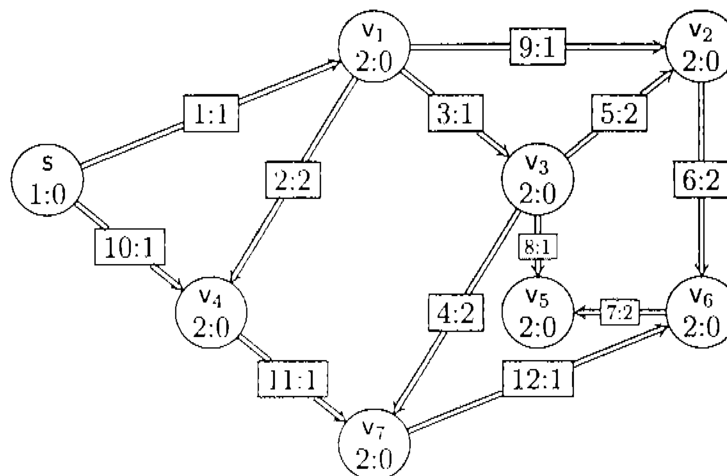
(a) Árbol de rango 1 **(b)** Árboles de rango 2.

Es interesante notar que se construyeron árboles de nivel $k = 1$ (siempre es así si la gráfica

es conexa) y árboles de nivel $k = 2$. Como ya mencionamos, el árbol de nivel $k = 1$ toca a todos los vértices de la gráfica y es, por lo tanto, un árbol generador. Tenemos 2 árboles de nivel $k = 2$ y los vértices de ambos, por supuesto, son subconjuntos de los vértices del árbol de nivel $k = 1$, como se puede observar en la Figura 8.13.

Es claro que la elección de la siguiente arista a usar es no determinística, pero considerando únicamente al subconjunto de aristas ya orientadas que no han sido usadas y que tienen asignado el rango con el valor más alto presente. Por supuesto que esta decisión puede cambiar la asignación final. En las Figuras 8.14 y 8.15 se muestran otras dos posibles estrategias para elegir al centro de acción. En la primera de ellas, conforme vamos eligiendo aristas las colocamos en una pila al estilo DFS. En un momento dado, la siguiente arista a usar queda determinada, siguiendo la estrategia DFS, por alguna que cumpla el criterio de tener en el rango un valor mayor o igual al de la arista en el tope de la pila, que sea el rango disponible más alto y que salga del vértice que es el destino de esa arista. Si ninguna de las aristas cumple con este criterio, se retrocede por el camino, quitando aristas de la pila, hasta que se llegue a una en la que estas condiciones se cumplan. Demostraremos más adelante que esta estrategia cumple con las condiciones que dimos para la elección no determinística.

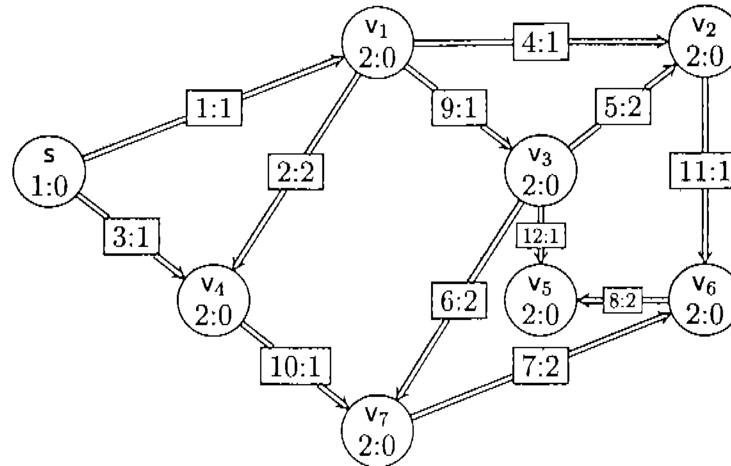
Figura 8.14: Se usan las aristas guiados por una exploración "al estilo" de DFS



En la segunda estrategia, seguimos el criterio de que en igualdad de condiciones, usaremos una arista que sale de un vértice descubierto antes, siguiendo de cierta manera la estrategia de BFS. Hay que notar, sin embargo, que estamos agregando condiciones que limitan la selección de la siguiente arista a usar, por lo que el algoritmo no se comportaría exactamente ni como BFS ni como DFS.

Cada recorrido, dependiendo de la estrategia que se use para elegir al siguiente centro de acción, y desde él el orden en que se elijan las aristas, determina una gráfica orientada distinta. Demostraremos más adelante que estas gráficas tienen en común propiedades de conexidad importantes.

Figura 8.15: Se usan las aristas guiados por una exploración "al estilo" de BFS



8.2. La especialización de ExploracionBasica a SFS

Hasta ahora, al estudiar cada una de las especializaciones de ExploracionBasica hemos procedido a revisar primero el caso de digráficas, para después revisar el de gráficas no dirigidas. Sin embargo, la característica de SFS que lo hace útil para las aplicaciones que vamos a revisar es la de que, al explorar una gráfica no dirigida, es la misma exploración la que orienta a las aristas, y es esta orientación la que presenta propiedades interesantes. Por ello, abordaremos únicamente el caso de gráficas no dirigidas.

Nuevamente usaremos como clase base la del algoritmo genérico (ExploracionBasica). La especialización en el proceso que exige SFS consiste en, al descubrir a un vértice sus artistas se deben "reorganizar" para asignarles el rango, de tal manera que al pedir una arista disponible del centro de acción sean entregadas de mayor a menor rango. Mientras se preserve que al pedir una arista disponible el método correspondiente entregue una y la elimine del conjunto de aristas disponibles, este preproceso no afecta las propiedades del ExploracionBasica. Debe quedar claro que como esta evaluación de las aristas es dinámica y se debe realizar al llegar por primera vez a un vértice, el preproceso no se puede hacer durante la construcción de la gráfica. Se debe tener cuidado, sin embargo, con la complejidad de esta reorganización, ya que si no lo hacemos bien podemos incrementar la complejidad del algoritmo. Por lo pronto supondremos que tenemos tal reorganización y que su complejidad es lineal en la suma de los grados de los vértices, por lo que la reorganización de todas las aristas de cada uno de los vértices tendrá complejidad lineal en el número de aristas. La nueva lista de aristas evaluadas y reorganizadas sustituye a la lista original, de tal manera que la entrega de aristas a recorrer sea transparente. Debe ser claro que en estas nuevas listas cada arista aparece una única vez asociada a alguno de los vértices, ya que estará orientada y no, como en las gráficas no dirigidas, una vez por cada vértice extremo de la arista. Daremos el diseño de este método más adelante, cuando tengamos completo el funcionamiento general del algoritmo.

Una arista queda orientada y evaluada cuando ya fue considerada en la reorganización de las aristas incidentes en un vértice y por lo tanto, en la lista de adyacencia de nada más uno

de los vértices extremos de la arista.

Queremos mantener la representación de la gráfica tal como la hemos manejado hasta ahora, pero tenemos que almacenar información adicional respecto a cada vértice y cada arista. Para ello, en la especialización para SFS tendremos arreglos paralelos a la estructura de vértices y a la estructura de aristas para anotar ahí la información que requerimos (omitiremos el método getPos() para denotar la posición en el arreglo tanto de vértices como aristas):

– ∀ e=v–u $\in E$,

rango[e] $= k$,

si el centro de acción es v y e es la k-ésima arista que llega a u en la ejecución del algoritmo.

– ∀ v $\in V$,

etiqueta[v] = max{rango[e] | e = v–x $\in E$}.

Este atributo refleja de cierta manera el máximo rango de los caminos que pasan por v, ya sea con aristas orientadas hacia v o desde v.

reorganizaAristas(Nodos v),

se encarga de convertir a la lista de incidencia del vértice v en una lista de aristas evaluadas y ordenadas de mayor a menor rango.

En el Listado 8.1 mostramos las declaraciones que requerimos, por lo pronto, en esta especialización.

Listado 8.1: Especialización para la clase SFS

```
1 public class ExploracionSFS extends ExploracionBasica {
2        /* Registra la evaluación de una arista.                    */
3     protected int[] rango;
4        /* Registra el rango más alto entre las aristas             *
5         * disponibles.                                             */
6     protected int[] etiqueta;
7        /* Evalúa a las aristas disponibles incidentes en el        *
8         * vértice, las ordena de acuerdo al rango.                 */
9     protected ListaDeAristas reorganizaAristas(Nodos v);
```

En el constructor de la especialización iniciamos a las estructuras de datos agregadas con los valores iniciales de 0 para ambos atributos. Veamos la implementación en el Listado 8.2.

Listado 8.2: Inicialización del rango de las aristas y la etiqueta de los vértices

```
100        /* Constructor del ejemplar de esta especialización.        *
101         * Inicia en 0 los valores de rango y etiqueta.             */
102     public ExploracionSFS(Grafica g) {
103        super(g);
104        rango = new int[g.getM() + 1];
105        etiqueta = new int[g.getN() + 1];
106        /* Todos los enteros se inician en 0 automáticamente.       */
```

La estrategia SFS nos indica que en el momento de descubrir a un vértice, se deben evaluar

y orientar a todas las aristas disponibles desde ese vértice, y elegir a la de mayor rango. Conforme vamos explorando la gráfica, recordamos el camino que se va formando para, de esa manera, cuando algún vértice ya no tiene aristas disponibles, la exploración regresa por el camino formado hasta que encuentre un vértice con aristas disponibles donde proseguir la exploración. SFS supone que en cada momento está eligiendo la arista con más alto rango disponible de entre los vértices ya descubiertos; estableceremos más adelante, con la correctez de esta especialización, que la estrategia que acabamos de delinear consigue precisamente eso.

Para implementar la estrategia que acabamos de delinear para esta especialización usamos una exploración similar a la que usamos para la especialización de Euler, agregándole a la frontera una pila auxiliar donde vaya almacenando los caminos que va construyendo. Esta pila auxiliar será manejada por métodos de la frontera. De esta forma, el centro de acción será el vértice con arista disponible que sea la primera que se encuentre al desandar el camino construido, pero verificando que el vértice que se va a usar como siguiente centro de acción, tenga una arista disponible de rango mayor o igual a la que se usó para llegar a ese vértice. La responsabilidad de esta elección recae en el método elige de la frontera.

Representaremos el camino que se construye exactamente de la misma forma en que lo hicimos en la especialización de Euler, excepto que cada vértice deberá evaluar y ordenar sus aristas en cuanto es incluido en el camino por primera vez. Hablaremos indistintamente de la última arista agregada al camino o del vértice en el tope de la pila auxiliar. Al igual que en la especialización de Euler, al empezar la exploración colocaremos a s como el origen de camino que se vaya a construir.

En el Listado 8.3 podemos ver la implementación de la frontera, que hereda de la frontera dada para ExploracionEuleriana. En el Capítulo 6 demostramos que los predicados dados para la frontera en ExploracionBasica también se cumplen. Los cambios introducidos a la manera como se maneja la frontera en ExploracionEuleriana son:

a. Al descubrir a un vértice debemos evaluar sus aristas y reorganizarlas para que al pedirle la siguiente arista disponible, entregue, en tiempo constante, a la de mayor rango.
b. Al regresar por el camino construido para seleccionar a un nuevo centro de acción, debemos elegir a uno que tenga una arista disponible de rango mayor o igual a la arista inmediata anterior en el camino.

Listado 8.3: Especialización de la frontera para ExploracionSFS 1/2

```
1  public class FronteraSFS extends FronteraConCamino     {
2          /* Constructor que inicia a la frontera y a los caminos.  */
3      public FronteraSFS() {
4          super();
5      }
6          /* Selecciona al siguiente centro de acción. Si el        *
7           * vértice "al final" del camino tiene aristas dispo-     *
8           * nibles (es GRIS), y la de mayor rango es de rango      *
9           * mayor o igual a la que la precede en el camino, ése    *
10          * es el siguiente centro de acción. Elimina a los vér-   *
11          * tices negros de la misma manera que lo hace Exploracion *
12          * Euleriana.                                             */
```

Listado 8.3: Especialización de la frontera para ExploracionSFS 2/2

```
13        public Nodos elige() {
14            if (!caminoIter.esNoVacia())
15                throw new EmptyCollectionException
16                    ("¡Tratando de elegir de un camino vacío!");
17            Aristas e = null;
18            Nodos vHacia = null;
19            if (caminoIter.esNoVacia()) // El del tope del camino
20                vHacia = (Nodos) caminoIter.peek();
21
22            boolean noEsRangoMayor = true;
23            while (caminoIter.esNoVacia()
24                    && (vHacia.yaExplorado() || noEsRangoMayor)) {
25                /* Debemos sacar del camino a todos los vértices      *
26                 * NEGROs que están encima de uno GRIS.              */
27                vHacia = (Nodos) caminoIter.pop();
28                if (caminoIter.esNoVacia())
29                    e = (Aristas) caminoIter.pop();
30                if (caminoIter.esNoVacia())
31                    vHacia = (Nodos) caminoIter.pop();
32                /* Vemos el rango de la siguiente arista del vértice  *
33                 * en el tope del camino.                            */
34                int rangoVHacia = ((Aristas) vHacia.getNext()).getD();
35                if (caminoIter.esNoVacia()) {
36                    e = (Aristas) caminoIter.peek();
37                /* Para dejar el camino igual.                      */
38                    caminoIter.push(vHacia);
39                    if (rangoVHacia >= e.getD())
40                        noEsRangoMayor = false;
41                }
42            }
43            if (!caminoIter.esNoVacia())
44                return null;
45            return vHacia;
46        }
47    }
```

Por supuesto que el método exploracionGenerica permanece igual, lo mismo que el método explora. El método yaDescubierto se copia de la especialización que dimos para caminos eulerianos – el método que da a un vértice como explorado, cierraNodo, no hace nada, al igual que en la superclase. Tenemos que redefinir al método descubriendo, ya que en esta especialización tenemos que evaluar las aristas del vértice que se está descubriendo y reorganizarlas. Asimismo, al iniciar la exploración en s ya no basta con poner a s como origen del camino, sino que tenemos también que evaluar y reorganizar a sus aristas. En el Listado 8.4 mostramos nuevamente la implementación de yaDescubierto, y agregamos las implementaciones de descubriendo y ponComoOrigen. Lo primero que hacemos en este listado es terminar la implementación del constructor de la clase, construyendo un ejemplar de la frontera descrita.

Listado 8.4: Exploración con revisión de aristas (SFS)

```
107       /* Constructor. Construye la frontera vacía.                */
108   public ExploracionSFS(Digrafica g) {
109       super(g);
110       frontera = new FronteraConCamino();
199   }
200       /* Coloca a s en el fondo del camino.                        */
201   protected void ponComoOrigen(Nodos s)  {
202       super.ponComoOrigen(s);
203         /* Reorganiza las aristas de s.                            */
204       s.setListaIncidencia(reorganizaAristas(s));
205       ((FronteraConCamino) frontera).pushCamino(s);
206   }
207       /* Agrega a un vértice GRIS al camino.                       */
208   protected void yaDescubierto(Nodos centroAccion, Aristas e,
209                                Nodos u) {
210       super.yaDescubierto(centroAccion, e, u);
211       ((FronteraConCamino) frontera).pushCamino(e);
212       ((FronteraConCamino) frontera).pushCamino(u);
213   }
214       /* Además de meter al vértice recién descubierto a           *
215        * frontera, lo coloca también en el tope del camino y       *
216        * reorganiza sus aristas.                                   */
217   protected void descubriendo(Nodos centroAccion, Aristas e,
218                               Nodos u) {
219       super.descubriendo(centroAccion, e, u);
220       centroAccion.setListaIncidencia(reorganizaAristas(
221                                            centroAccion));
222       ((FronteraConCamino) frontera).pushCamino(e);
223       ((FronteraConCamino) frontera).pushCamino(u);
224   }
```

Al empezar la exploración se debe elegir a un vértice como origen de la misma. Esto se hace exactamente de la misma manera que en la superclase. El proceso de elegir arista, una vez reorganizada la lista de aristas disponibles, se hace exactamente de la misma manera que en la superclase.

Como al inicializar la exploración se introduce a s a la frontera (evaluando y reorganizando a todas las aristas incidente en s) y se le coloca en el fondo del camino, al empezar a iterar la exploración el único vértice en la frontera y en la pila auxiliar es s, y por lo tanto el primer centro de acción.

Queda ya nada más por definir la manera en que vamos a reorganizar a las aristas, pues esto puede agregar un costo importante al método descubriendo y, si no se hace bien, también al método de la clase NodoBasico que elige la siguiente arista a recorrer. Trabajemos en ese problema.

8.2.1. Evaluación y reorganización de las aristas

En términos generales podemos pensar que lo que tenemos que hacer es ordenar una lista de *grado*(v) valores, donde cada valor será el rango asignado a la arista del vértice v. Una propiedad que presentan los valores para el rango de cada arista es que el número de valores distintos que pueden aparecer en la lista está acotado por el máximo grado de los vértices de la gráfica. Acá es conveniente aclarar que, aunque pudiera parecer que los valores están acotados por *grado*(v) para cada v, esto no es así, ya que el valor asignado a una arista que se está evaluando desde el vértice v depende del vértice destino de esa arista. Si en la clase NodoBasico agregamos el atributo grado para cada vértice, la construcción de la gráfica sigue costando lo mismo, ya que podemos contar las aristas conforme las vamos asociando al vértice. Obtener el grado máximo de la gráfica es, a su vez, una operación lineal en el número de vértices, por lo que tampoco eso altera de manera importante la complejidad de nuestra especialización. Si suponemos que contamos con esta información, podemos garantizar que el rango de una arista no puede tomar un valor mayor que el grado máximo de la gráfica. Podemos calcular este grado máximo en el constructor de ExploracionSFS, y como este cálculo consiste en comparar a los rangos de los vértices, el costo que se agrega a la complejidad del constructor de la superclase es lineal con $|V|$. En el Listado 8.5 se encuentra el cálculo del rango máximo de la gráfica en el constructor de la misma.

Listado 8.5: Cálculo del rango máximo en la gráfica

```
110         maxRango = g.getNodo(1).getGrado();
111         for (int i=2; i <= g.getM(); i++) {
112             if (maxRango > g.getNodo(i).getGrado()) {
113                 maxRango = g.getNodo(i).getGrado();
114             }
115         }
```

Utilizamos un ordenamiento por cubetas (*bucket sort*), en el que tenemos una cubeta para cada posible valor para el rango. Este espacio adicional está acotado por $|E|$, pero es compartido por todos los vértices. Como ya estamos utilizando espacio de orden *grado*(v) para las aristas incidentes en cada vértice, el usar estas cubetas no incrementa la cantidad de espacio usada por el algoritmo, asintóticamente. Al descubrir al vértice v, se coloca a la arista e_k en la i-ésima cubeta, si es que $e_k = v \rightarrow u$ y la etiqueta de u es i (o se modifica a i al evaluar a e_k). Si la arista ya fue orientada (y evaluada) desde otro vértice, no será tomada en cuenta para la nueva lista de incidencia del vértice que se está descubriendo – línea [412] del Listado 8.6.

Cada cubeta puede contener más de una arista, por lo que implementaremos cada cubeta como una lista ligada, ya que el orden en que se toman las salidas de una misma cubeta es irrelevante, y nos proporciona la ventaja de que el agregar o quitar un elemento a una lista ligada toma tiempo de orden constante. Requerimos entonces de un vector donde en la posición i se encuentre la cubeta para las aristas con rango i. Hay que notar que no podemos garantizar que primero entran las aristas con rango 1, después las de rango 2, etc., sino que se presentan en orden arbitrario; tampoco tenemos la garantía de que en todas las cubetas quede alguna arista, ya que si bien, como demostraremos más adelante, un vértice no puede

tener una arista de rango i si no tiene una de rango $i-1$, la de rango $i-1$ pudiera estar orientada hacia el vértice y no desde él.

Al terminarse de reorganizar la lista de incidencia del vértice que se está procesando, se procede a "recoger" las aristas que se encuentran en las cubetas, desde la de rango más alto hacia la de menor rango, ligándolas nuevamente en una lista, con estructura idéntica a la lista de incidencia original, pero en la que las aristas estarán ya ordenadas.

Tanto el ordenamiento por cubetas como la recolección de los elementos de las distintas cubetas para ligarlas en una lista tiene complejidad lineal en la suma de los grados de los vértices, por lo que la complejidad total de reorganizar a todas las aristas de la gráfica es lineal en el número de aristas, que ya aparece como un sumando en la complejidad total del algoritmo. En el Listado 8.6 tenemos la implementación del método que reorganiza las aristas cuando un vértice es descubierto.

Listado 8.6: Método que evalúa a las aristas al descubrirse un vértice 1/2

```
400    /* Reorganiza a las aristas para que estén ordenadas de         *
401     * acuerdo al rango que se le asigna al evaluarla. Asimismo     *
402     * se actualiza la etiqueta del centro de acción y, en su       *
403     * caso, la del vértice destino de cada arista.                 */
404    protected ListaDeIncidencia reorganizaAristas(Nodos v) {
405        LinkedList[] cubetas = new LinkedList[maxRango + 1];
406        ListaDeIncidencia lista = v.getListaIncidencia();
407        IteradorListaIncidencia itLista = new
408                              IteradorListaIncidencia(lista);
409        int rangoMax = 0;
410        while (!itLista.esVacia()) {
411            Arista e = itLista.next();
412            if (rango[e.getPos()] > 0) { // Ya fue orientada
413                continue;
414            }
415            Nodos u = e.getOtroNodo(v);
416            e.orientaArco(v, u);
417            int vi = v.getPos();
418            int ui = u.getPos();
419            int ei = e.getPos();
420            rango[ei] = ++ etiqueta[ui];
421            e.setD(rango[ei]); // Para ser manejado en la frontera
422            rangoMax = (rangoMax < rango[ei])
423                             ? rango[ei]
424                             : rangoMax;
425            if (etiqueta[vi] < rango[ei]) {
426                etiqueta[vi] = rango[ei];
427            }
428            cubetas[rango[ei]].push(ei);
429        } /* Ya están las cubetas "llenas".                            */
430        lista = new ListaDeIncidencia();
431        itLista = new IteradorListaIncidencia(lista);
```

Listado 8.6: Método que evalúa a las aristas al descubrirse un vértice 2/2

```
432         for (int i = rangoMax; i > 0; i--) {
433             while (! cubetas[i].isEmpty()) {
434                 Aristas e = (Aristas) cubetas[i].next();
435                 itLista.add(e);
436             }/* Acabamos de vaciar la i-ésima cubeta.                */
437         } /* Acabamos de recoger a las aristas en las cubetas.      */
438         return lista;
439     }
```

Es claro que esta implementación cumple con las propiedades que pedimos para mantener la complejidad del algoritmo, ya que:

i. Revisa todas las aristas en la lista de incidencia original de un vértice dado, ya que itera mientras haya aristas disponibles en la lista original.

ii. No vuelve a evaluar a una arista que ya haya sido evaluada - líneas [425–427] del Listado 8.6.

iii. Las aristas quedan ordenadas en la nueva lista de mayor a menor ya que se recogen las cubetas de mayor a menor – control de la iteración en la línea [432] del Listado 8.6.

iv. Tanto meter a las aristas a las cubetas como sacarlas de ellas lleva orden constante.

v. Revisar las cubetas para encontrar que no hay nada en ellas nos lleva, en total, la suma de los rangos de los vértices, y cada cubeta se revisa una vez para cada vértice, por lo que nuevamente, a lo largo de toda la ejecución, se ocupará tiempo lineal en el número de aristas para recoger y revisar las cubetas.

vi. Una vez que están reorganizadas las aristas, obtener una arista de la nueva lista se hace en tiempo constante, lo mismo que eliminarla de la misma.

vii. Saber cuál es el rango de la arista con rango más alto lleva tiempo constante, pues todo lo que hay que hacer es preguntar cuál es el rango de la arista en la cabeza de la lista.

En resumen, reorganizamos a las aristas del vértice que se descubre, para obtener una nueva lista que únicamente contiene a aquellas aristas que no hayan sido orientadas previamente. El reorganizar a las aristas nos lleva tiempo lineal en $|E|$ y el obtener la siguiente arista a procesar nos lleva tiempo constante. Asimismo, el saber el rango máximo disponible de un vértice también lleva tiempo constante.

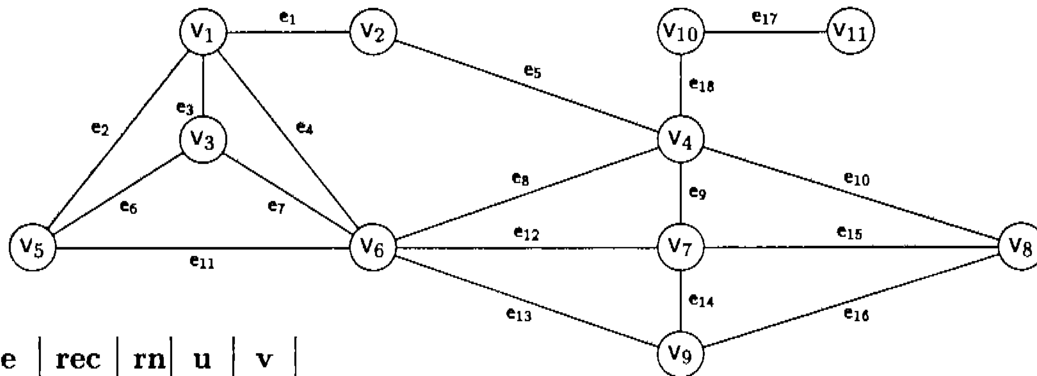
De todo lo anterior, tanto el meter a un vértice a la pila auxiliar como el sacarlo cuando se le pinta de NEGRO, queda como responsabilidad de los métodos de SFS que se encargan de recorrer la gráfica, y se hace explícitamente en ellos, como se hizo en ExploracionEuleriana.

Dada la estrategia SFS, la elección del siguiente centro de acción siempre va a ser alguno de los vértices que tenga aristas de rango máximo disponibles, pues conforme se van alcanzando a los vértices, si el vértice puede aumentar de rango lo hace, pero nunca disminuye. Estableceremos esta propiedad.

8.3. Ejemplo de ejecución de SFS

Revisemos la ejecución de exploracionGenerica desde ExploracionSFS en la gráfica de la Figura 8.16. En esta figura mostramos el estado inicial de las estructuras de datos.

Figura 8.16: Ejemplo para aplicar SFS



e	rec	rn	u	v
1	no		1	2
2	no		1	5
3	no		1	3
4	no		1	6
5	no		2	4
6	no		3	5
7	no		3	6
8	no		4	6
9	no		4	7
10	no		4	8
11	no		5	6
12	no		7	6
13	no		6	9
14	no		7	9
15	no		7	8
16	no		8	9
17	no		10	11
18	no		4	10

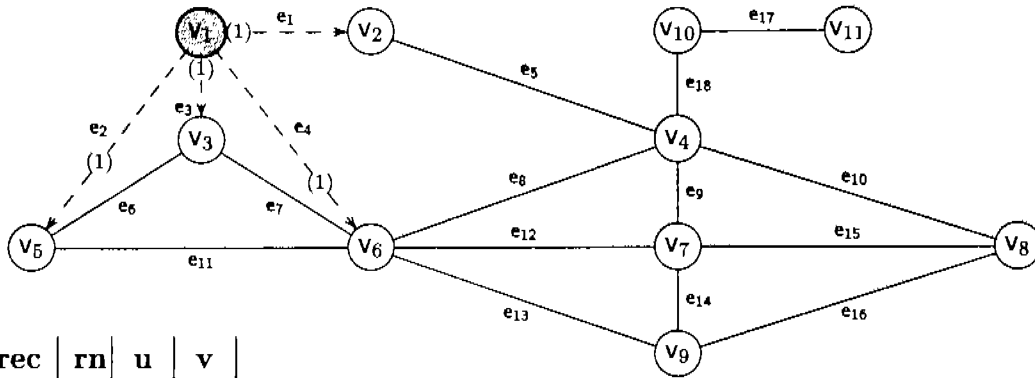
v	et	listas de incidencia ordenadas	listas de incidencia originales
1	0		$\boxed{1}$,2,3,4
2	0		$\boxed{1}$,5
3	0		$\boxed{3}$,6,7
4	0		$\boxed{5}$,8,9,18,10
5	0		$\boxed{2}$,6,11
6	0		$\boxed{4}$,7,11,8,12,13
7	0		$\boxed{12}$,9,15,14
8	0		$\boxed{10}$,15,16
9	0		$\boxed{13}$,14,16
10	0		$\boxed{18}$,17
11	0		$\boxed{17}$

frontera: ∅

Camino: ∅

Durante la exposición que sigue encerraremos en círculo en las listas de incidencia a la arista que será la siguiente en ser recorrida para cada vértice. La orientación de la arista consistirá en colocar al vértice origen de la arista orientada en la primera columna para los vértices extremos de la arista, y al vértice destino en la segunda columna, como está definido en la clase AristaBasica.

Figura 8.17: Al terminar con la lista de incidencia de s



e	rec	rn	u	v
1	no	1	1	2
2	no	1	1	5
3	no	1	1	3
4	no	1	1	6
5	no		2	4
6	no		3	5
7	no		3	6
8	no		4	6
9	no		4	7
10	no		4	8
11	no		5	6
12	no		7	6
13	no		6	9
14	no		7	9
15	no		7	8
16	no		8	9
17	no		10	11
18	no		4	10

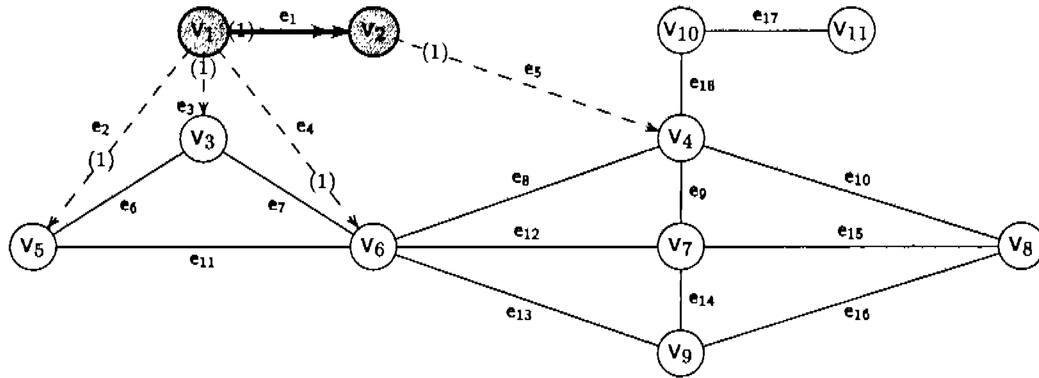
v	et	listas de incidencia ordenadas	listas de incidencia originales
1	1	$\boxed{1^1}, 2^1, 3^1, 4^1$	1,2,3,4
2	1		$\boxed{1}$,5
3	1		$\boxed{3}$,6,7
4	0		$\boxed{5}$,8,9,18,10
5	1		$\boxed{2}$,6,11
6	1		$\boxed{4}$,7,11,8,12,13
7	0		$\boxed{12}$,9,15,14
8	0		$\boxed{10}$,15,16
9	0		$\boxed{13}$,14,16
10	0		$\boxed{18}$,17
11	0		$\boxed{17}$

frontera: $\{v_1\}$

Camino: $\boxed{1}$ (v_1)

Usaremos la siguiente notación para denotar el rango de las aristas (en este ejemplo, 3 es el máximo rango que se alcanza): $\rightsquigarrow$ *Rango* $= 1$, $\Longrightarrow$ *Rango* $= 2$ y $\Rrightarrow$ *Rango* $= 3$. Empezamos la ejecución con $s = v_1$. En la Figura 8.17 mostramos el estado de la gráfica al terminar de explorar a v_1.

Figura 8.18: Al terminar de descubrir a v_2



e	rec	rn	u	v
1	sí	1	1	2
2	no	1	1	5
3	no	1	1	3
4	no	1	1	6
5	no	1	2	4
6	no		3	5
7	no		3	6
8	no		4	6
9	no		4	7
10	no		4	8
11	no		5	6
12	no		7	6
13	no		6	9
14	no		7	9
15	no		7	8
16	no		8	9
17	no		10	11
18	no		4	10

v	et	listas de incidencia ordenadas	listas de incidencia originales
1	1	$1^1\ \boxed{2^1}, 3^1, 4^1$	1,2,3,4
2	1	$\boxed{5^1}$	1,5
3	1		$\boxed{3}$,6,7
4	1		$\boxed{5}$,8,9,18,10
5	1		$\boxed{2}$,6,11
6	1		$\boxed{4}$,7,11,8,12,13
7	0		$\boxed{12}$,9,15,14
8	0		$\boxed{10}$,15,16
9	0		$\boxed{13}$,14,16
10	0		$\boxed{18}$,17
11	0		$\boxed{17}$

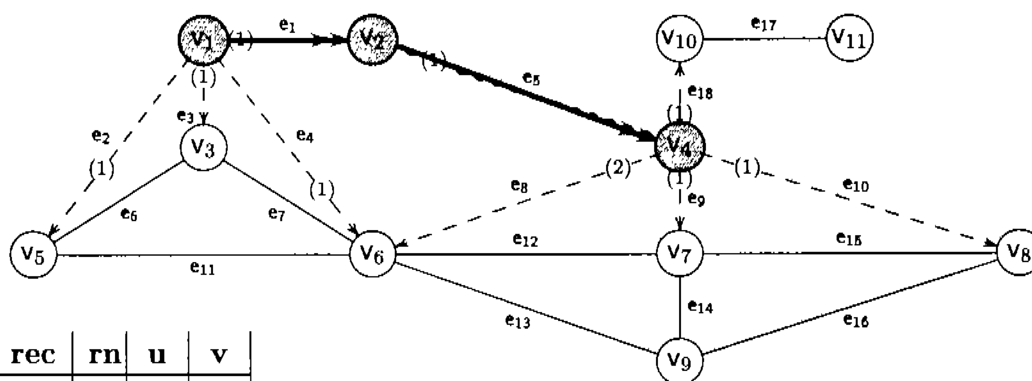
frontera: $\{v_1, v_2\}$

camino: $\boxed{1}$ (v_1) e_1 $\boxed{1}$ (v_2)

Tomamos a la arista e_1, ya que es la que está en el tope de la cubeta de máximo rango, la recorremos y descubrimos a v_2, quedando las estructuras de datos como se muestran en la Figura 8.18.

En este momento, la arista con mayor rango del vértice v_2 que es el vértice que se encuentra en el tope del camino, de hecho la única, es e_5, por lo que es la siguiente en ser recorrida, descubriéndose al vértice v_4. Se procede a orientar a las aristas no orientadas de v_4, quedando las estructuras de datos como se muestran en la Figura 8.19.

Figura 8.19: Al llegar el camino a v_4 por primera vez



e	rec	rn	u	v
1	sí	1	1	2
2	no	1	1	5
3	no	1	1	3
4	no	1	1	6
5	sí	1	2	4
6	no		3	5
7	no		3	6
8	no	2	4	6
9	no	1	4	7
10	no	1	4	8
11	no		5	6
12	no		7	6
13	no		6	9
14	no		7	9
15	no		7	8
16	no		8	9
17	no		10	11
18	no	1	4	10

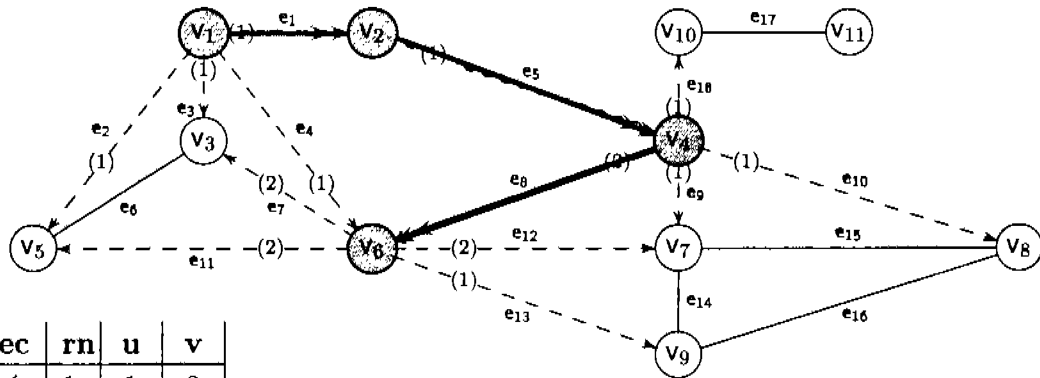
v	et	listas de incidencia ordenadas	listas de incidencia originales
1	1	1^1 $\boxed{2^1}$, 3^1, 4^1	1,2,3,4
2	1	5^1	1,5
3	1		$\boxed{3}$,6,7
4	2	$\boxed{8^2}$, 9^1, 18^1, 10^1	5,8,9,18,10
5	1		$\boxed{2}$,6,11
6	2		$\boxed{4}$,7,11,8,12,13
7	1		$\boxed{12}$,9,15,14
8	1		$\boxed{10}$,15,16
9	0		$\boxed{13}$,14,16
10	1		$\boxed{18}$,17
11	0		$\boxed{17}$

frontera: { v_1,v_2,v_4}

camino: $\boxed{1}$ (v_1) e_1 $\boxed{1}$ (v_2) e_5 $\boxed{1}$ (v_4)

El vértice en el tope del camino es v_4, por lo que es el siguiente centro de acción. Como tiene una arista incidente de rango 2, e_8, ésta es la siguiente a recorrer. Se le recorre y se descubre a v_6, por lo que se procede a orientar a las aristas incidentes en v_6 que no han sido todavía orientadas. El resultado de este proceso se puede ver en la Figura 8.20.

Figura 8.20: Al descubrir a v_6 desde v_4



e	rec	rn	u	v
1	sí	1	1	2
2	no	1	1	5
3	no	1	1	3
4	no	1	1	6
5	sí	1	2	4
6	no		3	5
7	no	2	6	3
8	sí	2	4	6
9	no	1	4	7
10	no	1	4	8
11	no	2	6	5
12	no	2	6	7
13	no	1	6	9
14	no		7	9
15	no		7	8
16	no		8	9
17	no		10	11
18	no	1	4	10

v	et	listas de incidencia ordenadas	listas de incidencia originales
1	1	$1^1, \boxed{2^1}, 3^1, 4^1$	1,2,3,4
2	1	5^1	1,5
3	2		$\boxed{3}$,6,7
4	2	$8^2, \boxed{9^1}, 18^1, 10^1$	5,8,9,18,10
5	2		$\boxed{2}$,6,11
6	2	$\boxed{7^2}, 11^2, 12^2, 13^1$	4,7,11,8,12,13
7	2		$\boxed{12}$,9,15,14
8	1		$\boxed{10}$,15,16
9	1		$\boxed{13}$,14,16
10	1		$\boxed{18}$,17
11	0		$\boxed{17}$

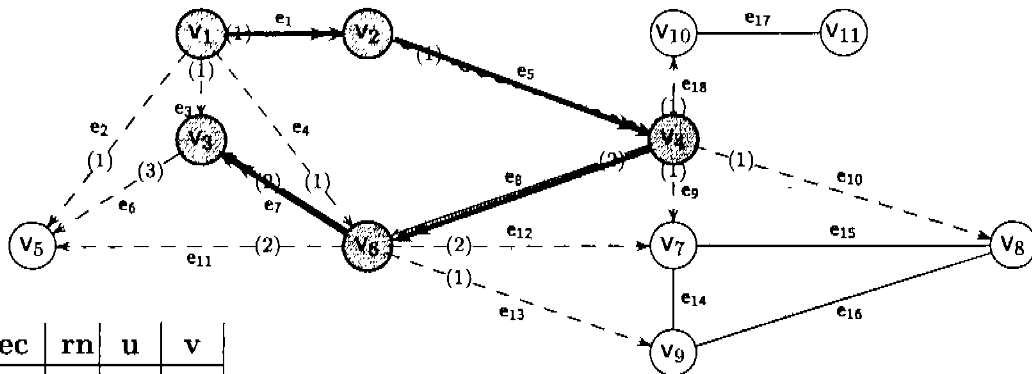
frontera: $\{v_1, v_2, v_4, v_6\}$

camino: $\boxed{1}(v_1) \cdot e_1 \cdot \boxed{1}(v_2) \cdot e_5 \cdot \boxed{1}(v_4) \cdot e_8 \cdot \boxed{2}(v_6)$

Se extiende el camino desde v_6, que es el vértice en el tope del mismo en este momento,

agregando la arista e_7, que tiene rango 2 (por lo que es válido elegirla) y es la que se encuentra al frente de la lista de incidencia ordenada del vértice. Al recorrer a e_7 se descubre a v_3 y se procede a orientar a la única arista incidente en v_3 que no ha sido orientada todavía. El resultado de estas acciones se pueden observar en la Figura 8.21.

Figura 8.21: Se descubre a v_3 al recorrer a e_7 desde v_6



e	rec	rn	u	v
1	sí	1	1	2
2	no	1	1	5
3	no	1	1	3
4	no	1	1	6
5	sí	1	2	4
6	no	3	3	5
7	sí	2	6	3
8	sí	2	4	6
9	no	1	4	7
10	no	1	4	8
11	no	2	6	5
12	no	2	6	7
13	no	1	6	9
14	no		7	9
15	no		7	8
16	no		8	9
17	no		10	11
18	no	1	4	10

v	et	listas de incidencia ordenadas	listas de incidencia originales
1	1	1^1 [2^1], 3^1, 4^1	1,2,3,4
2	1	5^1	1,5
3	3	[6^3]	3,6,7
4	2	8^2, [9^1], 18^1, 10^1	5,8,9,18,10
5	3		[2],6,11
6	2	7^2 [11^2], 12^2, 13^1	4,7,11,8,12,13
7	2		[12],9,15,14
8	1		[10],15,16
9	1		[13],14,16
10	1		[18],17
11	0		[17]

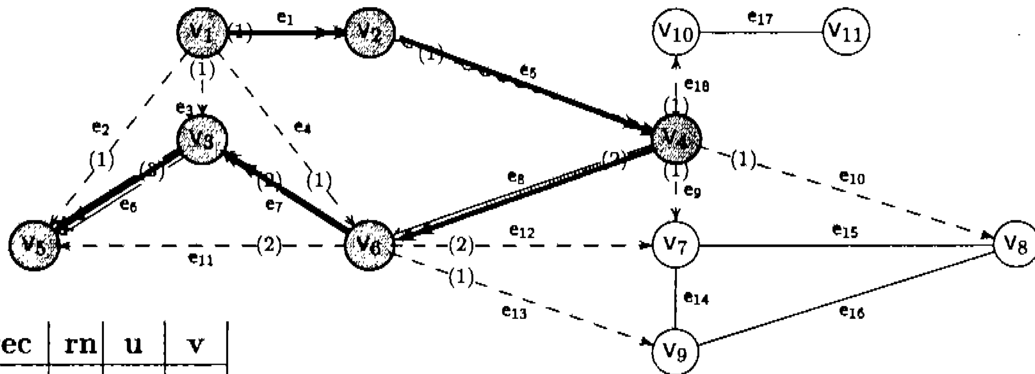
frontera: { v_1,v_2,v_4,v_6,v_3}

camino: [1] v_1 e_1 [1] v_2 e_5 [1] v_4 e_8 [2] v_6 e_7 [2] v_3

Se extiende el camino con la arista e_6, ya que tiene rango apropiado y tiene como origen

a v_3 que es el vértice en el tope del camino. Al hacerlo, se descubre a v_5, y como ya no tiene aristas incidentes sin orientar, su lista de incidencia ordenada queda vacía. Las estructuras de datos en este punto se muestran en la Figura 8.22.

Figura 8.22: Se extiende el camino con e_6, descubriendo a v_5



e	rec	rn	u	v
1	sí	1	1	2
2	no	1	1	5
3	no	1	1	3
4	no	1	1	6
5	sí	1	2	4
6	sí	3	3	5
7	sí	2	6	3
8	sí	2	4	6
9	no	1	4	7
10	no	1	4	8
11	no	2	6	5
12	no	2	6	7
13	no	1	6	9
14	no		7	9
15	no		7	8
16	no		8	9
17	no		10	11
18	no	1	4	10

v	et	listas de incidencia ordenadas	listas de incidencia originales
1	1	1^1 [2^1], 3^1, 4^1	1,2,3,4
2	1	5^1	1,5
3	3	6^3	3,6,7
4	2	8^2 [9^1], 18^1, 10^1	5,8,9,18,10
5	3		2,6,11
6	2	7^2 [11^2], 12^2, 13^1	4,7,11,8,12,13
7	2		[12],9,15,14
8	1		[10],15,16
9	1		[13],14,16
10	1		[18],17
11	0		[17]

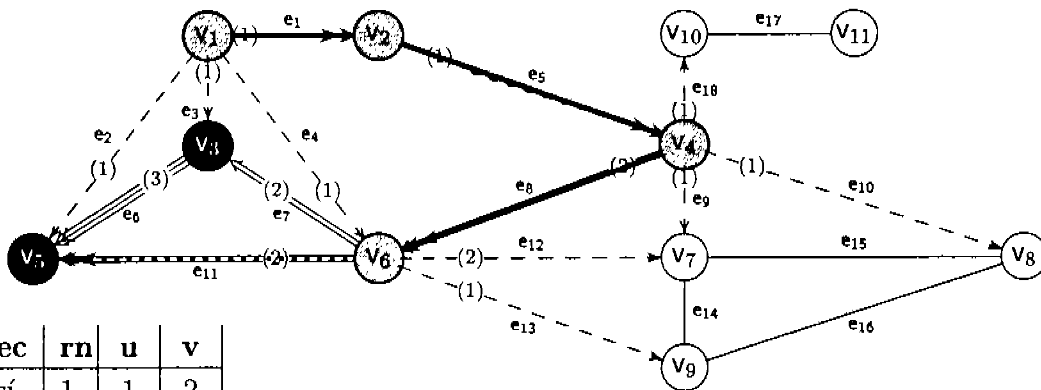
frontera: { $v_1, v_2, v_4, v_6, v_3, v_5$ }

camino: [1] v_1 e_1 [1] v_2 e_5 [1] v_4 e_8 [2] v_6 e_7 [2] v_3 e_6 [3] v_5

Desde v_5 ya no se puede extender el camino, ya que no tiene ninguna arista orientada desde él. Por lo tanto, siguiendo una estrategia similar a la de DFS, retrocedemos por el camino quitando vértices del tope del camino, y pintándolos de NEGRO en el caso de que ya

no tengan aristas en la lista ordenada. De esta forma, quitamos a v_5 del camino, y también lo pintamos de NEGRO. Pero como tampoco podemos salir de v_3, también lo quitamos del camino y lo pintamos de NEGRO. En este retroceso regresamos a v_6. Como a v_6 llegamos con una arista de rango 2, deberemos extender el camino con aristas de rango al menos 2. La siguiente arista por recorrer en la lista ordenada de v_6 tiene rango 2, por lo que la podemos usar. Sin embargo, como el vértice en el destino ya es NEGRO, no lo incluimos en el camino. El resultado de recorrer a e_{11} después del retroceso se muestra en la Figura 8.23.

Figura 8.23: Se avanza desde v_3 hasta v_5, y se retrocede por v_3 hasta v_6 donde se recorre e_{12} y se vuelve a retroceder a v_6



e	rec	rn	u	v
1	sí	1	1	2
2	no	1	1	5
3	no	1	1	3
4	no	1	1	6
5	sí	1	2	4
6	sí	3	3	5
7	sí	2	6	3
8	sí	2	4	6
9	no	1	4	7
10	no	1	4	8
11	sí	2	6	5
12	no	2	6	7
13	no	1	6	9
14	no		7	9
15	no		7	8
16	no		8	9
17	no		10	11
18	no	1	4	10

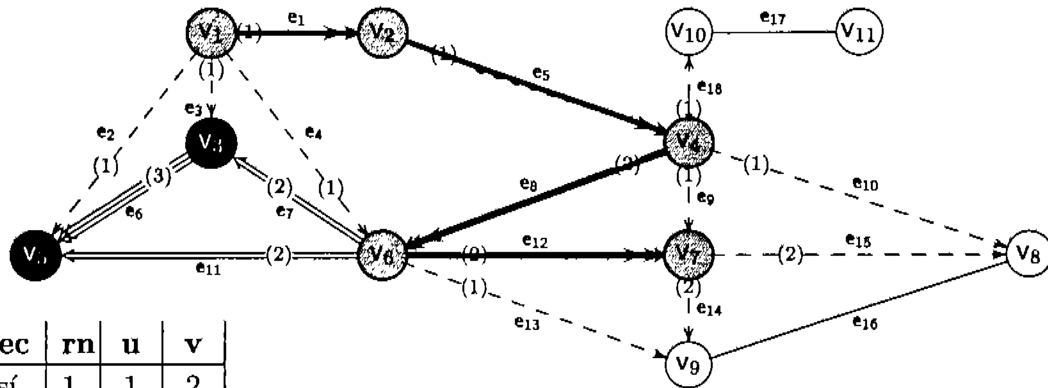
v	et	listas de incidencia ordenadas	listas de incidencia originales
1	1	1^1 [2^1], 3^1, 4^1	1,2,3,4
2	1	5^1	1,5
3	3	6^3	3,6,7
4	2	8^2 [9^1], 18^1, 10^1	5,8,9,18,10
5	3		2,6,11
6	2	7^2, 11^2 [12^2], 13^1	4,7,11,8,12,13
7	2		[12],9,15,14
8	1		[10],15,16
9	1		[13],14,16
10	1		[18],17
11	0		[17]

frontera: $\{v_1, v_2, v_4, v_6\}$

camino: [1] (v_1) e_1 [1] (v_2) e_5 [1] (v_4) e_8 [2] (v_6)

Nuevamente con v_6 en el tope del camino, podemos extenderlo recorriendo a la arista e_{12}, que tiene el rango apropiado (mayor o igual a 2). Al recorrer a e_{12} se descubre a v_7 y se procede a orientar a las aristas que no han sido orientadas. El resultado se puede observar en la Figura 8.24.

Figura 8.24: Se extiende el camino desde v_6 con e_{12}



e	rec	rn	u	v
1	sí	1	1	2
2	no	1	1	5
3	no	1	1	3
4	no	1	1	6
5	sí	1	2	4
6	sí	3	3	5
7	sí	2	6	3
8	sí	2	4	6
9	no	1	4	7
10	no	1	4	8
11	sí	2	6	5
12	sí	2	6	7
13	no	1	6	9
14	no	2	7	9
15	no	2	7	8
16	no		8	9
17	no		10	11
18	no	1	4	10

v	et	listas de incidencia ordenadas	listas de incidencia originales
1	1	$1^1, \boxed{2^1}, 3^1, 4^1$	1,2,3,4
2	1	5^1	1,5
3	3	6^3	3,6,7
4	2	$8^2, \boxed{9^1}, 18^1, 10^1$	5,8,9,18,10
5	3		2,6,11
6	2	$7^2, 11^2, 12^2, \boxed{13^1}$	4,7,11,8,12,13
7	2	$\boxed{15^2}, 14^2$	12,9,15,14
8	2		$\boxed{10}$,15,16
9	2		$\boxed{13}$,14,16
10	1		$\boxed{18}$,17
11	0		$\boxed{17}$

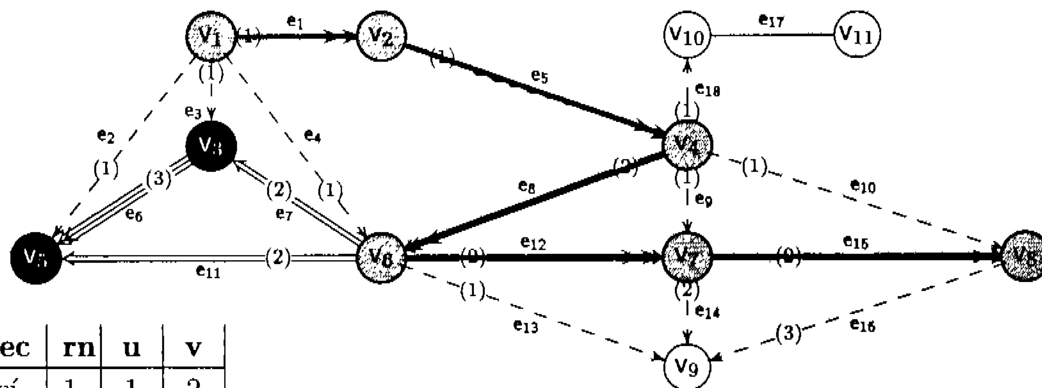
frontera: $\{v_1, v_2, v_4, v_6, v_7\}$

camino: $\boxed{1}\,v_1 \; e_1 \; \boxed{1}\,v_2 \; e_5 \; \boxed{1}\,v_4 \; e_8 \; \boxed{2}\,v_6 \; e_{12} \; \boxed{2}\,v_7$

Desde v_7 se extiende el camino con e_{15}, que es una arista de rango válido con origen en v_7. Al recorrer a e_{15} se descubre a v_8, por lo que se procede a orientar a e_{16}. El resultado se

puede ver en la Figura 8.25.

Figura 8.25: Se agrega al camino e_{15} desde v_7



e	rec	rn	u	v
1	sí	1	1	2
2	no	1	1	5
3	no	1	1	3
4	no	1	1	6
5	sí	1	2	4
6	sí	3	3	5
7	sí	2	6	3
8	sí	2	4	6
9	no	1	4	7
10	no	1	4	8
11	sí	2	6	5
12	sí	2	6	7
13	no	1	6	9
14	no	2	7	9
15	sí	2	7	8
16	no	3	8	9
17	no		10	11
18	no	1	4	10

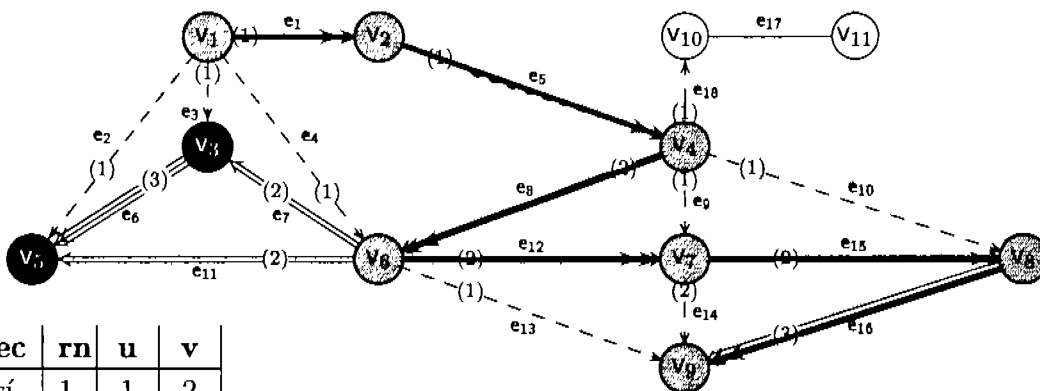
v	et	listas de incidencia ordenadas	listas de incidencia originales
1	1	$1^1\,\boxed{2^1}, 3^1, 4^1$	1,2,3,4
2	1	5^1	1,5
3	3	6^3	3,6,7
4	2	$8^2\,\boxed{9^1}, 18^1, 10^1$	5,8,9,18,10
5	3		2,6,11
6	2	$7^2, 11^2, 12^2\,\boxed{13^1}$	4,7,11,8,12,13
7	2	$15^2, \boxed{14^2}$	12,9,15,14
8	3	$\boxed{16^3}$	10,15,16
9	3		$\boxed{13}$,14,16
10	1		$\boxed{18}$,17
11	0		$\boxed{17}$

frontera: $\{v_1, v_2, v_4, v_6, v_7, v_8\}$

camino: $\boxed{1}\,v_1\ e_1\ \boxed{1}\,v_2\ e_5\ \boxed{1}\,v_4\ e_8\ \boxed{2}\,v_6\ e_{12}\ \boxed{2}\,v_7\ e_{15}\ \boxed{2}\,v_8$

Con v_8 en el tope del camino se extiende éste con e_{16}, ya que cumple con tener rango mayor o igual que con el que se llegó a v_8. Se descubre a v_9 y se le coloca en el tope del camino. Las estructuras de datos quedan como se muestra en la Figura 8.26, con v_9 sin aristas orientadas desde él.

Figura 8.26: Desde v_8 se recorre la arista e_{16} y se descubre a v_9



e	rec	rn	u	v
1	sí	1	1	2
2	no	1	1	5
3	no	1	1	3
4	no	1	1	6
5	sí	1	2	4
6	sí	3	3	5
7	sí	2	6	3
8	sí	2	4	6
9	no	1	4	7
10	no	1	4	8
11	sí	2	6	5
12	sí	2	6	7
13	no	1	6	9
14	no	2	7	9
15	sí	2	7	8
16	sí	3	8	9
17	no		10	11
18	no	1	4	10

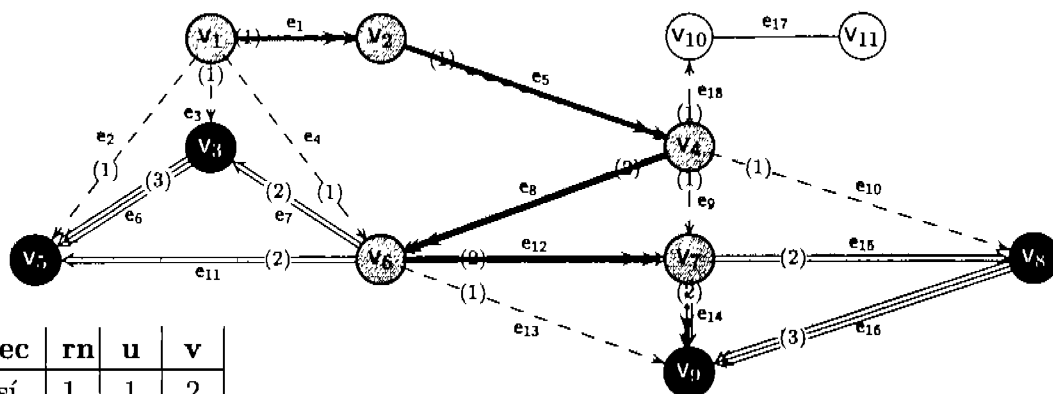
v	et	listas de incidencia ordenadas	listas de incidencia originales
1	1	1^1 $\boxed{2^1}$, 3^1, 4^1	1,2,3,4
2	1	5^1	1,5
3	3	6^3	3,6,7
4	2	8^2 $\boxed{9^1}$, 18^1, 10^1	5,8,9,18,10
5	3		2,6,11
6	2	7^2, 11^2, 12^2 $\boxed{13^1}$	4,7,11,8,12,13
7	2	15^2, $\boxed{14^2}$	12,9,15,14
8	3	16^3	10,15,16
9	3		13,14,16
10	1		$\boxed{18}$,17
11	0		$\boxed{17}$

frontera: { v_1,v_2,v_4,v_6,v_7,v_8,v_9 }

camino: $\boxed{1}$ (v_1) e_1 $\boxed{1}$ (v_2) e_5 $\boxed{1}$ (v_4) e_8 $\boxed{2}$ (v_6) e_{12} $\boxed{2}$ (v_7) e_{15} $\boxed{2}$ (v_8) e_{16} $\boxed{3}$ (v_9)

Como v_9 no tiene aristas en su lista ordenada, se le quita del tope del camino, se le pinta de NEGRO y se regresa a v_8, desde donde tampoco podemos ya extender el camino, por lo que también lo pintamos de NEGRO y retrocedemos a v_7. Ahí recorremos la arista e_{14}, que como tiene de destino a un vértice que ya fue pintado de NEGRO, no colocamos a v_9 en el camino. En este momento las estructuras de datos están como se muestra en la Figura 8.27.

Figura 8.27: Se retrocede hasta v_7 desde donde se recorre la arista e_{14}



e	rec	rn	u	v
1	sí	1	1	2
2	no	1	1	5
3	no	1	1	3
4	no	1	1	6
5	sí	1	2	4
6	sí	3	3	5
7	sí	2	6	3
8	sí	2	4	6
9	no	1	4	7
10	no	1	4	8
11	sí	2	6	5
12	sí	2	6	7
13	no	1	6	9
14	sí	2	7	9
15	sí	2	7	8
16	sí	3	8	9
17	no		10	11
18	no	1	4	10

v	et	listas de incidencia ordenadas	listas de incidencia originales
1	1	$1^1\,\boxed{2^1}, 3^1, 4^1$	1,2,3,4
2	1	5^1	1,5
3	3	6^3	3,6,7
4	2	$8^2\,\boxed{9^1}, 18^1, 10^1$	5,8,9,18,10
5	3		2,6,11
6	2	$7^2, 11^2, 12^2\,\boxed{13^1}$	4,7,11,8,12,13
7	2	$15^2, 14^2$	12,9,15,14
8	3	16^3	10,15,16
9	3		13,14,16
10	1		$\boxed{18}$,17
11	0		$\boxed{17}$

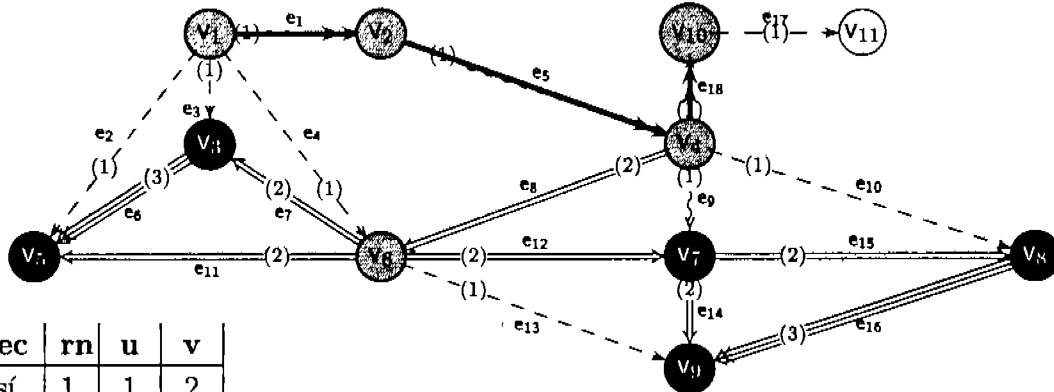
frontera: $\{ v_1, v_2, v_4, v_6, v_7 \}$

camino: $\boxed{1}\,(v_1)\,e_1\,\boxed{1}\,(v_2)\,e_5\,\boxed{1}\,(v_4)\,e_8\,\boxed{2}\,(v_6)\,e_{12}\,\boxed{2}\,(v_7)$

Ya no podemos extender el camino desde v_7 porque ya no tiene aristas disponibles, por lo que lo pintamos de NEGRO y retrocedemos a v_6, donde tampoco podemos extender el camino, pero en esta ocasión es porque llegamos a v_6 por una arista de rango 2 y la única disponible es de rango 1. Retrocedemos entonces a v_4, pero sin pintar a v_6 de NEGRO, y recorremos la arista e_9. Como esta arista tiene como destino a un vértice NEGRO, no colocamos a v_7 en la pila, y volvemos a extender el camino desde v_4 con la arista e_{18}, descubriéndose en este proceso al vértice v_{10}, por lo que procedemos a orientar sus aristas no orientadas todavía.

En la Figura 8.28 tenemos el estado de las estructuras de datos inmediatamente después de descubrir a v_{10}.

Figura 8.28: Se retrocede hasta v_4 desde donde se recorren las aristas e_9, e_{10} y e_{18}



e	rec	rn	u	v
1	sí	1	1	2
2	no	1	1	5
3	no	1	1	3
4	no	1	1	6
5	sí	1	2	4
6	sí	3	3	5
7	sí	2	6	3
8	sí	2	4	6
9	sí	1	4	7
10	sí	1	4	8
11	sí	2	6	5
12	sí	2	6	7
13	no	1	6	9
14	no	2	7	9
15	sí	2	7	8
16	sí	3	8	9
17	no	1	10	11
18	sí	1	4	10

v	et	listas de incidencia ordenadas	listas de incidencia originales
1	1	$1^1, \boxed{2^1}, 3^1, 4^1$	1,2,3,4
2	1	5^1	1,5
3	3	6^3	3,6,7
4	2	$8^2, 9^1, 18^1, \boxed{10^1}$	5,8,9,18,10
5	3		2,6,11
6	2	$7^2, 11^2, 12^2, \boxed{13^1}$	4,7,11,8,12,13
7	2	$15^2, 14^2$	12,9,15,14
8	3	16^3	10,15,16
9	3		13,14,16
10	1	$\boxed{17^1}$	18,17
11	1		$\boxed{17}$

frontera: $\{v_1, v_2, v_4, v_{10}\}$

camino: 1 (v_1) e_1 1 (v_2) e_5 1 (v_4) e_{18} 1 (v_{10})

Desde v_{10} extendemos el camino con e_{17} y se descubre al vértice v_{11}, procesando su lista de incidencia. El resultado de este paso se puede observar en la Figura 8.29.

Figura 8.29: Se extiende el camino desde v_{10} con la arista e_{17} y se descubre a v_{11}

e	rec	rn	u	v
1	sí	1	1	2
2	no	1	1	5
3	no	1	1	3
4	no	1	1	6
5	sí	1	2	4
6	sí	3	3	5
7	sí	2	6	3
8	sí	2	4	6
9	sí	1	4	7
10	no	1	4	8
11	sí	2	6	5
12	sí	2	6	7
13	no	1	6	9
14	sí	2	7	9
15	sí	2	7	8
16	sí	3	8	9
17	sí	1	10	11
18	sí	1	4	10

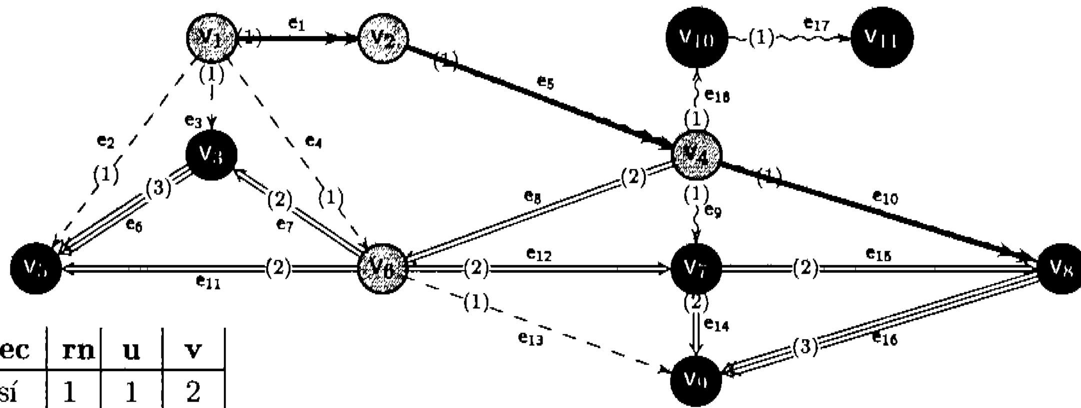
v	et	listas de incidencia ordenadas	listas de incidencia originales
1	1	$1^1, \boxed{2^1}, 3^1, 4^1$	1,2,3,4
2	1	5^1	1,5
3	3	6^3	3,6,7
4	2	$8^2, 9^1, 18^1, \boxed{10^1}$	5,8,9,18,10
5	3		2,6,11
6	2	$7^2, 11^2, 12^2, \boxed{13^1}$	4,7,11,8,12,13
7	2	$15^2, 14^2$	12,9,15,14
8	3	16^3	10,15,16
9	3		13,14,16
10	1	17^1	18,17
11	1		17

frontera: $\{ v_1, v_2, v_4, v_{10}, v_{11} \}$

camino: 1 (v_1) e_1 1 (v_2) e_5 1 (v_4) e_{18} 1 (v_{10}) e_{17} 1 (v_{11})

Como ni v_{11} ni v_{10} tienen ya aristas disponibles, retrocedemos por el camino quitándolos de la pila y pintándolos de NEGRO, hasta que llegamos a v_4, desde donde recorremos la arista e_{10}. Como el vértice destino de e_{10} ya está pintado de NEGRO, no la colocamos en la pila, y nos encontramos nuevamente en v_4 en este momento. El estado de las estructuras de datos se puede observar en la Figura 8.30.

Figura 8.30: Se extiende el camino desde v_{10} con la arista e_{17} y se descubre a v_{11}



e	rec	rn	u	v
1	sí	1	1	2
2	no	1	1	5
3	no	1	1	3
4	no	1	1	6
5	sí	1	2	4
6	sí	3	3	5
7	sí	2	6	3
8	sí	2	4	6
9	sí	1	4	7
10	sí	1	4	8
11	sí	2	6	5
12	sí	2	6	7
13	no	1	6	9
14	sí	2	7	9
15	sí	2	7	8
16	sí	3	8	9
17	sí	1	10	11
18	sí	1	4	10

v	et	listas de incidencia ordenadas	listas de incidencia originales
1	1	$1^1, \boxed{2^1}, 3^1, 4^1$	1,2,3,4
2	1	5^1	1,5
3	3	6^3	3,6,7
4	2	$8^2, 9^1, 18^1, 10^1$	5,8,9,18,10
5	3		2,6,11
6	2	$7^2, 11^2, 12^2, \boxed{13^1}$	4,7,11,8,12,13
7	2	$15^2, 14^2$	12,9,15,14
8	3	16^3	10,15,16
9	3		13,14,16
10	1	17^1	18,17
11	1		17

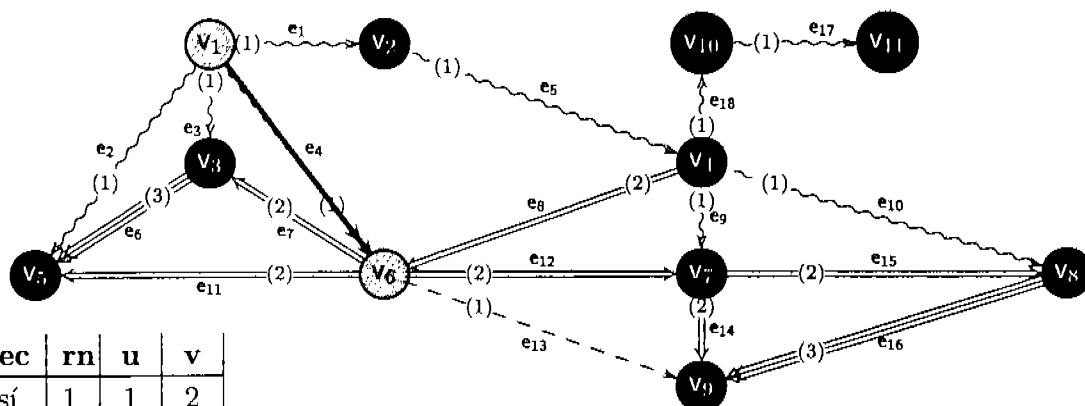
frontera: $\{v_1, v_2, v_4\}$

camino: $\boxed{1}\,(v_1)\; e_1\; \boxed{1}\,(v_2)\; e_5\; \boxed{1}\,(v_4)$

Como v_4 ya no tiene aristas disponibles, lo pintamos de NEGRO. Lo mismo hacemos con v_2 al retroceder hacia v_1, en donde tenemos nuevamente aristas disponibles. Recorremos las aristas e_2 y e_3, pero como ambas tienen como destinos a vértices NEGROs, no colocamos a sus destinos en la pila. Al recorrer a e_4, como su destino v_6 es un vértice GRIS, extendemos el camino y pasamos a v_6 para intentar proseguir. Las estructuras de datos se encuentran en

este momento como se muestran en la Figura 8.31.

Figura 8.31: Se retrocede desde v_4, pasando por v_2 hasta v_1, desde donde se recorren e_2, e_3 y e_4



e	rec	rn	u	v
1	sí	1	1	2
2	sí	1	1	5
3	sí	1	1	3
4	sí	1	1	6
5	sí	1	2	4
6	sí	3	3	5
7	sí	2	6	3
8	sí	2	4	6
9	sí	1	4	7
10	sí	1	4	8
11	sí	2	6	5
12	sí	2	6	7
13	no	1	6	9
14	sí	2	7	9
15	sí	2	7	8
16	sí	3	8	9
17	sí	1	10	11
18	sí	1	4	10

v	et	listas de incidencia ordenadas	listas de incidencia originales
1	1	$1^1, 2^1, 3^1, 4^1$	1,2,3,4
2	1	5^1	1,5
3	3	6^3	3,6,7
4	2	$8^2, 9^1, 18^1, 10^1$	5,8,9,18,10
5	3		2,6,11
6	2	$7^2, 11^2, 12^2, \boxed{13^1}$	4,7,11,8,12,13
7	2	$15^2, 14^2$	12,9,15,14
8	3	16^3	10,15,16
9	3		13,14,16
10	1	17^1	18,17
11	1		17

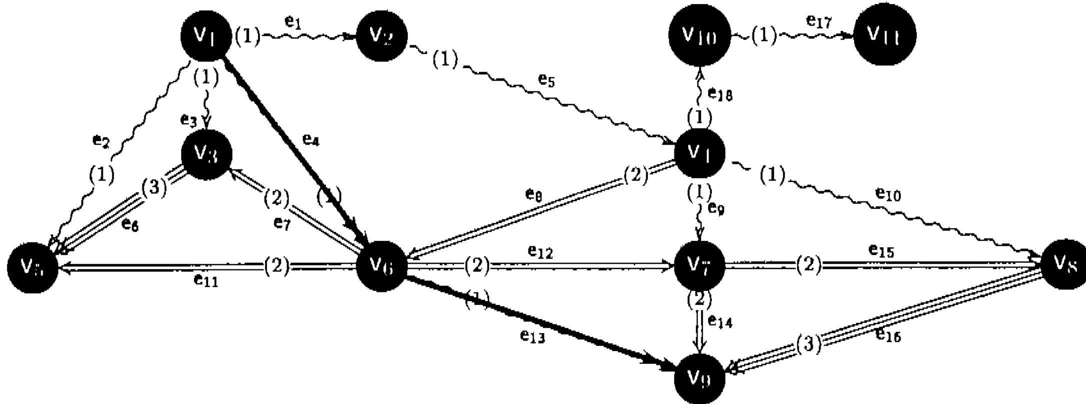
frontera: { v_1,v_6}

camino: [1](v_1) -e_4- [1](v_6)

En este momento podemos extender el camino desde v_6 con e_{13}, pero como el destino de esta arista ya está pintado de NEGRO, no incluimos a v_9 en la pila. Como ya no hay aristas disponibles para v_6, lo pintamos de NEGRO y retrocedemos a v_1, donde también ya no hay aristas disponibles, por lo que lo pintamos de NEGRO. En este momento ya se vació la

frontera, por lo que la ejecución termina, con las estructuras de datos como se muestran en la Figura 8.32.

Figura 8.32: Se retrocede desde v_4, pasando por v_2 hasta v_1, desde donde se recorren e_2, e_3 y e_4



e	rec	rn	u	v
1	sí	1	1	2
2	sí	1	1	5
3	sí	1	1	3
4	sí	1	1	6
5	sí	1	2	4
6	sí	3	3	5
7	sí	2	6	3
8	sí	2	4	6
9	sí	1	4	7
10	sí	1	4	8
11	sí	2	6	5
12	sí	2	6	7
13	sí	1	6	9
14	sí	2	7	9
15	sí	2	7	8
16	sí	3	8	9
17	sí	1	10	11
18	sí	1	4	10

v	et	listas de incidencia ordenadas	listas de incidencia originales
1	1	$1^1, 2^1, 3^1, 4^1$	1,2,3,4
2	1	5^1	1,5
3	3	6^3	3,6,7
4	2	$8^2, 9^1, 18^1, 10^1$	5,8,9,18,10
5	3		2,6,11
6	2	$7^2, 11^2, 12^2, 13^1$	4,7,11,8,12,13
7	2	$15^2, 14^2$	12,9,15,14
8	3	16^3	10,15,16
9	3		13,14,16
10	1	17^1	18,17
11	1		17

frontera: ∅

camino: ∅

8.4. Propiedades de esta implementación de SFS

Como mencionamos en la sección anterior, dado que SFS es una especialización de ExploracionBasica que respeta todas las características de este último, como establecimos que exploracionGenerica desde ExploracionBasica termina, también lo hace desde ExploracionSFS. También se hereda la característica de que va a explorar a todos los vértices alcanzables desde s y va a recorrer a todas las aristas incidentes en esos vértices. Por supuesto que también construye a G_π, exactamente de la misma manera que lo hace en ExploracionBasica.

Hay varias propiedades inherentes a ExploracionSFS que tienen que ver con la manera como se reorganizan las aristas, la disciplina de elegirlas cuando un vértice es centro de acción y la manera de elegir al próximo centro de acción. También entra en juego la función de evaluación particular que se eligió para asignar rango a las aristas. Dado que modificamos la manera de elegir al siguiente centro de acción a que se haga utilizando la pila auxiliar, primero demostraremos que esta modificación no altera las especificaciones dadas de manera general en SFS. Empezaremos por demostrar que exploracionGenerica desde ExploracionSFS explora a toda la componente conexa en la que se encuentra el origen de la exploración, ya que esto pudiera no ser evidente. Asimismo, tenemos que demostrar que nuestra especialización elige como siguiente centro de acción a un vértice que se encuentra pintado de GRIS. Una vez hecho esto, estableceremos varias propiedades particulares de ExploracionSFS que se dan por la manera de elegir al siguiente centro de acción y a la siguiente arista a recorrer.

Lema 8.1 *exploracionGenerica desde ExploracionSFS explora a todos los vértices de la componente conexa en la que se encuentra* s.

Demostración:

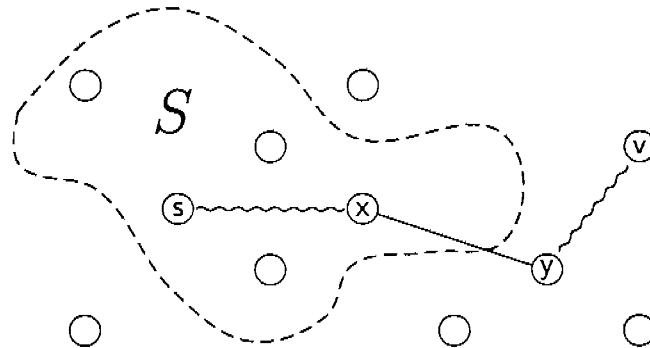
Por contrapositivo, supongamos que al terminar exploracionGenerica desde ExploracionSFS quedó al menos un vértice v sin explorar. Sea S el conjunto de los vértices que sí fueron explorados (que quedaron pintados de NEGRO). Podemos garantizar que $S \neq \varnothing$, porque al poner a s como origen de la exploración – líneas [201–206] del Listado 8.4 – se mete a s a la frontera y se le pinta de GRIS. El algoritmo no podría terminar si s sigue pintado de GRIS, por lo que al menos $s \in S$ y $S \neq \varnothing$.

También podemos garantizar que en la componente conexa únicamente quedaron vértices NEGROSy vértices BLANCOS, ya que si hubiesen quedado vértices GRISes, el algoritmo no hubiese terminado.

Como estamos trabajando con la componente conexa que contiene a s, $\exists$ un camino $P = s \rightsquigarrow v$. Sea x el último vértice que toca P en S, y sea y el primer vértice que toca P fuera de S (x podría ser s y y podría ser v), como se muestra en la Figura 8.33.

Consideremos a la arista x—y. Si esta arista hubiera sido recorrida, el vértice y hubiera sido descubierto y pintado de GRIS, por lo que no se encontraría fuera de S. $\therefore$ x—y no fue recorrida. Como x fue explorado, todas sus aristas fueron orientadas, y como y no fue descubierto, la arista x—y debió quedar orientada x$\rightarrow$y. Pero como no fue recorrida, no se pudo haber pintado de NEGRO a x, ya que tiene todavía aristas sin recorrer.

Figura 8.33: Demostración del Lema 8.1



∴ la arista x→y debió ser recorrida para que se pudiera haber pintado de NEGRO a x, y y debió ser descubierto, contradiciendo la hipótesis de que no lo fue. ❑

Lema 8.2 *Toda arista incidente en algún vértice de la componente conexa de* s *es orientada y recorrida por* exploracionGenerica *de* ExploracionSFS.

Demostración:

Es similar a la demostración del Lema 8.1. ❑

Lema 8.3 *Una vez que un vértice es descubierto, ninguna arista adicional procesada después tendrá como destino a ese vértice.*

Demostración:

Sea u un vértice que ya fue descubierto. Cuando fue descubierto se orientaron todas aquellas aristas incidentes en el vértice que no habían sido orientadas previamente. Esto quiere decir que orienta a todas las aristas que quedan todavía sin evaluar (orientar) desde u hacia afuera. Una vez hecho esto, no queda ninguna arista incidente en u sin estar evaluada, por lo que ningún otro vértice la podrá colocar en su lista de aristas disponibles.

∴ Después de descubrir a un vértice u, ninguna arista será orientada hacia él. ❑

Lema 8.4 *Para cada vértice* v *hay a lo más una arista orientada* e = u→v *con rango* i.

Demostración:

Por contrapositivo supongamos que existen dos aristas orientadas e_1 = x→v y e_2 = y→v y ambas tienen rango j. Sin pérdida de generalidad, supongamos que e_1 se orientó a v antes que e_2. El rango de una arista depende únicamente del valor del atributo etiqueta en el vértice destino. Al orientarse a e_1 supongamos que el rango asignado a e_1 fue j. Como se ejecutó la línea [420] del Listado 8.6, sabemos que al invocar a reorganizaAristas desde x, antes de orientar a e_1, v.etiqueta vale $j - 1$, se incrementa en la línea [420] y cuando se le copia a e_1.rango, v.etiqueta queda valiendo j, valor que mantiene hasta que se oriente a alguna otra arista hacia v. Por lo que al orientarse a e_2, como se hace después de orientar a e_1, nuevamente el valor de v.etiqueta vale al menos j (suponiendo que no se haya orientado a ninguna otra arista hacia v entre estas dos orientaciones), nuevamente en la línea [420]

se incrementa el valor de v.etiqueta en 1, dejándolo en al menos $j+1$, y después copia ese valor a e_2.rango. De esto, tenemos que en el momento en que se asigna rango a e_2,

$$e_1.\text{rango} = j < \text{v.etiqueta} \leq e_2.\text{rango}$$

lo que demuestra que no pueden tener el mismo valor. ❑

Lema 8.5 *Si un vértice* v *tiene una arista orientada hacia él de rango m, entonces tiene una arista orientada hacia él de rango i, para cada i tal que $1 \leq i < m$.*

Demostración:

La demostración se hará por inducción en m.

Base: Si $m = 1$, se cumple por vacuidad, ya que ninguna arista puede tener rango menor a 1.

Inducción: Supongamos como hipótesis de inducción que se cumple para $m < k$ y veamos qué pasa si $m = k$. Por el Lema 8.3, en el momento en que se descubre a v todas las aristas que quedan orientadas hacia él ya han sido evaluadas. Esta evaluación depende del atributo etiqueta de v, que se incrementa en 1 cada vez que se orienta a una arista hacia él. Si a la arista se le asignó el rango k, es porque este atributo se había incrementado $k-1$ veces, lo que implica que $k-1$ aristas antes que ésta habían sido orientadas hacia v. Por el Lema 8.5, hay a lo más una arista de rango i, $1 \leq i < k$, orientada hacia v, por lo que estas aristas tuvieron que tomar los rangos $1, \ldots, k-1$ conforme fueron evaluadas. ❑

Decimos que un vértice v es *alcanzado* cuando es el destino de una arista que se recorre. La primera vez que un vértice es alcanzado, es descubierto cuando se le alcanza, independientemente de que el vértice alcanzado sea o no el siguiente centro de acción. Una vez que un vértice ha sido alcanzado una vez, como demostramos en el Lema 8.3, está completo el conjunto de aristas orientadas hacia él.

Lema 8.6 *Todo vértice* v $\neq$ s *en la componente conexa de* s *es alcanzado por una arista de rango 1, esto es, colocado en el tope de la pila auxiliar en algún momento durante la ejecución de ExploracionSFS, habiendo llegado a él por una arista de rango 1.*

Demostración:

Supongamos, por contrapositivo, que existe un vértice v que no fue alcanzado por una arista de rango 1. Por el Lema 8.1, v fue explorado, por lo que fue descubierto y por lo tanto alcanzado. Sea $k > 1$ el rango de la arista de menor rango desde la cual fue alcanzado. Por el Lema 8.4, si v tiene una arista orientada hacia él de rango k, debe tener aristas orientadas hacia él de rango $i, i = 1, \ldots, k-1$. De donde podemos garantizar que tiene una arista de rango 1 orientada hacia él. Y por el Lema 8.2, toda arista incidente en los vértices de la componente conexa de s es orientada y recorrida, por lo que no puede ser que esta arista se haya dejado sin recorrer, y por lo tanto colocada en la pila. ❑

Lema 8.7 *El centro de acción es siempre un vértice cuya referencia se encuentra en el tope de la pila auxiliar.*

Demostración:

- El centro de acción se determina en el método de la frontera elige, al inicio de cada iteración en el método exploracionGenerica.
- Al terminar una iteración del ciclo en exploracionGenerica, se encuentra en el tope de la pila auxiliar la referencia a un vértice, que es el candidato a ser el centro de acción en la siguiente iteración.
- Si el vértice cuya referencia se encuentra en el tope de la pila ya no tiene aristas disponibles se le pinta de NEGRO y se le quita de la la pila; al hacer esto, igual que en ExploracionEuleriana, regresa por la pila quitando a los vértices NEGROs, hasta que se vacíe la pila o quede un vértice GRIS en el tope de la misma. Si la pila no se vacía, el vértice que queda en el tope de la misma cumple con una de las dos características que siguen:

  i. No tiene ya aristas disponibles de ningún rango.
  ii. Tiene alguna arista disponible de rango mayor o igual al de la arista por la que se le alcanzó y colocó en la pila auxiliar.

- Si el vértice cuya referencia se encuentra en el tope de la pila es susceptible de ser centro de acción, simplemente regresa esa referencia como siguiente centro de acción.

  En ambos casos, lo que este método hace es buscar en la pila hasta que en el tope quede un vértice GRIS. Como estableceremos más adelante, nunca se va a poder dar el caso de que la pila auxiliar se vacíe sin que se haya encontrado a un vértice GRIS. ❑

Lema 8.8 *En la pila auxiliar se encontrarán los vértices* u *y* v *en posiciones consecutivas, con* u *más cerca del fondo que* v*, si y sólo existe la arista* u—v *y queda orientada* u→v.

Demostración:

⟹ Supongamos que en la pila auxiliar, en algún momento durante la ejecución de exploracionGenerica desde ExploracionSFS, se encontraron referencias a los vértices u y v en posiciones consecutivas de la pila, con u más cerca del fondo que v. Esto quiere decir que cuando se colocó a v en la pila, u era el centro de acción. Si se eligió a v para colocar en la pila, esto fue porque v era el destino de una arista elegible, cuyo origen era u.

$\therefore$ $\exists$ la arista u—v que fue recorrida desde u, por lo que tuvo que ser orientada u→v.

⟸ Supongamos que al terminar el algoritmo, la arista u—v quedó orientada u→v. Por el Lema 8.2, toda arista es orientada y recorrida por ExploracionSFS. Para ser recorrida con esta orientación, se requiere que u sea el centro de acción, esto es, por el Lema 8.6, que en el tope de la pila auxiliar se encuentre una referencia a u. Como el destino de esta arista es v, ya sea que v ya hubiese sido descubierto o no, una referencia a v se coloca en la pila auxiliar, inmediatamente arriba de la referencia a u – método descubriendo, línea [223] en el Listado 8.4 y método yaDescubierto, línea [212] en el mismo listado.

$\therefore$ u y v se encontrarán en posiciones consecutivas de la pila auxiliar, con la referencia a u más cercana del fondo de la pila que la referencia a v. ❑

Lema 8.9 *En la pila auxiliar se encontrarán las referencias a los vértices* v_0, $v_1, \ldots, v_k$ *en este orden con* v_k *en el tope de la pila, si y sólo si existe un camino formado por las aristas orientadas* $v_i \rightarrow v_{i+1}$, *con* $i = 0, \ldots, k-1$, *tal que los rangos de estas aristas forman una sucesión no decreciente.*

Demostración:

Por el Lema 8.8, dos vértices u y v se encuentran consecutivos en la pila auxiliar si y sólo si existe una arista u—v que al terminar la ejecución queda orientada u→v. Esto es cierto para cada pareja de vértices consecutivos en la sucesión dada.

∴ se cumple la primera parte del teorema.

Nos falta establecer que la sucesión de aristas implícitas en este camino es no decreciente. Pero esto es claro de la manera como se elige un centro de acción, ya que un vértice v es centro de acción y se extiende el camino desde él sólo si tiene una arista orientada desde él disponible, cuyo rango sea mayor o igual que el de la arista que se recorrió para colocar a v en la pila auxiliar – método elige de FronteraSFS, líneas [34–41] en el Listado 8.3, que quita de la pila auxiliar a todos aquellos vértices que no cumplen con esta condición.

∴ la sucesión formada por los rangos de las aristas implícitas es una sucesión no decreciente. ❏

Lema 8.10 *Si* u *y* v *se encuentran en posiciones consecutivas en la pila auxiliar, con* u *más cerca del fondo que* v*, entonces* u *fue descubierto antes que* v.

Demostración:

Si u y v se encuentran en posiciones consecutivas en la pila, con u más cerca del fondo, quiere decir que v fue colocado en la pila cuando u estaba en el tope de la misma, esto es, recorriendo una arista orientada u→v. Esto implica que cuando se descubrió a u la arista u—v todavía no había sido evaluada, por lo que se le orientó desde u. Si v hubiera sido descubierto antes que u, por el Lema 8.3, esta arista habría estado sin evaluar al momento de descubrir a v, y por lo tanto se hubiera orientado v → u, y al descubrirse a u ya no hubiera sido orientada nuevamente.

∴ Cuando se descubrió a u, v todavía no había sido descubierto. ❏

Lema 8.11 *Si en la pila auxiliar se encuentran referencias a los vértices* v_0, $v_1, \ldots, v_k$ *en este orden con* v_k *en el tope de la pila, entonces el vértice* v_i *fue descubierto antes que el vértice* v_j*, para* $0 \leq i < j \leq k$.

Demostración:

La demostración se sigue del Lema 8.10. ❏

Lema 8.12 *Sea* $\langle v_0, v_1, \ldots, v_k \rangle$ *el contenido de la pila auxiliar en un momento dado de la ejecución de* exploracionGenerica *desde* ExploracionSFS*. Entonces, ningún vértice se repetirá en esta sucesión.*

Demostración:

Por contrapositivo, supongamos que el vértice v aparece en dos posiciones de la pila, $\langle v_0, v_1, \ldots, v, \ldots, v, \ldots, v_k \rangle$. Por el Lema 8.11, el orden en el que aparecen en la pila es el orden relativo en el que fueron descubiertos. La doble aparición de v implica que v fue descubierto antes que v, lo que no es posible. Por lo que ningún vértice podrá aparecer dos veces en la pila auxiliar. ❏

Lema 8.13 *Al terminar ExploracionSFS, para todo vértice* v *en la componente conexa de* s *existe un camino simple* $P = \mathsf{s} \rightsquigarrow \mathsf{v}$ *tal que todas sus aristas están etiquetadas con rango 1.*

Demostración:

Construimos el camino $P = \mathsf{s} \rightarrow \mathsf{u}_1 \rightarrow \ldots \rightarrow \mathsf{u}_k \rightarrow \mathsf{v}$ desde s hacia v. Por el Lema 8.6, existe una arista orientada hacia v con rango 1. Tomamos al origen de esa arista y lo colocamos como vértice u_k. Si $\mathsf{u}_k = \mathsf{s}$, entonces ya terminamos. Si no, tomamos a u_k, y por el mismo lema, tomamos a la arista orientada hacia u_k de rango 1, y al vértice origen lo colocamos como u_{k-1}. Y así sucesivamente hasta que el vértice origen de la arista considerada sea s.

Tenemos que garantizar que este procedimiento termina con s. Pero s es el único vértice con el que puede terminar, ya que es el único vértice que no tiene una arista orientada hacia él de rango 1 (de hecho, no tiene *ninguna* arista orientada hacia él). Lo que falta por establecer es que termina. Esto lo garantizamos si es que demostramos que el camino construido de esta manera es simple, pues si no se repite ningún vértice en él, como el número de vértices es finito, se tendrá que terminar de agregar vértices al camino.

Por contradicción, supongamos que no es simple, esto es, algún vértice se repite. Por el Lema 8.2 todas las aristas son recorridas. Por el Lema 8.9, como la sucesión que forman los rangos de las aristas es no decreciente ya que consiste únicamente de 1's, en algún momento durante la ejecución de SFS se encontrarán en la pila auxiliar los vértices del camino en el orden en que están, con v en el tope. Pero por el Lema 8.12, ningún vértice se puede repetir, de donde un vértice no puede aparecer dos veces en la pila como perteneciente al mismo camino.

$\therefore$ no se pueden repetir vértices en este camino. ❏

Lema 8.14 *Un vértice es pintado de* NEGRO *únicamente cuando se encuentra en el tope de la pila auxiliar.*

Demostración:

Un vértice únicamente puede ser pintado de NEGRO si es el centro de acción, y el centro de acción, por el Lema 8.7 se encuentra en el tope de la pila auxiliar. Asimismo, como por el Lema 8.12 ningún vértice aparece más de una vez, en un momento dado, en la pila auxiliar, no hay otra referencia al vértice que se está pintando de NEGRO que no sea la del tope de la pila auxiliar. ❏

Lema 8.15 *En la pila auxiliar se encuentran únicamente referencias a vértices pintados de* GRIS, *excepto posiblemente por el tope de la pila, donde pudiera haber una referencia a un vértice pintado de* NEGRO.

Demostración:

Sea v un vértice cuya referencia se va a ingresar a la pila auxiliar. Si $\mathsf{v} = \mathsf{s}$, se le ingresa cuando se le pone como origen, inmediatamente después de que se le ingresa a la frontera, por lo que estará pintado de GRIS. Por el Lema 8.12, mientras permanezca esta referencia en la pila auxiliar, no aparecerá ninguna otra referencia a s en ella. La única manera de sacar a s de la pila auxiliar es cuando se le agote su lista de aristas disponibles, ya que cada vez que s es centro de acción, como su etiqueta es 1, cualquiera de las aristas disponibles

cumple con el criterio de elegibilidad. De esto, cuando se le pinta de NEGRO se le saca de la pila auxiliar, por lo que s permanecerá en la pila auxiliar únicamente mientras esté pintado de GRIS.

Si $v \neq s$, hay dos métodos en el algoritmo, además de cuando se le pone como origen, cuando se ingresa la referencia a un vértice a la pila auxiliar:

i. En el método descubriendo, línea [222]. Como esto se hace si es que el vértice destino de la arista era BLANCO, y al meterlo a la frontera se le pinta de GRIS, la referencia que se está colocando en la pila es a un vértice GRIS.
ii. En el método yaDescubierto, línea [212]. Éste pudiera ser un momento peligroso, ya que se pudieran dar dos casos:
   a. La referencia que se coloca en el tope de la pila auxiliar es a un vértice que ya no tiene aristas disponibles, por lo que será pintado de NEGRO y sacado de la frontera en la siguiente iteración. Si éste es el caso, se le sacará de la pila auxiliar también en la siguiente iteración y no se colocará ninguna referencia encima de él entre que se le colocó en el tope de la pila auxiliar y se le pinta de NEGRO. Por el Lema 8.12, en la pila auxiliar en ningún momento se repite una referencia, por lo que ninguna referencia presente en la pila, por debajo de la del vértice que se pinta de NEGRO, puede ser a este vértice. $\therefore$ en este caso, únicamente el tope de la pila podrá tener una referencia a un vértice NEGRO.
   b. La referencia que se coloca en el tope de la pila auxiliar es a un vértice que ya fue pintado de NEGRO, y que por lo tanto no se encuentra en la frontera. Observemos qué pasa en este caso: al terminar la iteración y regresar a elegir el siguiente centro de acción, el método elige se encarga de decidir si el vértice en el tope de la pila auxiliar puede o no ser centro de acción. En la línea [24] del Listado 8.3, si el vértice es un vértice que ya está pintado de NEGRO, simplemente lo bota de la pila auxiliar y continúa buscando hacia el fondo de la pila una referencia a un vértice GRIS, que deberá encontrar inmediatamente. Por las mismas razones que en el caso anterior, la referencia al vértice NEGRO no aparece nuevamente en la pila.

$\therefore$ toda referencia que se mete a la pila es GRIS; en cuanto se mete a la pila una a un vértice NEGRO se bota esta referencia de la pila en la siguiente iteración; cuando se pinta de NEGRO al que se encuentra en el tope de la pila en el momento en que se le pinta de NEGRO se le bota de la pila auxiliar; y como no hay manera de que una referencia a un vértice GRIS, que no se encuentre en el tope de la pila auxiliar, pudiera convertirse a una referencia a un vértice NEGRO, nunca aparecerá un vértice NEGRO más que en el tope de la pila. ❑

Lema 8.16 *La pila auxiliar no se vacía hasta en tanto no se haya vaciado la frontera.*

Demostración:

La única presencia de s en la pila auxiliar es en el fondo de la misma. Por el Lema 8.14, únicamente se puede pintar de NEGRO – y sacar de la frontera – a vértices que se encuentren en el tope de la pila. Cada vez que s queda en el tope de la pila, y mientras tenga aristas disponibles, cualquiera de sus aristas es elegible para recorrer, ya que su etiqueta es 1 y s es el origen de la exploración. Por lo tanto, la única manera de quitar a s de la pila auxiliar es cuando se le pinte de NEGRO, en el método cierraNodo.

$\therefore$ s permanecerá en la pila auxiliar hasta que se le saque de la frontera.

Faltaría por demostrar que s es el último vértice al que se saca de la frontera. Por contradicción, supongamos que no, y que al sacar a s de la frontera queda un vértice, digamos v, que todavía está pintado de GRIS.

Como se ingresó a v a la frontera, quiere decir que hay un camino de aristas orientadas desde s hasta v, todas ellas de rango 1 – Lema 8.13. Sea ese camino $P = \mathsf{s} \rightarrow \mathsf{v}_0 \rightarrow \ldots \rightarrow \mathsf{v}_k \rightarrow \mathsf{v}$ tal que todas las aristas tienen rango 1. De ello, y del Lema 8.9, habrá algún momento en que estos vértices se encuentren en la pila auxiliar, en el orden en que se encuentran en el camino, con s en el fondo de la pila auxiliar y v en el tope. Para quitar a v de la pila, y dado que se llegó a él por una arista de rango 1, la única posibilidad es que ya no tenga aristas disponibles. Pero entonces, al quitarlo de la pila auxiliar se le pinta de NEGRO y se le saca de la frontera, contradiciendo la hipótesis de que continúa en la frontera después de que se sacó a s de la frontera. ❑

Teorema 8.17 *Cuando un vértice es centro de acción se cumple alguna de las condiciones que siguen:*

i. Ya no tiene aristas disponibles y se le debe pintar de NEGRO.
ii. Cumple con tener una arista disponible de rango tan alto como cualquier otra arista disponible.

Demostración:

Por el Lema 8.7, el centro de acción es siempre un vértice cuya referencia se encuentra en el tope de la pila auxiliar. Pero para que sea centro de acción, el método elige lo tiene que dejar en el tope de la pila auxiliar, para tomarlo de allí.

Para que la iteración en el método elige termine, se requiere que en el tope de la pila auxiliar se encuentre un vértice que ya no tiene aristas sin recorrer, o bien que sí tenga una arista disponible de rango mayor o igual, esto es, que no se cumpla la condición en la línea [24] en el Listado 8.3. En cualquier otro caso, en este iteración se procede a botar el tope de la pila auxiliar y seguir buscando hacia abajo.

El método elige regresa como siguiente centro de acción al vértice cuya referencia haya quedado en el tope de la pila auxiliar, que cumple con una de las dos propiedades listadas.

Es importante notar que por el Lema 8.16, mientras s no haya sido sacado de la pila auxiliar, la pila auxiliar no se podrá vaciar. Y como ya vimos en la demostración del Lema 8.16, s solo será sacado de la pila auxiliar cuando se le pinte de NEGRO. ❑

Lema 8.18 *Al terminar ExploracionSFS, $G' = \{V, E'\}$, con $E' = \{\mathsf{e} \in E \mid \mathsf{e.rango} = 1\}$ es un árbol generador de G.*

Demostración:

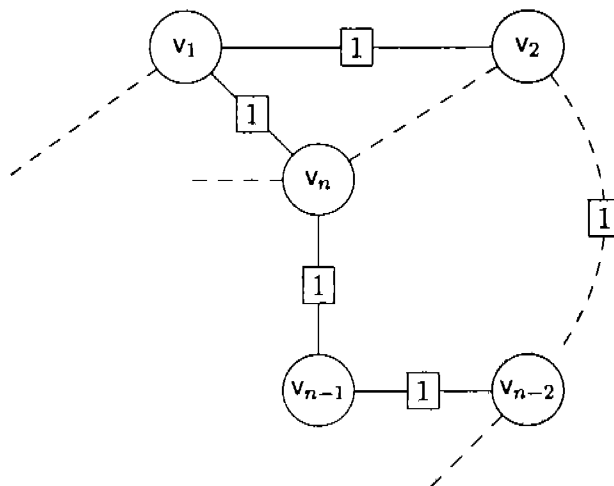
Queremos demostrar que G' es acíclica y que tiene $|V| - 1$ aristas. Empecemos por contar la cardinalidad de E'.

- Por el Lema 8.6, todo vértice $\mathsf{v} \neq \mathsf{s}$ es alcanzado por una arista de rango 1. Por el Lema 8.4, sólo puede haber una arista de rango 1 que alcance a v. En el caso de s, que es la raíz del árbol, ninguna arista lo alcanza.

$\therefore\ |E'| = |V| - 1.$

- Supongamos, por contradicción, que G' tiene un ciclo con n vértices, en el cual todas las aristas tienen rango 1, como se muestra en la Figura 8.34. Pero esto no es posible, porque estaríamos formando un camino con aristas cuyos rangos forman una sucesión no decreciente, y por el Lema 8.9, habrían estado consecutivos en algún momento en la pila auxiliar; pero entonces, por el Lema 8.12 ningún vértice se repite, contradiciendo la hipótesis de que hay un ciclo.

Figura 8.34: Ciclo de aristas con rango 1



$\therefore\ G'$ es un árbol generador de G. ❑

Lema 8.19 *Consideremos una componente conexa $T_k = (V_k, E_k)$ tal que $V_k \subseteq V$, $E_k \subseteq E$ y $\forall \mathsf{u}, \mathsf{v} \in V_k \quad \exists P = \mathsf{u} \rightsquigarrow \mathsf{v}$ tal que $\forall \mathsf{e} \in P, \mathsf{e.rango} = k$. Consideremos a las aristas orientadas por ExploracionSFS. Entonces T_k es un árbol dirigido.*

Demostración:

Para demostrar que cada componente conexa T_k es un árbol dirigido usaremos la caracterización que sigue:

> Un árbol dirigido es una gráfica acíclica cuya gráfica subyacente es conexa y donde para todo vértice en la gráfica su ingrado es exactamente 1, excepto para la raíz, cuyo ingrado es 0.

– El que la gráfica subyacente es conexa es parte de las hipótesis, ya que exigimos que exista un camino no dirigido entre cualesquiera dos vértices, formado por aristas con rango k.

– Debemos demostrar que T_k es acíclica. Pero esto quedará demostrado si demostramos que el camino entre cualesquiera dos vértices es un camino orientado. Supongamos que no; entonces en el camino no dirigido existirá un vértice v tal que dos de sus aristas orientadas hacia él tienen rango k, como se ve en la Figura 8.35.

Figura 8.35: Contradicción por la inexistencia de un camino dirigido

Pero por el Lema 8.4 no puede haber más de una arista orientada hacia v con rango k, $\therefore$ la orientación de las aristas constituye un camino orientado.

– Como existe un camino orientado u⇝v, por el Lema 8.9, los vértices de este camino se encontrarán en la pila auxiliar en el orden dado por el camino. Por el Lema 8.12 ningún vértice se repetirá en esta sucesión, de donde no podrá haber ningún ciclo.

– Como no puede haber más de una arista con rango k incidiendo en cada uno de los vértices, el ingrado de cada vértice es 1. Veamos por qué esto se cumple para todos los vértices menos 1. Consideraremos los valores posibles para k:

· Para $k = 1$, establecimos en el Lema 8.18 que T_1 es un árbol generador, y por lo tanto un árbol donde todos los vértices menos s tienen ingrado 1, y s tiene ingrado 0.

· Para $k > 1$, T_k no contiene a s, ya que todas las aristas que se orientan *desde* s tienen rango 1. En algún momento debemos colocar en la pila auxiliar a un vértice desde el cual sale una arista de rango k, pero es la primera arista de rango k en ese camino que se está formando. El vértice que es el centro de acción en ese momento, no tiene ninguna arista de rango k orientada hacia él, por lo que su ingrado en T_k es 0. Este vértice no podrá aparecer nuevamente en la pila sin que se haya quitado su primera referencia, por el Lema 8.12, por lo que permanecerá con ingrado 0 en T_k. ❑

Lo que tenemos con el recorrido SFS es la construcción de k bosques, cada uno de ellos maximal. En el Lema 8.19 estableceremos, de hecho, que en cada subgráfica $G^k = (V, E^k)$, con $E^k = \{\text{e} \in E \mid \text{e.rango} = k\}$, toda componente conexa es un árbol dirigido. Pasamos a establecer que G^k es maximal.

Lema 8.20 *La gráfica G^i es maximal.*

Demostración:

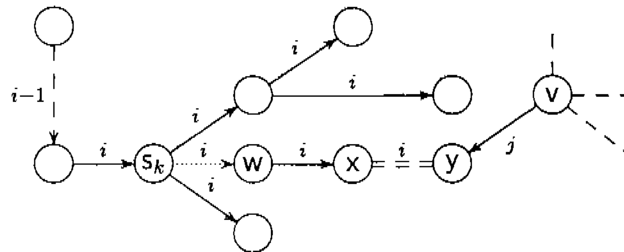
Por contradicción, supongamos que G^i no es maximal, esto es que hay alguna arista que debió quedar en G^i pero no fue así.

Supongamos que tenemos una arista e = x—y de rango i que debió agregarse a G^i.

Ya establecimos – Lema 8.19 – que cada componente conexa de G^i es un árbol dirigido T_i. Por ello, no puede ser que tanto x como y se encuentren en la misma componente conexa T de G^i, porque como T_i es un árbol, al agregarle la arista e se crea un ciclo y T_i deja de ser un árbol en G^i.

Supongamos que x y y están en componentes conexas distintas de G^i, como se muestra en la Figura 8.36, con la arista x—y con rango $j = i$.

Figura 8.36: Existencia de arista orientada que debió ser incluida



Si ambas componentes son árboles en G^i, no hay manera de orientar a la arista e. Si e se orienta x → y, y tendrá dos aristas incidentes en él de rango i, contrario a lo que se estableció en el Lema 8.4. Tampoco puede estar orientada y → x, porque entonces es el vértice x el que tiene más de una arista incidente en él de rango i.

Supongamos que uno de los vértices de e está en una componente conexa de G^i y el otro no, y sin pérdida de generalidad supongamos que x $\in T_i$. La arista x—y no puede estar orientada x→y, porque si lo estuviera querría decir que se exploró a x antes que a y, por lo que en el momento en que se exploró a x, este vértice se encontraba en el tope de la pila. No se puede quitar a x de la pila hasta entonces no se usen todas las salidas disponibles de rango $\geq i$, por lo que antes de quitar a esta referencia de x de la pila debe suceder que e es la salida disponible desde x de mayor rango, por lo que e se tuvo que agregar a esa componente de T_i.

e no puede estar orientada y → x porque entonces x tendría más de una arista orientada hacia x de rango i, contraviniendo el Lema 8.4.

Supongamos ahora que ni x ni y están en G^i. La argumentación que sigue es simétrica para ambos vértices, aunque trabajemos con x. Supongamos que e está orientada x→y, por lo que x fue descubierto antes que y. Por el Lema 8.13, hay un camino dirigido desde s hasta x donde todas las aristas tienen rango 1. Por ello, x va a ser colocado en el tope de la pila al menos una vez, habiendo sido alcanzado por una arista de rango 1. Si hasta ese momento no se había recorrido la arista x—y, antes de terminar de explorar a x se tiene que hacer, y cuando se hace se habrá incluido tanto a x como a y en G^i, contrario a lo que supusimos.

$\therefore \nexists$e tal que e.rango $= i$ y no esté incluida ya en algún árbol de G^i. ❑

Notemos que G_π (construido como producto colateral del ExploracionBasica) no coincide con ninguna de las subgráficas G^k, a menos que G sea un árbol y entonces $G_1 = G = G_\pi$. Esto se debe a que en G_π vamos colocando a las aristas por las cuáles se descubre a cada vértice, esto es, la primera que se usa que tiene como destino a ese vértice. En ExploracionSFS esta arista corresponde a la de rango más alto de entre las que quedan orientadas hacia el vértice. Esto implica que G_π contiene al menos a una arista de cada uno de los árboles de cada G^k, y por lo tanto G_π puede verse como una especie de hipergráfica de G respecto a los árboles de las G^k.

Lema 8.21 *Siempre que se coloca a un vértice* v *en el tope de la pila auxiliar, la arista por la que se le alcanzó (o descubrió) es la de mayor rango disponible de entre las orientadas hacia* v.

Demostración:

La demostración será por inducción en el número de aristas orientadas hacia v y el orden en que se van recorriendo.

Base: Si el número de aristas orientadas hacia v es 1, su rango debe ser 1. Como ExploracionSFS recorre todas las aristas de la componente conexa, la única manera de alcanzar a v es recorriendo esta arista, por lo que esta arista será recorrida y cumple con ser la de mayor rango disponible (la única) orientada hacia v.

Inducción: Supongamos como hipótesis de inducción que cuando ExploracionSFS recorre las primeras $k-1$ aristas orientadas hacia un vértice lo hace recorriendo siempre la de mayor rango disponible, y veamos que pasa cuando alcanza al vértice por k-ésima vez.

Si v tiene k aristas orientadas hacia él, los rangos son los enteros consecutivos $1 \ldots k$. Por la hipótesis de inducción, las primeras $k-1$ veces que se le alcanzó se hizo por las aristas de rangos $k \ldots 2$, en orden descendente. Por lo que la última vez que se le alcanza, la única arista disponible orientada hacia v es de rango 1, que cumple con ser la de más alto rango disponible. ❏

Corolario 8.22 *Todo vértice es descubierto por la arista de más alto rango de entre las que quedan orientadas hacia él.*

Corolario 8.23 G_π *consiste de las aristas de más alto rango de entre las que quedan orientadas hacia cada uno de los vértices.*

8.5. Complejidad de SFS

Es claro que la complejidad de ExploracionSFS es la misma que la de ExploracionBasica, excepto posiblemente por el costo que tenga el preproceso de las aristas para asignarles el rango, y el costo involucrado en el manejo de la pila auxiliar.

Respecto a la reorganización (orientación y asignación de rango) de las aristas, el método que se eligió para ordenarlas tiene complejidad de orden lineal con el número de aristas, ya que el número máximo de cubetas está predeterminado por el máximo grado de los vértices; este ordenamiento es posible en un tiempo lineal, donde nos lleva tiempo constante decidir el rango que le corresponde a la arista, dos unidades de tiempo el quitarla de una lista y meterla en la cubeta, y otra unidad de tiempo sacarla de la cubeta y agregarla a la lista correspondiente. Esto para cada arista, por lo que la evaluación y reorganización de las aristas lleva tiempo lineal en el número total de aristas.

En cuanto al manejo de la pila auxiliar, podemos utilizar complejidad amortizada para calcular el número total de veces que vamos a colocar, consultar y botar referencias de ella.

El número de veces que vamos a colocar a una referencia en la pila es, exactamente, el número de aristas que queden orientadas hacia ese vértice, más el costo de colocar a s en la pila auxiliar.

Averiguar cuál es la arista de mayor rango tiene un costo constante, ya que únicamente se consulta el rango de la arista al frente de la lista. Se hará una consulta de estas cada vez que la referencia al vértice se encuentre en el tope de la pila auxiliar, y esto sucederá a lo más, el número de aristas incidentes en el vértice (una vez por cada ocasión que se desee extender el camino desde ese vértice).

Finalmente, el número de veces que se va a sacar a una referencia de la pila auxiliar tiene que ver con el número de aristas que quedan orientadas hacia el vértice, ya que esta operación se efectuará cada vez que se haya llegado al vértice y ya no se pueda extender el camino. Pero una vez quitado el vértice de la pila auxiliar, si se le vuelve a colocar allí es porque se llegó a él recorriendo otra arista que no había sido recorrida.

El acceso a los vértices de la frontera lleva tiempo constante, ya que mantenemos una referencia en el vértice a la posición que ocupa en la frontera. Por lo tanto, elegir el siguiente centro de acción, una vez que queda determinado el tope de la pila auxiliar, lleva tiempo constante para cada vértice.

De todo lo anterior, el costo amortizado del manejo de la pila auxiliar es lineal con el número total de aristas en la gráfica.

Determinado este costo, podemos decir que el costo de esta implementación de ExploracionSFS es el mismo que para el ExploracionBasica:

Complejidad de ExploracionSFS: $O(|V| + |E|)$

Pasamos ahora a ver aplicaciones importantes de este algoritmo. Trabajaremos en el contexto de conexidad en gráficas[1], ya que el algoritmos SFS nos proporciona un mecanismo para trabajar con esta propiedad.

8.6. Aplicaciones de SFS

Resulta interesante que el algoritmo ExploracionSFS nos proporcione un mecanismo para determinar propiedades de conexidad local, global y S-mixta (las definiciones de estos términos y sus propiedades se encuentran en el Apéndice C). En una red de computadoras, como Internet, podemos representar a los procesadores con vértices y a las líneas de comunicación con aristas. Nos interesa la pregunta ¿seguirá habiendo comunicación entre cualquier pareja de vértices (o línea de comunicación para transmitir información) aun cuando la red sufra desperfectos? ¿Cuál es el umbral para dichos desperfectos? En el contexto de esta pregunta, la conexidad S-mixta nos permite definir un subconjunto de procesadores que podrán jugar un papel más importante que otros; por ejemplo, si tenemos en nuestra red procesadores menos confiables, podemos organizar la red de tal manera que esos procesadores no sean vértices de corte en la gráfica; esto es, que no pase más de un camino por ellos, de tal manera que al desconectarse de la red no afecten a nadie más que a sí mismos.

[1]En el Apéndice C presentamos los principales conceptos y propiedades de conexidad en gráficas, que serán utilizados para las aplicaciones de esta especialización.

8.6.1. Certificados delgados de k-conexidad

Se sabe que toda gráfica $G = (V, E)$ $k-$conexa (por vértices o aristas) contiene una *subgráfica generadora* $G' = (V, E')$, $E' \subseteq E$, tal que G' es $k-$conexa y $|E| = O(k \cdot |V|)$. Desafortunadamente, encontrar G' con el mínimo número de aristas es un problema NP-completo para cualquier $k \geq 2$ fija (para $k = 2$ se reduce al problema del ciclo hamiltoniano[2]). Aunque encontrar la G' mínima no es realizable, sí es posible encontrar una buena aproximación, que se conoce como un *certificado de k-conexidad*:

Definición 8.1 (Certificado de k-conexidad por aristas) Una subgráfica $G' = (V, E')$ es un certificado de k-conexidad por aristas de una gráfica $G = (V, E)$ k-conexa si cumple que:

i. $E' \subseteq E$
ii. G' es k-conexa por aristas, esto es, $\forall \mathsf{u}, \mathsf{v} \in V, \mathsf{u} \neq \mathsf{v}$ existen al menos k caminos ajenos por aristas entre u y v.

En el Teorema 8.24 estableceremos la existencia de este certificado de manera constructiva.

Teorema 8.24 *Sea $G = (V, E)$ una gráfica $k-$conexa por aristas. Entonces G contiene una subgráfica (generadora) $G' = (V, E')$ $k-$conexa por aristas con $|E'| \leq k \cdot |V|$.*

Demostración:

Construiremos G' de la siguiente manera:

$F_1 = (V, E_1)$ un árbol generador de G, es decir, una subgráfica generadora acíclica maximal.
$F_2 = (V, E_2)$ una subgráfica acíclica maximal (un bosque que consiste de un árbol generador para cada componente conexa) de $G - E_1$.
$F_3 = (V, E_3)$ una subgráfica acíclica maximal de $G - (E_1 \cup E_2)$.
......
$F_k = (V, E_k)$ una subgráfica acíclica maximal de $G - (E_1 \cup E_2 \cup \ldots \cup E_{k-1})$.

Es importante notar que F_j pudiera ser vacío a partir de cierta $j \leq k$, pero al menos $F_1 \neq \varnothing$.

Debemos establecer que $G' = (V, E' = E_1 \cup E_2 \cup \ldots \cup E_k)$ es la subgráfica buscada. Veamos por qué sí:

1. $|E'| \leq k(|V| - 1)$ porque cada una de las F_i, con $1 \leq i \leq k$ es acíclica – Teorema E.2 inciso ii del Apéndice E.
2. Veremos que G' es $k-$conexa por aristas. Supongamos, por contradicción, que no lo es. Entonces G' contiene un corte de aristas $U = \{\mathsf{e}_1, \ldots \mathsf{e}_p\}$, con $p \leq k - 1$ – ver Figura 8.37 – que separa a G' en dos componentes S_1 y S_2.

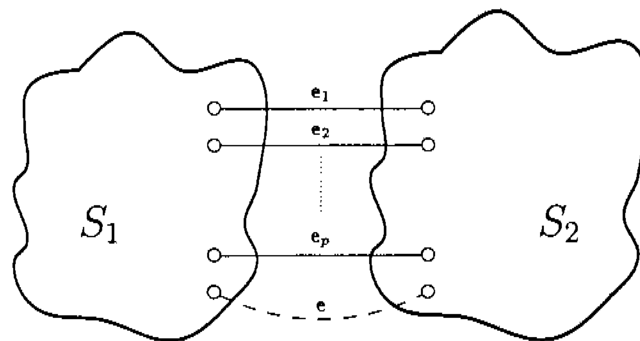
   Veamos cómo están distribuidas las aristas de este corte en los distintos F_i. F_1 contiene al menos a una de las aristas del corte, porque si no fuera así, como F_1 es maximal y las aristas del corte cruzan entre S_1 y S_2, si no hay ninguna arista del corte en F_1 le podríamos agregar al menos una de ellas a F_1 sin que se cerrara un ciclo. De esto,

[2] Garey & Johnson

F_1 contiene al menos a una de las aristas del corte. Es posible que F_1 contenga a más de una arista del corte, pero al menos contiene a una. Si al terminar de construir a F_1 éste no contiene a todas las aristas del corte, F_2 deberá contener al menos una, por un razonamiento similar al que acabamos de hacer. Si razonamos igual para cada uno de los F_i's, podemos notar que el bosque F_k no incluye aristas de U, ya que $|U| \leq k-1$, y cada uno de los F_i, $1 \leq i \leq m \leq p \leq k-1$ debieron incluir al menos a una de las aristas de U.

Pero como G es $k-$conexa, U no es un corte de G – ningún conjunto con menos de k aristas puede ser un corte de G. Esto quiere decir que hay al menos una arista más entre S_1 y S_2 en G, digamos e, que no está en G' y por lo tanto tampoco en F_k. Pero como F_k no contiene aristas de U, entonces no hay un camino de S_1 a S_2 en F_k.

Figura 8.37: Corte U que separa a S_1 y S_2



Por lo tanto, si añadimos e a F_k no se forma un ciclo, lo que contradice que F_k sea maximal, y con ello el que G' no sea k-conexa.

$\therefore$ G' es un certificado delgado de k-conexidad de G. ❏

Construir un certificado delgado de k-conexidad por vértices no es tan directo, aunque si es de manera similar. Estableceremos su existencia en el Teorema 8.25

Teorema 8.25 *Sea $G = (V, E)$ una gráfica $k-$conexa (por vértices). Entonces G contiene una subgráfica (generadora) $G' = (V, E')$ $k-$conexa (por vértices) con $|E'| \leq k \cdot |V|$.*

Demostración:

Construiremos G' de la siguiente manera:

$F_1 = (V, E_1)$ un árbol generador de G, es decir, una gráfica generadora acíclica maximal. Al construir a F_1, cada vez que llegamos a un vértice orientamos a todas las aristas que no han sido incluidas ya en F_1 desde el vértice en el que se está. Es importante notar que si bien no todas las aristas están forzosamente en F_1, como F_1 es un árbol generador, todas las aristas de G quedaron orientadas, aunque por supuesto no todas quedaran incluidas en F_1. Llamemos $\vec{G}$ a G con sus aristas orientadas.

$F_2 = (V, E_2)$ una gráfica acíclica maximal (un bosque que consiste de un árbol generador para cada componente conexa) de $\vec{G} - E_1$.

$F_3 = (V, E_3)$ una gráfica acíclica maximal de $\vec{G} - (E_1 \cup E_2)$.

......

$F_k = (V, E_k)$ una gráfica acíclica maximal de $\vec{G} - (E_1 \cup E_2 \cup \ldots \cup E_{k-1})$.

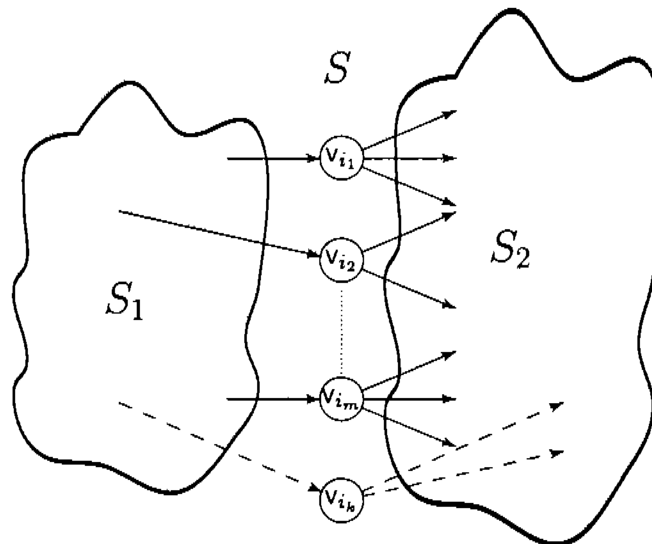
$G' = (V, E' = E_1 \cup E_2 \cup \ldots \cup E_k)$.

De manera similar a la demostración del Teorema 8.24, debemos notar que a partir de cierta i, las E_i's pudieran estar vacías.

Debemos establecer ahora que G' es un certificado de k-conexidad (por vértices) para G.

Por contradicción, supongamos que no, y supongamos que tenemos un corte de vértice S para G', $S = \{v_{i_1}, v_{i_2}, \ldots, v_{i_m}\}$, con $m \leq k-1$, que particiona a G' en dos conjuntos, S_1 y S_2, como se muestra en la Figura 8.38.

Figura 8.38: Corte S de vértices que separa a S_1 y S_2



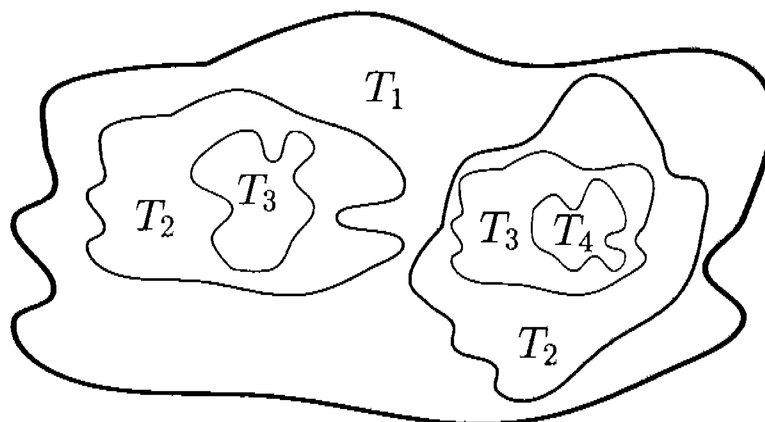
Cada uno de los F_i son bosques, y las aristas quedaron orientados desde S_1 hacia S_2 al construir a F_1. Estamos suponiendo que en los vértices de S inciden todas las aristas que comunican a S_1 con S_2 en G'. Por un razonamiento similar al del Teorema 8.24, cada conjunto F_j de los primeros r que se construyen, $1 \leq j \leq r \leq m \leq k-1$, contiene al menos a una de estas aristas que va de S_1 al vértice y a una de las que van del vértice a S_2, y F_k no contiene a ningún camino que pase por alguno de estos vértices. Pero S no es un corte (de vértices) para G, ya que G es k-conexa por vértices. Por lo que existe al menos un vértice adicional v_{i_k} tal que hay un camino de S_1 a S_2 que pasa por v_{i_k}. Observemos únicamente a una arista que llega desde S_1 al vértice v_{i_k} y a una que sale desde v_{i_k} hacia S_2 (al menos hay una de cada una). Estas dos aristas no están en G' o en F_k. Pero como ninguna de las aristas de F_k está orientada desde S_2 hacia S_1, al agregarle a F_k estas aristas no se forma un ciclo, contradiciendo con esto la maximalidad de F_k, y por lo tanto, la existencia de un corte de $m < k$ vértices en G'. ❑

De las demostraciones de los Teoremas 8.24 y 8.25 obtenemos que es posible construir una $G' = (V, E')$ $k-$conexa, $E' \subseteq E$, $|E'| \leq k \cdot |V|$ en tiempo $O(k \cdot |V| + |E|)$, corriendo k veces

el algoritmo genérico de exploración para encontrar los k bosques generadores, eliminando en cada corrida las aristas del bosque que se genere, y asignando a las subgráficas el índice k cuando se trate de la k-ésima ejecución. Esta gráfica G' corresponde a un certificado delgado de $k-$conexidad por aristas (o por vértices, si se orientan a las aristas de la manera indicada).

A todas luces éste es un algoritmo muy ineficiente. Como ya ha de intuir el lector, el algoritmo ExploracionSFS, con complejidad lineal como ya demostramos, obtiene un certificado de k-conexidad, ya que los bosques que construye, como ya también demostramos, cumplen con las propiedades dadas en la Definición 8.1 y en el Teorema 8.24, quedando los árboles generadores como se muestra esquemáticamente en la Figura 8.39.

Figura 8.39: Construcción de los árboles para el certificado de $k-$conexidad



La diferencia entre la construcción dada en la demostración de los teoremas y la dada en el algoritmo ExploracionSFS es únicamente el que la construcción de los árboles para las distintas k's se va "mezclando", mientras que en la demostración del teorema se construyen de manera exclusiva para cada k. Dado que se trata de una gráfica no dirigida, el resultado deberá ser el mismo.

Queremos insistir en la ventaja de obtener los certificados delgados de k-conexidad con este algoritmo, dada su eficiencia. Si lo que se desea es el certificado únicamente para determinados valores de k, todo lo que se hace es eliminar las aristas que queden con un rango mayor a k. Dado que el algoritmo es lineal, el costo asociado a obtener todo el certificado no es mayor.

8.7. Conclusiones

En este capítulo establecimos al algoritmo SFS como una especialización del algoritmo genérico de exploración. Una vez más tenemos un algoritmo con un proceso local, que selecciona el siguiente vértice para procesar en términos de las aristas disponibles para el centro de acción actual. La extensión del camino que se va formando es una decisión que depende

exclusivamente de las aristas incidentes desde el centro de acción actual.

El trabajo adicional en esta especialización tiene que ver con la evaluación de las aristas y el reordenamiento para que sean tomadas en el orden dado por la evaluación. Esto se logra agregando atributos a la super clase, y extendiendo los métodos que descubren un vértice. Estas modificaciones están perfectamente localizadas y encapsuladas, lo que hace más sencilla la estructura del algoritmo. También, como las modificaciones están perfectamente localizadas, es poco el trabajo que hay que hacer para demostrar que la exploración es correcta y calcular su complejidad. La demostración de las propiedades adicionales de la especialización se apoya en (hereda de) las propiedades establecidas para la exploración genérica.

Parte III

Ejemplos Adicionales

Capítulo 9

Otras familias de algoritmos genéricos

En este capítulo presentamos otras tres familias de algoritmos genéricos, donde los algoritmos de cada familia comparten una estrategia genérica.

En el caso de la familia de árboles generadores establecen una estrategia de elegir consecutivamente aristas que no cierren un ciclo, y si se trata de árboles generadores de peso mínimo, el orden para elegir las aristas candidato es el que determina el peso del árbol generador.

La familia de apareamiento exacto de texto tiene la estrategia genérica de preprocesar el patrón, para realizar el proceso de apareamiento usando el resultado de este preproceso. Las garantías que da es que en cada iteración avanza sobre la cadena de texto y que no pierde ninguna ocurrencia del patrón en el texto.

Finalmente, la familia de flujo en redes comparte una estrategia genérica de iterar, buscando en cada iteración aumentar el flujo obtenido hasta ahora, hasta que no sea posible hacerlo. Para ello, utiliza dos subestrategias principales, la de trayectorias aumentantes y la de preflujo.

Presentamos estas tres familias para subrayar el hecho de que un enfoque genérico es posible en una gran cantidad de problemas, y que este tipo de abstracción facilita la comprensión de los algoritmos específicos y la demostración de correctez de los mismos.

9.1. Árboles generadores

Un *árbol generador* de una gráfica $G = (V, E)$ es una subgráfica $G'(V, E')$, con $E' \subseteq E$, que es un árbol. Es de *peso mínimo* si la suma de los pesos de las aristas del árbol es tan pequeña como es posible[1].

Cuando revisamos la parte correspondiente a exploración en gráficas, $G_\pi = (V, E_\pi)$ corresponde a un árbol generador de $G = (V, E)$. Nos ocupan en esta ocasión árboles generadores

[1] En inglés, *Minimum spanning trees*, MST.

de peso mínimo[2].

La solución burda para este problema sería el obtener todos los posibles árboles generadores y elegir uno de peso mínimo. Pero esta solución es, obviamente, sumamente onerosa ya que el número de árboles generadores es el el número de subconjuntos de $n-1$ aristas, donde n es el número de vértices en la gráfica.

En [CLRS01, pág. 563] se presenta un algoritmo genérico para árboles generadores de peso mínimo, el cuál tomamos como modelo para nuestro algoritmo genérico para árboles generadores. Este algoritmo genérico usa una estrategia glotona para construir el árbol generador, escogiendo en cada iteración la arista que en ese momento menos peso aporta al árbol que se está construyendo y que mantiene la propiedad de ser un árbol para la subgráfica construida. Además de las especializaciones para el algoritmo de Prim y el de Kruskal para construir árboles generadores de peso mínimo, incluimos bajo esta abstracción los algoritmos DFS, BFS y de distancias de Dijkstra. En el caso de DFS y BFS se construyen árboles generadores, y si consideramos que el peso de las aristas es uniforme, los árboles construidos son de peso mínimo. En el caso del algoritmo de Dijkstra para distancias mínimas, también se construye un árbol generador, pero en este caso el peso mínimo se refiere a la suma de las distancias a cada uno de los vértices de la gráfica.

9.1.1. Algoritmo genérico para la construcción de árboles generadores

Como mencionamos, el algoritmo genérico para árboles generadores, GenericoDeArbolesGeneradores, va eligiendo aristas de la gráfica original con un cierto criterio, que garantiza que el peso que aportan es tan pequeño como es posible, y verificando que la arista elegida, al agregarla al árbol en construcción, no provoque que la gráfica que se conforma deje de ser un árbol. La clase abstracta se puede ver en el Listado 9.1.

Listado 9.1: Algoritmo genérico para construir árboles generadores. 1/2

```
1   public abstract class ArbolGeneradorEnAbstracto {
2     /* Gráfica a ser procesada.                          */
3   protected Grafica grafica;
4     /* Árbol generador a construirse.                    */
5   protected Grafica arbol;
6     /* Para recolectar las aristas del árbol.            */
7   protected Aristas[] aristasArbol;
8     /* Para recolectar los vértices del árbol.           */
9   protected Nodos[] nodosArbol;
10    /* Número de nodos en la gráfica.                    */
11  protected int N;
```

[2] Puede haber más de un árbol generador con el mismo peso, y que este peso sea mínimo.

Listado 9.1: Algoritmo genérico para construir árboles generadores. 2/2

```
12       /* Número de aristas en la gráfica.                                  */
13     protected int M;
14       /* Para contar el número de aristas agregadas al árbol.             */
15     protected int aristasAgregadas = 0;
16       /* Constructor del ejemplar. Inicializa el árbol, n y m.            */
17     public ArbolGeneradorEnAbstracto(Grafica g) {
18         grafica = g;
19         N = grafica.getN();
20         M = grafica.getM();
21         aristasArbol = new Aristas[N + 1];
22         nodosArbol = new Nodos[N + 1];
23     }
24       /* Estrategia genérica para la construcción de árboles             *
25        * generadores.                                                      */
26     public final Grafica genericoDeArbolGenerador() {
27         Aristas e;
28         int cont = 0;
29         while (!esArbolGenerador()) {
30             e = obtenAristaSegura();
31             if (e != null) {
32                 agregaAlArbol(e);
33             }
34         }
35         armaNodosYAristas(aristasArbol, nodosArbol);
36         arbol = grafica.construyeGraf(grafica, aristasArbol,
37                                        nodosArbol);
38         return arbol;
39     }
40       /* Determina si ya termina el proceso.                              */
41     protected abstract boolean esArbolGenerador();
42       /* Devuelve una arista segura que no cierra un ciclo.               */
43     protected abstract Aristas obtenAristaSegura();
44       /* Agrega una arista y modifica el árbol que se construye.          */
45     protected void agregaAlArbol(Aristas e) {
46         aristasAgregadas++;
47         e.marcaEnArbol();
48     }
49       /* Manipula los arreglos para que se construya la gráfica.          */
50     protected void armaNodosYAristas(Aristas[] aristasArbol,
51                                       Nodos[] nodosArbol) {
       ...
99     }
100 }
```

Correctez del algoritmo genérico

La única propiedad que podemos extraer a este nivel de definición es que el algoritmo debe construir, en efecto, un árbol generador. Esto es sencillo de demostrar si se supone que el método que entrega una arista segura es correcto, y que el método que avisa cuando ya se terminó de construir el árbol también es correcto.

Complejidad del algoritmo genérico

La complejidad de este algoritmo se puede definir con la siguiente fórmula:

$$\begin{aligned}\text{Complejidad de ArbolGeneradorEnAbstracto} &= f_{\text{constr}}(n,m)+\\ &+\sum_{m}\left(f_{\text{aristaSegura}}(n,m)+f_{\text{agregaArbol}}(n,m)\right)\end{aligned} \quad (9.1)$$

En este nivel abstracto no contamos con estimaciones para ninguno de los tiempos, ya que su costo dependerá de la especialización. A este nivel de abstracción se tienen dos especializaciones que quedan todavía a nivel abstracto, dependiendo de si el árbol generador es de una gráfica con pesos heterogéneos en las aristas o no. Examinemos primero la familia de algoritmos para árboles generadores de gráficas con pesos heterogéneos en sus aristas. La especialización que debe producir un árbol con características respecto al peso trabaja con una cola de prioridades, donde los elementos de esa cola, con su respectiva llave, pueden ser vértices o aristas, dependiendo del algoritmo particular. En el caso de los árboles generadores sin restricciones de peso, el algoritmo utiliza una estructura más genérica para almacenar a los vértices y de ahí elegir el siguiente a procesar. Estas clases abstractas se encuentran respectivamente en los Listados 9.2 y 9.3.

Listado 9.2: Superclase para árboles generadores con peso 1/2

```
 1 public abstract class AGPMEnAbstracto
 2                     extends ArbolGeneradorEnAbstracto {
 3     /* Cola de prioridades para elegir de ahí a los elementos        *
 4      * (arista/nodo).                                                */
 5     protected HeapBinMin objetosProcesados;
 6     /* Para encontrar relación de orden entre objetos.              */
 7     protected ComparadorEnGrafica comp;
 8     /* Constructor.                                                  */
 9     public AGPMEnAbstracto(Grafica g) {
10         super(g);
11         if (!(g instanceof GraficaConPesos))
12           throw new IllegalArgumentException("No se puede "
13             + "construir un AGPM si no es Q una GraficaConPesos");
14         GraficaConPesos wG = (GraficaConPesos) g;
15         cnstryEstrctrsAuxiliares(wG);
16         llenaHeapBin(wG);
17     }
```

Listado 9.2: Superclase para árboles generadores con peso 2/2

```
18      /* Estructuras que determinan el orden en el que los        *
19       * objetos van a ser seleccionados.                          */
20     protected abstract void cnstryEstrctrsAuxiliares(
21                         GraficaConPesos g);
22      /* Llena la cola de prioridades para determinar el orden.    */
23     protected abstract void llenaHeapBin(GraficaConPesos g);
24      /* Determina si ya se tiene un árbol generador.              */
25     protected boolean esArbolGenerador() {
26         if (aristasAgregadas >= N - 1)
27             return true;
28         return objetosProcesados.esVacia();
29     }
30 }
```

Listado 9.3: Superclase para árboles generadores con pesos homogéneos 1/2

```
 1 public abstract class ArbolGeneradorSinPesoAbstracto
 2             extends ArbolGeneradorEnAbstracto {
 3      /* Para que se considere a todos los vértices, no nada más    *
 4       * a los de la componente conexa del origen.                   */
 5     protected MiListaLigada nodosDisponibles;
 6      /*Para almacenar los nodos mientras son procesados.           */
 7     protected Frontera estructNodos;
 8      /* Las referencias de los nodos hacia la lista de nodos       *
 9       * disponibles.                                                */
10     protected ElementoEnMiLista[] refLista;
11      /* Constructor. Inicia la gráfica a generar y el árbol que    *
12       * se va a construir, así como la lista de nodos disponibles  */
13     public ArbolGeneradorSinPesoAbstracto(Grafica grafica) {
14         super(grafica);
15         refLista = new ElementoEnMiLista[N + 1];
16         nodosDisponibles = new MiListaLigada();
17         for (int i = 1; i <= N; i++) {
18             Nodos v = grafica.getNodo(i);
19             refLista[i] = (ElementoEnMiLista)
20                                 nodosDisponibles.agrega(v);
21         }
22     }
23      /* Determina si ya se terminó de construir el árbol. Si es    *
24       * una sola componente conexa, esto sucede si se incorporan   *
25       * al árbol n-1 aristas. Si no, si ya no quedan vértices      *
26       * sin procesar.                                               */
27     protected boolean esArbolGenerador() {
28         if (aristasAgregadas >= N - 1)
29             return true;
```

Listado 9.3: Superclase para árboles generadores con pesos homogéneos 2/2

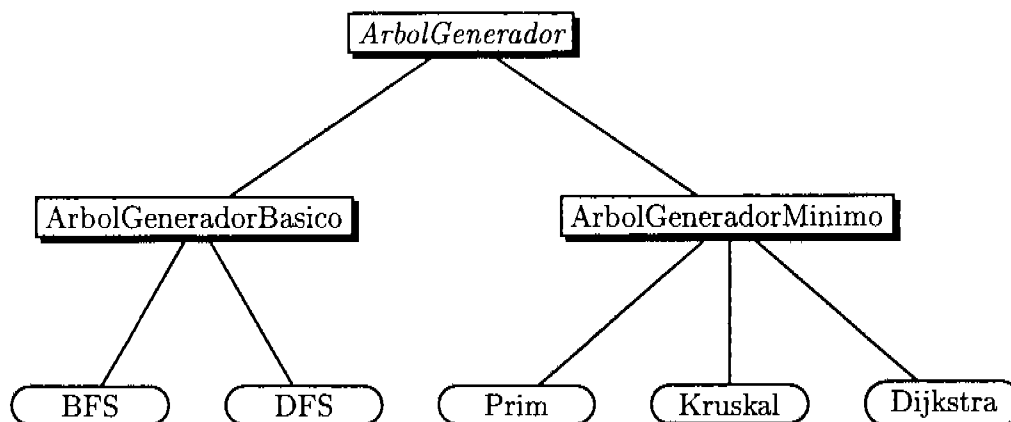
```
30          if (!estructNodos.esNoVacia()) {
31              if (nodosDisponibles.estaVacia())
32                  return true;
33              Nodos v = (Nodos) ((ElementoEnMiLista) nodosDisponibles.
34                                          getPrimero()).getInfo();
35              nodosDisponibles.elimina((ElementoEnMiLista)
36                                      nodosDisponibles.getPrimero());
37              estructNodos.agregaNodo(v);
38              refLista[v.getPos()] = null;
39              return false;
40          }
41          return false;
42      }
43      /* Regresa una arista segura, que es la primera que llega      *
44       * al vértice.                                                 */
45      protected Aristas obtenAristaSegura() {
46          while (estructNodos.esNoVacia()) {
47              Nodos vDesde = estructNodos.selecciona();
48              while (vDesde.masAristas()) {
49                  Aristas e = vDesde.obtenArista();
50                  if (e.usada())                    continue;
51                  Nodos vHacia = e.obtenElOtroNodo(vDesde);
52                  if (vHacia.yaFueAlcanzado())      continue;
53                  procesaPrimeroAlcanzado(vDesde, e, vHacia);
54                  if (refLista[vHacia.getPos()] != null)
55                      nodosDisponibles.
56                             elimina(refLista[vHacia.getPos()]);
57                  e.setUsada();
58                  return e;
59              }
60              terminaNodo(vDesde);
61          }
62          return null;
63      }
64  }
```

En la Figura 9.1 podemos ver el esquema jerárquico de esta familia de algoritmos.

9.1.2. Árboles generadores de gráficas con pesos heterogéneos en sus aristas

En esta especialización, la elección de la siguiente arista segura tiene que hacerse buscando que el peso que se acumule en el árbol sea el mínimo posible. Tradicionalmente se tiene dos algoritmos que construyen el árbol generador de peso mínimo, refiriéndose este peso a la suma de los pesos de las aristas, el algoritmo de Prim y el de Kruskal. Se tiene que

Figura 9.1: Jerarquía de las clases para Árboles Generadores



agregar como propiedad de este nivel abstracto que el árbol obtenido es de peso mínimo. Las especializaciones estarán encargadas de demostrar que en efecto es así.

La especialización para el algoritmo de Prim

Para esta especialización se ingresa a todos los vértices en una cola de prioridades, donde la llave de cada vértice es el peso de la arista con la cual se le alcanza. Al iniciar el proceso, todos los vértices tienen llave con valor ∞, por lo que cualquiera de ellos se va a encontrar al frente de la cola. Se va tomando al vértice que se encuentre al frente de la cola y se usan todas las aristas incidentes en él que no hayan sido usadas previamente. A los vértices destino de cada una de estas aristas se le anota como llave el mínimo entre la llave que tenían y el peso de la arista que los está alcanzando. Es un algoritmo glotón porque en cada momento elige al vértice con llave menor, suponiendo que esta "mejor elección" a corto plazo va a resultar en una mejor elección definitiva. Y así es, como se puede demostrar – no entramos en esta demostración por falta de espacio, pero puede ser consultada, entre otros, en [CLRS01]. Esta especialización se presenta en el Listado 9.4.

Listado 9.4: Especialización de Prim para obtener árboles generadores de peso mínimo 1/3

```
1 public class AGPMPrim
2          extends AGPMEnAbstracto {
3     /* Registra el peso de la arista a través de la cuál se      *
4      * alcanzó al i-ésimo nodo.                                  */
5     private AristaLigera[] nodosLigeros;
6     /* Constructor. Simplemente llama a la superclase.           */
7     public AGPMPrim(Grafica g) {
8         super(g);
9     }
```

Listado 9.4: Especialización de Prim para obtener árboles generadores de peso mínimo 2/3

```
10      /* Construye las estructuras auxiliares para los nodos,          *
11       * registrando su valor inicial como ∞.                           */
12    protected void cnstryEstrctrsAuxiliares(GraficaConPesos g) {
13        nodosLigeros = new AristaLigera[N + 1];
14        for (int i = 1; i <= N; i++)
15            nodosLigeros[i] = new AristaLigera(g.getNodo(i),
16                                              INFINITO, null);
17    }
18      /* Puebla la cola de prioridades con todos los nodos, con         *
19       * su "peso" registrado como INFINITO.                             */
20    protected void llenaHeapBin(GraficaConPesos g) {
21        comp = new ComparadorAristasLigeras(nodosLigeros);
22        objetosProcesados = new HeapBinMin(g.getN(), comp);
23        for (int i = 1; i <= N; i++) {
24            Nodos v = g.getNodo(i);
25            v.setPosCola((ElementoEnMiLista)
26                            objetosProcesados.agregaObjeto(v));
27        }
28    }
29      /* Elige a la arista ligera por la que se llegó al nodo en        *
30       * el frente de la cola de prioridades como candidato, y          *
31       * verifica que no cierre un ciclo en lo que se lleva             *
32       * construido del árbol.                                           */
33    protected Aristas obtenAristaSegura() {
34        Nodos v = (NodoBasico) (objetosProcesados.daPrimero());
35        if (nodosLigeros[v.getPos()].getPeso() == INFINITO) {
36            // El nodo es una nueva raíz.
37            nodosLigeros[v.getPos()].setPeso(0);
38        }
39        Aristas aristaSegura = nodosLigeros[v.getPos()].obtenArista();
40        objetosProcesados.borraElemento(v.getPosCola());
41        ElementoListaDeAristas actual =
42                    (ElementoListaDeAristas) v.getSigArista();
43        Nodos desde, hacia;
44        while (actual != null) {
45            boolean usada = true;
46            AristaBasica e = null;
47            while (usada && actual != null) {
48                e = (AristaBasica) actual.obtenArista();
49                usada = e.usada();
50                if (e.usada())
51                    actual = actual.getNext();
52            }
53            if (actual == null)
54                return aristaSegura;
55              // Instala a la arista orientada desde→hacia.
56            desde = v;
57            e.orienta(desde, hacia);
```

Listado 9.4: Especialización de Prim para obtener árboles generadores de peso mínimo 3/3

```
58                 e.setUsada();
59                 if (e.getPeso() <
60                         nodosLigeros[hacia.getPos()].getPeso()) {
61                     Aristas te = hacia.getAristaEnArbol();
62                     hacia.setD(desde.getD() + e.getPeso());
63                     if (te != null)
64                         hacia.setD(hacia.getD() - te.getPeso());
65                     nodosLigeros[hacia.getPos()].setPeso(e.getPeso());
66
67                     nodosLigeros[hacia.getPos()].setEdge(e);
68                     hacia.setPi(desde);
69                     hacia.setAristaEnArbol(e);
70                     objetosProcesados.reduceLlave
71                         ((ElementoEnMiLista) hacia.getPosCola());
72                 }
73                 actual = actual.getNext();
74             }
75             return aristaSegura;
76         }
77           /* Determina si el árbol generador ya está completo.              */
78         protected boolean esArbolGenerador() {
79             return objetosProcesados.estaVacia();
80         }
81 }
```

El método obtenAristaSegura debe devolver una arista segura y con peso mínimo de entre las disponibles para ese vértice, además de actualizar las llaves para todos los vértices alcanzados desde el vértice al frente de la cola. Se demuestra que este criterio en efecto construye un árbol generador de peso mínimo.

Correctez de la especialización de Prim

Como ya mencionamos, es fácil demostrar que esta especialización cumple con los predicados pedidos en la clase abstracta. No lo hacemos acá por falta de espacio.

Complejidad de la especialización de Prim

Sustituyendo en la fórmula (9.1) que dimos para el algoritmo genérico, el método obtenAristaSegura es el más costoso ya que tiene que mantener la cola de prioridades ordenada. Trabaja de manera similar al algoritmo de Dijkstra para caminos más cortos, con la diferencia de que el valor de la llave en este caso es el peso de la arista con menor peso y no la distancia. El método trabaja con las m aristas, por lo que el término $O(m \log n)$ es el que domina si se

utilizan heaps binarios[3] (como en el caso de esta especialización). Sin embargo, si el número de vértices es relativamente pequeño comparado con el número de aristas, es conveniente utilizar heaps de Fibonacci[4], obteniendo entonces una complejidad de $O(m + n \log n)$.

Especialización de Kruskal

El algoritmo de Kruskal obtiene también un árbol generador de peso mínimo. En la cola de prioridades este algoritmo forma a todas las aristas ordenadas de acuerdo a su peso, y al iniciarse el algoritmo presenta a los vértices de la gráfica formando un bosque, donde cada vértice es raíz de su propio árbol (trivial). Para agregar una arista al árbol generador verifica que los extremos de la arista estén en árboles distintos, para garantizar que la arista que se está agregando no cierra un ciclo; si es así, la incluye; si no, la descarta. Como las aristas se van eligiendo de acuerdo a su peso, eso garantiza que el árbol construido es de peso mínimo. Los árboles se representan como conjuntos ajenos, de tal forma de que sea relativamente eficiente determinar el árbol al que pertenece un vértice, y si dos vértices están o no en un mismo árbol. En el Listado 9.5 se encuentra esta especialización.

Listado 9.5: Especialización de Kruskal para árboles generadores de peso mínimo 1/3

```
 1 public class AGPMKruskal    extends AGPMEnAbstracto {
 2       /* Raíces de los subárboles donde cada vértice es una raíz.    */
 3     private ElementoBosque[] raices;
 4       /* Constructor.                                                 */
 5     public AGPMKruskal(Grafica g) {
 6         super(g);
 7     }
 8       /* Construye el bosque con cada vértice en su propio árbol.     *
 9        * Utiliza representación de conjuntos ajenos.                  */
10     protected void cnstryEstrctrsAuxiliares(GraficaConPesos g) {
11         raices = new ElementoBosque[N + 1];
12         for (int i = 1; i <= N; i++) {
13             Nodos v = g.getNodo(i);
14             raices[i] = new ElementoBosque(v);
15             raices[i].setTamanho(0);
16             raices[i].setRango(0);
17             raices[i].setV(v);
18             raices[i].setCabeza(new ElementoArbol(raices[i]));
19             raices[i].getCabeza().setPi(raices[i].getCabeza());
20         }
21     }
```

[3]Los heaps binarios se presentan en el contexto del ordenamiento por montículo, y se presentan y definen en el Apéndice E.

[4]Se define lo que son los heaps binomiales y de Fibonacci en el Apéndice E.

Listado 9.5: Especialización de Kruskal para árboles generadores de peso mínimo 2/3

```
22        /* Puebla la cola de prioridad con todas las aristas, donde  *
23         * las mismas van a ser devueltas en orden creciente de su   *
24         * peso.                                                     */
25      protected void llenaHeapBin(GraficaConPesos g) {
26          comp = new ComparadorPesosAristas();
27          objetosProcesados = new HeapBinMin(M, comp);
28          for (int i = 1; i <=M; i++) {
29              Aristas e = g.obtenArista(i);
30              ElementoEnMiLista he = (ElementoEnMiLista)
31                              objetosProcesados.agregaObjeto(e);
32              e.setPosCola(he);
33          }
34      }
35        /* Devuelve una arista segura verificando que integre a dos  *
36         * conjuntos ajenos. Si no es así, quiere decir que cierra   *
37         * un ciclo.                                                 */
38      protected Aristas obtenAristaSegura() {
39          Aristas e;
40          Nodos vDesde, vHacia;
41          int iDesde, iHacia, iRepDesde, iRepHacia;
42          while (!objetosProcesados.estaVacia()) {
43              for (int i = 1; i <= objetosProcesados.size(); i++) {
44                  ElementoEnMiLista eml = (ElementoEnMiLista)
45                                      objetosProcesados.get(i);
46                  Aristas ed = (Aristas) eml.getInfo();
47              }
48              e = (AristaBasica) objetosProcesados.daPrimero();
49              e.setUsada();
50              objetosProcesados.borraElemento(e.getPosCola());
51              vDesde = e.getDesde();
52              vHacia = e.getHacia();
53              iDesde = vDesde.getPos();
54              iHacia = vHacia.getPos();
55              ElementoArbol repDesde = raices[iDesde].
56                                      getCabeza().getRepr();
57              ElementoArbol repHacia = raices[iHacia].
58                                      getCabeza().getRepr();
59              iRepDesde = repDesde.getVert().getPos();
60              iRepHacia   = repHacia.getVert().getPos();
61              if (iRepDesde != iRepHacia)
62                  return e;
63          }
64          return null;
65      }
```

Listado 9.5: Especialización de Kruskal para árboles generadores de peso mínimo 3/3

```
66      /* Agrega al árbol haciendo la unión de conjuntos ajenos.   */
67      protected void agregaAlArbol(Aristas e) {
68          super.agregaAlArbol(e);
69          Nodos x = e.getDesde();
70          Nodos y = e.getHacia();
71          int xi = x.getPos();
72          int yi = y.getPos();
73          ElementoArbol rX = raices[xi].getCabeza();
74          ElementoArbol rY = raices[yi].getCabeza();
75          raices[xi].union(rX, rY);
76      }
77  }
```

Correctez de la especialización de Kruskal

Es fácil corroborar, apoyados en el código, que la arista que se elige es segura y que se agrega al bosque de manera adecuada. Respecto a que el árbol es de peso mínimo, se argumenta con el hecho de que se van eligiendo las aristas en orden ascendente respecto a su peso.

Complejidad de la especialización de Kruskal

En implementaciones tradicionales del algoritmo de Kruskal se ordena al conjunto de aristas antes de entrar al ciclo de elegir aristas seguras. En nuestra implementación, en cambio, las mantenemos en una cola de prioridades, y vamos obteniendo la de menor peso. En realidad, con nuestra implementación podemos ahorrar algunos pasos ya que pudiéramos obtener las $n-1$ aristas que requerimos sin tener que revisar las m aristas, a lo que está obligado el ordenamiento. Ambas operaciones tienen un costo de $O(m \log m) = O(m \log n)$ si se usa un algoritmo eficiente para ordenamiento como el ordenamiento de montículo (heapsort). Si se tiene una implementación eficiente de conjuntos ajenos, el método agregaAlArbol tiene costo constante.

Especialización para árboles generadores de caminos más cortos

Para esta especialización colocamos en la cola de prioridades a los vértices de la gráfica con una estimación de su distancia al origen, que al empezar es ∞. Se empieza con cualquiera de los vértices al frente de la cola de prioridades, y se actualiza la estimación para todos aquellos vértices que se encuentren en el otro extremo del vértice elegido. Conforme se actualiza la estimación, se reorganiza la cola de prioridades. Cada vértice registra cuál es la arista que le da el menor costo. Cuando un vértice llega al frente de la cola, la arista elegida para verificar si es arista segura es la que dio al vértice el menor costo. La especialización para este algoritmo se puede ver en el Listado 9.6.

Listado 9.6: Especialización para árboles generadores de caminos más cortos 1/2

```
 1 public class AGPMDijkstra extends AGPMEnAbstracto {
 2        /* Constructor.                                                  */
 3     public AGPMDijkstra(Grafica g) {
 4         super(g);
 5     }
 6       /* No requiere de estructuras auxiliares, además de la           *
 7        * cola de prioridades.                                          */
 8     protected void cnstryEstrctrsAuxiliares(
 9                               GraficaConPesos g) { }
10       /* Llena la cola de prioridades con los nodos, con una           *
11        * distancia inicial de ∞.                                       */
12     protected void llenaHeapBin(GraficaConPesos g) {
13         comp = new ComparadorDeNodos();
14         objetosProcesados = new HeapBinMin(g.getN(), comp);
15         for (int i = 1; i <= N; i++) {
16             Nodos v = g.getNodo(i);
17             v.setPosCola((ElementoEnMiLista)
18                          objetosProcesados.agregaObjeto(v));
19         }
20     }
21       /* Devuelve una arista "segura", esto es, que no cierra un      *
22        * ciclo en lo que se lleva construido.                          */
23     protected Aristas obtenAristaSegura() {
24         Nodos v = (Nodos) objetosProcesados.daPrimero();
25         ElementoEnMiLista hel = (ElementoEnMiLista) v.getPosCola();
26         Aristas e = null;
27         if (v.getD() == INFINITO)
28             v.setD(0);
29         if (v.masAristas())
30             e = v.obtenArista();
31         else
32             objetosProcesados.borraElemento(hel);
33         if (e != null) {
34             Nodos vHacia;
35             if (e.getDesde() == v) {
36                 vHacia = e.getHacia();
37             } else {
38                 e.setHacia(e.getDesde());
39                 e.setDesde(v);
40                 vHacia = e.getHacia();
41             }
42             e.setUsada();
43             if (v.getD() + e.getPeso() < vHacia.getD()) {
44                 procesaLlaveReducida(v, e, vHacia);
45                 objetosProcesados.reduceLlave(hel);
46                 return e;
47             }
48         }
```

Listado 9.6: Especialización para árboles generadores de caminos más cortos 2/2

```
49             return null;
50         }
51           /* Determina si ya se construyó el árbol generador.          */
52         protected boolean esArbolGenerador() {
53             return objetosProcesados.esVacia();
54         }
55           /* Modifica la cola de prioridades si se reduce una llave.   */
56         protected void procesaLlaveReducida(Nodos v, Aristas e,
57                                                   Nodos vHacia) {
58             if (vHacia.getAristaEnArbol() != null) {
59                 vHacia.getAristaEnArbol().setSaleDeArbolGenrdr();
60                 aristasAgregadas--;
61             }
62             vHacia.setPi(v);
63             vHacia.setAristaEnArbol(e);
64             vHacia.setD(v.getD() + e.getPeso());
65         }
66 }
```

Correctez de la especialización de Dijkstra

Es fácil verificar – ver Capítulo 7 – que con este proceso se produce un árbol generador de caminos más cortos. El método obtenAristaSegura hace precisamente esto, pues a cada vértice hay asociada una y sólo una arista, la que completa el camino más corto a ese vértice desde el origen.

Complejidad para la especialización de Dijkstra

El mantenimiento de la cola de prioridades en este caso tiene un costo de $O(m \log n)$, ya que se tienen que revisar todas las aristas para determinar el camino más corto a cada vértice – se puede ver una demostración más precisa de esto en el Capítulo 7.

9.1.3. Árboles generadores con peso homogéneo en las aristas

En el Capítulo 3, al revisar exploración en gráficas vimos que un producto colateral de esta exploración es la construcción de un árbol generador de la gráfica. Revisamos en ese momento las especializaciones ExploracionDFS y ExploracionBFS. En esta sección construiremos estos árboles como una especialización de la clase abstracta que construye árboles generadores en base a ir eligiendo aristas seguras.

Especialización DFS

Esta especialización va colocando a los vértices en una pila conforme los va alcanzando. La pila se inicia con el vértice origen, y se coloca en ella a alguno de los vértices adyacentes al que se encuentra en el tope de la pila. Conforme se van eligiendo aristas, se descartan; pero si la arista es tal que el vértice es alcanzado por primera vez con ella, entonces es una arista segura y se incluye en el árbol. La especialización para este algoritmo se encuentra en el Listado 9.7. Adicionalmente a la construcción del árbol generador, esta especialización presenta las características usuales de un recorrido DFS.

Listado 9.7: Especialización para árbol generador DFS

```
 1 public class ArbolgeneradorDFS
 2                 extends ArbolGeneradorSinPesoAbstracto  {
 3      /* Para registrar las marcas de tiempo de DFS.                  */
 4     protected MarcasDeTiempo[]  marcasDeTiempo;
 5      /* Reloj discreto para marcar hora de apertura y clausura       *
 6       * del vértice.                                                 */
 7     protected int              reloj = 0;
 8      /* Constructor. Inicia la gráfica y el árbol, así como la       *
 9       * pila desde donde se selecciona al siguiente vértice.         */
10     public ArbolgeneradorDFS(Grafica grafica) {
11         super(grafica);
12         marcasDeTiempo = new MarcasDeTiempo[N + 1];
13         for (int i = 1; i <= N; i++) {
14             marcasDeTiempo[i] =
15                         new MarcasDeTiempo(grafica.getNodo(i));
16         }
17         estructNodos = new FronteraEnPila();
18         Nodos s = grafica.getNodo(1);
19         estructNodos.agregaNodo(s);
20         s.setAlcanzado();
21         s.setD(0);
22         s.setPi(null);
23     }
24      /* Hace el registro cuando un nodo es alcanzado por primera    *
25       * vez.                                                         */
26     protected void procesaPrimeroAlcanzado(Nodos vDesde,
27                                     Aristas e, Nodos vHacia) {
28         super.procesaPrimeroAlcanzado(vDesde, e, vHacia);
29         marcasDeTiempo[vHacia.getPos()].setAbre(++reloj);
30     }
31      /* Hace el registro de la clausura de un vértice.              */
32     protected void terminaNodo(Nodos vDesde) {
33         super.terminaNodo(vDesde);
34         marcasDeTiempo[vDesde.getPos()].setCierra(++reloj);
35     }
36 }
```

Correctez para la especialización de DFS

En cada invocación a obtenAristaSegura, el método recorre todas las aristas necesarias hasta que encuentre a una que alcance a algún vértice por primera vez, y devuelve a ésta. Por lo tanto, se va a construir el árbol generador de la componente conexa en la que se encuentra el primer vértice con que se inicia la pila – para construir el bosque generador hay que garantizar que todos los vértices llegan a la pila, lo que no se hace en esta especialización.

Es fácil ver que las propiedades de las marcas de tiempo expuestas en el Capítulo 5 también se presentan en esta especialización.

Complejidad para la especialización de DFS

Dado que para elegir las aristas del árbol generador el método obtenAristaSegura debe revisar todas las aristas, la complejidad agregada de este método es de $O(m)$ por la revisión de cada arista, más $O(n)$ para determinar cuando un vértice ya no tiene aristas disponibles. El costo agregado de incluir a una arista en el árbol es de $O(m)$, por lo que esta especialización tiene una complejidad de $O(n+m)$, como vimos en el capítulo correspondiente a ExploracionDFS.

Listado 9.8: Especialización para árboles generadores BFS

```
1 public class ArbolGeneradorBFS
2                 extends ArbolGeneradorSinPesoAbstracto {
3     /* Constructor. Inicia la gráfica y el árbol. Instala una  *
4      * cola para seleccionar al siguiente vértice.              */
5     public ArbolGeneradorBFS(Grafica grafica) {
6         super(grafica);
7         estructNodos = new FronteraEnCola();
8         Nodos s = grafica.getNodo(1);
9         estructNodos.agregaNodo(s);
10        s.setAlcanzado();
11        s.setD(0);
12        s.setPi(null);
13        s.setAristaEnArbol(null);
14        int i = s.getPos();
15        nodosArbol[i].setAlcanzado();
16        nodosArbol[i].setD(0);
17        nodosArbol[i].setPi(null);
18        nodosArbol[i].setAristaEnArbol(null);
19    }
20 }
```

9.1.4. Especialización BFS

La única diferencia entre esta especialización y la que corresponde a DFS es que la estructura en la que se acomodan los vértices es una cola en lugar de una pila, así que lo único que trae la especialización es la construcción de la estructura frontera como una cola. El resto de los métodos hereda la implementación de la superclase inmediata. El código para la especialización se encuentra en el Listado 9.8.

Correctez para la especialización de DFS

Como la arista que entrega el método obtenAristaSegura es siempre la primera que llega a un vértice, cada vértice tiene ingrado 1, y sólo el vértice origen tiene ingrado 0, lo que corresponde a una de las definiciones que tenemos para lo que es un árbol. Asimismo, todos los vértices que pertenecen a la componente conexa del vértice origen van a ser incorporados al árbol – lo que se demuestra en detalle en el Capítulo 4.

Complejidad para la especialización de BFS

Al igual que para la especialización de árboles DFS, y por los mismos argumentos, esta especialización tiene una complejidad de $O(n+m)$.

9.1.5. Relación entre los dos algoritmos abstractos que producen árboles generadores

Claramente, no es posible comparar la familia de algoritmos de exploración con la de árboles generadores a nivel del código en las clases abstractas, ya que la organización de estas dos familias es muy distinta, y aun así, deben producir el mismo resultado en cuanto a los árboles generadores. La manera como podríamos hacer una comparación de estas familias es a través del *estado* de las estructuras de datos durante la ejecución de las especializaciones respectivas. Podríamos utilizar un método similar al dado en [Lam02], en donde los sistemas se analizan en términos de un lenguaje específico que describe estados del sistema. Si lo vemos desde este punto de vista, podemos observar que se usan las mismas estructuras de datos en las especializaciones que se corresponden, y que dichas estructuras pasan por los mismos estados. De esta observación podemos concluir que podemos tener varias abstracciones para una familia dada de algoritmos.

9.2. Apareamiento de cadenas de texto

Empezaremos por definir de manera precisa lo que entendemos por una cadena (de texto):

Definición 9.1 (Cadena) Una *cadena* es una sucesión de símbolos (carácteres) que ocupan posiciones consecutivas de la posición 1 a la n, donde n es el tamaño de la cadena, el número de símbolos presentes en la misma.

El problema de apareamiento exacto de cadenas de texto[5] se puede expresar de la siguiente manera:

Definición 9.2 (Apareamiento de cadenas) Sea patron una cadena, a la que identificamos con un *patrón* de carácteres, y texto otra cadena a la que identificamos con un *texto*. El apareamiento de cadenas consiste en localizar todas las apariciones de patron en texto.

El apareamiento de cadenas ha tenido siempre mucha importancia. Su uso en editores de texto para encontrar y/o sustituir cadenas en un archivo está siempre presente. En la última década, con los trabajos en ADN y el proceso de este código en forma de cadenas de símbolos (letras), el apareamiento de cadenas ha cobrado mucha importancia. También, por supuesto, el desarrollo de la *web*[6] requiere de apareamiento de cadenas en sus búsquedas.

En esta sección trabajaremos con *apareamiento exacto*, que no permite ninguna diferencia entre patron y su aparición en texto. Asimismo, supondremos que el patrón a buscar es relativamente pequeño respecto al tamaño del texto en el que se le va a buscar, además de que el texto puede ir variando con el tiempo.

9.2.1. El algoritmo genérico para apareamiento de cadenas

Al diseñar un algoritmo genérico para apareamiento de cadenas tomamos en cuenta los métodos más conocidos para tal fin, como lo son el de Boyer-Moore, el Knuth-Morris-Pratt, y aun el método ingenuo que compara, alineando el patrón con cada posición del texto, los símbolos en patron[p] contra los de texto[p]. En este tipo de algoritmos reconocemos los siguientes métodos:

preproceso	Construye una representación del patrón que va a facilitar y hacer más eficiente el apareamiento.
compara	Se encarga de comparar el patrón contra el texto, símbolo por símbolo, hasta que haya una diferencia o se termine el patrón.
recorre	Se encarga de recorrer el patrón a la derecha, para continuar la búsqueda de otra presencia.
apareamiento	Organiza el apareamiento, estableciendo una estrategia genérica.

En el Listado 9.9 se muestra el código para la clase abstracta que define el apareamiento genérico. Como lo hemos hecho hasta ahora, no mostramos aquellos métodos que sirven para depurar o para mostrar resultados.

[5]En adelante, simplemente "apareamiento de cadenas".
[6]No encontramos una traducción adecuada; consideramos *telaraña* y *maraña*, pero ninguna de estas dos nos pareció adecuada.

Listado 9.9: Estrategia genérica para el apareamiento de cadenas 1/2

```
1 public abstract class ApareamientoAbstracto {
2       /* Arreglo que contiene el patrón a buscar.                    */
3     protected char[] patron;
4       /* Texto donde se va a buscar el patrón.                       */
5     protected char[] texto;
6       /* Apunta a la posición en el patrón que será la siguiente     *
7        * a comparar.                                                 */
8     protected int p;
9       /* Apunta a la posición en el texto que será la siguiente      *
10       * a comparar.                                                 */
11    protected int t;
12      /* Tamaño del patrón.                                          */
13    protected int m;
14      /* Tamaño del texto.                                           */
15    protected int n;
16      /* Listas de posiciones en texto para reportar el apa-         *
17       * reamiento.                                                  */
18    protected ListaDePosiciones listPos = null;
19      /* Proporciona un constructor por omisión, para herencia.      */
20    public ApareamientoAbstracto() {
21    }
22      /* Constructor. Copia los argumentos a las variables           *
23       * propias de la clase.                                        */
24    public ApareamientoAbstracto(char[] patron, char[] texto) {
25        m = patron.length - 1;
26        n = texto.length - 1;
27        if (m > n) {
28            throw new IllegalArgumentException("No hay proceso.");
29        this.patron = patron;
30        this.texto = texto;
31    }
32      /* Determina si queda suficiente texto sin procesar para       *
33       * intentar nuevamente el apareamiento. El resultado indica    *
34       * si se realiza una nueva iteración o no. Usa directamente    *
35       * los atributos de la clase.                                  */
36    protected final boolean masTextoPorAparear(int t) {
37        return t <= texto.length - patron.length + 1;
38    }
39      /* Preproceso del patrón.                                      */
40    protected abstract void preprocesaPatron();
41      /* Compara símbolo por símbolo el patrón contra el texto.      *
42       * Si llega al final del patrón, reporta éxito.                */
43    protected abstract int compara(RefInt refT, RefInt refP);
44      /* Recorre el patrón hacia la derecha, para alinearse con      *
45       * el texto e iniciar un nuevo apareamiento.                   */
46    protected abstract int recorre(RefInt refP, RefInt refT);
```

Listado 9.9: Estrategia genérica para el apareamiento de cadenas. 2/2

```
47      /* Estrategia genérica para el apareamiento exacto de ca-   *
48       * denas. Usa directamente los atributos de la clase. Re-    *
49       * gresa una lista con las posiciones en las que el patrón  *
50       * aparece en el texto.                                      */
51     public final ListaDePosiciones apareamientoGenerico() {
52         listPos = new ListaDePosiciones();
53         preprocesaPatron();
54         p = 1;
55         t = 1;
56           /* Para pasar a t por referencia.                       */
57         RefInt refT = new RefInt(t);
58           /* Para pasar a p por referencia.                       */
59         RefInt refP = new RefInt(p);
60         while (masTextoPorAparear(refT)) {
61             t = refT.intValue();    // Actualizamos a t.
62             if (compara(refT, refP))
63                 reporta(refT.intValue());
64             t += recorre(refP, refT);
65             refT.setValue(t); // Actualizamos a refT.
66         } // aparear texto con patrón
67         return listPos;
68     }
69 }
```

Correctez de apareamientoGenerico

Es claro que la propiedad que debe presentar todo apareamiento exacto de cadenas es el que localice todas y cada una de las apariciones de patronen texto. Para ello, y dado que apareamientoGenerico trabaja en una iteración, deberemos establecer invariantes para el ciclo. El teorema general de correctez es el Teorema 9.1.

Teorema 9.1 *Al invocar al método apareamientoGenerico desde alguna subclase concreta de ApareamientoAbstracto; si este método trabaja a partir de dos arreglos de símbolos,* patron *y* texto*; y si cada uno de los métodos invocados desde él es correcto, entonces:*

i. *apareamientoGenerico siempre termina.*

ii. *En la lista de posiciones construida en este proceso se encuentran todas y cada una de las posiciones en* texto *donde aparece* patron.

Antes de demostrar este teorema, definamos qué queremos decir con declarar "correcto" a cada uno de los métodos invocados desde apareamientoGenerico.

Correctez del constructor. Queremos que el constructor deje listos los arreglos con el patrón y el texto. Lo expresamos a continuación en forma de postcondiciones para el método.

El constructor ApareamientoAbstracto(patron, texto) siempre termina. (9.2a)

$1 \leq m = |patron| < \infty.$ (9.2b)

$m \leq n = |texto| < \infty.$ (9.2c)

$listPos = \varnothing.$ (9.2d)

$t = 1.$ (9.2e)

Correctez de preprocesaPatron. Prepara el patrón para su adecuado apareamiento. Su correctez es la que permite a recorre hacer los desplazamientos correctos del patrón, por lo que su correctez estará en términos de recorre. Sin embargo, podemos pedir como postcondición mínima que deje bien preparado el proceso. Suponemos como precondiciones las postcondiciones del constructor.

El método preprocesaPatron siempre termina. (9.3a)

$(t = 1) \wedge (p = 1).$ (9.3b)

La postcondición (9.3b) exige que el proceso inicie con T apuntando al primer carácter de texto y P apuntando al primer carácter de patron.

Correctez de compara. Compara carácter por carácter, a partir de las posiciones alineadas del patrón y el texto. Como está inmerso en un ciclo, exigiremos de este método que en cada iteración cumpla con pre y postcondiciones.

Precondiciones: Para que este método se ejecute correctamente requerimos que:

$(n - t + 1) \geq m$ (9.4a)

texto[t] está alineado con patron[1] (9.4b)

$1 \leq p \leq m$ (9.4c)

Postcondiciones: Al terminar la ejecución de compara para esta iteración, el siguiente predicado es válido:

El método compara siempre termina. (9.4d)

Además, si denotamos con t' al valor de t cuando este método es invocado, y con t al valor de la misma variable cuando compara termina, entonces

$t' = t;$ (9.4e)

compara regresa VERDADERO $\iff$ $\text{texto}[\text{t} \ldots \text{t} + \text{m} - 1] = \text{patron}[1 \ldots \text{m}].$ (9.4f)

En las precondiciones estamos requiriendo que el texto no se acabe antes que el patrón (postcondición (9.4a); que la primera posición del patrón esté alineada con la posición actual del texto (precondición (9.4b); y que P esté apuntando a una posición válida del patrón. Como postcondiciones pedimos que no se mueva al apuntador de texto y que se regrese un valor de verdadero si y sólo si en la posición t empieza una aparición completa de patron.

Correctez del método recorre. Este método es el encargado de reubicar al patrón, para que se inicie nuevamente la comparación. Debe recorrer al patrón al menos un lugar a la derecha, y si lo recorre más lugares, no debe brincar ninguna aparición de patron en texto.

Se suponen como precondiciones a las postcondiciones de compara. Para las postcondiciones identificamos con t al valor de t antes de la ejecución de recorre y con t' al valor que presenta t al terminar esta ejecución. Entonces, las postcondiciones quedan como sigue:

El método recorre siempre termina. (9.5a)

$t < t'$. (9.5b)

$\nexists\, i,\ t < i < t'$, tal que $\mathsf{texto}[\mathsf{t}+\mathsf{i}\ldots\mathsf{t}+\mathsf{i}+\mathsf{m}-1] = \mathsf{patron}[1\ldots\mathsf{m}]$. (9.5c)

Pensamos en cada uno de estos métodos como operaciones primitivas. Las implementaciones particulares de estos métodos resultan en especializaciones que representan a los algoritmos de apareamiento exacto de cadenas más conocidos.

Estamos ya listos para demostrar el Teorema 9.1.

Demostración (Correctez de apareamientoGenerico):

Inciso 9.1.*i* : Para demostrar que apareamientoGenerico siempre termina, basta con demostrar que el constructor termina y que el método masTextoPorAparear eventualmente va a evaluarse a FALSO. Esto es claro, veamos por qué.

- Estamos exigiendo al constructor que siempre termine – postcondición (9.2a).
- El método masTextoPorAparear es un método final, lo que indica que ninguna especialización lo puede redefinir. Como estamos garantizando que todos los métodos invocados en el ciclo terminan, este método será evaluado más de una vez, con los valores de t dejados ahí por recorrer, o por el constructor la primera vez. Se exige al método compara que no modifique el valor de t – postcondición (9.4e), y dado que la postconidición (9.5b) de recorre exige que el valor de t se incremente en al menos una unidad en cada iteración, eventualmente se cumplirá que $t > n - m + 1$, y este método regresará el valor FALSO.

Inciso 9.1.*ii* : Si el método compara regresa VERDADERO cada vez que hay un apareo de patron con texto a partir de la posición t en texto – suponemos que es así – y si recorre no mueve a t brincando el principio de una aparición de patron – también es una de nuestras hipótesis – entonces en la línea [63] del Listado 9.1, apareamientoGenerico ingresará a la lista a todas y cada una de las apariciones de texto en patron. ❏

Fórmula de complejidad para ApareamientoAbstracto

Aún cuando todavía no tengamos la implementación de los dos métodos principales, podemos presentar una fórmula para la complejidad de esta familia de algoritmos en la Fórmula (9.6), donde:

$\mathsf{f}_{\mathsf{preProc}}(\mathsf{m})$	Tiempo que cuesta preprocesar el patrón, para usarlo en el apareamiento y reducir la complejidad del algoritmo. Depende del tamaño de patron.
r	representa al número de veces que se invoca al método recorre. Depende de la manera en que haga el corrimiento del patrón.
n, m	Tamaño de texto y de patron respectivamente.
$\mathsf{f}_{\mathsf{constr}}(\mathsf{m},\mathsf{n})$:	Tiempo que consume el constructor de la especialización. Depende del tamaño de patron (m) y de texto (n).

$\mathsf{f}_{\mathsf{masTPA}}()$:	Tiempo que consume evaluar si hay más texto por procesar. Como es una comparación simple, el tiempo es constante – ver línea [37] del Listado 9.1.
$\mathsf{f}_{\mathsf{reporta}}(\mathsf{n})$:	Tiempo que toma reportar una posición en la que hay apareamiento. Está acotado por el tamaño del texto.
$\mathsf{f}_{\mathsf{compara}}(\mathsf{m})$:	Tiempo que consume una comparación de texto contra patron. Depende del tamaño del patrón.
$\mathsf{f}_{\mathsf{recorre}}(\mathsf{m})$:	El tiempo que le toma calcular el desplazamiento del patrón. Depende del tamaño del patrón.

Con estas aclaraciones, la fórmula para calcular la complejidad de los algoritmos de la familia ApareamientoAbstracto se muestra en la Ecuación (9.6).

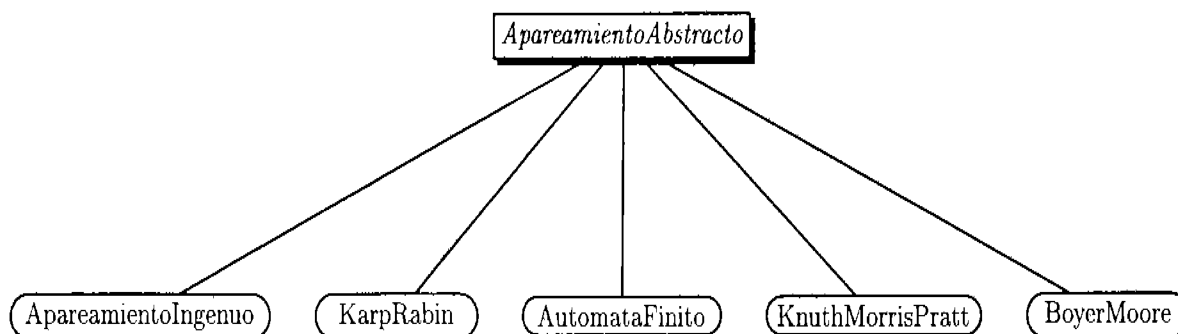
$$\begin{aligned}\text{Complejidad de ApareamientoAbstracto} &= \mathsf{f}_{\mathsf{constr}}(\mathsf{n},\mathsf{m}) + \mathsf{f}_{\mathsf{preProc}}(\mathsf{m}) + \\ &+ \sum_{r}\left(\mathsf{f}_{\mathsf{masTPA}}(\mathsf{n}) + \mathsf{f}_{\mathsf{compara}}(\mathsf{m}) + \mathsf{f}_{\mathsf{recorre}}(\mathsf{m})\right) \quad (9.6)\end{aligned}$$

donde ya sabemos los siguientes valores:

$\mathsf{f}_{\mathsf{constr}}(\mathsf{n},\mathsf{m})$	=	c_0	Lo que lleve colocar los apuntadores al texto y al patrón.
$\mathsf{f}_{\mathsf{masTPA}}(\mathsf{n})$	=	c_h	Consiste únicamente de una comparación de valores enteros.

El árbol jerárquico que corresponde a esta familia se encuentra en la Figura 9.2. Pasamos ahora a trabajar con las especializaciones.

Figura 9.2: Esquema Jerárquico para la familia de apareamiento de cadenas



9.2.2. La especialización para el apareamiento ingenuo

Esta especialización no preprocesa el patrón. Simplemente alinea el patrón sucesivamente con todos y cada uno de los carácteres en el texto, haciendo una comparación exhaustiva. El código para la especialización se encuentra en el Listado 9.10.

Listado 9.10: Apareamiento ingenuo de cadenas

```
 1 public class ApareamientoIngenuo
 2     extends ApareamientoAbstracto {
 3       /* Arma los arreglos para el apareamiento de cadenas.          */
 4     public ApareamientoIngenuo(char[] patron, char[] texto) {
 5         super(patron, texto);
 6     }
 7       /* No hay preproceso del patrón en esta especialización.      */
 8     protected void preprocesaPatron() {
 9     }
10       /* Compara carácter por carácter de patron contra texto        *
11        * alineado. Ejecuta a lo más m comparaciones.                 */
12     protected boolean compara(RefInt refT, RefInt refP) {
13         int n = texto.length - 1;
14         int m = patron.length - 1;
15         int p = 1;
16         int t = refT.intValue();
17         while (p <= m && texto[t + p - 1] == patron[p]) {
18             p++;
19         }
20         if (p > m) {
21             return true;
22         } else {
23             return false;
24         }
25     }
26       /* Recorre el patrón una posición a la derecha.                */
27     protected int recorre(RefInt refP, RefInt refT) {
28         return 1;
29     }
30 }
```

Como se puede observar en el Listado 9.10, no hay preproceso para el patrón; cada invocación de compara simplemente compara a las cadenas alineadas hasta terminar con éxito o encontrar una desavenencia, mientras que el método recorre simplemente recorre el patrón un lugar a la derecha. Es fácil ver que los tres métodos cumplen con los predicados dados para que el algoritmo genérico sea correcto. En cuanto a la fórmula de la complejidad, queda como sigue:

$$
\begin{aligned}
\text{Complejidad de ApareamientoIngenuo} &= c_0 + 0 + \sum_{n} \left(c_h + m + 1 \right) \\
&= c_0 + n(m + c_h + 1) \\
&= O(n \cdot m) \qquad (9.7)
\end{aligned}
$$

El método compara pudiera hacer menos de m comparaciones, dependiendo de la frecuencia con que el patrón aparezca en el texto. Por ejemplo, si el primer carácter del patrón no

aparece en el texto, el número de comparaciones sería 1 en cada invocación. m es entonces una cota superior para el número de comparaciones en cada invocación.

Este algoritmo es tan ineficiente como puede ser un algoritmo de apareamiento de cadenas, $O(n \cdot m)$.

La especialización para el algoritmo de Karp-Rabin

Este algoritmo utiliza una función de dispersión que aplica al patrón y, sucesivamente al subtexto que se está apareando con el patrón. La función tiene que ser tal que cada carácter pueda ser agregado o quitado del subtexto, independiente de los otros carácteres. Si se tiene una función de dispersión con estas características, cuando el patrón se recorre un lugar a la derecha se elimina el valor asignado al carácter que ya no está en el subtexto y se le agrega el valor del último carácter del subtexto que se incorpora en esta iteración.

Si el valor de la función de dispersión aplicada al subtexto es igual al valor que se obtuvo del patrón, el algoritmo procederá a comparar carácter por carácter del subtexto con el patrón. Esto último se tiene que hacer porque las funciones de dispersión en general presentan colisiones. El código correspondiente a esta especialización se muestra en el Listado 9.11.

Listado 9.11: Especialización para el algoritmo de Karp-Rabin 1/2

```
 1 public class ApareamientoKarpRabin
 2              extends ApareamientoAbstracto  {
 3     /* Para registrar el valor de dispersión (hash) de patron .    */
 4    protected int valPatron;
 5     /* Para calcular el valor de dispersión de texto que se        *
 6      * está comparando.                                            */
 7    protected int valTexto;
 8     /* Valor de dispersión del primer carácter en la porción de    *
 9      * texto a comparar.                                           */
10    protected int primerCar;
11     /* Parámetros para la función de dispersión.                   */
12    protected static final int Q = 31;
13    protected static final int D = 2;
14     /* Construye un apareador de cadenas que sigue el método       *
15      * de Karp-Rabin.                                              */
16    public ApareamientoKarpRabin(char[] patron, char[] texto) {
17        super(patron, texto);
18    }
19     /* Obtiene el valor de dispersión para patron.                 *
20      * Procesa al patrón y a los primeros m carácteres de          *
21      * texto.                                                      */
22    protected void preprocesaPatron(char[] patron)  {
23        int m = patron.length - 1;
24        int n = texto.length - 1;
```

Listado 9.11: Especialización para el algoritmo de Karp-Rabin 2/2

```
25         int d = 1;
26         if (n < m)
27             return;
28         valPatron = (int) patron[1] % Q;
29         valTexto = 0;
30         primerCar = 1;
31         for (int i = 2; i <= m; i++) {
32             primerCar = (primerCar * D) % Q;
33             valPatron = (valPatron * D + (int) patron[i]) % Q;
34             valTexto = (valTexto * D + (int) texto[i - 1]) % Q;
35         }
36     }
37       /* Calcula el nuevo valor de dispersión para la nueva          *
38        * porción de texto, restando el valor del carácter que ya    *
39        * no está en la porción de texto y sumando el valor del      *
40        * carácter que se acaba de agregar.                          */
41     protected boolean compara(RefInt refT, RefInt refP) {
42         int p = 1;
43         int t = refT.intValue();
44         valTexto = (valTexto * D + (int) texto[t + m - 1]) % Q;
45         if (valTexto == valPatron) {
46             while (p <= m && texto[t+p-1] == patron[p] )  p++;
47           /* Quita del valor de dispersión del texto el valor del    *
48            * primer carácter.                                        */
49         valTexto = (255 * Q + valTexto - (int) texto[t]
50                         * primerCar) % Q;
51         if (p > m)
52             return true;
53         else
54             return false;
55     }
56       /* Recorre al patrón un lugar a la derecha para iniciar un     *
57        * nuevo apareo.                                               */
58     protected int recorre(RefInt refP, RefInt refT) {
59         return 1;
60     }
61 }
```

Es fácil ver que la implementación de los métodos para esta especialización cumplen con los predicados requeridos para la correctez del algoritmo genérico. El método encargado de comparar va a dictaminar correctamente, ya que si la función de dispersión da el mismo valor que el del patrón se procederá a comparar carácter por carácter del patrón – es imposible que la misma cadena dé dos valores distintos. Y el método encargado de recorrer el patrón lo hace únicamente en una posición, por lo que no puede brincarse ninguna presencia del patrón.

Si hacemos un análisis de peor caso para este algoritmo, deberíamos considerar como peor caso aquél en el que la mayoría de las veces vamos a tener que hacer la comparación carácter por carácter entre el patrón y el texto, resultando la complejidad de esta especialización similar a la de la especialización ingenua, con el cálculo de la función de dispersión siendo trabajo adicional. La ecuación correspondiente al cálculo de la complejidad se encuentra en la Ecuación (9.8).

$$\begin{aligned} \textsf{Complejidad de ApareamientoKarpRabin} &= c_0 + c_1 \cdot m + \sum_n \Big(c_h + (m+1) + 1 \Big) \\ &= c_0 + c_1 \cdot m + n(m + c_h + 2) \\ &= O(n \cdot m) \end{aligned} \tag{9.8}$$

Sin embargo, muchas veces el número de posiciones donde el texto y el patrón se aparean no son muchas, digamos k. Entonces, el número total de veces que en el método **compara** las cadenas serían comparadas carácter por carácter sería k. Esto lo podemos expresar dividiendo la aportación de **compara** a la complejidad entre aquellas invocaciones donde la comparación se lleva a cabo y aquéllas donde no. Sin embargo, la comparación entre el patrón y el texto carácter por carácter también se va a llevar a cabo con los falsos positivos, esto es, cadenas distintas que produzcan el mismo valor de dispersión. Como estamos obteniendo el valor de dispersión módulo Q, la probabilidad de que esto suceda es de $1/Q$, por lo que probablemente el número de casos que se nos van a presentar son n/Q. De esto, requerimos tomar en cuenta esto. Entonces, el número de casos donde la comparación **no** se va a hacer es $O(n-k-n/Q)$. Sabemos que en esta especialización $r = n$, porque el patrón se recorre únicamente una posición en cada iteración. Partimos entonces los casos en la sumatoria de la siguiente manera:

$$\sum_n \Big(\mathsf{f}_{\mathsf{masTPA}}(\mathsf{n}) + \mathsf{f}_{\mathsf{compara}}(\mathsf{m}) + \mathsf{f}_{\mathsf{recorre}}(\mathsf{m}) \Big) =$$

$$= \sum_n \Big(\mathsf{f}_{\mathsf{masTPA}}(\mathsf{n}) + \mathsf{f}_{\mathsf{recorre}}(\mathsf{m}) \Big) + \sum_{n-k-n/Q} c_1 + \sum_{k+n/Q} c_2 \cdot m$$

Suponemos que k es pequeña relativa a n, por lo que

$$\sum_{n-k-n/Q} c_1 = O(n).$$

Para la segunda componente de esta suma, el tiempo se reduciría si $k = O(1)$, esto es, que no depende de m o n; y si elegimos $Q > m$. Entonces tendríamos que la complejidad de esta especialización es de $O(n+m)$, lo que la haría atractiva para este caso especial. Si no, entonces tendríamos una complejidad de $O(n \cdot m)$, lo que no la haría atractiva.

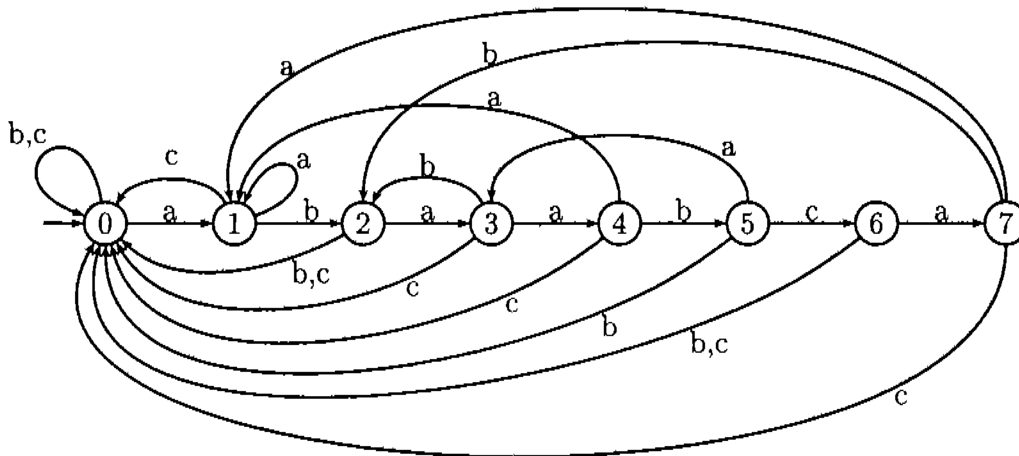
9.2.3. Especialización de apareamiento con autómatas finitos

Un autómata finito es una máquina aceptadora de cadenas, $M = (Q, \Sigma, \delta, q_0, A)$ donde Q es un conjunto de estados, Σ es el alfabeto del que se construyen las cadenas, δ es una

función de transición que determina a qué estado se transfiere el autómata, dependiendo del estado en el que se encuentra y el símbolo que lee; q_0 es el estado inicial y A corresponde a un subconjunto de Q, aquellos estados que si el autómata termina de leer en algún estado en A decimos que acepta a la cadena. Estos implementos han sido muy bien estudiados y poseen propiedades muy importantes para el apareamiento de texto. Para cada patrón P existe un AF que lo *reconoce*: M empieza en el estado q_0; se le alimenta P y nada más; el AF termina en un estado que acepta. Entonces, uno podría construir el autómata que reconozca al patrón dondequiera que se encuentre en el texto.

En este caso, el AF tiene ocho estados, uno por cada símbolo en el patrón, además del estado inicial. El AF iría del estado i al estado $i+1$ conforme va leyendo cada uno de los símbolos del patrón en el texto. Cuando el AF lee un símbolo que no coincide con el del patrón, entonces tiene que determinar si ya leyó algún prefijo del patrón. Por ejemplo, para el patrón abaabca, si la subcadena abaab ya fue apareada, pero se encuentra en el texto con algo que no es c, que

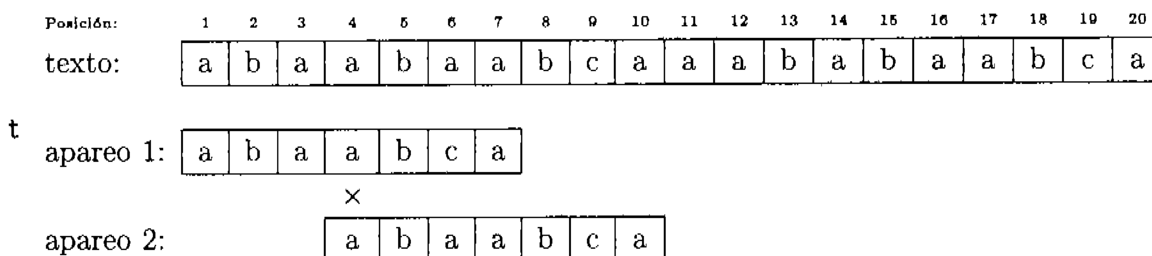
Figura 9.3: AF que reconoce al patrón P=abaabca



es el siguiente carácter en el patrón, debe usar la información que ya tiene y recorrer al patrón a que se alinee con el prefijo más grande posible, para disminuir el número de comparaciones que se hacen. Por ejemplo, si tenemos el texto de la Figura 9.4 con este patrón, al encontrar la desavenencia en la posición 6, tenemos a la subcadena abaab de la cadena apareada con el texto; el patrón empieza con ab y el sufijo de la cadena apareada también es ab, por lo que podemos recorrer el patrón hasta la posición en la que el prefijo ab del patrón esté apareada con el sufijo ab apareado en el texto, lo que se puede apreciar en la tercera línea de la Figura 9.4. Si no se considera esta información, tendríamos que recorrer tres veces el patrón en una posición cada vez para llegar al mismo punto.

Se debe notar que no puede haber una ocurrencia del patrón en la posición 2 del texto, porque ahí hay una b y el patrón empieza con a. Tampoco la va a haber en la posición 3 porque la siguiente posición contiene una a mientras que en el patrón tenemos una b; esta información ya fue capturada por el autómata conforme transita por los estados al ir leyendo

Figura 9.4: Reacomodo del patrón para disminuir el número de comparaciones



el texto.

En este algoritmo de apareamiento de cadenas, el preproceso consiste en construir al autómata finito que va a hacer el reconocimiento. Para cada prefijo del patrón – incluyendo a ε, la palabra vacía – el algoritmo construye cadenas que consisten del prefijo, seguido de cada símbolo del alfabeto; a continuación, para cada cadena construida, verifica la longitud de un prefijo en el patrón que también sea sufijo de esta cadena; el estado actual corresponde a la longitud del prefijo tomado del patrón; el tamaño del sufijo determina la transición del AF bajo el símbolo que fue concatenado al prefijo.

Una vez que el AF es construido, el apareamiento de cadenas se lleva a cabo simplemente siguiendo la tabla de transiciones. Cada vez que el AF pasa por el estado m, el número de símbolos en el patrón, se reporta un apareamiento.

La especialización para este algoritmo se encuentra en el Listado 9.12.

Listado 9.12: Especialización para apareamiento con un autómata finito 1/3

```
 1 public class ApareamientoAutomataFinito
 2                         extends ApareamientoAbstracto {
 3     /* Tabla de transición para el autómata finito.                 */
 4   protected int[][] tablaTrans;
 5     /* Alfabeto con el cual texto y patron están construidos.      */
 6   protected char[] alfabeto;
 7     /* Se registra el estado previo, para que se recuerde a         *
 8      * través de invocaciones de compara y recorre.                 */
 9   protected int pPrev;
10     /* Construye texto y patron y registra las posiciones ini-      *
11      * ciales para que se lleve a cabo la comparación.              */
12   public ApareamientoAutomataFinito(char[] patron,
13                                      char[] texto) {
14       super(patron, texto);
15       alfabeto = StringUtils.cnstryeAlfabeto(patron);
16       tablaTrans = new int[patron.length][alfabeto.length];
17   }
```

Listado 9.12: Especialización para apareamiento con un autómata finito 2/3

```
18    /* Verifica la periodicidad de patron, armando transiciones  *
19     * de un estado a uno previo siempre que el carácter actual  *
20     * aparezca antes en patron. Regresa la tabla de transicio-  *
21     * nes del autómata.                                         */
22  protected void preprocesaPatron(char[] patron) {
23    /* Construye la tabla de transiciones.                       */
24      int alf = alfabeto.length;
25      char[] prefijo = new char[m + 2];
26      for (int q = 0; q <= m; q++) {
27          boolean esPrefijo;
28          prefijo[q] = patron[q];
29          for (int ch = 1; ch < alf; ch++) {
30              prefijo[q + 1] = alfabeto[ch];
31              int k = Math.min(m + 1, q + 2);
32              int j;
33              do {
34                  k--;
35                  j = 1;
36                  while (j <= k
37                          && patron[j] == prefijo[q - k + j + 1])
38                      j++;
39                  if (j > k) {
40                      esPrefijo = true;
41                  } else {
42                      esPrefijo = false;
43                  }
44              } while (k > 0 && !esPrefijo);
45              tablaTrans[q][ch] = k;
46          }
47      }
48  }
49    /* Hace una revisión de texto usando el autómata que se      *
50     * construyó, leyendo carácteres de texto. Cada estado del   *
51     * autómata identifica a una posición en patron. Cuando se   *
52     * alcanza el estado m se ha encontrado un apareamiento.     *
53     * Cada vez que hay una transición a un estado "menor", se   *
54     * ha encontrado una desavenencia.                           */
55  protected boolean compara(RefInt refT, RefInt refP) {
56      int t = refT.intValue();
57      int p = refP.intValue();
58      if (t == 1)
59          p = 0; // Primera vez que se entra.
60      int tT = t + p;
61      pPrev = p;
62      for (; tT <= n; tT++) {
63          pPrev = p;
64          int car = StringUtils.getChar(texto[tT], alfabeto);
```

Listado 9.12: Especialización para apareamiento con un Autómata Finito 3/3

```
65              if (car == 0)
66                  p = 0;
67              else
68                  p = tablaTrans[p][car];
69              refP.setValue(p);
70              if (pPrev == m)
71                  return true;
72              if (p <= pPrev)     // se debe recorrer a patron.
73                  return false;
74          }
75          return false;
76      }
77      /* Recorre patron a la derecha para empezar nuevo apareo.      *
78       * El número de posiciones a recorrer se determina por la      *
79       * diferencia entre el estado anterior y el estado actual.     */
80      protected int recorre(RefInt refP, RefInt refT) {
81          int p = refP.intValue();
82          return Math.max(1, pPrev - p + 1);
83      }
84 }
```

Correctez de ApareamientoAutomataFinito

Es claro que el constructor cumple con los predicados dados en ApareamientoGenerico, ya que después de llamar al constructor de la superclase, lo único que hace es construir el alfabeto que corresponde al patrón y preparar la tabla para el autómata finito – líneas [15] y [16] del Listado 9.4 – por lo que no modifica ni al patrón ni al texto una vez que los dejó listos para proceso.

Respecto al método preprocesaPatron es claro que cumple con el predicado (9.3b). Hay que comprobar, sin embargo que siempre termina. Esto se demuestra siguiendo las condiciones para los ciclos anidados en las lineas [26–47] del Listado 9.4. La correctez del método compara presenta únicamente el problema de que no siempre que es invocado empieza a comparar desde el primer símbolo en el patrón, sino que aprovecha el conocimiento que le da la tabla del autómata. Pero lo único que debemos demostrar es que reporta bien las ocurrencias del patrón en el texto, lo cual se demuestra siguiendo la tabla del autómata. El método recorre calcula bien el desplazamiento que se debe hacer del patrón, como la diferencia entre el estado en el que se dio la desavenencia y el estado al que se transfirió el autómata. Por último hay que demostrar la correctez del autómata que se construyó, analizando la periodicidad que pueda tener el patrón y propiedades de sufijos.

Complejidad ApareamientoAutomataFinito

Pasamos a ver de qué manera contribuye cada método a la complejidad de esta especialización.

$f_{constr}(n, m) = O(m)$. Además de llamar al constructor de la superclase, se tiene que construir el alfabeto, restringiendo el alfabeto a los símbolos que aparecen en el patrón.

$f_{preProc}(m) = O(m^3)$. La construcción del autómata se hace con cuatro ciclos anidadas – líneas [26–47], [29–46], [33–44] y [36–38] del Listado 9.4. En el ciclo exterior se ejecutan $m + 2$ iteraciones. El primer ciclo anidado se controla con el tamaño del alfabeto, que está acotado a su vez por el tamaño del patrón. En el segundo ciclo anidado se busca el prefijo más grande, lo que podría ejecutarse $m + 1$ veces si no hay prefijo propio. Por último, en el tercer ciclo anidado tiene que verificar si la subcadena es o no prefijo. Todas estas comparaciones están acotadas por m.

$f_{masTPA}(n) = c_h$. Consiste únicamente de una comparación de valores enteros.

Debemos observar al conjunto de las invocaciones tanto a **compara** como a **recorre**, ya que haremos análisis amortizado de estos métodos:

$\sum_r f_{compara}(m) = O(n)$ Cada posición en el texto se va a comparar contra el patrón nada más una vez, por lo que tenemos tiempo amortizado.

$\sum_r f_{recorre}(m) = O(n)$ Este método será invocado cada vez que el patrón se aparee o que haya una desavenencia. De esto r está acotado por n .

Con esto tenemos ya la clase de complejidad a la que pertenece esta especialización:

$$\text{Complejidad de ApareamientoAutomataFinito} = O(n + m^3 \cdot |\Sigma|), \tag{9.9}$$

que puede resultar muy atractivo si el tamaño del patrón es mucho más pequeño que el tamaño del texto.

9.2.4. La especialización para el apareamiento de Knuth-Morris-Pratt

Una de las ventajas del apareamiento de cadenas usando autómatas finitos es que el texto puede ser proporcionado conforme se va requiriendo, ya que cada símbolo en el texto es comparado contra el patrón exactamente una vez. Sin embargo, el preproceso del patrón y el espacio necesario para guardar la tabla de transiciones puede resultar muy oneroso si el alfabeto con el que se trabaja es grande.

El algoritmo de Knuth-Morris-Pratt – [KMP77], en adelante KMP – utiliza el concepto del apareamiento con un autómata finito, donde la etiqueta del estado indica el número de carácteres apareados hasta el momento, introduciendo algunas diferencias para facilitar el preproceso y disminuir la cantidad de espacio requerida:

– En lugar de la tabla de transiciones se construye una tabla de transferencias (**goto**) similar a un diagrama de flujo, donde desde cada nodo únicamente hay dos posibili-

dades: éxito o fracaso en el apareamiento.

- Cada nodo del diagrama compara el símbolo actual en el texto con el que está denotado por el nodo.
- Si el resultado es éxito, se procede a leer el siguiente carácter en el texto. Cuando se sigue la ruta del fracaso, no se lee carácter del texto.
- Hay un nodo privilegiado, el del inicio, que tiene únicamente un arco saliendo de él y que fuerza una lectura. El proceso de apareamiento empieza en este nodo.
- Existe un nodo aceptador que cuando es alcanzado señala la aparición del patrón en el texto. Como este nodo es alcanzado por un arco exitoso, se lleva a cabo una lectura adicional. Desde este nodo no hay posibilidad de éxito, así que de él sale un arco de fracaso que lleva al máximo prefijo posible que es sufijo del patrón.

Si se llega al final del texto simultáneamente que al final del patrón, el algoritmo debe reconocer la ocurrencia del apareamiento y terminar exitosamente. Si el final del texto se alcanza en cualquier otro nodo, simplemente el patrón no ocurre al final del texto y deberá también terminar bien.

El problema en esta especialización es la construcción de la tabla de transferencias. Para la construcción de la tabla de transferencias se procede de la siguiente manera:

Cuando los símbolos en el texto y el patrón se aparean, el proceso simplemente va al siguiente nodo. Cuando se da una desavenencia, el proceso trata de rescatar la parte del patrón que lleva apareada, hasta el último carácter antes del que provocó la desavenencia. Por lo tanto, para construir la tabla de flujo ((tablaDeFlujo) al preprocesar el patrón, para cada posición i en el patrón, uno tiene que encontrar el mayor prefijo de patron que es un sufijo de patron[1 ... i − 1]. Este cálculo directo es, sin embargo, muy costoso. El algoritmo KMP propone un método recursivo para construir las tablas que tiene un costo lineal en m. La especialización para este método se encuentra en el Listado 9.13.

Listado 9.13: Especialización para el algoritmo de Knuth-Morris-Pratt 1/3

```
 1    public class ApareoKnuthMorrisPratt
 2                               extends ApareamientoAbstracto   {
 3      /* Marca la presencia anterior del carácter con código car    *
 4       * en patron.                                                  */
 5    private int[] tablaDeFlujo;
 6      /* Para recordar la posición anterior en texto.               */
 7    private int pPrev;
 8      /* Verifica si el carácter en el texto aparece en patron .    */
 9    private boolean[] existeCar = new boolean[256];
10      /* Inicia patron y texto; crea las tablas para los brincos    *
11       * y las "desavenencias" (mismatch).                          */
12    public ApareoKnuthMorrisPratt(char[] patron, char[] texto)  {
13        super();
14        patron = new char[patron.length + 1];
15        texto = texto;
16        for (int i = 0; i < patron.length; i++)
17            patron[i] = patron[i];
```

Listado 9.13: Especialización para el algoritmo de Knuth-Morris-Pratt 2/3

```
18          Integer car = new Integer(0);
19          byte bcar = car.byteValue();
20          patron[patron.length - 1] = (char) bcar;
21          tablaDeFlujo = new int[patron.length];
22          p = 1;
23          pPrev = 0;
24          m = patron.length - 1;
25          n = texto.length - 1;
26      }
27        /* Se preprocesa a patron para determinar la periodicidad    *
28         * del mismo. Siempre que tengamos una desavenencia, bus-    *
29         * camos la posición de una subcadena previa que empiece     *
30         * con ese carácter. Llena las tablas necesarias.            */
31      protected void preprocesaPatron(char[] patron) {
32          tablaDeFlujo[0] = 0;
33          tablaDeFlujo[1] = 0;
34          existeCar[(int) patron[1] % 256] = true;
35          int k, s = 0;
36          for (k = 2; k <= m + 1; k++) {
37              existeCar[(int) patron[k] % 256] = true;
38              while (s > 0 && patron[k] != patron[s + 1])
39                  s = tablaDeFlujo[s];
40
41              if (patron[s + 1] == patron[k])
42                  s++;
43              tablaDeFlujo[k] = s;
44          }
45      }
46        /* Compara carácter por carácter, de izquierda a derecha,    *
47         * patron contra texto. Cada comparación empieza en la po-   *
48         * sición en la que la última invocación terminó, indicada   *
49         * por refT. Para saber cómo recorrer a patron se guarda el  *
50         * valor previo de p en prevP.                               */
51      protected boolean compara(RefInt refT, RefInt refP) {
52          int tT;
53          int t = refT.intValue();
54          int p = refP.intValue();
55          if (t == 1)
56              pPrev = p = tT = 1;
57          else
58              tT = t + p - 1;
59          while (tT <= n) {
60              if (p == m + 1) {
61                  pPrev = p;
62                  p = tablaDeFlujo[p];
63                  if (p == 0) {
64                      p = 1;
65                      tT++;
66                  }
```

Listado 9.13: Especialización para el algoritmo de Knuth-Morris-Pratt 3/3

```
67                  refP.setValue(p);
68                  return true;
69              }
70              if (patron[p] == texto[tT]) {
71                  tT++;
72                  pPrev = p++;
73              } else {
74                  int carI = (int) texto[tT] % 256;
75                  pPrev = p;
76                  if (existeCar[carI]) {
77                      p = tablaDeFlujo[p];
78                  } else
79                      p = 0;
80                  if (p == 0) {
81                      p = 1;
82                      tT++;
83                      refP.setValue(p);
84                      return (p >= m) ? true : false;
85                  }
86                  refP.setValue(p);
87                  return (p >= m) ? true : false;
88              }
89          }
90          refP.setValue(p);
91          return (p >= m) ? true : false;
92      }
93       /* El número de posiciones que se tiene que desplazar a      *
94        * patron para continuar con el apareamiento.                */
95      protected int recorre(RefInt refP, RefInt refT) {
96          int p = refP.intValue();
97          return (pPrev > m
98                      ? ((pPrev - 1) > 1
99                                  ? pPrev - p : 1)
100                     : Math.max((pPrev - Math.max(1, p) + 1), 1));
101     }
102 }
```

Correctez de la especialización ApareoKnuthMorrisPratt

La principal aportación de este algoritmo es el preproceso del patrón con un costo de orden lineal respecto a m. La demostración de que esta construcción es correcta es un proceso no trivial, que omitimos en el presente trabajo por falta de espacio, pero el lector interesado lo puede encontrar en, entre otros, [Ata99, BG00, CLRS01] o directamente en [KMP77]. Una vez que suponemos esta construcción correcta, la demostración de correctez de los métodos invocados desde apareamientoGenerico es un trabajo de mucho detalle pero sin gran dificultad.

Complejidad de la especialización ApareoKnuthMorrisPratt

Si el preproceso del patrón se lleva $O(m)$, se puede demostrar, con argumentos de costos amortizados – función potencial – que el método compara trata de aparear a cada símbolo del texto únicamente una vez, lo que a lo largo de la ejecución de esta especialización significa que compara tiene un costo amortizado de $O(n)$. El método recorre tiene complejidad $O(1)$, al igual que el método masTPA. Por lo tanto, la fórmula dada para ApareamientoGenérico, con los valores sustituidos para esta especialización, queda como sigue:

$$\begin{aligned} T_{\text{ApKMP}}(n,m) &= c_0 + O(m) + \sum_q \Big(O(1) + O(1)\Big) + \sum_q T_{\text{compara}}(m) \\ &= O(m) + O(n) \\ &= O(m+n) \end{aligned} \tag{9.10}$$

9.2.5. Especialización para el apareamiento de Boyer-Moore

En la misma época en que se publica el algoritmo de Knuth-Morris-Pratt para el apareamiento de cadenas, R. Boyer y J. Moore publican un algoritmos similar – [BM77] – que hace uso también de las propiedades de los prefijos y sufijos en un patrón de texto para lograr que los corrimientos dados por el método recorre sean más amplios, sin que por eso se salte ninguna ocurrencia del patrón. La principal diferencia entre este algoritmo y el de KMP es que en aquel la comparación del patrón con el texto se hace de derecha a izquierda, mientras que en el de KMP la comparación se hace de izquierda a derecha. Aunque en teoría el algoritmo de Boyer-Moore presenta un orden de complejidad similar al que presenta el algoritmo de KMP, es en las constantes donde se encuentra la diferencia. Además, dependiendo del tipo particular de patrón, en la práctica se pueden presentar tiempos muy eficientes para este algoritmo. La presentación de este algoritmo como especialización de ApareamientoGenerico se encuentra en el Listado 9.14.

Listado 9.14: Especialización para el algoritmo de Boyer-Moore 1/4

```
 1 public class ApareamientoBoyerMoore
 2                 extends ApareamientoAbstracto {
 3     /* Auxiliar para llevar la cuenta de la posición en patron.    */
 4     private int tempP;
 5     /* Da los patrones de periodicidad para los corrimientos.      */
 6     private int[] unCorrimiento;
 7     /* Da la regla del mal carácter para este algoritmo.            *
 8      * Correspondiendo a cada carácter hay una lista con todas      *
 9      * las posiciones en patron donde ese carácter aparece.         */
10     private ListaDePosiciones[] unSigma =
11                                 new ListaDePosiciones[256];
12     /* Construye una instancia de esta especialización, creando     *
13      * las tablas que se van a necesitar.                           */
14     public ApareamientoBoyerMoore(char[] patron, char[] texto) {
15         super(patron, texto);
```

Listado 9.14: Especialización para el algoritmo de Boyer-Moore 2/4

```
16          unCorrimiento = new int[m + 1];
17          t = 1;  p = 1;
18      }
19        /* Construye las tablas unCorrimiento y unSigma que indican      *
20         * dónde continuar las comparaciones entre el texto y el         *
21         * patrón.                                                       */
22      protected void preprocesaPatron(char[] patron)  {
23          int[] posN = new int[m + 1];
24          int[] ultPosP = new int[m + 1];
25          int[] lp = new int[m + 1];
26          int i = 1,
27              k = m - 1,
28              r = m,
29              l = m,
30              j = 0;
31          /* Se calcula posN en tiempo lineal: subcadena más larga       *
32           * que termina en la posición k y que es un sufijo de         *
33           * patron.                                                    */
34          for (k = m - 1; k > 0; k--) {
35              i = 1;
36              if (k <= l) {
37                  while (i <= k
38                          && patron[m - i + 1] == patron[k - i + 1])
39                      i++;
40                  posN[k] = i - 1;
41                  if (posN[k] > 0) {
42                      r = k;
43                      l = k - posN[k] + 1;
44                  }
45              } else {
46                  int primerK = m - r + k;
47                  if (posN[primerK] < k - l + 1)
48                      posN[k] = posN[primerK];
49                  else {
50                      i = k - l + 1;
51                      while (i <= k && patron[m - i + 1]
52                              == patron[k - i + 1])        i++;
53                      posN[k] = i - 1;
54                      l = k - posN[k] + 1;
55                      r = k;
56                  }
57              }
58          }
59            /* Calculando L'[i] en tiempo lineal: patron[i..n]=w         *
60             * aparea a un sufijo de patron[1..L'[i]].                   */
61          for (k = 1; k < m; k++)  {
62              i = m - posN[k] + 1;
63              if (i > 0 && i <= m)
64                  ultPosP[i] = k;
65          }
```

Listado 9.14: Especialización para el algoritmo de Boyer-Moore 3/4

```
66          /* Calculando l'[i] en tiempo lineal: tamaño del sufijo   *
67           * más grande de patron[i..m] que es también prefijo de   *
68           * patron.                                                 */
69         int max = 0;
70         int cont = 0;
71         for (i = m, j = 1; i >= 1; i--, j++) {
72             if (j == posN[j] && max < j)
73                 max = j;
74             lp[i] = max;
75         }
76          /* Calculando unCorrimiento en tiempo lineal.             */
77         unCorrimiento[0] = m - lp[2]; // Si patron casó.
78         for (k = 1; k <= m; k++) {
79             if (ultPosP[k] == 0)
80                 unCorrimiento[k] = m - lp[k];
81             else
82                 unCorrimiento[k] = m - ultPosP[k];
83         }
84          /* Calculando la regla del mal carácter . Encuentra la    *
85           * presencia más cercana del carácter a la izquierda      *
86           * del carácter desavenido.                               */
87         for (int b = 0; b < 256; b++)
88             unSigma[b] = null; // Carácteres no en patron.
89         for (p = 1; p <= m; p++) {
90             int q = (int) ((byte) patron[p]);
91             Posicion pPos = new Posicion(p);
92             if (unSigma[q] == null)
93                    unSigma[q] = new ListaDePosiciones();
94             unSigma[q].prepend(pPos);
95         }
96      /* Compara patron y texto carácter por carácter. Debe regresar *
97       * si se encuentra una presencia de patron en texto. patron y  *
98       * texto están alineados. La comparación se lleva a cabo       *
99       * de derecha a izquierda.                                     */
100     protected boolean compara(RefInt refT, RefInt refP) {
101         p = m;
102         int h = t + m - 1;
103         while (p > 0 && patron[p] == texto[t + p - 1])
104             p--;
105         tempP = p;
106         refP.setValue(p);
107         if (p == 0) { // Se completó el apareo.
108             p = 1;
109             refP.setValue(p);
110             return true;
111         } else {
112             return false;
113         }
114     }
```

Listado 9.14: Especialización para el algoritmo de Boyer-Moore 4/4

```
115         /* Recorre patron a la derecha para iniciar una nueva com-      *
116          * paración. El número de posiciones a recorrer depende del     *
117          * sufijo del patron y el texto que se están comparando, y      *
118          * está dado por las tablas unCorrimiento y unSigma.           */
119     protected int recorre(RefInt refP, RefInt refT) {
120         int t = refT.intValue();
121         int pbcr = 1;
122         int sh;
123           /* Obtiene el corrimiento dado por el carácter               *
124            * desavenido.                                               */
125         Posicion bcr;
126         if (unSigma[(int) ((byte) texto[t + tempP - 1])] == null) {
127             bcr = null;
128         } else {
129             bcr = unSigma[(int) ((byte) texto[t + tempP - 1])].
130                                                         getFirst();
131         }
132     }
133         if (bcr == null && tempP == m)  {
134             pbcr = m;
135         } else {
136             while (bcr != null && bcr.getPos() >= tempP) {
137                 bcr = bcr.getNext();
138             }
139             if (bcr == null)  {
140                 pbcr = tempP;
141             } else {
142                 pbcr = tempP - bcr.getPos();
143             }
144         }
145         sh =  (tempP == m
146                ? pbcr
147                : Math.max(unCorrimiento[tempP + 1], pbcr));
148         return sh;
149     }
150}
```

Correctez de la especialización ApareoBoyerMoore

Para demostrar la correctez de los métodos implementados para esta especialización se debe demostrar la correctez de la construcción de las tablas que se hace en el preproceso, y que la manera de usarlas es la correcta. Sin embargo, como esta demostración es muy elaborada queda fuera del alcance de esta sección (se encuentra, entre otros, en [Ata99, BG00, SF96] con una presentación muy clara en [Gus97], aunque alejada de la exposición original dada en [BM77]. Si se supone la construcción y el uso de las tablas como correctas, es fácil ver que el demostrar que los predicados del algoritmo genérico se cumplen en esta especialización,

a pesar de que conllevan una cierta cantidad de trabajo, requieren únicamente de vigilar la ejecución de cada uno de los métodos.

Complejidad de la especialización de ApareoBoyerMoore

El cálculo de la complejidad de este algoritmo llevó varios años y distintas versiones. El preproceso del patrón, como se puede apreciar en el Listado 9.14, líneas [22–132], aunque elaborado tiene un costo de $O(m)$.

Al igual que la especialización de Knuth-Morris-Pratt, el peor caso del algoritmo es cuadrático y se presenta cuando todo el patrón consiste del mismo carácter. Sin embargo, es sorprendente que cuando se usa para encontrar la primera ocurrencia del patrón en el texto, la especialización corre en tiempo lineal. En la práctica el algoritmo es $O(n)$ cuando se está procesando un texto tomado de alfabetos relativamente grandes comparados con el tamaño del patrón. La demostración de estas aseveraciones depende de propiedades en la periodicidad del patrón y frecuencias de subcadenas del patrón en el texto, lo que determina el número de veces que cada carácter en el texto va a ser comparado contra el patrón. Se puede ver la evolución de las medidas de complejidad de este algoritmo en [Teo04].

9.3. Flujo máximo en redes*

Una *red* es una digráfica con pesos asignados a los arcos. Una *red de flujo* es una red cuyos arcos están etiquetados con una *capacidad*. El problema de flujo máximo consiste en enviar la mayor cantidad de flujo posible desde un nodo de la red hacia otro, sin exceder la capacidad de ninguno de los arcos.

En [AMO93] los autores hacen una excelente exposición del estado del arte en el tema de redes de flujo. Para el caso particular de flujo máximo hacen una presentación de varios algoritmos que trabajan con flujos enteros bajo un enfoque unificado, clasificándolos en una primera instancia por la manera cómo proceden para incrementar el flujo. En este trabajo seguimos la clasificación dada allí para diseñar una familia de algoritmos, con una estrategia genérica que resuelve el problema de flujo máximo entero en tiempo polinomial.

Especializamos la implementación que tenemos para representar gráficas, a que representen adecuadamente redes de flujo. Esta especialización incluye una extensión a la implementación de vértices (nodos) y arcos, pues estos algoritmos requieren de acceso directo a arcos incidentes desde o hacia un vértice, y no basta con el acceso secuencial dado por las listas de incidencia. Estas extensiones garantizan el costo de orden constante en el acceso, modificación y eliminación de arcos, acciones requeridas por este tipo de algoritmos.

* Una versión extendida de este tema, con una implementación previa y, en algunos puntos, distinta, de las clases se encuentra en [Rey00].

9.3.1. Algoritmo genérico para el problema de flujo máximo entero en redes

En el nivel más alto de abstracción se define una estrategia genérica que consiste en iterar mientras se logre un incremento en el flujo obtenido. El control del ciclo invoca a un método que define si vale la pena volver a procesar la red, y si así se considera, se invoca a métodos que deberían incrementar el flujo. Cada especialización define cómo se determina si se debe o no hacer el esfuerzo de incrementar el flujo, y la manera de hacer esto último. La clase abstracta que representa a este nivel de abstracción se encuentra en el Listado 9.15.

Listado 9.15: Clase abstracta que define la estrategia genérica

```
1 public abstract class FlujoMaximoAbstracto {
2      /* La red de flujo con que va a trabajar el algoritmo.          */
3    protected RedFlujo redFljo;
4      /* Registra el flujo máximo obtenido hasta ahora.               */
5    protected int    flujoMaximo;
6      /* Instala la red de flujo con la que se va a trabajar.         */
7    public FlujoMaximoAbstracto (RedFlujo redFljo) {
8        this.redFljo   = redFljo;
9        flujoMaximo    = 0;
10   }
11     /* Estrategia genérica para el problema.                        */
12   public final int flujoMaximoGenerico () {
13       preprocesa ();
14       while (!esMaximoFlujo ())
15           incrementaFlujo ();
16       return valorDelFlujoMaximo ();
17  }
18     /* Preprocesa a la red de flujo para darle condiciones          *
19      * adecuadas de inicio.                                          */
20   protected abstract void preprocesa ();
21     /* Verifica si el flujo obtenido en la última iteración         *
22      * alcanzó ya el valor máximo posible, o todavía puede ser      *
23      * incrementado.                                                 */
24   protected abstract boolean esMaximoFlujo ();
25     /* Trata de incrementar el flujo obtenido en la última          *
26      * iteración.                                                    */
27   protected abstract void incrementaFlujo ();
28     /* Regresa el valor del flujo máximo, registrado en el          *
29      * atributo correspondiente.                                     */
30   protected abstract int valorDelFlujoMaximo ();
31 }
```

En esta clase abstracta simplemente, como ya mencionamos, se establecen las condiciones iniciales, una primera aproximación, del flujo sobre la red en preprocesa; a continuación se entra en un ciclo en el cual se verifica si se alcanzó ya el máximo flujo posible (esMaximoFlujo), y si no es así, se procede a intentar incrementar el flujo en incrementaFlujo. Una vez que se sale de este ciclo, simplemente se entrega el valor calculado como máximo

flujo (valorDelFlujoMaximo).

En la estrategia genérica para esta familia de algoritmos podemos demostrar que si:

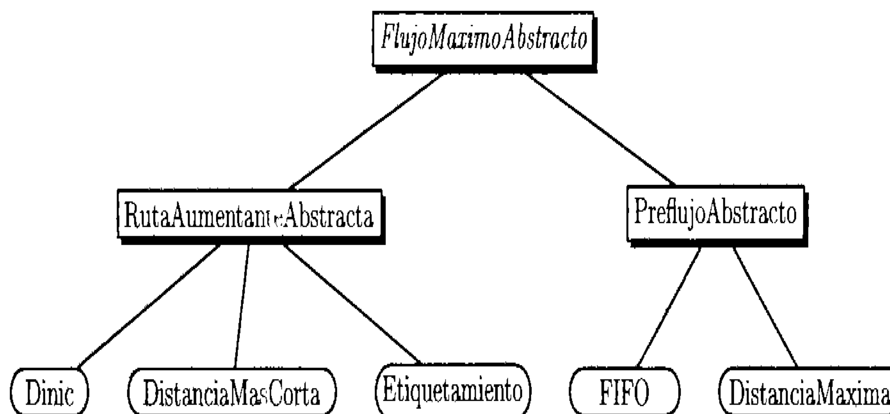
preprocesa	construye la red de flujo y establece las estructuras de datos requeridas;
esMaximoFlujo	es correcto y determina de manera correcta si se debe iterar una vez más;
incrementaFlujo	Consigue que el valor del flujo calculado antes de la invocación de este método sea menor que el que calcula este método;
valorDelFlujoMaximo	regresa el valor del flujo calculado por la última iteración del ciclo;

entonces el algoritmo genérico calcula el valor del máximo flujo posible – una demostración detallada de esto se puede ver en [AMO93, Rey00].

La fórmula genérica para la complejidad de esta familia de algoritmos es la siguiente:

$$\begin{aligned}\text{Complejidad de FlujoMaximoAbstracto} = {} & f_{\text{constr}}(n,m) + f_{\text{preprocesa}}(n,m) + \\ & + f_{\text{valorFlujoMax}}(n,m) + \sum_{q} \left(f_{\text{esMaxFlujo}}(n,m) + f_{\text{incrementaFljo}}(n,m) \right) \qquad (9.11)\end{aligned}$$

Figura 9.5: Jerarquía de la familia de algoritmos de flujo máximo



El árbol jerárquico de esta familia de algoritmos se puede ver en la Figura 9.5. En este árbol se puede observar que, al igual que en la familia para obtener árboles generadores, se encuentran varios niveles de abstracción. En el segundo nivel se definen tipos de estrategias para incrementar el flujo. La primera de ellas es mediante rutas o trayectorias aumentantes, mientras que la segunda subfamilia trabaja en base a enviar preflujo. Seguiremos una estrategia DFS para la exposición de estas subfamilias.

9.3.2. Subfamilia de cálculo de flujo con trayectorias aumentantes

Las especializaciones para la familia de trayectorias aumentantes deben encontrar una trayectoria aumentante y después proceder a incrementar el flujo a través de esa trayectoria. Esta manera de resolver el problema de flujo máximo en una red fue propuesta por Ford y Fulkerson en [FF62], por lo que cuando se inicia el estudio del tema de flujo máximo obligatoriamente se hace mención del algoritmo de Ford-Fulkerson – ver, por ejemplo, [Eve79, CLRS01] – englobándose acá, como lo hacemos en este trabajo, a un conjunto de algoritmos que trabajan con esta filosofía. La clase abstracta para esta subfamilia se muestra en el Listado 9.16.

Listado 9.16: Clase abstracta para la subfamilia que trabaja con trayectorias aumentantes 1/3

```
 1 public abstract class TrayectoriaAumentanteAbstracta
 2                         extends FlujoMaximoAbstracto {
 3      /* Almacena la trayectoria aumentante encontrada.              */
 4    protected ColaPrioridades trayctria;
 5      /* Almacena el flujo máximo encontrado hasta ahora.            */
 6    protected int capTryctria = 0;
 7      /* Inicializa la trayectoria aumentante, y la capacidad de    *
 8       * esa trayectoria en 0.                                       */
 9    public TrayectoriaAumentanteAbstracta (RedFlujo redFljo) {
10        super (redFljo);
11        trayctria  = new ColaPrioridades();
12        capTryctria = 0;
13    }
14      /* El flujo es susceptible de aumentarse si es que existe     *
15       * una trayectoria aumentante.                                 */
16    protected boolean esMaximoFlujo () {
17        return !(encuentraTryctriaAumntnte ());
18    }
19      /* El flujo es incrementado a lo largo de la trayectoria      *
20       * aumentante encontrada.                                      */
21    protected void incrementaFlujo ()  {
22        aumentaCapcdadTryctria();
23        corrigeRedFlujoResidual();
24        corrigeFlujoEnRedFlujo();
25    }
26      /* Regresa el valor del atributo.                              */
27    protected int valorDelFlujoMaximo () {
28        return flujoMaximo;
29    }
30      /* Aumenta la capacidad de la trayectoria construyendo la     *
31       * red residual, hasta que alcanza un flujo máximo en esa     *
32       * trayectoria.                                                */
33    protected void aumentaCapcdadTryctria () {
34        int tmp;
35        IteradorCola itrdrTryctria = trayctria.IteradorCola();
```

Listado 9.16: Clase abstracta para la subfamilia que trabaja con trayectorias aumentantes. 2/3

```
36          NodoDeFlujo  u = (NodoDeFlujo) itrdrTryctria.next();
37          NodoDeFlujo  v = (NodoDeFlujo) itrdrTryctria.next();
38          capTryctria =  u.capacidad(v);
39          while (itrdrTryctria.hasNext()) {
40              u = v;
41              v = (NodoEtiquetado) itrdrTryctria.next();
42              tmp = u.capacidad(v);
43              if (tmp < capTryctria)
44                  capTryctria = tmp;
45          }
46      }
47        /* Corrige el flujo en la red utilizando tanto la trayec-    *
48         * toria aumentante como la red residual.                   */
49       protected void corrigeFlujoEnRedFlujo ()  {
50          flujoMaximo += capTryctria;
51          ArrayList nodosNR = redFljo.redFlujoResidual().obtenNodos();
52          ArrayList nodosN  = redFljo.redFlujo().obtenNodos();
53          ListIterator itrdrTryctria = trayctria.listIterator();
54          if (trayctria.isEmpty())
55              return;
56          NodoEtiquetado ur = null;
57          NodoEtiquetado vr = (NodoEtiquetado) itrdrTryctria.next();
58          NodoEtiquetado un = null;
59          int vPos = vr.getPos();
60          NodoEtiquetado vn = (NodoEtiquetado) nodosN.get(vPos);
61          while (itrdrTryctria.hasNext()) {
62              ur = vr;
63              un = vn;
64              vr = (NodoEtiquetado) itrdrTryctria.next();
65              vn = (NodoEtiquetado) nodosN.get(vr.getPos());
66              ArcoDeFlujo a = (ArcoDeFlujo) un.getArco(vn);
67              if (a != null)
68                  a.aumentaFlujo(capTryctria);
69              else {
70                  a = (ArcoDeFlujo) vn.getArco(un);
71                  if (a != null)
72                      a.dismnyeFlujo(capTryctria);
73              }
74          }
75      }
76        /* Corrige el flujo en la red residual conforme aumenta en   *
77         * la trayectoria aumentante.                                */
78      protected void corrigeRedFlujoResidual () {
79          NodoEtiquetado v1, v2;
80          ListIterator itrdrTryctria = trayctria.listIterator();
81          if (trayctria.isEmpty())
82              return;
83          v2 = (NodoEtiquetado) itrdrTryctria.next();
```

Listado 9.16: Clase abstracta para la subfamilia que trabaja con trayectorias aumentantes. 3/3

```
84          while (itrdrTryctria.hasNext()) {
85              v1 = v2;
86              v2 = (NodoEtiquetado) itrdrTryctria.next();
87              v1.dismnyeCapacidad(v2, capTryctria);
88              if (v2.conectadoCon(v1))
89                  v2.incrmntaCapacidad(v1, capTryctria);
90              else
91                  v2.conectaCon(v1, capTryctria);
92          }
93      }
94       /* Construye una trayectoria aumenta si es que ésta existe.  */
95      protected void constryeTryctria () {
96          NodoEtiquetado v = (NodoEtiquetado) redFljo.nodoPozo();
97          trayctria.add (v);
98          while (v != redFljo.nodoFuente()) {
99              v = v.getPi();
100             trayctria.addFirst(v);
101         }
102         ListIterator itrdrLista = trayctria.listIterator();
103         while (itrdrLista.hasNext())
104             NodoDeFlujo tmp = (NodoDeFlujo) itrdrLista.next();
105     }
106      /* Determina si una trayectoria aumentante existe para el    *
107       * valor actual de flujo máximo. La manera precisa en que    *
108       * esto se haga quedará determinado por las especializa-     *
109       * ciones.                                                   */
110     protected abstract boolean encuentraTryctriaAumntnte ();
111 }
```

Correctez de la familia de trayectorias aumentantes

El método esMaximoFlujo se traduce en este nivel de abstracción a que el algoritmo encuentre una trayectoria aumentante. Por lo tanto, refinamos el predicado que se refiere a este método a pedirle que reporte correctamente la existencia de una trayectoria aumentante, y que de existir ésta, la deje construida.

El predicado del método incrementaFlujo también se refina para pedir la correctez de los métodos aumentaCapcdadTryctria, que consiste en que anote bien los nodos y los arcos con el flujo modificado; para que el método corrigeRedFlujoResidual sea correcto, se requiere de él que construya adecuadamente la red residual; por último, a partir de ésta, el método corrigeFlujoEnRedFlujo debe actualizar correctamente los valores del flujo en los nodos y arcos de la red de flujo. Deberemos demostrar con detalle que cada una de las especializaciones cumplen con estos predicados.

Se demuestra la validez de trabajar con la red residual y que el flujo se puede incrementar a través de una trayectoria aumentante, de existir ésta.

Complejidad para la familia de trayectorias aumentantes

De esta estrategia genérica para el método incrementaFlujo, este término de la Fórmula (9.1) se descompone de la siguiente manera:

$$f_{esMaxFlujo}(n,m) = f_{encTryAum}(n,m) \quad (9.12)$$

$$f_{incrementaFljo}(n,m) = f_{aumCapTryc}(n,m) + f_{corrigeRedResdl}(n,m) + f_{corrigeRed}(n,m) \quad (9.13)$$

9.3.3. Algoritmo de Dinic de trayectorias aumentantes

Pasemos a revisar las especializaciones en esta subfamilia. En ella se encuentra el algoritmo de Dinic [Din70], que encuentra trayectorias aumentantes construyendo una red donde los nodos que se encuentran en una misma capa están a la misma distancia del origen del árbol BFS. El flujo se puede incrementar a través de trayectorias aumentantes a lo más m veces. La especialización se puede ver en el Listado 9.17.

Listado 9.17: Especialización de flujo máximo para el algoritmo de Dinic 1/4

```
1 public class TrayectoriasAumentantesDeDinic
2                 extends TrayectoriaAumentanteAbstracta {
3     /* Establece la red de flujo, una trayectoria aumentante   *
4      * inicial, y la red residual.                              */
5     public TrayectoriasAumentantesDeDinic (RedFlujo redFljo) {
6         super(redFljo);
7     }
8     /* Encuentra una trayectoria aumentante siguiendo caminos   *
9      * del mismo tamaño en la red residual.                     */
10    protected boolean encuentraTryctriaAumntnte () {
11        NodoEtiquetado s = (NodoEtiquetado) redFljo.nodoFuente();
12        NodoEtiquetado t = (NodoEtiquetado) redFljo.nodoPozo();
13        NodoEtiquetado v = s;
14        int numNodos = redFljo.redFlujoResidual().numNodos();
15        do {
16            ArcoDeFlujo a = (ArcoDeFlujo) v.arcoAdmisible();
17            if (a != null)
18                v = (NodoEtiquetado) avanza(v, a);
19            else
20                v = (NodoEtiquetado) retrocede(v);
21            if (v == s && ((NodoEtiquetado) s).estaEtiquetado()) {
22                preprocesa();
23                s = (NodoEtiquetado) redFljo.nodoFuente();
24                t = (NodoEtiquetado) redFljo.nodoPozo();
25                v = s;
26            }
27        } while ((v != t) && (s.getD() < numNodos));
```

Listado 9.17: Especialización de flujo máximo para el algoritmo de Dinic 2/4

```
28          if (v == t)  {
29              trayctria = new ColaPrioridades();
30              capTryctria = 0;
31              construyeTryctria ();
32              return true;
33          }
34          else
35              return false;
36      }
37        /* Establece la red con su red residual y construye el         *
38         * árbol BFS, para que los nodos queden en capas de acuerdo    *
39         * a su  distancia a la raíz.                                  */
40      public void preprocesa ()  {
41          cnstryeRedFlujoResidual ();
42          Digrafica    tmp = (GraficaFlujo) redFljo.
43                                   redFlujoResidual().invierte();
44          NodoDeFlujo   tR = (NodoDeFlujo) redFljo.nodoPozo();
45          ArrayList  nodos = ((GraficaFlujo) tmp).obtenNodos();
46          ArrayList nodosR = redFljo.redFlujoResidual().obtenNodos();
47          int i;
48          int iTR = tR.getPos();
49          Nodos t = (NodoEtiquetado) tmp.getTodosLosNodos()[iTR];
50          ExploracionBFS bfs = new ExploracionBFS(tmp);
51          bfs.exploracionGenerica(t);
52          for (i = 1; i < nodos.size();  i++) {
53              NodoEtiquetado v  =  (NodoEtiquetado) nodos.get(i);
54              NodoEtiquetado vR = (NodoEtiquetado) nodosR.get(i);
55              vR.setD(v.getD());
56              vR.desetiqueta();
57              vR.setPi(null);
58          }
59          NodoEtiquetado s  = (NodoEtiquetado) redFljo.nodoFuente();
60          if (s.getD() == -1)
61              s.setD (nodos.size());
62          creaRedFlujoPorCapas();
63      }
64        /* Construye la red residual correspondiente al flujo actual. */
65      public void cnstryeRedFlujoResidual () {
66          redFljo.copiaRedFlujoEnRedFlujoResidual ();
67          ArrayList nodos = redFljo.redFlujoResidual().obtenNodos();
68          ArrayList arcos = redFljo.redFlujoResidual().obtnArcos();
69          for (int i = 0; i < nodos.size(); i++)  {
70              if (nodos.get(i) == null)
71                  continue;
72              NodoEtiquetado u = (NodoEtiquetado) nodos.get(i);
73              ListaBasicaDeAristas ady =
74                                   (ListaBasicaDeAristas) u.getAdy();
75              ItrdrListaDeAristas iterAdy =
76                                         new ItrdrListaDeAristas(ady);
```

Listado 9.17: Especialización de flujo máximo para el algoritmo de Dinic 3/4

```
77              if (iterAdy.isEmpty())
78                  continue;
79              while (iterAdy.hasNext()) {
80                  ArcoDeFlujo aristaUV  = (ArcoDeFlujo) iterAdy.next();
81                  int    capacidad = aristaUV.getCapacidad();
82                  int    flujo     = aristaUV.getFlujo();
83                  NodoEtiquetado v = (NodoEtiquetado)
84                                              aristaUV.getHacia();
85                  if ((capacidad - flujo) <= 0)
86                      u.desconectaDe(v);
87                  else
88                      aristaUV.setCapacidad(capacidad - flujo);
89                  if (flujo > 0)
90                      v.conectaCon(u, flujo);
91              }
92          }
93      }
94      /* Construye la red por capas para que se pueda encontrar    *
95       * la trayectoria aumentante.                                */
96      public void creaRedFlujoPorCapas () {
97          ArrayList nodos = redFljo.redFlujoResidual().obtenNodos();
98          ArrayList nodosN = redFljo.redFlujo().obtenNodos();
99          int i;
100         for (i = 0; i < nodos.size(); i++) {
101             NodoEtiquetado u = (NodoEtiquetado) nodos.get(i);
102             if (u == null)      continue;
103             ListaBasicaDeAristas ady =
104                                     (ListaBasicaDeAristas) u.getAdy();
105             ItrdrListaDeAristas iterAdy =
106                                     new ItrdrListaDeAristas(ady);
107             if (iterAdy.isEmpty())     continue;
108             while (iterAdy.hasNext()) {
109                 ArcoDeFlujo aristaUV  =
110                                      (ArcoDeFlujo) iterAdy.next();
111                 int iHacia = aristaUV.getHacia().getPos();
112                 NodoEtiquetado v =
113                                     (NodoEtiquetado) nodos.get(iHacia);
114                 if (u.getD() != (v.getD() + 1))
115                     u.desconectaDe(v);
116             }
117         }
118     }
119     /* Si no se puede avanzar con la trayectoria aumentante,     *
120      * el algoritmo retrocede por el padre del nodo para         *
121      * intentar otra trayectoria.                                */
122     public Nodos retrocede (NodoEtiquetado v)  {
123         v.etiqueta();
```

Listado 9.17: Especialización de flujo máximo para el algoritmo de Dinic 4/4

```
124         if (v != redFljo.nodoFuente())
125             return ((Nodos) v.getPi());
126         else
127             return v;
128     }
129     /* Extiende la trayectoria aumentante con un arco.                */
130     public Nodos avanza(NodoEtiquetado u, ArcoDeFlujo a) {
131         NodoEtiquetado v = (NodoEtiquetado) a.getHacia();
132         v.setPi(u);
133         return v;
134     }
135 }
```

Correctez de la especialización de Dinic

En el método preprocesa se construye la red residual correspondiente al flujo actual. A continuación se construye la red de flujo por capas, dejando en la red residual únicamente los arcos que son admisibles. Cada vez que se forma la red de capas, el flujo se incrementa. Por otro lado, el algoritmo ejecuta a lo más n fases, ya que la distancia se incrementa en cada iteración al menos una unidad; esto hace que finalmente se llegue a n fases, donde n es la máximo distancia (sobre caminos simples) desde el origen a alguno de los vértices. Una vez que la distancia desde s es mayor o igual a n, sabemos que ya no existe ninguna trayectoria aumentante, por lo que el flujo ha alcanzado su máximo, lo que corresponde a la correctez de esta especialización.

Complejidad de la especialización de Dinic

El método esMaximoFlujo verifica si existe una trayectoria aumentante en la red. Para encontrarla,en cada iteración del ciclo hay que localizar al vértice con menor capacidad residual, y enviar flujo por uno de sus arcos, lo que lleva $O(|E| + |V|^2)$ pasos. El método que incrementa el flujo, lo hace a lo largo de la trayectoria aumentante, lo mismo que la corrección de la red residual, por lo que no va a colaborar con un término superior a $|V|^2$ en la complejidad del algoritmo, y no los tomamos en cuenta. Como el número de iteraciones está acotado por $|V|$, la complejidad de este algoritmos es $O(|E| + |V|^3) = O(|V|^3)$.

9.3.4. Especialización de trayectorias aumentantes de distancias mínimas

El segundo algoritmo de esta subfamilia es el que trabaja en base a construir la trayectoria aumentante encontrando en cada iteración a los nodos a distancia mínima, empezando con s. El algoritmo propuesto por Edmonds y Karp en [EK72] ejecuta una exploración BFS para colocar a los vértices por capas respecto al origen, tomándose la distancia como el número

de arcos desde el origen al vértice. El código para esta especialización de encuentra en el Listado 9.18.

Listado 9.18: Especialización para trayectorias aumentantes a distancia mínima 1/2

```
1  public class MinimasDistancias
2      extends TrayectoriaAumentanteAbstracta {
3        /* Instala la red de flujo e inicializa la trayectoria    *
4         * aumentante.                                            */
5      public MinimasDistancias(RedFlujo redFljo)  {
6          super(redFljo);
7      }
8
9        /* Encuentra una trayectoria aumentante, avanzando un arco  *
10        * a la vez utilizando distancias mínimas. Si se coloca a   *
11        * s en la trayectoria aumentante entonces el algoritmo ha  *
12        * encontrado una. Si no hay ninguna trayectoria aumentante *
13        * que contenga a t, el método devuelve falso.              */
14     protected boolean encuentraTryctriaAumntnte ()  {
15         NodoEtiquetado s = (NodoEtiquetado) redFljo.nodoFuente();
16         NodoEtiquetado t = (NodoEtiquetado) redFljo.nodoPozo();
17         NodoEtiquetado u = s;
18         int numNodos = redFljo.redFlujoResidual().numNodos();
19         do {
20             ArcoDeFlujo a = u.arcoAdmisible ();
21             if (a != null) {
22                 u = (NodoEtiquetado) avanza(u, a);
23             } else {
24                 u = (NodoEtiquetado) retrocede(u);
25             } // end of else
26         } while ((u != t) && (s.getD() < numNodos));
27         if (u == t) {
28             trayctria = new LinkedList();
29             capTryctria = 0;
30             constryeTryctria();
31             return true;
32         } else
33             return false;
34     }
35       /* Asigna etiquetas válidas de distancia a cada nodo en la  *
36        * red.                                                     */
37     public void preprocesa ()  {
38         GraficaFlujo   tmp = redFljo.redFlujo().invierte();
39         NodoDeFlujo     tR = (NodoDeFlujo) redFljo.nodoPozo();
40         ArrayList    nodos = tmp.obtenNodos();
41         ArrayList   nodosR = redFljo.redFlujoResidual().obtenNodos();
42         NodoDeFlujo      t = (NodoDeFlujo) nodos.get(tR.getPos());
43         ExploracionBFS     bfs = new ExploracionBFS(tmp);
44         bfs.exploracionGenerica(t);
```

Listado 9.18: Especialización para trayectorias aumentantes a distancia mínima 2/2

```
45            for (int i = 0; i < nodos.size(); i++) {
46                if (nodos.get(i) == null)
47                    continue;
48                NodoEtiquetado v  =  (NodoEtiquetado) nodos.get(i);
49                NodoEtiquetado vR =  (NodoEtiquetado) nodosR.get(i);
50                vR.setD(v.getD());
51            }
52        }
53          /* Retrocede al padre del nodo si la trayectoria aumentante   *
54           * ya no puede crecer y el último nodo visitado no es t.      */
55        public NodoEtiquetado retrocede(NodoEtiquetado v)  {
56            int d = v.distanciaMinimaNodosAdyacentes();
57            if (d == -1)
58                d = v.getD();
59            v.setD(d + 1);
60            if (v != redFljo.nodoFuente())
61                return  ((NodoEtiquetado) v.getPi());
62            else
63                return v;
64        }
65          /* Agrega un arco a la trayectoria aumentante, estable-       *
66           * ciendo al nodo actual como el padre del que se está        *
67           * alcanzando.                                                */
68        public NodoDeFlujo avanza(NodoEtiquetado u, ArcoDeFlujo a)  {
69            NodoEtiquetado  v  =  (NodoEtiquetado) a.getHacia();
70            v.setPi(u);
71            return v;
72        }
73 }
```

Correctez para la especialización de distancias mínimas

Dado que esta especialización únicamente altera la manera en que se van eligiendo los vértices para encontrar la trayectoria aumentante, se aplicarían acá las demostraciones de correctez dadas para la especialización anterior, simplemente revisando que el código funcione correctamente.

Complejidad para la especialización de distancias mínimas

Se demuestra para esta especialización, que la distancia más corta sobre la red residual para todo vértice v crece monótonamente cada vez que se aumenta el flujo. Apoyado en este resultado, se demuestra que el número de iteraciones que se van a llevar a cabo en el ciclo externo está acotado por $O(|V||E|)$. Sin embargo, como cada trayectoria aumentante se puede encontrar, usando BFS, en $O(|E|)$, este algoritmo está en $O(|V||E|^2)$.

9.3.5. Especialización con etiquetamiento de trayectorias aumentantes

La tercera especialización para esta subfamilia consiste en encontrar una trayectoria aumentante etiquetando todos los nodos para los cuales hay una trayectoria desde el origen. Si en algún momento se etiqueta al nodo pozo se encontró una trayectoria aumentante. El código para esta especialización se encuentra en el Listado 9.19.

Listado 9.19: Trayectorias aumentantes etiquetando nodos 1/2

```
1 public class EtiquetamientoTrayectoriasAumentantes
2                      extends TrayectoriaAumentanteAbstracta {
3     /* Instala la red de flujo e inicializa la trayectoria      *
4      * aumentante.                                               */
5   public EtiquetamientoTrayectoriasAumentantes(RedFlujo redFljo) {
6       super(redFljo);
7   }
8     /* Inicializa el estado de los nodos y la trayectoria       *
9      * aumentante.                                               */
10  protected void preprocesa () {
11      GraficaFlujo residual = redFljo.redFlujoResidual ();
12      ArrayList nodosR = residual.obtenNodos();
13      int numNodos = nodosR.size() - 1;
14      trayctria = new LinkedList();
15      capTryctria = 0;
16      for (int i = 0; i < nodosR.size(); i++) {
17          if (nodosR.get(i) == null)  continue;
18          NodoEtiquetado ve;
19          ve = (NodoEtiquetado) nodosR.get(i);
20          ve.desetiqueta();
21          ve.setPi(null);
22      }
23  }
24    /* Etiqueta a los nodos que pueden estar en la trayectoria  *
25     * aumentante. Si se reetiqueta al nodo pozo, se termina de *
26     * construir una trayectoria aumentante.                    */
27  public boolean encuentraTryctriaAumntnte ()  {
28      NodoEtiquetado s = (NodoEtiquetado) redFljo.nodoFuente ();
29      NodoEtiquetado t = (NodoEtiquetado) redFljo.nodoPozo ();
30      NodoEtiquetado s = (NodoEtiquetado) redFljo.nodoFuente ();
31      NodoEtiquetado t = (NodoEtiquetado) redFljo.nodoPozo ();
32      LinkedList  lista = new LinkedList ();
33      preprocesa();
34      s.etiqueta();
35      lista.add(s);
36      NodoEtiquetado u;
```

Listado 9.19: Trayectorias aumentantes etiquetando nodos 2/2

```
37          while (lista.size() > 0 && !(t.estaEtiquetado())) {
38              u = (NodoEtiquetado) lista.getLast();
39              lista.removeLast();
40              ListaDeAristas ady = (ListaBasicaDeAristas) u.getAdy();
41              ItrdrListaDeAristas iterAdy =
42                                      new ItrdrListaDeAristas(ady);
43              while (iterAdy.hasNext()) {
44                  ArcoDeFlujo arc = (ArcoDeFlujo) iterAdy.next();
45                  NodoEtiquetado v = (NodoEtiquetado) arc.getHacia();
46                  if (!v.estaEtiquetado())
47                      v.setPi(u); v.etiqueta(); lista.add(v);
48              }
49          }
50          if (t.estaEtiquetado())  {
51              construyeTryctria();
52              return true;
53          }
54          return false;
55      }
56 }
```

Correctez de la especialización con etiquetamiento de nodos

Se demuestra que las etiquetas de distancia aplicadas a los nodos son correctas y que encuentra correctamente los arcos admisibles – ver [AMO93, pp.217-219] – por lo que las trayectorias aumentantes se construyen correctamente.

Complejidad de la especialización con etiquetamiento de nodos

El costo total de encontrar arcos admisibles y reetiquetando a los nodos es de $O(|V||E|)$, y se demuestra que entonces el número de aumentaciones en trayectorias aumentantes es de $O(|V| \cdot |E|)$. Como cada aumentación requiere de $O(|V|)$, el costo total en tiempo es de $O(|V^2| \cdot |E|)$.

9.3.6. Obtención de flujo máximo por preflujo

En la segunda familia de algoritmos, el incremento en el flujo en cada iteración se hace en base a empujar preflujo, esto es, ignorar la ley de conservación de flujo y empujar tanto flujo por un nodo como su capacidad lo permite. A los nodos cuya capacidad no está saturada se les denomina *nodos activos*. El exceso de flujo, sin embargo, se regresa, por lo que al final se preserva la ley de conservación de flujo. Este concepto fue presentado por Goldberg en [GT88]. Los algoritmos de preflujo trabajan de una manera más local que los de trayectorias aumentantes, fijándose en cada momento en el nodo alcanzado. El código para la clase abstracta se encuentra en el Listado 9.20.

Listado 9.20: Clase abstracta para flujo máximo por preflujo 1/4

```
1  public abstract class PorPreflujoAbstracta
2                          extends FlujoMaximoAbstracto {
3     /* Inicia la red de flujo. Adicionalmente produce un árbol    *
4      * BFS con la gráfica invertida, dando de esta manera la      *
5      * distancia más corta de cada nodo hacia el pozo.            */
6     public PorPreflujoAbstracta (RedFlujo redFljo) {
7         super (redFljo);
8         asignaEtiqsDistancia ();
9     }
10    /* Devuelve verdadero si el flujo actual es máximo.           */
11    protected boolean esMaximoFlujo () {
12        return !(tieneNodoActivo ());
13    }
14    /* Incrementa el flujo encontrando un nodo activo y empu-     *
15     * jando flujo desde él a través de un arco admisible.        */
16    protected void incrementaFlujo () {
17        NodoEtiquetado u = seleccionaNodoActivo();
18        ArcoDeFlujo a = u.arcoAdmisible();
19        if (a != null)
20            empuja (u, a);
21        else
22            reetiqueta(u);
23    }
24    /* Asigna el exceso de flujo para el vértice t como el        *
25     * flujo actual.                                              */
26    protected int valorDelFlujoMaximo () {
27        NodoEtiquetado t = (NodoEtiquetado) redFljo.nodoPozo();
28        flujoMaximo = t.obtnExceso();
29        return flujoMaximo;
30    }
31    /* Asigna a cada nodo la distancia mínima desde él al nodo    *
32     * t. Utiliza bfs en la red de flujo invertida.               */
33    protected void asignaEtiqsDistancia () {
34        GraficaFlujo tmp = redFljo.redFlujo().invierte();
35        NodoEtiquetado tR = (NodoEtiquetado) redFljo.nodoPozo();
36        ArrayList nodos = tmp.obtenNodos();
37        ArrayList nodosR = redFljo.redFlujoResidual().obtenNodos();
38        NodoEtiquetado   t = (NodoEtiquetado) nodos.get(tR.getPos());
39        ExploracionBFS bfs = new ExploracionBFS(tmp);
40        bfs.exploracionGenerica (t);
41        for (int i = 0; i < nodos.size(); i++) {
42            if (nodos.get(i) == null)
43                continue;
44            NodoEtiquetado v  = (NodoEtiquetado) nodos.get(i);
45            NodoEtiquetado vR = (NodoEtiquetado) nodosR.get(i);
46            vR.setD(v.getD());
47        }
48    }
```

Listado 9.20: Clase abstracta para Flujo Máximo por preflujo 2/4

```
49       /* Empuja un preflujo para todos los nodos v tales que       *
50        * exista un arco s→v. También registra la distancia desde   *
51        * s a v.                                                    */
52     protected void preprocesa () {
53         NodoEtiquetado s = (NodoEtiquetado) redFljo.nodoFuente();
54         ListaBasicaDeAristas ady = (ListaBasicaDeAristas) s.getAdy();
55         if (ady.isEmpty()) {
56            s.setD(redFljo.redFlujoResidual().obtenNodos().size()-1);
57            return;
58         }
59         ItrdrListaDeAristas iterAdy = new ItrdrListaDeAristas(ady);
60         while (iterAdy.hasNext()) {
61             ArcoDeFlujo   sv = (ArcoDeFlujo) iterAdy.next();
62             NodoEtiquetado v = (NodoEtiquetado) sv.getHacia();
63             int            c = sv.getCapacidad();
64             s.dismnyeCapacidad(v, c);
65             v.aumentaExceso(c);
66             s.dismnyeExceso(c);
67             agregaNodoActivo(v);
68             if (v.conectadoCon(s))
69                 v.incrmntaCapacidad(s, c);
70             else
71                 v.conectaCon(s, c);
72             corrigeFlujoEnRedFlujo(s, v, c);
73         }
74         s.setD(redFljo.redFlujoResidual().numNodos());
75     }
76       /* Empuja flujo desde el nodo u a través del arco a.          */
77     protected void empuja (NodoEtiquetado u, ArcoDeFlujo a)  {
78         NodoEtiquetado v = (NodoEtiquetado) a.getHacia();
79         int        delta = u.obtnExceso();
80         if (delta > a.getCapacidad())
81             delta = a.getCapacidad();
82         else
83             eliminaNodoActivo(u);
84         if (v != (NodoEtiquetado) redFljo.nodoPozo()
85             && v != (NodoEtiquetado) redFljo.nodoFuente()) {
86             agregaNodoActivo(v);
87         }
88         u.dismnyeCapacidad(v, delta);
89         v.aumentaExceso(delta);
90         u.dismnyeExceso(delta);
91         if (v.conectadoCon(u))
92             v.incrmntaCapacidad(u, delta);
93         else
94             v.conectaCon(u, delta);
95         corrigeFlujoEnRedFlujo(u, v, delta);
96     }
```

Listado 9.20: Clase abstracta para Flujo Máximo por preflujo 3/4

```
 97       /* Si u no tiene arco admisible, reetiqueta la distancia     *
 98        * desde u, incrementándola en al menos una unidad.          */
 99     protected void reetiqueta (NodoEtiquetado u) {
100         ListaBasicaDeAristas ady = (ListaBasicaDeAristas)
101                                                 u.getAdy();
102         if (ady.isEmpty())
103             return;
104         ItrdrListaDeAristas iterAdy = new ItrdrListaDeAristas(ady);
105         eliminaNodoActivo(u);
106         ArcoDeFlujo   uv = (ArcoDeFlujo) iterAdy.next();
107         NodoEtiquetado v = (NodoEtiquetado) uv.getHacia();
108         int              d = v.getD();
109         while (iterAdy.hasNext()) {
110             uv = (ArcoDeFlujo) iterAdy.next();
111             v  = (NodoEtiquetado) uv.getHacia();
112             if (v.getD() < d)
113                 d = v.getD();
114         }
115         u.setD(d + 1);
116         agregaNodoActivo(u);
117     }
118       /* Incrementa el flujo a través del arco u→v en delta          *
119        * unidades.                                                   */
120     protected void corrigeFlujoEnRedFlujo(NodoDeFlujo u,
121                                     NodoDeFlujo v, int delta) {
122         ArrayList nodosRedFlujoResidual = redFljo.
123                               redFlujoResidual().obtenNodos();
124         ArrayList nodosRedFlujo = redFljo.redFlujo().obtenNodos();
125         int indiceU = u.getPos();
126         int indiceV = v.getPos();
127         NodoEtiquetado u1 = (NodoEtiquetado)
128                               nodosRedFlujo.get(indiceU);
129         NodoEtiquetado v1 = (NodoEtiquetado)
130                               nodosRedFlujo.get(indiceV);
131         ArcoDeFlujo a = (ArcoDeFlujo) u1.getArco(v1);
132         if (a != null)
133             a.aumentaFlujo(delta);
134         else {
135             a = (ArcoDeFlujo) v1.getArco(u1);
136             if (a != null)
137                 a.dismnyeFlujo(delta);
138         }
139     }
140       /* Devuelve un nodo que tenga exceso de flujo mayor que 1.   */
141     protected abstract NodoEtiquetado seleccionaNodoActivo();
```

Listado 9.20: Clase abstracta para Flujo Máximo por preflujo 4/4

```
142      /* Elimina a un nodo del conjunto de nodos activos.          */
143     protected abstract void eliminaNodoActivo (NodoEtiquetado u);
144      /* Agrega al nodo u al conjunto de nodos activos.             */
145     protected abstract void agregaNodoActivo (NodoEtiquetado u);
146      /* Decide si la red de flujo tiene algún nodo activo.        */
147     protected abstract boolean tieneNodoActivo ();
148 }
```

Correctez para la familia de preflujo

No es difícil demostrar la correctez de esta subfamilia en términos de que se están construyendo trayectorias aumentantes, pero en lugar de en cada iteración construirlas completas, se procede a ir creciendo la trayectoria aumentante en cada iteración, aprovechando subtrayectorias comunes.

Complejidad para la familia de preflujo

Examinando en detalle – ver [AMO93, págs. 221-229] – el número de veces que se examina cada nodo, el número de veces que se modifican las etiquetas de las distancias y el número de veces que manda preflujo, se obtiene una cota para esta subfamilia de $O(|V|^2 \cdot |E|)$.

9.3.7. Especialización con una cola para los nodos activos

La primera especialización de esta subfamilia es aquella donde la lista de nodos activos funciona como una cola, de acá su nombre de FIFO (*first in first out*). El código correspondiente a esta especialización se encuentra en el Listado 9.21.

Listado 9.21: Especialización que empuja flujo a través de nodos activos de una cola 1/2

```
1 public class PreflujoFIFO
2              extends PorPreflujoAbstracta {
3      /* Guarda a los nodos activos a través de los cuáles se va   *
4       * a empujar flujo.                                           */
5     protected LinkedList lista;
6      /* Construye la red de flujo e inicializa la cola para       *
7       * registrar en ella a los nodos activos.                     */
8     public PreflujoFIFO (RedFlujo n) {
9         super(n);
10        cola = new LinkedList();
11    }
```

Listado 9.21: Especialización que empuja flujo a través de nodos activos de una cola 2/2

```
1     /* Agrega el nodo a la cola, si es que no está ya en ella.  */
2    protected void agregaNodoActivo (NodoEtiquetado u) {
3        if (cola.indexOf(u) == -1)
4            cola.addFirst(u);
5    }
6     /* Elimina al primer nodo de la cola.                         */
7    protected void eliminaNodoActivo (NodoEtiquetado u) {
8        cola.removeLast();
9    }
10    /* Selecciona al último nodo de la cola.                      */
11   protected NodoEtiquetado seleccionaNodoActivo() {
12       int i = cola.size();
13       NodoEtiquetado n = (NodoEtiquetado) cola.get(i - 1);
14       return n;
15   }
16    /* Indica si la cola de nodos activos está vacía.             */
17   protected boolean tieneNodoActivo () {
18       int i = cola.size();
19       return (i > 0);
20   }
21 }
```

Correctez para preflujo con cola

En esta especialización, el proceso una vez que elige a un nodo para empujar flujo a través de él, seguirá empujando flujo hasta que el exceso sea 0 o se reetiquete a ese nodo. Si el nodo sigue activo, se le manda al final de la cola. Es claro que esta manera de examinar a los nodos mantendrá activos a todos aquellos que lo sean, y la cola se vaciará cuando ya no haya ningún nodo activo. Por lo que dada la correctez de la superclase, tenemos también la correctez de esta especialización.

Complejidad para preflujo con cola

Se puede demostrar que el número de veces que se ejecuta el ciclo exterior de la superclase es a lo más $2|V|$, utilizando para ello análisis amortizado. En cada ciclo se examina a cada nodo a lo más una vez y se empuja preflujo por el nodo también a lo más una vez. Por lo que esta especialización tiene un costo de $O(|V|^3)$, que cuando el número de arcos es superior al número de nodos – la mayoría de los casos en una red de flujo – está acotada por arriba por $O(|V|^2 \cdot |E|)$.

9.3.8. Especialización de preflujo por etiquetas mayores

La segunda especialización en esta subfamilia es la que trabaja etiquetando a los nodos que están a máxima distancia. Manda el flujo desde los nodos activos con máxima distancia d^* hacia aquéllos con distancia $d^* - 1$, y de éstos a los que están a distancia $d^* - 2$ y así sucesivamente hasta que no haya más nodos sin etiquetar. El algoritmo termina cuando el único nodo que le queda para trabajar con él es el nodo pozo. Los nodos se almacenan en pilas o colas, así que seleccionarlos toma tiempo $O(1)$. En el Listado 9.22 se encuentra el código para esta especialización.

Listado 9.22: Especialización de preflujo desde distancia máxima 1/2

```
 1 public class PreflujoDesdeMaximaDistancia
 2                                 extends PorPreflujoAbstracta {
 3       /* Contiene las listas para las distintas distancias desde   *
 4        * el nodo origen. En la i–ésima lista se encuentran los      *
 5        * nodos a distancia i.                                      */
 6     private ArrayList arregloDeListas = new ArrayList();
 7       /* Marca el nivel con el que está trabajando el algoritmo.  */
 8     private int  nivel = 0;
 9       /* Instala la red de flujo, inicia los nodos activas y crea *
10        * las primeras listas de distancia.                         */
11     public PreflujoDesdeMaximaDistancia (RedFlujo redFljo) {
12         super(redFljo);
13         creaListasDistancia();
14     }
15       /* Selecciona a un nodo activo del conjunto de nodos ac–     *
16        * tivos. Selecciona a un nodo de la lista de listas con     *
17        * el máximo índice.                                         */
18     protected NodoEtiquetado seleccionaNodoActivo  ()  {
19         LinkedList lista = (LinkedList) arregloDeListas.get(nivel);
20         NodoEtiquetado v = (NodoEtiquetado) lista.getLast();
21         return v;
22     }
23       /* Dice si queda algún nodo activo.                          */
24     protected boolean tieneNodoActivo ()  {
25         return (nivel > 0);
26     }
27       /* Crea las listas de nivel donde cada nodo es incluido en   *
28        * el nivel que corresponde a su distancia al origen.        */
29     private void creaListasDistancia ()  {
30         arregloDeListas = new ArrayList();
31         for (int i = 0;
32              i <= (2 * redFljo.redFlujo().numNodos()); i++)  {
33             LinkedList  lista = new LinkedList();
34             arregloDeListas.add(lista);
35         }
36     }
```

Listado 9.22: Especialización de preflujo desde distancia máxima 2/2

```
37      /* Agrega un nodo activo a la lista que corresponde a su      *
38       * distancia al origen.                                      */
39     protected void agregaNodoActivo (NodoEtiquetado u)  {
40         int d = u.getD();
41         LinkedList lista = (LinkedList) arregloDeListas.get(d);
42         if (!u.enListaD())  {
43             u.ponEnListaD(true);
44             lista.addLast(u);
45             if (d > nivel)
46                 nivel = d;
47         }
48     }
49      /* Elimina al nodo u de la lista correspondiente a su         *
50       * nivel.                                                    */
51     protected void eliminaNodoActivo (NodoEtiquetado u)  {
52         int d = u.getD();
53         LinkedList lista = (LinkedList) arregloDeListas.get(d);
54         lista.removeLast();
55         u.ponEnListaD(false);
56         if (lista.size() == 0)  {
57             int i = nivel;
58             while (i > 0 && ((LinkedList) arregloDeListas.
59                                          get(i)).size() <= 0)
60                 i--;
61             nivel = i;
62         }
63     }
64 }
```

Correctez de preflujo por elección de etiquetas

Aunque con este método de empuje de preflujo no se consideran en realidad todos los arcos (en eso radica la reducción de su complejidad), se puede demostrar fácilmente que los que se consideran son aquellos que mueven más flujo, y como lo que se desea es maximizar el flujo, los arcos que quedaron fuera aportan menos que aquéllos que fueron considerados en su lugar.

Complejidad de preflujo por elección de etiquetas

El empujar preflujo a través de los nodos con las etiquetas más altas evita el empujar preflujo por arcos que tienen poca capacidad. Se puede demostrar que esta especialización empuja preflujo $O(|V|^2 \cdot |E|^{1/2})$ – se presenta en [AMO93, págs. 234-237] – pero es una demostración elaborada que no tiene caso en esta breve exposición.

Con este resultado, tenemos que esta especialización tiene un costo de $O(|V|^2 \cdot |E|^{1/2})$,

inferior a la dada como cota de la subfamilia.

9.3.9. Otros enfoques para el problema de máximo flujo

En [AMO93] en esta familia de algoritmos (de complejidad polinomial) se listan otros dos algoritmos que utilizan un valor escalado, ya sea para encontrar las trayectorias aumentantes o para enviar preflujo. Si bien se pueden hacer caber en nuestra clasificación, requiere de transformaciones en el control de los ciclos de los algoritmos, y por eso no los incluimos acá.

9.4. Conclusiones

De las tres familias que presentamos en este capítulo, dos de ellas nuevamente tienen que ver con gráficas. Esto se debe a que el trabajo con gráficas responde a patrones intuitivos que permiten de manera más sencilla dilucidar una estrategia genérica con determinado fin. En el caso del apareamiento exacto de texto, la estrategia genérica es sencilla y también, de cierta manera, intuitiva.

Estamos convencidos que este tipo de abstracción es un auxiliar importante no solamente en cuanto a la comprensión de los distintos algoritmos, sino también en lo referente a pensar novedosas especializaciones que pudieran aportar, en algunos casos, mejores algoritmos.

Conclusiones

Conclusiones

Las Ciencias de la Computación han estado, desde sus inicios, preocupadas con la abstracción de procesos y cómputos. En primera instancia esta abstracción se dio – y se sigue dando – en términos de lo que significa y representa un cómputo o un proceso; con el advenimiento de la computadora como herramienta para llevar a cabo estos procesos o cómputos se desarrolla también un interés por abstraer la manera de diseñar y desarrollar estos cómputos. Así, a lo largo de la historia del diseño de soluciones usando una computadora se ha estudiado la programación funcional, la programación estructurada, lenguajes declarativos, y en los últimos años el paradigma de diseño y programación que ocupa de manera muy importante el interés de los profesionales de la computación: la orientación a objetos.

La orientación a objetos juega un papel importante en el diseño de soluciones y en la programación de las mismas. Como cualquier otro paradigma de programación, lo que importa realmente es el marco de referencia desde el que se diseña la solución, más que el lenguaje utilizado – finalmente todo estará ejecutado en lenguaje de máquina. Entre los objetivos fundamentales de la orientación a objetos se encuentran la reutilización y el encapsulamiento de datos y métodos relativos a aquéllos. La reutilización se logra con los mecanismos de herencia y polimorfismo, abstrayendo las características comunes a un grupo de problemas, diseñando una solución genérica que cubra a todos estos problemas – y posiblemente a algunos que pudieran, en un futuro, estar caracterizados de manera similar – y especializando posteriormente para cubrir aquellas características que distinguen a unos miembros de otros dentro de una misma clase.

El diseño orientado a objetos ha sido ampliamente difundido y perfeccionado, primero a través de la construcción de estructuras jerárquicas de clases, principalmente relacionadas con estructuras de datos abstractas y sus implementaciones, y desde 1994 con el concepto de patrones [GHJV94]. En este tenor, el objetivo de la reutilización directa de código, un interés siempre presente, está empezando a ser una realidad, cuando menos en aquellas áreas relacionados con la ingeniería de software.

La reutilización se puede dar en dos sentidos: ascendente, cuando se van construyendo conceptos sencillos y se usan para construir objetos más complejos, o bien cuando una solución más general se construye utilizando soluciones ya dadas para partes del problema; ésta es la forma usual en que se trabaja en matemáticas y en el área de algoritmos, donde el desarrollo de estas disciplinas, sin reutilización, es impensable; por otro lado, tenemos una reutilización descendente, que es la que se usa en este trabajo en cuanto a las demostraciones de correctez, que establece un esqueleto o marco de referencia general, y éste es reutilizado llenando algunos aspectos del mismo de maneras diferentes. Si bien en muchos problemas la orientación a

objetos ha satisfecho las aspiraciones de reutilización y abstracción, tanto desde el punto de vista ascendente como descendente, este paradigma, en cuanto a la reutilización y abstracción se refiere, no ha sido llevado a las demostraciones de correctez de algoritmos o al cálculo de su complejidad.

La reutilización en las demostraciones de correctez, en su modalidad descendente, se logra encontrando abstracciones adecuadas para familias de algoritmos, de tal manera que se pueda dar una demostración de correctez a nivel del algoritmo abstracto, sujeto a que en el nivel de especialización los métodos implementados presenten las propiedades exigidas por la demostración de correctez del algoritmo abstracto. Estas propiedades se pueden ver como pre y post-condiciones en el contexto de los métodos abstractos, pero constituyen las invariantes de la estrategia genérica, ya que se especifica, en cada punto de ésta qué es lo que debe cumplirse. El cálculo de la complejidad de los algoritmos concretos se consigue dando una fórmula general para el algoritmo abstracto, en términos de cada uno de los métodos por definir, para llenar únicamente estos últimos al llegar a las especializaciones para los algoritmos concretos.

En este trabajo observamos que puede haber más de una abstracción para un mismo algoritmo, esto es, éste puede pertenecer a familias distintas. Si lo que estamos haciendo es conformar una estrategia genérica que al especializarse resulte en un algoritmo concreto, un mismo resultado (la solución que provee el algoritmo) puede ser alcanzado por distintos caminos, utilizando distintas estrategias genéricas.

La aportación principal de este trabajo es la idea de estructurar una familia de algoritmos alrededor de un algoritmo abstracto, cuyas propiedades se heredan a las especializaciones, las cuáles representan cada una de ellas a un algoritmo concreto. Esta reutilización, como ya mencionamos, es descendente. Esta estructuración debe de ser en términos de métodos polimorfos, de los que se asumen determinadas propiedades. Se reutiliza tanto la demostración de la estrategia general en cada especialización, como la fórmula genérica para el cálculo de la clase de complejidad a la que pertenece cada especialización.

Un *algoritmo genérico*, en el contexto de este trabajo, usa la abstracción para ocultar detalles que distraen de los aspectos esenciales de la solución. También la usa para permitir nuevos métodos que pueden ser más específicos que los originales en comportamiento, y utilizan atributos que están disponibles únicamente para el objeto derivado, facilitando de esta manera el diseño de algoritmos más complejos para problemas diversos, aunque relacionados con el que resuelve el algoritmo genérico.

Esta manera de adoptar los principios de la orientación a objetos, y en particular del lenguaje Java, da un marco de referencia y un lenguaje para describir algoritmos donde las similitudes entre algoritmos pueden ser enunciadas directamente en una clase abstracta o interfaz de Java – aunque podrían quedar a nivel de pseudocódigo sin restarle precisión a nuestro enfoque.

Este enfoque va más allá, por supuesto, que otras técnicas para derivar algoritmos tales como la parametrización (como en algoritmos de análisis de flujo en [AMO93]), o la derivación de programas completos a partir de especificaciones, como se indica en [Gri89]. En el primer caso, la parametrización no da en realidad especializaciones distintas de un mismo algoritmo,

sino que posibilita trabajar sobre distintos tipos de datos; en el segundo caso, hay muchos problemas, como por ejemplo la exploración en gráficas, que el conocer *nada más* las pre y post-condiciones del problema general no da ninguna idea de cómo proceder. En el caso de la construcción de soluciones que se lleva a cabo, por ejemplo, en lenguajes funcionales, donde se construyen soluciones complejas a partir de incorporar soluciones sencillas[ASS96], cuáles soluciones sencillas incorporar o cómo hacerlo no es provisto *a priori*. En este enfoque damos una estrategia genérica, que deberá en ciertos puntos cumplir con determinadas invariantes, para que cualquier especialización dada cumpla con objetivos generales de esta estrategia. Más aún, el paradigma de los algoritmos genéricos *promueve* esta manera de pensar, alentando al programador a alejarse de los detalles y reusar soluciones para encontrar nuevas aplicaciones, y al mismo tiempo adquirir mejores técnicas de programación. Un atributo de este enfoque es que las especializaciones se comportan de acuerdo a la complejidad asintótica que presentan las aplicaciones dedicadas, ya que se tuvo mucho cuidado que las implementaciones respetaran este aspecto.

Tanto la demostración de correctez de un algoritmo como el cálculo de la clase de complejidad a la que pertenece se encuentra en la intersección de la computación con las matemáticas. En esta última disciplina, la reutilización se hace de abajo hacia arriba, construyendo poco a poco los conceptos básicos, y utilizándolos en niveles superiores. Nuestro enfoque es distinto, pues primero nos preocupamos por demostrar los conceptos superiores (el algoritmo abstracto), suponiendo que los conceptos más primitivos (la implementación concreta de los métodos) son correctos. Una gran ventaja de usar el paradigma de orientación a objetos en este contexto es que los métodos concretos, cada uno de ellos, tiene una implementación generalmente sencilla, donde es fácil demostrar que las propiedades que se requieren de ellos en efecto se presentan.

No importa, sin embargo, si demostramos correctez de abajo hacia arriba o de arriba hacia abajo: la intersección con matemáticas es siempre uno de los aspectos que dificultan – y al mismo tiempo enriquecen – el desarrollo de la disciplina. Nuestra propuesta permite la reutilización de la infraestructura matemática construida para el algoritmo abstracto, tanto en lo relativo a las demostraciones de correctez como al cálculo de la complejidad.

En la actualidad los estudiantes de computación están trabajando cada vez más a niveles altos de abstracción. Los lenguajes de programación que están aprendiendo manejan conjuntos de abstracciones a partir de cierto momento. Este enfoque para estudiar y diseñar algoritmos resulta ser un puente entre el tipo de programación que están llevando a cabo y el área de algoritmos, puente que no existe hoy en día.

Queda por definir de manera más precisa qué es un algoritmo genérico, y qué es lo que identifica a una familia de algoritmos como candidatos a conformar un algoritmo genérico. Esta definición formal pudiera tener consecuencias en la teoría de algoritmos y en las metodologías para el diseño de éstos.

Otra de las vertientes a seguir es el examinar nuevas familias de algoritmos, con el objetivo de reconocer propiedades unificadoras en las mismas. Las familias que nos interesan incluyen algoritmos de planaridad en gráficas, ordenamientos más generales que funcionen por intercambio, extender el algoritmo genérico para máximo flujo en redes y estudiar algoritmos sobre apareamientos en gráficas.

Apéndices

Apéndice A

Análisis de algoritmos

En este apéndice presentamos conceptos generales de complejidad de algoritmos, como las clases de complejidad y la relación entre ellas, cómo "medir" un algoritmo y algunas fórmulas de recurrencias útiles en el análisis de algoritmos.

A.1. Conceptos generales

Este material está enfocado a definir y evaluar soluciones constructivas a problemas bien planteados. Para ello es conveniente definir de manera precisa lo que queremos decir con problema, solución y costo de una solución.

Un *problema* se define mediante un conjunto de entradas posibles, y para cada una de esas entradas, un conjunto de soluciones posibles. Define una relación entrada/salida, especificándose cuál es la entrada que se tiene y cuál es la salida que se desea obtener.

Una *ejemplar*[1] *de un problema* es una entrada específica para un cierto problema. En particular, por proceso de solución entendemos algoritmo. Un algoritmo es *correcto* si para cualquier ejemplar del problema, la ejecución termina y produce la salida correcta.

Definición A.1 Un *algoritmo* es un procedimiento bien definido para resolver un problema, y que tiene las siguientes características:

Entrada: El algoritmo trabaja a partir de una entrada (o *datos*). Excepcionalmente, el algoritmo trabajará a partir de 0 entradas.

Salida: El algoritmo produce una salida (*resultado*), que corresponde a la solución del problema planteado.

Secuencia finita de pasos: El algoritmo define una secuencia de pasos cuya ejecución transforma a la entrada en la salida.

Definición precisa: Cada uno de estos pasos debe estar bien definido y tener un nivel adecuado de detalle.

Terminación: El algoritmo debe siempre terminar su ejecución proporcionando una solución correcta.

Los algoritmos se escriben en un lenguaje informal, llamado *seudo-código*, que muchas veces esconde ciertos niveles de detalle. La diferencia entre un algoritmo y un programa es que el programa está escrito en un lenguaje de programación, en el que no se puede obviar absolutamente nada respecto a los detalles de implementación. Si se utiliza directamente

[1] En inglés *instance*, de lo que *instancia* no es una traducción adecuada.

un lenguaje de programación, en ocasiones los detalles de implementación se esconden en métodos que aportan simplicidad al algoritmo, suponiendo que en esos métodos se cumplen restricciones dadas para que el análisis de los algoritmos sea válido.

A.2. Analizando algoritmos

Nuestro principal interés es el de diseñar algoritmos *correctos* y *eficientes*. La *eficiencia* del algoritmo corresponde a una medida de qué tan bueno es el algoritmo para resolver un problema dado. El análisis de un algoritmo consiste en, una vez demostrado que el algoritmo es correcto, predecir la cantidad de recursos que el algoritmo utiliza. Muchos de estos recursos son en teoría (en el contexto de las Máquinas de Turing) infinitos, como el tiempo o el espacio, pero en la práctica no es así. Los recursos más relevantes que se toman en cuenta hoy en día son:

Tiempo: Queremos tener una estimación del tiempo que se va a tardar el algoritmo, como *función del tamaño de la entrada*. Usaremos la convención de medirlo contando el número total de operaciones elementales que involucra, donde una operación elemental es una suma, resta, multiplicación, división, comparación de dos números, o una asignación a una variable de un número[2]. A esta medida la denotaremos con $T(n)$, una función del tiempo de ejecución del algoritmo, en términos del tamaño de la entrada n, . Es importante hacer notar que estamos suponiendo que cada uno de estos pasos primitivos toma una cantidad constante de tiempo, independientemente del tamaño de la entrada, y que es el mismo para todas las operaciones. Esto en general no va a ser cierto, aunque justificaremos el porqué de este enfoque.

Espacio: El análisis de algoritmos está relacionado, de alguna manera, con las máquinas de Turing, donde el espacio (la memoria, el tamaño de la cinta) es teóricamente infinito. Sin embargo esto no es cierto en las computadoras actuales, por lo que es relevante también medir la cantidad de memoria, en función del tamaño de la entrada, que se requiere para ejecutar un determinado algoritmo. Aún cuando la cantidad de memoria que puede tener una computadora está creciendo mucho, los tiempos de acceso a la misma no se reducen en la misma proporción que los tiempos de procesador. En ocasiones se tendrá que hacer un balance entre tiempo y espacio, ya que para reducir el tiempo se requerirá de más espacio y viceversa.

Comunicaciones: La comunicación entre computadoras es de alto costo y requiere de infraestructura compleja. Muchos algoritmos se miden en términos de la cantidad de comunicación que requieren, pues esto puede convertirse en un cuello de botella importante que debe ser tomado en cuenta. Este aspecto es cada día más importante, con el planteamiento de sistemas colaborativos, paralelos y/o distribuidos.

Circuitos: Cuando se está diseñando alguna arquitectura específica que implemente a un cierto algoritmo (como pudiera ser la unidad aritmética que multiplique dos números), es importante el costo que el hardware pueda tener y que estará dado por el número

[2]Este modelo es, básicamente, el que se conoce como el *Modelo RAM de cómputo*, que consiste de una computadora abstracta en la cual cada una de estas operaciones tiene el mismo costo, no hay diferencia en el costo o tiempo entre tener accesos a memoria RAM o cache, y la cantidad de memoria es, en principio, infinita.

de compuertas que se requieren, por el área que ocupe en la tableta y otros recursos de interés.

Tamaño del programa: Cuando se están "quemando" programas para microprocesadores, o procesadores dedicados, el área que ocupa la tableta es muy importante, y en cada tableta cabe una cantidad fija de memoria. El programa debe grabarse en esa cantidad de memoria, y un solo byte más que se necesite puede querer decir otra tableta completa de memoria, por lo que es muy importante hacer que el programa ocupe no más de un cierto número de bytes.

Muchas veces el tamaño de la entrada va a estar dado por el número de datos en ella. Sin embargo si deseamos, por ejemplo, hacer la multiplicación de enteros arbitrariamente grandes, esta multiplicación necesariamente tomará más tiempo conforme los números crezcan en tamaño. Por lo tanto, no sería razonable suponer que el tamaño de la entrada es p, donde p es el número de enteros a multiplicar. En este caso, consideramos en el tamaño de la entrada al número de dígitos del entero (o el número de bits en su representación binaria) con lo que el parámetro de tamaño de la entrada sería $p \times m$ donde m es el número de dígitos en cada entero. De cualquier manera, deseamos poder especificar el tiempo de ejecución de un algoritmo en términos del tamaño de la entrada, aún cuando tengamos que especificar qué es lo que estamos considerando como *tamaño de la entrada*, que dependerá del problema en cuestión.

Para ejemplificar veamos uno de los problemas clásicos en análisis de algoritmos, que es el de ordenar una lista de n valores, suponiendo que esos valores son números que pueden ser comparados en un solo paso. Este problema es típico pues existen muchos algoritmos que resuelven este problema, unos más eficientes que otros, y algunos muy recomendables, dependiendo de lal ejemplar del problema que se desee resolver. Empecemos por enunciar el problema:

Problema: Ordenamiento por inserción.

Entrada: Una sucesión de números $A = \langle a_1, a_2, \ldots, a_n \rangle$.

Salida: Una permutación de A, $A' = \langle a'_1, a'_2, \ldots, a'_n \rangle$, tal que se cumple $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Algoritmo: (se encuentra en el Listado A.1.)

El algoritmo supone, en cada momento, que los números en las posiciones 1 a $j - 1$ se encuentran bien ordenados. Toma entonces al elemento en la posición j y lo va a insertar donde le corresponde entre los elementos 1 a $j - 1$. Esto lo hace comparando al elemento en la posición j (al que se coloca en la variable *llave*) con cada uno de los elementos en las posiciones $j - 1$ a 1, hasta que se encuentre uno que es menor (supongamos que en la posición i, $1 \leq i \leq j - 1$) o bien se haya comparado contra el primero (en cuyo caso i queda valiendo 0). En el transcurso de cada iteración, al comparar al elemento j contra cada uno de los elementos en las posiciones $j - 1$ a i, se recorren a la derecha todos aquellos elementos que no son menores que el elemento que se desea insertar. Ya sea que la iteración se suspenda por haber comparado inclusive al elemento en el primer lugar, o por haber encontrado un elemento menor, el lugar que le corresponde a *llave* es el que se encuentra a la derecha (lugar $i + 1$).

Listado A.1: Ordenamiento por Inserción

```
1   void InsertSort(int A[1..n], int n) {
2       int i ← 0;
3       int j ← 0;
4       /* Acomodar a todos los elementos                           */
5       for(j = 2; j ≤ n; j++) {
6           llave ← A[j];
7           /* Se toma al j−ésimo elemento para insertarlo
8              en la parte ordenada A[1...j−1].                     */
9           i ← j−1;
10          while (i > 0 && A[i] > llave) {
11              /* Se recorren hacia j.                             */
12              A[i+1] ← A[i];
13              i ← i−1;
14          }
15          A[i+1] ← llave;
16             /* Se insertó en el lugar adecuado.                  */
17      }
18  } // InserSort
```

Para trabajar con este algoritmo debemos cubrir dos aspectos. El primero de ellos consiste en demostrar la correctez del algoritmo, esto es, que el algoritmo termina y produce la salida correcta frente a cualquier ejemplar del problema. La segunda consiste en hacer el análisis del algoritmo.

A.3. Correctez

Las demostraciones de correctez de un algoritmo generalmente se hacen por inducción en el número de veces que se repiten los enunciados del algoritmo, o si el algoritmo trabaja de manera recursiva, por una fórmula que contempla el costo de cada invocación. En este caso tenemos dos iteraciones, una anidada dentro de la otra. Debemos encontrar una *invariante*, que se refiere a predicados que deben ser verdaderos antes de que se inicie la ejecución (la base de la inducción) y al terminar la ejecución. Generalmente se demuestra que si en el ciclo k el predicado se evalúa a verdadero, con las operaciones que se ejecutan en el ciclo, el predicado seguirá siendo verdadero en el ciclo $k+1$ (paso inductivo).

No es tan difícil encontrar esa invariante, pues tiene que ver con la manera en que describimos la salida del algoritmo. En el caso del problema que estamos revisando, el de ordenar una secuencia de números, el predicado es, simplemente:

$$perm(A, A') \text{ y } \; a'_1 \leq a'_2 \leq \ldots \leq a'_n,$$

donde $perm(A, A')$ es un predicado que se evalúa a verdadero siempre que A' sea una permutación de A.

Ahora debemos expresar esta invariante en términos de los ciclos del algoritmo. Para ello nuestra variable para la inducción puede ser j, que es la que va a "contar" los ciclos. Entonces, nuestra demostración queda de la siguiente manera:

Teorema A.1 (Correctez del algoritmo) *Dada una sucesión de números $A = \langle a_1, a_2, \ldots, a_n \rangle$, al ejecutar el algoritmo de ordenamiento por inserción con esta sucesión como entrada, tenemos que la salida del algoritmo es la sucesión $A' = \langle a'_1, a'_2, \ldots, a'_n \rangle$ donde se cumple el siguiente predicado:*

$$perm(A, A') \ \wedge \ a'_1 \leq a'_2 \leq \ldots \leq a'_n$$

Demostración:

Haremos la demostración, como lo mencionamos, por inducción sobre el valor de j, la posición del elemento a insertar. La invariante del algoritmo es una pequeña transformación del predicado que define al ordenamiento y que debe cumplirse antes de empezar cada iteración.

$$perm(A, A') \ \wedge \ a'_1 \leq a'_2 \leq \ldots \leq a'_{j-1}$$

Si demostramos lo anterior para el inicio de cada iteración, al terminar el algoritmo - antes de que empiece la iteración $j = n + 1$ - se va a cumplir el predicado para toda la lista de elementos.

Base: La primera iteración se da para $j = 2$ – línea [5] del Listado A.1 – por lo que tenemos que verificar

$$perm(A, A') \ \wedge \ a'_1 \leq a'_{j-1=1}.$$

Es claro que $A' = A$ pues no hemos hecho nada, por lo que se cumple $perm(A, A')$. Y se cumplen las desigualdades, ya que sólo hay un elemento en el rango $[1 \ldots 1]$.

Paso inductivo: Supongamos como hipótesis de inducción que este predicado se cumple para valores de k con $1 < k \leq j - 1$ y veamos qué pasa cuando el algoritmo se ejecuta para $j \geq 2$.

i. Verifiquemos primero que $perm(A, A')$ se verifica, viendo si todos los elementos que se colocan en la sucesión pertenecen a la sucesión y que no se elimina a ninguno. Se colocan elementos en la sucesión en dos puntos del algoritmo, la línea [12], donde claramente se está nada más desplazando a un elemento que ya estaba en la sucesión; y en la línea [15], donde se coloca al contenido de la variable *llave*. Pero la única vez que se le asigna valor a la variable *llave* es en la línea [6], y el valor corresponde a alguno de los elementos de A, precisamente el del j-ésimo elemento. En la línea [6] copiamos el valor del elemento en el lugar j a la variable *llave*. En las líneas [10–14] lo que hacemos es recorrer a los elementos un lugar a la derecha. Examinemos las condiciones bajo las cuales se ejecuta esta iteración.

   a) Puede no ejecutarse ni una sola vez, pero dado que el valor menor de j para el que se ejecuta esta iteración es 2, si no se entra a la iteración es porque la segunda condición de la iteración no se cumple, esto es, $A[i] \leq llave$. Si esto sucede, La iteración termina con $i = j - 1$ y tenemos un hueco en la posición $i + 1 = j$ con $llave = A[j]$.

b) Se ejecuta la iteración y termina cuando se deja de cumplir la primera condición, esto es, con $i = 0$. Quiere decir que los elementos en las posiciones $j-1$ a 1 fueron copiados a la posición inmediata a su derecha. Excepto por la posición j, cada posición que se reescribe fue copiada antes, por lo que al salir de la iteración tenemos que los elementos que estaban en las posiciones 1 a $j-1$ ahora ocupan las posiciones 2 a j; el elemento en la posición 2 está repetido en la posición 1, y el elemento que estaba en la posición j está protegido en la variable *llave*. Podemos pensar que tenemos un espacio disponible en la posición $1 = i+1$.

c) Se ejecuta la iteración y termina cuando se deja de cumplir la segunda condición, esto es, $i > 0$ pero $A[i] \leq llave$. De manera similar al caso anterior, recorrimos los elementos desde la posición $i+1$ hasta la $j-1$ a las posiciones $i+2 \ldots j$, dejando un hueco en la posición $i+1$ y con el elemento que originalmente ocupaba la posición j protegido en la variable *llave*.

Al salir de la iteración, en cualquiera de los tres casos, se coloca el valor protegido en *llave* en la posición $i+1$, que es donde se hizo el hueco. El único elemento que no pertenece al arreglo y que se acomoda en él es *llave*, que copió el valor de un elemento del arreglo, aquel que se reescribió, por lo que no estamos agregando ningún elemento que no estuviera. Al recorrer los elementos, el único que se duplica es el último que se recorrió; pero el lugar duplicado es en el que se sobreescribe el valor de llave, por lo que en el arreglo quedan precisa y exactamente los elementos que estaban en él antes de empezar la j-ésima iteración.

$\therefore$ se cumple el predicado $perm(A, A')$.

ii. Nos falta demostrar que se cumple la invariante que involucra las desigualdades, suponiendo que se cumple para $1 < k < j$ y viendo qué es lo que sucede para j. Tenemos dos casos:

$$a_j \geq a_{j-1} \qquad \text{o bien} \qquad \text{(a)}$$
$$a_j < a_{j-1} \qquad \text{(b)}$$

Si se cumple (a), ya terminamos. Veamos por qué. *llave* contendrá el valor de $A[j]$. Llegamos a la línea [10] con $i = j-1$, por lo que la expresión $A[j-1] > llave$ es falsa y la ejecución continúa en la línea [15], sin modificar a i. Por lo que en la línea [15] tenemos $i+1 = (j-1)+1 = j$ y en *llave* tenemos el valor de $A[j]$. De todo esto, la asignación se convierte en $A[j] \leftarrow llave$, o sea que no modificamos nada. Y como por la hipótesis de inducción tenemos

$$a_1 \leq \ldots \leq a_{j-1};$$

como se cumple (a) tenemos $a_{j-1} \leq a_j$; juntando las dos desigualdades, tenemos:

$$a_1 \leq \ldots \leq a_{j-1} \leq a_j$$

que es lo que queremos demostrar.

Si se cumple (b) tenemos que al llegar a la línea [9] la i toma el valor de $j-1$. Por un razonamiento similar a los casos (b) y (c) del inciso i, salimos de la iteración con $0 \leq i \leq j-1$ y cumpliéndose que $a_k > llave$ para $i+1 < k < j$. Recordemos que los elementos se recorrieron un lugar a la derecha, y que en la línea [15] colocamos el

valor de *llave* en la posición $i+1$. Por la hipótesis de inducción, y como no tocamos a los elementos en las posiciones $[1 \ldots i]$, tenemos

$$a_1 \leq \ldots \leq a_i \quad \text{y} \quad a_{i+2} \leq \ldots \leq a_j.$$

Además, por la ejecución de la iteración en las líneas [10] a [14], sabemos que

$$a_i \leq llave = a_{i+1} \quad \text{y} \quad llave = a_{i+1} \leq a_k, \qquad i+2 \leq k \leq j.$$

Nuevamente, juntando estas cuatro desigualdades, tenemos

$$a_1 \leq \ldots \leq a_j,$$

que es lo que queríamos demostrar.

❏

A.4. Costo

Hagamos el análisis de este algoritmo, con algunas consideraciones adicionales.

Mejor caso: Se dice que estamos haciendo un análisis de *mejor caso* si suponemos que los datos se van a presentar de tal manera que tengamos que hacer el menor trabajo posible. En general, éste es un análisis fácil de hacer y poco útil, ya que surge la pregunta de qué tan válido puede ser el suponer que nuestros datos van a tener una distribución que justifiquen este análisis optimista. Para poder usar este tipo de análisis habrá que justificar plenamente que en efecto las instancias del problema que vamos a manejar, con una muy alta probabilidad, van a presentar el mejor caso, para poder medir el tiempo de ejecución de un algoritmo en estos términos. En el algoritmo del ordenamiento por inserción, el mejor caso se da cuando los datos vienen ya ordenados, por lo siguiente:

Ejecución de InsertSort para mejor caso:

1. Vamos a pasar por el enunciado [5] exactamente $n-1$ veces, ya que esto no depende de cómo vengan los datos.
   Vamos a ejecutar, entonces, los enunciados [6] y [9] cada uno, $n-1$ veces.
   Número de pasos: $2n-2$.
2. El enunciado [10] involucra 2 comparaciones, y lo vamos a ejecutar exactamente $n-1$ veces.
   Número de pasos: $2n-2$.
3. Como los datos vienen ordenados, la comparación $A[i] > llave$ va a ser inmediatamente falsa, por lo que ese bloque no se ejecutará ni una sola vez (únicamente la comparación, que ya la contamos en el inciso anterior).
   Número de pasos: 0.
4. Finalmente, el enunciado [15] se ejecuta también $n-1$ veces.
   Número de pasos: $n-1$.

Total para InsertSort:

$$T(n) = \quad \mathbf{5n - 5}$$

Deberemos considerar también el costo que tiene ejecutar el la línea [10]. Si consideramos una sola unidad de costo por la evaluación de la expresión booleana en este paso podemos decir que el tiempo de ejecución de mejor caso es de $5n - 5$ unidades.

Peor caso: Decimos que estamos haciendo un análisis de *peor caso* si suponemos que los datos vienen de tal forma que vamos a tener que ejecutar el máximo posible de pasos. Este tipo de análisis es muy común, pues nos proporciona una garantía del funcionamiento del algoritmo, una cota superior para el tiempo de ejecución del algoritmo: sabemos que nada peor puede suceder, y por lo tanto, no podemos tener un tiempo de ejecución mayor que el dado por el peor caso. Si no podemos garantizar la manera como se van a presentar los datos del problema, este tipo de análisis nos proporciona una cota superior de costo en tiempo de ejecución. En el algoritmo de ordenamiento por inserción, el peor caso se va a dar cuando los números vengan ordenados en forma descendente, pues en este caso vamos a tener que voltear la lista, insertando a cada uno de los elementos en el primer lugar del arreglo.

Ejecución de InsertSort para peor caso:

1. Al igual que en el análisis de mejor caso, vamos a ejecutar el bloque que corresponde al enunciado [5] exactamente $n - 1$ veces.

   Los enunciados [6] y [9] vamos a ejecutarlos, cada uno, $n - 1$ veces.

   Número de pasos: $2n - 2$.

2. La expresión booleana de la línea [10] la vamos a contar con una función que depende de j y de n, $f(j, n)$, y nos dice, para cada pareja, el número de veces que vamos a ejecutar este paso, que consiste en evaluar la expresión booleana:

   Número de pasos: j $\quad \boldsymbol{f(j,n)}$

Justificación:	j	$f(j,n)$
Para $j = 2$, la i empieza valiendo 1, por lo que se compara contra 0 y se comparan $A[1]$ con *llave*. Se decrementa a i y se vuelve a comparar con 0, pero esta vez la expresión booleana se evalúa a falsa y ya no se entra en el ciclo.	2	2
Para $j = 3$, en el peor caso, sale del ciclo porque la i tome el valor de 0, y como empezó con el valor 2 entra al ciclo 2 veces y evalúa la expresión booleana una vez más para poder salir del ciclo.	3	3
$\vdots$	$\vdots$	$\vdots$
En la última vuelta entra al ciclo $n - 1$ veces, cada una de ellas después de evaluar la expresión booleana. La debe evaluar una vez más para poder saltarlo.	n	n.

$$\begin{aligned} \sum_{j=2}^{n} f(j,n) &= 2 + 3 + \ldots + n \\ &= \tfrac{1}{2} \cdot n(n+1) - 1 \\ &= \boldsymbol{\tfrac{1}{2}(n^2 + n) - 1} \end{aligned}$$

3. Se entra a ejecutar los pasos [12] y [13], una vez menos que el número de evaluaciones de la expresión booleana en la línea [10].

   Número de pasos: $\frac{1}{2}n \cdot (n-1)$

   Total para la iteración interior: Número de pasos: $\boldsymbol{\frac{1}{2}(n^2 - n)}$

4. Por último, la línea [15] se ejecuta $n-1$ veces.

   Número de pasos: $\boldsymbol{n-1}$

Total para el peor caso:

$$\boldsymbol{T(n) = n^2 + 3n - 4}$$

¿Cómo sabemos que éste es realmente el peor caso? Porque independientemente de cómo venga la lista, el ciclo en la línea [10] del `while` se ejecuta a lo más j veces. Debemos notar que en el análisis de peor caso no es necesario describir una instancia específica que induzca el peor caso; es suficiente con probar que el algoritmo nunca excede la cota dada.

Caso promedio: Este es el análisis más difícil pero quizá el más útil, porque pretende predecir cómo se comporta el algoritmo "en general". La dificultad radica en que se debe determinar, antes de hacer cualquier otra cosa, la frecuencia de distribución de los datos. Se debe tener un análisis detallado que garantice que lo que estamos suponiendo como caso promedio, en efecto es el que va a ocurrir más frecuentemente, promediándose entonces casos mejores con casos peores. En este texto no nos detendremos con este tipo de análisis, ya que es más complicado y requiere de técnicas distintas y sofisticadas.

A.5. Orden de crecimiento

Dado que estamos simplificando el problema de medir, al considerar a todas las operaciones con el mismo costo, podemos simplificar aún más el análisis considerando únicamente órdenes de crecimiento de las funciones de tiempo de ejecución. Por ejemplo, para la función n^2+3n+5, cuando la n crece, los términos $3n$ y 5 van a aportar ya poco al valor de la función:

$\boldsymbol{n}$	$\boldsymbol{n^2+3n+5}$	$\boldsymbol{n^2}$	$\boldsymbol{n^2/f(n)}$
5	45	25	0.55
10	135	100	0.74
500	251,505	250,000	0.99
1,000	1,003,005	1,000,000	0.997

Como se puede observar en la tabla anterior, si nos interesan valores grandes para el tamaño de la entrada (n), con considerar únicamente el término más significativo tendremos una aproximación bastante buena al valor de la función. También ignoraremos, por razones similares y debido a que estamos considerando a todas las operaciones con un costo constante, los coeficientes de estos términos. A esta aproximación es a lo que llamamos la *complejidad* del algoritmo y como la misma, de hecho, agrupa a distintas funciones bajo una misma aproximación, decimos que estas aproximaciones definen clases de equivalencia, a las que denotamos por el momento con $\Theta(g(n))$ (más adelante formalizaremos esta notación). Es

decir, si $f(n) = 4n - 4$, $g(n) = n$, $f(n) \in \Theta(g(n))$. O si $f(n) = \frac{1}{2}n^2 + 3.5n - 4$, $f(n) \in \Theta(n^2)$, por lo que $g(n) = n^2$. Comparar la complejidad de distintos algoritmos, cuando la n es grande, nos va a proporcionar un buen método para decidir si un algoritmo es mejor que otro, sobre todo cuando estemos comparando algoritmos de distinta complejidad. Veamos algunos ejemplos que muestran lo que acabamos de comentar:

$f(n)$	**Complejidad**
$4n - 4$	$\Theta(n)$
$\frac{1}{2}n^2 + 3.5n - 4$	$\Theta(n^2)$
$3n^3 + 5n^2 - 2n + 1000$	$\Theta(n^3)$
$2^{n/2} + n^8$	$\Theta(a^n)$

Podemos asegurar que un algoritmo que tiene complejidad n^2 es más eficiente que uno cuya complejidad es n^3 independientemente de los coeficientes (insistimos: para n suficientemente grandes). Para corroborar esto veamos la gráfica de la Figura A.1.

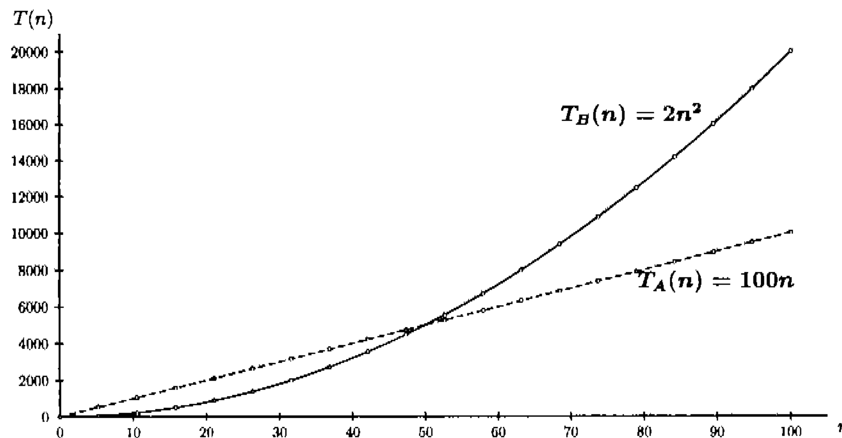


Figura A.1: Comparación de $T_a = 100n$ y $T_b = 2n^2$

Volviendo al ejemplo que vimos de ordenamiento por inserción, podemos decir que $n^2 + 3n - 4 \in \Theta(n^2)$ para el análisis de peor caso.

Estas medidas de complejidad nos dan una muy buena idea de qué tan bueno es un algoritmo, independientemente de la computadora en la que lo vayamos a ejecutar. Supongamos, por ejemplo, que tenemos una computadora que es capaz de ejecutar 1,000 operaciones por segundo. Veamos el tamaño de problemas que puede resolver en 1 segundo, suponiendo

distintas complejidades de nuestros algoritmos:

Complejidad del algoritmo	**Tamaño que se resuelve en**		
	1 *seg*	1 *min*	1 *hora*
n	1,000	6×10^4	3.6×10^6
$n \log n$	140	4893	2.0×10^5
n^2	31	244	1,897
n^3	10	39	153
2^n	9	15	21

Como se puede ver en la tabla anterior con las distintas funciones de complejidad, si tenemos un algoritmo de complejidad *lineal* ($\Theta(n)$), el tamaño de los problemas que podemos resolver al aumentar el tiempo es proporcional al aumento del tiempo. Con algoritmos de complejidad *polinomial* ($\Theta(n^k)$, $k > 1$) el número de casos que podemos resolver al aumentar el tiempo no se incrementa a la misma tasa que se incrementa n. En el caso de complejidad *exponencial* ($\Theta(a^n)$) la situación es aún más drástica, ya que incrementando el tiempo de ejecución millones de veces, sólo logramos resolver unos cuantos casos más.

Supongamos ahora que la velocidad de la computadora se incrementa 10 veces. Veamos la manera en que se modifica la tabla anterior:

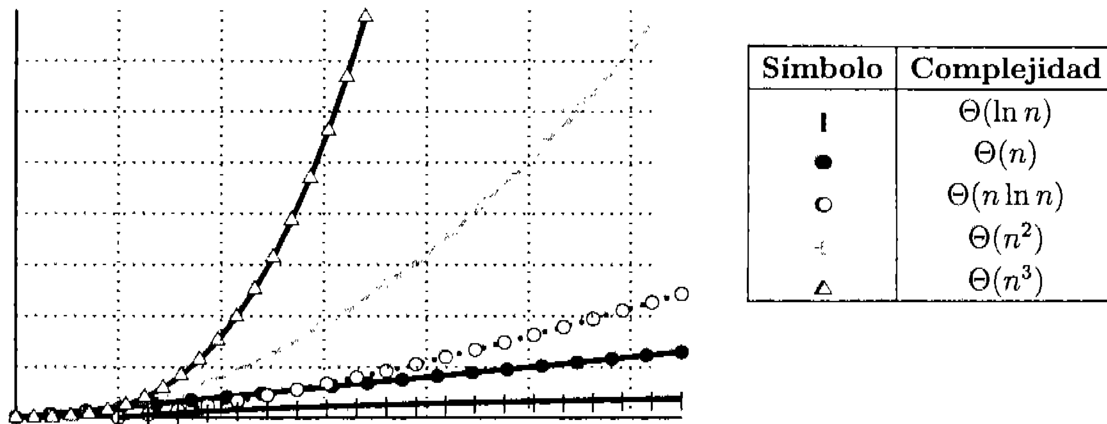
Complejidad del algoritmo	**Tamaño con velocidad original**	**Tamaño con velocidad incrementada**
n	s_1	$10 \times s_1$
$n \log n$	s_2	aprox. $10 \times s_2$
n^2	s_3	$3.16 \times s_3$
n^3	s_4	$2.15 \times s_4$
2^n	s_5	$s_5 + 3.3$

Todo esto que acabamos de mencionar es válido cuando la n es suficientemente grande, ya que si la n es pequeña los coeficientes y los términos menos significativos pueden tener impacto en el tiempo de ejecución que no se verá reflejado en la medida de complejidad, como se puede observar en la tabla anterior. Es muy importante tomar en cuenta, además, que cuando queramos comparar algoritmos con la misma medida de complejidad (ambos con complejidad $\Theta(n^2)$ por ejemplo) son precisamente los coeficientes y los términos menos significativos los que van a decidir, dependiendo del rango de valor para n, cuál de ellos es más eficiente. Una ventaja de la notación asintótica (sin *constantes* es que la evaluación de un algoritmo es independiente de la máquina, lenguaje, compilador, etc. Veamos la siguiente tabla:

n	$T_1(n) = 22n^2 + 10n + 175$	$T_2(n) = 2n^3 + n^2$	$\Theta(n^2)$	$\Theta(n^3)$
3	403	63	9	27
5	775	265	25	125
7	1,323	735	49	343
11	2,947	2,711	121	1,331
13	4,023	4,563	169	2,197
20	9,175	16,400	400	8,000
50	55,675	252,500	2,500	125,000
75	124,675	849,375	5,625	421,875

En ella podemos observar que para n pequeña ($n < 12$), mientras que en el cálculo preciso de $T(n)$ tenemos que $T_2(n) < T_1(n)$, la comparación de las medidas respectivas de complejidad no reflejan esto. Por ello queremos insistir en que el decidir si un algoritmo es mejor que otro utilizando para ello las medidas de complejidad tiene sentido únicamente si se cumple que los algoritmos estén en distintas clases de complejidad. Esto queda de manifiesto en la gráfica de la Figura A.2 donde se muestran distintas medidas de complejidad. Se puede ver que cerca del origen (para n pequeña) la relación entre las distintas medidas no se mantiene de la misma forma que cuando la n crece.

Figura A.2: Comparación de complejidad de algoritmos



Símbolo	Complejidad
I	$\Theta(\ln n)$
●	$\Theta(n)$
○	$\Theta(n \ln n)$
⸗	$\Theta(n^2)$
△	$\Theta(n^3)$

A.6. Notación asintótica

Supongamos que tenemos una función $f(n) \geq 0$ con dominio $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ que mide el tiempo de ejecución de un algoritmo.

Definición A.2 (Notación Θ:) $f(n) \in \Theta(g(n))$ si $\exists\ c_1,\ c_2\ >\ 0$ y n_0 tal que

$$0 \leq c_1 \cdot g(n) \ \leq \ f(n) \ \leq \ c_2 \cdot g(n), \qquad \forall n \ \geq \ n_0.$$

Ejemplos:

(a) Probar que $3n^2 + 5n + 22 \ \in \ \Theta(n^2)$.

Tenemos que encontrar c_1, c_2 y n_0 de la definición, por lo que establecemos la siguiente relación:

$$c_1 \cdot n^2 \ \leq \ 3n^2 + 5n + 22 \ \leq \ c_2 \cdot n^2;$$

dividiendo entre n^2 tenemos:

$$c_1 \ \leq 3 + \frac{5}{n} + \frac{22}{n^2} \ \leq \ c_2.$$

Para $n \geq 8$, tenemos

$$c_1 \ \leq \ 3 + \frac{5}{8} + \frac{22}{64} = \ 3 + \frac{5}{8} + \frac{11}{32} = \frac{96}{32} + \frac{20}{32} + \frac{11}{32} = \frac{127}{32} < 4.$$

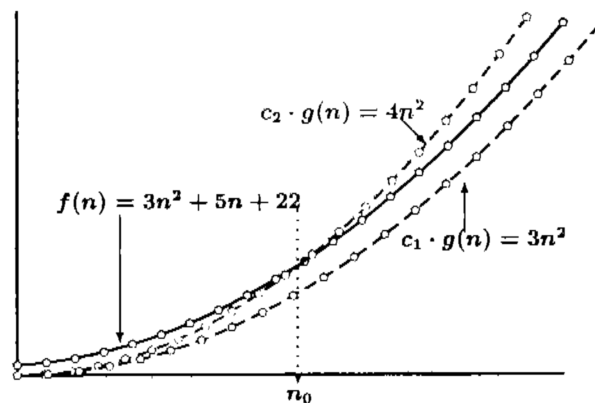
Por lo que si hacemos las siguientes asignaciones:

$$c_1 = 3, \qquad c_2 = 4, \qquad n_0 = 8,$$

tendremos la situación que se muestra en la Figura A.3, donde

$$3 \cdot n^2 \ \leq \ 3 \cdot n^2 + 5 \cdot n + 22 \ \leq \ 4 \cdot n^2 \qquad \text{para } n \geq 8$$

Figura A.3: $3n^2 + 5n + 22 \in \Theta(n^2)$



(b) Probar que $\frac{1}{2}n^2 - 3n \in \Theta(n^2)$

Similarmente al caso anterior, debemos encontrar c_1, c_2 y n_0 tales que se cumpla

$$c_1 \cdot n^2 \ \leq \ \frac{1}{2}n^2 - 3n \ \leq \ c_2 \cdot n^2, \qquad \text{para } n \ \geq \ n_0.$$

Trabajemos con esta última desigualdad.

$$c_1 n^2 \;\leq\; \frac{1}{2}n^2 - 3n \;\leq\; c_2 n^2.$$

Dividiendo entre el término con mayor exponente:

$$c_1 \;\leq\; \frac{1}{2} - \frac{3}{n} \;\leq\; c_2.$$

Para $n \;\geq\; 12$:

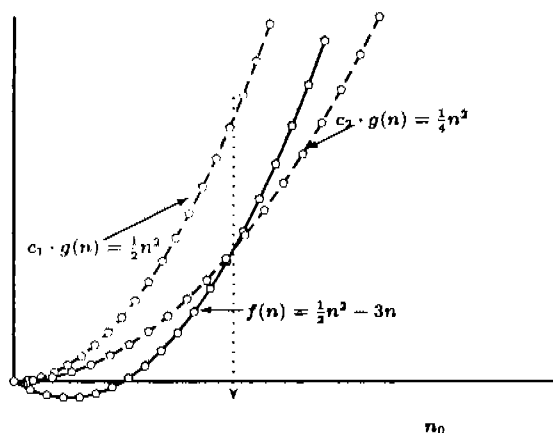
$$c_1 \;\leq\; \frac{1}{2} - \frac{3}{12} = \frac{1}{2} - \frac{1}{4} = \frac{1}{4}.$$

De donde:

$$c_1 = \frac{1}{4}, \qquad c_2 = \frac{1}{2}, \qquad \forall n \;\geq\; n_0 \;=\; 12.$$

Esta relación se muestra en la Figura A.4.

Figura A.4: $\frac{1}{2}n^2 - 3n \in \Theta(n^2)$



Definición A.3 (Notación O:) Decimos que $f(n) \in O(g(n))$ - se lee $f(n)$ es de *orden de* $g(n)$ y se escribe frecuentemente como $f(n) = O(g(n))$ - si $\exists \;\; c > 0 \;\; \text{y} \;\; n_0$ tales que

$$f(n) \;\leq\; c \cdot g(n), \qquad\qquad \forall n \;\geq\; n_0.$$

La notación O nos sirve para dar una cota superior para nuestro algoritmo, garantizándonos que no se va a tardar más de eso, para n suficientemente grande. Es importante aclarar que esta cota debe estar lo más ajustada posible pues lo que nos interesa es tener una buena idea de la complejidad del algoritmo.

Definición A.4 (Notación Ω:) $f(n) \in \Omega(g(n))$ si $\exists \;\; c > 0 \;\; \text{y} \;\; n_0$ tales que

$$c \cdot g(n) \;\leq\; f(n), \qquad\qquad \forall \;\; n \geq n_0.$$

La notación Ω nos da una cota inferior, que como en el caso de O intentaremos que esté lo más ajustada posible; es decir, que nos dé la mayor cantidad de información posible.

Tenemos otras dos notaciones relacionadas con O y Ω pero que hacen la desigualdad estricta:

Definición A.5 (Notación o:) Decimos que $f(n) \in o(g(n))$ – se lee $f(n)$ es *o chica de* $g(n)$ – si $\forall\ c > 0$ existe $n_0 > 0$ tales que

$$0 \leq f(n) < c \cdot g(n), \qquad \forall n \geq n_0.$$

Esta notación nos permite hablar de una función que crece *estrictamente* más lento que otra.

Definición A.6 (Notación ω:) $f(n) \in \omega(g(n))$ si $\forall\ c > 0$ existe $n_0 > 0$ tales que

$$0 \leq c \cdot g(n) < f(n), \qquad \forall\ n \geq n_0.$$

Esta notación nos permite hablar de una función que crece *estrictamente* más rápido que otra. Veamos algunos ejemplos de órdenes de complejidad con el nombre que se les asocia:

$O(1)$	constante
$O(\log n)$	logarítmico
$O(n)$	lineal
$O(n \log n)$	n log n
$O(n^2)$	cuadrático
$O(n^3)$	cúbico
$O(n^k), k \geq 2$	polinomial
$O(a^n)$	exponencial

Los distintos órdenes de complejidad, como ya mencionamos, particionan a las funciones en clases de equivalencia, una por cada orden de complejidad (lo mismo se aplica para las distintas notaciones). Como estamos hablando de pertenencia a clases de equivalencia tenemos la propiedad expresada en el Lema A.1:

Lema A.2

$$g(n) \in \Theta(f(n)) \iff f(n) \in \Theta(g(n))$$

Demostración:

Supongamos que $g(n) \in \Theta(f(n))$. Eso quiere decir que existen $c_1,\ c_2 > 0$ y n_0 tales que

$$c_1 f(n) \leq g(n) \leq c_2 f(n), \quad \forall n \geq n_0.$$

Tomando la primera desigualdad y dividiendo ambos términos entre c_1 tenemos:

$$f(n) \leq \frac{1}{c_1} g(n),$$

y tomando la segunda desigualdad y dividiendo entre c_2 tenemos:

$$\frac{1}{c_2} g(n) \leq f(n).$$

Combinando nuevamente ambas desigualdades, tenemos

$$\frac{1}{c_2} g(n) \leq f(n) \leq \frac{1}{c_1} g(n),$$

que es precisamente la definición de $f(n) \in \Theta(g(n))$. ❏

A.6.1. Consideraciones prácticas

Los órdenes de complejidad están ordenados entre sí. A continuación listamos algunos de ellos que ocurren con frecuencia, calificándolos de buenos o malos:

$$\left.\begin{array}{l} \Theta(1) \\ \Theta(\log n) \\ \Theta(n) \\ \Theta(n \log n) \\ \Theta(n^2) \\ \Theta(n^3) \\ \vdots \end{array}\right\} \textit{bueno}$$

$$\left.\begin{array}{l} \vdots \\ \Theta(a^n) \\ \vdots \end{array}\right\} \textit{malo}$$

Para n muy grande, generalmente $\Theta(n \log n)$ o menos es considerado muy bueno.

Es importante notar que cuando tenemos distintas secciones de un algoritmo con complejidad distinta, la manera de calcular la complejidad del algoritmo es "agregando" ambas complejidades, de la siguiente manera:

$$T_1(n) + T_2(n) \in O(\text{máx}\{O(T_1(n)), O(T_2(n))\})$$

Esto se puede mostrar regresando a las definiciones de complejidad que dimos antes.

Con estas definiciones, pasamos a demostrar algunas propiedades de las clases de complejidad:

Teorema A.3 *Para cualquier $f(n)$ y $g(n)$, tenemos*

$$f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n)) \quad y \quad f(n) \in \Omega(g(n)),$$

es decir,

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n)).$$

Demostración:

$\Longrightarrow$ Queremos demostrar

$$f(n) \in \Theta(g(n)) \implies f(n) \in O((g(n)) \text{ y } f(n) \in \Omega(g(n))$$

Suponemos, por hipótesis, que $f(n) \in \Theta(g(n))$. Entonces existen $c_1,\ c_2\ >\ 0$ y n_0 tales que si $n \geq n_0$ tenemos

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \qquad \forall n \geq n_0.$$

Pero entonces tenemos

$$\begin{array}{lll} c_1 \cdot g(n) \leq f(n), & \forall\, n \geq n_0; & \text{definición de} f(n) = \Omega(g(n)). \\ f(n) \leq\ c_2 \cdot g(n), & \forall\, n \geq n_0; & \text{definición de} f(n) = O(g(n)). \end{array}$$

Con lo que queda demostrado en esta dirección.

$\Longleftarrow$ Queremos demostrar

$$f(n) \in O((g(n)) \text{ y } f(n) \in \Omega(g(n)) \Longrightarrow f(n) \in \Theta(g(n))$$

Suponemos, por hipótesis, que $f(n) = O((g(n))$ y $f(n) = \Omega(g(n))$. Como $f(n) \in \Omega(g(n))$, existen $c_1 > 0$ y n_1 tales que

$$c_1 \cdot g(n) \leq f(n), \qquad \forall n \geq n_1.$$

Y como $f(n) \in O(g(n))$, existe $c_1 > 0$ y n_1 tales que

$$f(n) \leq c_1 \cdot g(n), \qquad \forall n \geq n_2.$$

Sea $n_0 = \text{máx}\{n_1, n_2\}$. Entonces, combinando ambas desigualdades se cumple que

$$c_1 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n) \quad \forall n \geq n_0,$$

que es precisamente la definición de que $f(n) \in \Theta(g(n))$. ❏

Teorema A.4 *Para cualesquiera dos funciones $f(n)$ y $g(n)$,*

$$f(n) \in o(g(n)) \quad \Longrightarrow \quad f(n) \in O(g(n)) - \Omega(g(n)),$$

es decir,

$$o(g(n)) \subseteq O(g(n)) - \Omega(g(n)).$$

Demostración:

Sea $f(n) \in o(g(n))$. Demostraremos que $f(n) \in O(g(n))$ pero que $f(n) \notin \Omega(g(n))$. Para demostrar la primera parte procedemos como sigue:

Como $f(n) \in o(g(n))$, dada $c > 0 \ \exists \ n_0$ tal que

$$0 \leq f(n) < c(g(n)) \quad \forall n \geq n_0,$$

y como

$$o(f(n)) \subset O(f(n)),$$

tenemos

$$f(n) \in O(g(n)).$$

Ahora, supongamos por contrapositivo que $f(n) \in \Omega(g(n))$. Entonces existen $c > 0$ y $n_1 > 0$ tal que $\forall n > n_1$, $0 < c \cdot g(n) \leq f(n)$. Tomemos ahora la c a la que corresponde n_1. Como $f(n) \in o(g(n))$, dada esta c existe $n_2 > 0$ tal que $0 \leq f(n) < cg(n)$, $\forall n \geq n_2$. Sea $n_0 = \text{máx}\{n_1, n_2\}$. Entonces tenemos:

$$\begin{aligned} &0 \leq cg(n) \leq f(n), && \text{para } n \geq n_0 \geq n_1, \text{ y} \\ &0 \leq f(n) < cg(n), && \text{para } n \geq n_0 \geq n_2, \end{aligned}$$

lo que es una contradicción, por lo que $f(n) \notin \Omega(g(n))$. ❏

Es importante notar que el teorema anterior no es "si y sólo si", ya que existen funciones en $O(g(n)) - \Omega(g(n))$ que no están en $o(f(n))$. Veamos un ejemplo:

$$g(n) = \begin{cases} n & \text{si } n \text{ es par} \\ 1 & \text{si } n \text{ es impar} \end{cases}$$

En esta función, $g(n) \in O(n) - \Omega(n)$ pero $g(n) \notin o(n)$.

Para terminar esta sección, listaremos varias propiedades que preservan las clases de complejidad y que se demuestran recurriendo a las definiciones que dimos de ellas.

(1) $g(n) \in O(f(n)) \iff f(n) \in \Omega(g(n))$

(2) $g(n) \in \Theta(f(n)) \iff f(n) \in \Theta(g(n))$

(3) Si $b > 1, a > 1, \quad \log_a n \in \Theta(\log_b n)$

(4) Si $0 < a < b, \quad a^n \in o(b^n)$

(5) $\forall a > 0 \quad a^n \in o(n!)$

(6) Si $c \geq 0,\ d > 0 \quad g(n) \in O(f(n))$ y $h(n) \in \Theta(f(n))$ entonces $cg(n) + dh(n) \in \Theta(f(n))$

A.7. Medición de procesos recursivos

Cuando describimos el cómo medir algoritmos, nos ocupamos únicamente de cómo medir la complejidad de algoritmos iterativos. Sin embargo, muchísimos algoritmos se presentan con recursividad, como por ejemplo el algoritmo de ordenamiento por intercalación (*Mergesort*), que se presenta en el listado A.2.

Listado A.2: Seudo-código para Ordenamiento por Intercalación *Mergesort* 1/2

```
1       public void mergeSort(long[] data, int n) {
2           merge(data, n);
3       }
4
5       public void merge(long[] data, int n) {
6           /* Si sólo hay 1 o dos elementos, ordenarlos a pie.            */
7           if(n == 1) return;
8           if(n == 2) {
9               if(data[0] > data[1]) {
10                  long temp = data[0];
11                  data[0] = data[1];
12                  data[1] = temp;
13              }
14              return;
15          }
16          /* Se parte en dos el arreglo, se copia a dos arreglos        *
17           * arreglos más pequeños.                                       */
18          int nl = (int)(n/2);
19          int nr = n - (int)(n/2);
20          /* Éste no es un algoritmo ``en su lugar''.                     */
21          long[] dataLeft = new long[nl];
22          long[] dataRight = new long[nr];
23          for(int i = 0; i < nl; i++)
24              dataLeft[i] = data[i];
25          for(int i = 0; i < nr; i++)
26              dataRight[i] = data[nl + i];
27          /* Se hacen las llamadas recursivas.                            */
28          merge(dataLeft, nl);
29          merge(dataRight, nr);
30          combine(dataLeft, dataRight, data);
31      }
32
33      public void combine(long[] dataLeft, long[] dataRight,
34                                             long data[]) {
35          /* Lleva a cabo la intercalación de dos arreglos,              *
36           * cada uno de ellos ya ordenado.                               */
37          int nl = dataLeft.length;
38          int nr = dataRight.length;
39          int n = data.length;
```

Listado A.2: Ordenamiento por Intercalación *Merge Sort* 2/2

```
40          int i=0;
41          int j=0;
42          int k=0;
43          while(i < nl && j < nr) {
44              if(dataLeft[i] <= dataRight[j])
45                  data[k++] = dataLeft[i++];
46              else
47                  data[k++] = dataRight[j++];
48          }
49          if(i == nl)
50              for(; j < nr; j++)
51                  data[k++] = dataRight[j];
52          else if(j == nr)
53              for(; i < nl; i++)
54                  data[k++] = dataLeft[i];
55      }
```

La llamada externa a la función se hace en la línea [2]. Al entrar a ejecutar la función, la cláusula de escape de la recursión se encuentra en las líneas [7] a [15] donde se verifica que ya no tiene sentido seguir partiendo el arreglo – hay implementaciones donde en cuanto el arreglo alcanza un cierto tamaño, se utiliza algún otro método de ordenamiento que sea eficiente con un número pequeño de datos.

En las líneas [18] a [26] se preparan los datos para las llamadas recursivas, copiando el arreglo que entró como parámetro a dos arreglos, donde cada uno de ellos va a tener a la mitad de los elementos del original. En las líneas [28] y [29] se llevan a cabo las llamadas recursivas, cada una de ellas con el arreglo que contiene respectivamente a la mitad izquierda y derecha del arreglo original. Una vez que se regresó de ambas llamadas recursivas, se procede, en la línea [30] a combinar los dos arreglos obtenidos. Cada llamada a merge regresa al arreglo pasado como parámetro ordenado.

La función combine simplemente hace la intercalación de dos arreglos respetando el orden de los elementos. Dado que es una función ampliamente conocida, no la explicamos con más detalle.

Si nos fijamos en el tiempo de ejecución de la primera llamada, y llamamos a este tiempo $T(n)$, podemos en primera instancia expresarlo de la siguiente manera:

$$T\left(n\right) = T_{\mathsf{copia}}\left(n\right) + 2 \cdot T\left(\frac{n}{2}\right) + T_{\mathsf{combine}}\left(n\right)$$

Como tanto $T_{\mathsf{copia}}(n)$ como $T_{\mathsf{combine}}(n)$ son $O(n)$, la expresión puede simplificarse de la siguiente manera:

$$T\left(n\right) = 2 \cdot T\left(\frac{n}{2}\right) + O\left(n\right).$$

De lo anterior, la fórmula para calcular la complejidad del ordenamiento por intercalación

está dada por una recurrencia. Demostraremos que la evaluación de esta recurrencia da

$$T(n) = n \log n.$$

Veamos primero el caso para $n = 2^k$. Entonces tenemos

$$\begin{aligned} T(2^k) &= 2 \cdot T(2^{k-1}) + 2^k && \text{para } k \geq 2; \\ T(1) &= 0. \end{aligned}$$

Si dividimos ambos lados de la ecuación por 2^k obtenemos

$$\frac{T(2^k)}{2^k} = \frac{T(2^{k-1})}{2^{k-1}} + 1 = \frac{T(2^{k-2})}{2^{k-2}} + 2 = \frac{T(2^{k-3})}{2^{k-3}} + 3 = \ldots = \frac{T(2^0)}{2^0} + k = k.$$

Como $n = 2^k$, $k = \log_2 n$. Y como para obtener este resultado dividimos entre $n = 2^k$, tenemos que

$$T(n) = O(n \log n).$$

De manera similar se puede demostrar para $n \neq 2^k$, pero dado que requiere más de habilidades matemáticas que algorítmicas, no lo hacemos acá.

En general, cuando tenemos una función o procedimiento recursivo, tenemos que establecer una recurrencia que debe tomar en cuenta:

- El número de sub-casos en los que se divide el problema original.
- El costo de dividir el problema original.
- El costo de cada sub-caso dado en términos del tamaño del sub-caso en relación con el tamaño original del caso.
- El costo de combinar las soluciones dadas por los subproblemas.

En el caso del ordenamiento por intercalación, cada problema se divide a su vez en dos subproblemas. Cada subproblema trabaja con la mitad de los datos del problema original. Para dividir el problema debemos copiar los datos a dos arreglos nuevos, cada uno de la mitad de tamaño del problema original. El costo de cada subproblema es el mismo que el costo del problema original, pero aplicado a la mitad de los datos. Finalmente, para combinar las soluciones dadas por los subproblemas, tenemos que intercalar los dos arreglos en el orden adecuado.

De lo anterior, una fórmula general para obtener el costo de un programa recursivo, cuando se está usando el método de *divide y vencerás*, es la siguiente:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + \Theta(n^k) \tag{1}$$

donde a es el número de subproblemas en los que se divide el problema original, n/b es el tamaño de cada subproblema y $\Theta(n^k)$ es el costo de dividir y combinar las soluciones dadas a los subproblemas. La solución a esta recurrencia, con $a \geq 1$ y $b > 1$, toma la siguiente forma:

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{si } a > b^k & (2.a) \\ O(n^k \log n) & \text{si } a = b^k & (2.b) \\ O(n^k) & \text{si } a < b^k & (2.c) \end{cases}$$

En el caso del ordenamiento por intercalación, $k = 1$ y $a = b = 2$, de donde la solución a la recurrencia está dada por $O(n^1 \log n)$.

Como el planteamiento de esta recurrencia indica, no siempre se divide al problema original en partes iguales.

Una vez establecida la función de recurrencia se deberá acudir a las matemáticas para resolverlas mediante distintos métodos – ver [CLRS01] – y obtener una función que describa la clase de complejidad del algoritmo. Es importante conocer el valor de algunas sumas discretas que se presentan continuamente en la solución de recurrencias. A continuación mostramos las más usadas.

$$\sum_{0 \le k < n} x^k = \frac{1 - x^n}{1 - x}$$ serie geométrica.

$$\sum_{0 \le k < n} k = \frac{n(n-1)}{2} = \binom{n}{2}$$ serie aritmética.

$$\sum_{0 \le k \le n} \binom{k}{m} = \binom{n+1}{m+1}$$ coeficientes binomiales.

$$\sum_{0 \le k \le n} \binom{n}{k} x^k y^{n-k} = (x + y)^n$$ teorema binomial.

$$\sum_{0 \le k \le n} \frac{1}{k} = H_n$$ números armónicos.

$$\sum_{0 \le k < n} H_k = nH_n - n$$ suma de números armónicos.

$$\sum_{0 \le k \le n} \binom{n}{k}\binom{m}{t-k} = \binom{n+m}{t}$$ convolución de Vandermonde.

A continuación mostramos algunas de las recurrencias que aparecen más frecuentemente en el cálculo de la complejidad de algoritmos.

Recurrencia	**Equivalente a:**	
$a_n = x_n a_{n-1}$	$a_n = \prod_{1 \le k \le n} x_k$	para $n > 0$, con $a_0 = 1$
$a_n = a_{n-1} + y_n$	$a_n = \sum_{1 \le k \le n} y_k$	para $n > 0$, con $a_0 = 0$
$a_n = x_n a_{n-1} + y_n$	$a_n = y_n + \sum_{1 \le j \le n} y_j x_{j+1} x_{j+2} \ldots x_n$	para $n > 0$, con $a_0 = 0$
$a_n = 3a_{n-1} - 2a_{n-2}$	$a_n = 2^n - 1$	para $n < 0$, con $a_0 = 0$ y $a_1 = 1$
$a_n = 5a_{n-1} - 6a_{n-2}$	$a_n = 3^n - 2^n$	para $n < 0$, con $a_0 = 0$ y $a_1 = 1$

Con esto damos por terminada esta introducción al tema de complejidad de algoritmos. Esperamos haber proporcionado algunas herramientas básicas que auxiliarán en el análisis de distintos algoritmos.

Apéndice B

Conceptos de ordenamientos

En este apéndice incluimos algunos conceptos básicos de ordenamientos, útiles para comprender de mejor manera este tipo de proceso.

B.1. Ordenamientos

Hblaremos un poco sobre los conceptos involucrados en los ordenamientos. El problema de ordenamiento precede realmente a las computadoras, como por ejemplo el orden necesario en un diccionario o en un directorio. Sin embargo, es con el advenimiento de las computadoras – y con el uso de clasificadoras como la de Hollerith – que se hace conciencia de la necesidad de tener algoritmos de ordenamiento eficientes – ver [IHB$^+$00]. Para muchos problemas, la eficiencia del algoritmo presenta un incremento notable si los datos vienen ordenados, como en el caso de la construcción de índices con ciertos algoritmos, o el proceso de un verificador de ortografía. Otros algoritmos tienen como parte integral algún ordenamiento parcial o total de sus datos, por lo que las diferencias en eficiencia de los distintos algoritmos de ordenamiento puede ser determinante en la eficiencia del algoritmo que únicamente utiliza al ordenamiento como una subrutina.

En general, lo que buscamos que haga un algoritmo de ordenamiento es que *reorganice* el orden de los registros originales y nos entregue los mismos registros, pero en un orden tal que de acuerdo a un criterio dado, estos registros se encuentren *ordenados.*

El problema general de ordenamiento no exige que todas las llaves sean distintas, sino que se puede repetir alguna llave en más de un registro. Esto trae a colación la distinción entre ordenamientos que, frente a la misma llave, preservan o no el orden original de los registros:

Definición B.1 (Ordenamiento estable) Un ordenamiento $perm(R, R')$ es *estable* si frente a repetición de llaves $r_j = r_\ell,\ j < \ell$, si tenemos que r_j aparece en la posición m de R' $(r_j = r'_m)$ y r_ℓ aparece en la posición p en R' $(r_\ell = r'_p)$,entonces $m < p$.

En ocasiones es muy importante que un algoritmo de ordenamiento sea estable. Por ejemplo, si consideramos los movimientos en una cuenta de cheques donde la llave corresponde al número de cuenta, queremos que los movimientos se procesen cronológicamente de acuerdo al tiempo de llegada del movimiento, por lo que un algoritmo de ordenamiento que no sea estable no serviría para este proceso.

Un algoritmo *interno* de ordenamiento supone lo siguiente:

i. Participan en el ordenamiento exclusivamente las llaves de los registros y una referencia al registro original al que le corresponde esa llave. Este último campo participa en el ordenamiento en la medida en que es movido y copiado junto con la llave. En casos en que la información adicional a la de la llave no ocupa mucho espacio, ésta es incluida en

el registro que se manipula para obtener el ordenamiento. Se habla indistintamente de estos registros que participan directamente en el ordenamiento y los registros originales.

ii. Todos los registros a ordenar se encuentran en memoria de acceso directo, por lo que no se considera el tiempo que pudiera llevar obtener un registro desde almacenamiento secundario. Además se supone que se cuenta con acceso directo al i-ésimo elemento mediante el índice i – esto es, se supone que lleva una unidad de tiempo obtener a un elemento mediante su índice. Como es costumbre en computación, se usa $r[i]$ para denotar a r_i. Se usa R para denotar a todo el arreglo, y $R[i \ldots j]$ para denotar a la porción del arreglo que incluye a todos los registros que están entre las posiciones i y j inclusives.

iii. A menos que se indique lo contrario, se supone que las comparaciones de llaves se pueden realizar en una sola operación, lo mismo que la copia de registros.

B.2. Algoritmos de ordenamiento por comparación

Cuando hablamos de algoritmos de ordenamiento por comparación debemos tener claro que existe una cota mínima en cuanto a la eficiencia que cualquier algoritmo de esta clase puede tener – el lector puede ver en [CLRS01] y en [Knu98b] un estudio detallado y preciso alrededor de este tema. Pensemos en el árbol de comparaciones de los datos. Por ejemplo, si queremos ordenar tres datos, $\langle a_1, a_2, a_3 \rangle$, el árbol que contempla todas las posibles comparaciones sería como se muestra en la Figura B.1. En este árbol, cada vez que comparamos dos elementos, el resultado de la comparación es verdadero o falso. Si es falso, suponemos que los elementos comparados se intercambian para que tengan la relación deseada.

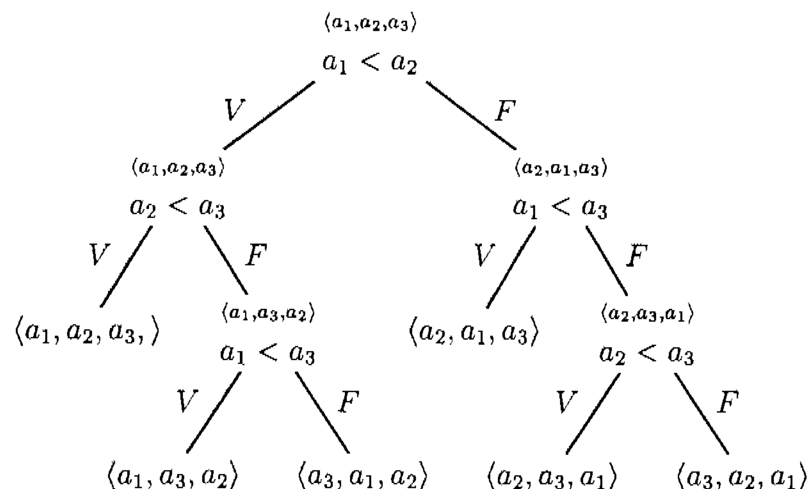


Figura B.1: Árbol de comparaciones para ordenar tres elementos

Como se puede observar de la Figura B.1, tenemos un árbol binario cuya raíz es la permutación original que se nos presenta, siendo posible cualquiera de las seis permutaciones de tres elementos para obtener una lista ordenada. Cada una de las hojas del árbol presenta los elementos en orden y corresponde a cada una de las posibles permutaciones que presentaban

los datos originales, incluyendo a aquella que corresponde a la permutación original – los datos ya venían ordenados. Por lo tanto, cualquier ordenamiento por comparación que llevemos a cabo estará representado por un árbol binario, cuyo resultado debe corresponder a haber ordenado a cada una de las permutaciones posibles de los datos. Si tenemos una lista de n datos, el número de hojas que debe tener el árbol de comparaciones deberá ser, al menos $n!$, que es el número de permutaciones de n datos. Para que un árbol binario tenga al menos $n!$ hojas, debe tener un camino en el árbol de longitud al menos $\log_2 n!$. Pero entonces tenemos las siguientes relaciones. Como $n! \leq n^n$, $\log_2 n! \leq \log_2 n^n = n \log_2 n$. De donde, para cualquier algoritmo de ordenamiento, habrá al menos una permutación que exija que se hagan $n \log_2 n$ comparaciones (e intercambios), si se desea que el algoritmo pueda transformar una permutación arbitraria en una ordenada.

Apéndice C

Conceptos de teoría de gráficas

En este apéndice daremos una revisión a los conc eptos y propiedades de teoría de gráficas que se requiere para plantear algunas de las especializaciones que presentamos, y en algunos casos demostrar su correctez y calcular su complejidad.

C.1. Conceptos básicos

Una gráfica $G = (V, E)$ es una pareja ordenada de conjuntos V y E, donde V son los vértices de la gráfica y E son parejas de una relación entre los vértices. Si esta relación es reflexiva, decimos que sus elementos se llaman *aristas*, y si no lo es entonces sus elementos se denominan *arcos*. Si E consiste de arcos decimos que la gráfica es *dirigida* o *digráfica*; si no, es simplemente una gráfica.

A la arista (u, v), donde u y $\mathsf{v} \in V$, la denotamos con $\mathsf{u}-\mathsf{v}$; al arco (u, v) lo denotamos con $\mathsf{u}\rightarrow\mathsf{v}$.

C.1.1. Caminos más cortos y distancias

Definición C.1 (Camino) Un camino en una gráfica es una sucesión de vértice $\langle \mathsf{v}_0, \mathsf{v}_1, \ldots \mathsf{v}_k \rangle$, en la que entre cualesquiera dos vértices v_i y v_{i+1}, $i = 0, \ldots k-1$, la arista (el arco) $(\mathsf{v}_i, \mathsf{v}_{i+1}) \in E$.

Definición C.2 (Longitud de un camino) Sea $P = \mathsf{u}_i, \mathsf{u}_2, \ldots, \mathsf{u}_k$ un camino en G. La longitud (o tamaño) de p, denotada por $|P|$, es el número de aristas (arcos) en P.

Otra definición de camino está dada en términos de una sucesión de aristas o arcos.

Un camino es *simple* si no se repite ningún vértice en él. Un camino es un *ciclo* si empieza y termina en el mismo vértice.

Definición C.3 (Distancia,δ(s,v)) Denotemos con $\mathsf{u}\overset{P}{\rightsquigarrow}\mathsf{v}$ a un camino simple de u a v. Entonces, definimos la *distancia* δ de un vértice s a un vértice v,

$$\delta(\mathsf{s}, \mathsf{v}) = \begin{cases} \text{mín}(|P_i|) & \mathsf{s} \overset{P_i}{\rightsquigarrow} \mathsf{v} \text{ en } G \\ \infty & \nexists\, \mathsf{s} \rightsquigarrow \mathsf{v} \end{cases}$$

Definición C.4 (Camino más corto) Un camino más corto $P = \mathsf{u}_0, \mathsf{u}_1, \ldots, \mathsf{u}_k$ es aquél donde $|P| = \delta(\mathsf{u}_0, \mathsf{u}_k)$.

A continuación veremos una propiedad de caminos más cortos.

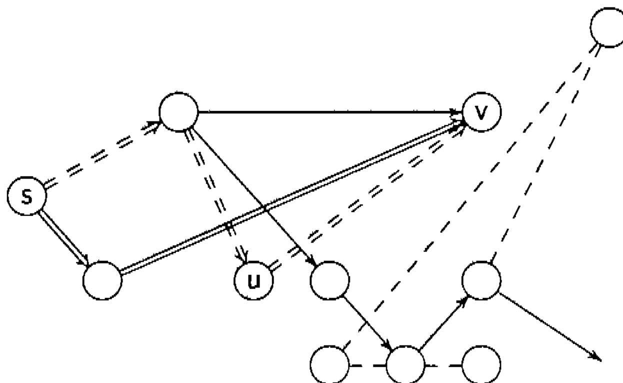
Lema C.1 $\forall$ u$\rightarrow$v $\in E$, $\delta(\mathsf{s},\mathsf{v}) \leq \delta(\mathsf{s},\mathsf{u}) + 1$.

Demostración:

Tomemos un camino más corto de s a v, s⇝v, y un camino más corto de s a u, s⇝u. Si un camino más corto de s a v pasa por u (s ⇝ $u \rightarrow$ v), el último arco que se incluyó en el camino es, precisamente, u$\rightarrow$v y entonces se da la igualdad:

$$\delta(\mathsf{s},\mathsf{v}) = \delta(\mathsf{s},\mathsf{u}) + 1$$

Si ningún camino más corto de s a v pasa por u, entonces un camino más corto debe tener menos arcos que el que pasa por u. El camino que pasa por u primero llega a u por un camino más corto de s a u ($\delta(\mathsf{s},\mathsf{u})$); una vez en el vértice u, la forma más directa de llegar a v es utilizando el arco u $\rightarrow$ v, por lo que tenemos que la longitud de s ⇝ $u \rightarrow$ v es $\delta(\mathsf{s},\mathsf{u}) + 1$. Pero como existe un camino más corto de s a v que no pasa por u, tenemos $\delta(\mathsf{s},\mathsf{v}) < \delta(\mathsf{s},\mathsf{u}) + 1$. Esto se ilustra en la gráfica de la Figura C.1.



$\delta(\mathsf{s},\mathsf{u}) = 2$ (==⇒)

$\delta(\mathsf{s},\mathsf{v}) = 2$ sin pasar por u (⟹)

$\leq \delta(\mathsf{s},\mathsf{u}) + 1 = 3$ (==⇒)

Figura C.1: Propiedad de la distancia

C.2. Gráficas bipartitas

Definición C.5 (Gráfica bipartita) Sea $G = (V, E)$ una gráfica no dirigida. G es *bipartita* si existe una partición de V en dos subconjuntos S_1 y S_2, tales que $S_1 \cap S_2 = \varnothing$ y $V = S_1 \cup S_2$; y $\forall$e = u—v $\in E$ se tiene que u $\in S_1$ y v $\in S_2$ o bien u $\in S_2$ y v $\in S_1$.

Lo que nos dice esta definición es que debemos poder separar a los vértices de la gráfica en dos conjuntos, de tal manera que todas las aristas de la gráfica vayan de un vértice en un conjunto a un vértice en el otro conjunto.

Una manera de ver si una gráfica G es bipartita es probando todas las parejas de subconjuntos de V y ver si se cumple la condición con las aristas. Este algoritmo tiene complejidad

$\Omega(2^{|V|})$, por lo que deseamos un algoritmo más eficiente. Para justificar la correctez de BFS para detrminar si una gráfica es bipartita o no, caracterizamos a las gráficas bipartitas de manera alternativa en el Teorema C.2:

Teorema C.2 *Sea $G = (V, E)$ una gráfica no dirigida. G es bipartita $\iff$ G no tiene ciclos de longitud impar.*

Demostración:

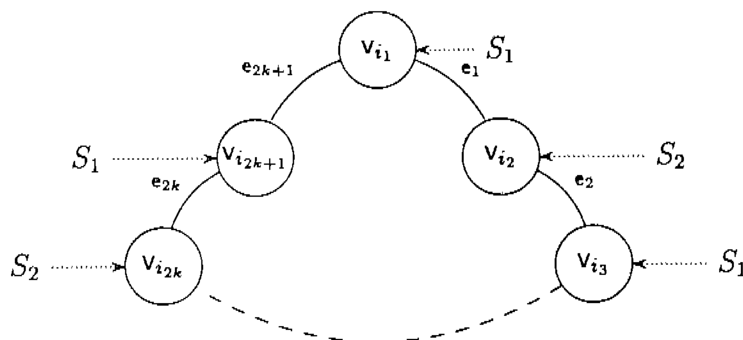
$\Longrightarrow$ Por contrapositivo, supongamos que G tiene algún ciclo de longitud impar. Sea ese ciclo

$$\mathcal{C} = \langle v_{i_1} - v_{i_2} - \ldots - v_{i_{2k+1}} - v_{i_1} \rangle.$$

Al asignar a estos vértices a los conjuntos S_1 y S_2 conforme recorremos el ciclo vamos alternando la asignación de los vértices a los conjuntos S_1 y S_2, como se muestra en la Figura C.2. La asignación se hace con la siguiente regla:

$v_{i_j} \in S_1 \iff$ el número de aristas desde v_{i_1} a v_{i_j} en el ciclo es par o 0.
$v_{i_j} \in S_2 \iff$ el número de aristas desde v_{i_1} a v_{i_j} en el ciclo es impar.

Figura C.2: Gráfica con ciclo de longitud impar



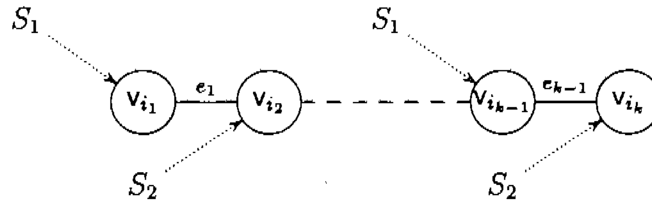
La arista e_{2k+1} va de un vértice en S_1 a otro vértice también en S_1, por lo que la gráfica no es bipartita. Es obvio que no hay manera de asignar a los vértices de un ciclo de longitud impar a dos conjuntos ajenos, de tal manera que no haya ninguna arista entre vértices del mismo conjunto.

$\Longleftarrow$ Supongamos ahora que la gráfica no tiene ningún ciclo de longitud impar. Tenemos, entonces, dos casos:

i. La gráfica no tiene ciclos.
ii. Todos los ciclos de la gráfica son de longitud par.

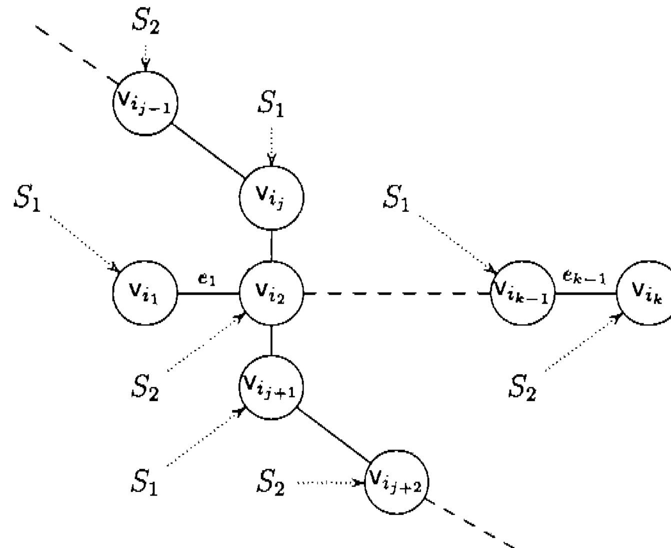
En el primer caso, en el que la gráfica no tiene ciclos, la asignación de los vértices a los conjuntos S_1 y S_2 es trivial. Supongamos que la gráfica consiste de un camino simple. Entonces para la asignación de cada vértice a uno de los conjuntos S_1 y S_2 se usa la regla que dimos para la primera parte de la demostración y se van alternando los vértices como se muestra en la Figura C.3.

Figura C.3: Asignación de vértices en camino simple



Si la gráfica consiste de más de un camino simple, como la gráfica no tiene ciclos, lo más que puede pasar es que se crucen en alguno de los vértices, como se muestra en la Figura C.4. En este caso, se alternan los conjuntos para el primer camino simple y a continuación, partiendo del vértice en el que se cruzan dos caminos, se asignan alternando desde ese vértice en cada una de las direcciones del camino simple que cruza. Como la gráfica no tiene ciclos, no puede suceder que dos caminos se crucen en más de un vértice.

Figura C.4: Asignación de vértices en más de un camino simple



De la Figura C.4, y como cada componente de G es un árbol (no tiene ciclos), se asignan correctamente los vértices a cada uno de los conjuntos, de tal manera que no va a existir ninguna arista que vaya de un vértice a otro en el mismo conjunto.

En el segundo caso, la asignación de vértices a los conjuntos S_1 y S_2 que hemos estado realizando, dado que los ciclos que pudiera haber en la gráfica son de longitud par, también asigna de manera correcta a los vértices.

De lo anterior, logramos encontrar dos conjuntos S_1 y S_2 que cumplen las condiciones para que la gráfica sea bipartita. ❏

C.3. Conexidad en gráficas

Definición C.6 Una gráfica $G = (V, E)$ es *conexa* si $\exists$ un camino entre cualesquiera dos vértices en la gráfica.

Involucrados como estamos en tratar de determinar propiedades estructurales de gráficas, utilizando para ello algoritmos de exploración, un aspecto de mucho interés resulta ser qué tan "comunicados"están los vértices de una gráfica entre sí, o dicho de otra manera, cuántos caminos hay entre cualesquiera dos vértices. De este tipo de propiedades podemos obtener la "resistencia" que puede oponer una gráfica a partirse. Nos referimos a cuántos y cuáles vértices o aristas – o una combinación de ambos – se pueden quitar de la gráfica sin que ésta pierda la propiedad de ser una gráfica conexa.

C.3.1. Componentes no separables

Definición C.7 Un vértice v es un *vértice de corte* de $G = (V, E)$ si existen vértices a y b distintos de v, tales que todos los caminos entre a y b pasan por v. Una gráfica con vértices de corte se llama *separable.*

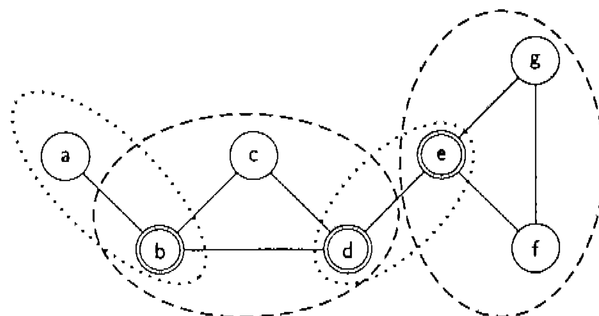
Otra definición de vértice de corte en términos de la conexidad de la gráfica la tenemos en la Definición C.8.

Definición C.8 (vértice de corte)
Un vértice v es de *corte* si $G = (V, E)$ es conexa, pero $G - \{v\}$ es trivial o no conexa.

Un vértice de corte es aquél por el que pasan todos los caminos entre dos vértices. Al quitar a ese vértice – junto con todas las aristas incidentes en él – se obtienen dos o más componentes de la gráfica original.

En la gráfica de la Figura C.5, b, d y e son vértices de corte. En el caso de b, es vértice de corte porque todos los caminos entre a y c, a—b—c y a—b—d—c, pasan por b; c no es vértice de corte porque no hay ninguna pareja se vértices tales que todo camino entre ellos pase por c.

Figura C.5: Componentes no separables de una gráfica



Definición C.9 (Componente no separable) Sea $V' \subseteq V$ un subconjunto de V. La subgráfica inducida $G' = (V', E')$ es una *componente no separable* si G' es no separable, y para todo V'' más grande tal que $V' \subset V'' \subseteq V$, la gráfica inducida $G'' = (V'', E'')$ es separable, esto es, V' es *maximal no separable.*

En el ejemplo que dimos en la Figura C.5, cada uno de los siguientes subconjuntos corresponde a una componente no separable:

- $\{\mathsf{a}, \mathsf{b}\}$.
- $\{\mathsf{b}, \mathsf{c}, \mathsf{d}\}$.
- $\{\mathsf{d}, \mathsf{e}\}$.
- $\{\mathsf{e}, \mathsf{f}, \mathsf{g}\}$.

Para el caso de la componente {a, b}, por ejemplo, le podemos agregar los vértices c o d, o ambos. Si le agregamos c, b se convierte en vértice de corte ya que el único camino entre a y c pasa por b, por lo que la gráfica $G'' = (\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}, \{\mathsf{a}{-}\mathsf{b}, \mathsf{b}{-}\mathsf{c}\})$ es separable.

C.3.2. Propiedades

De las definiciones de vértice de corte y componentes no separables, podemos demostrar las siguientes propiedades para una gráfica $G = (V, E)$ conexa.

Lema C.3 *Si G no contiene vértices de corte entonces toda G es una sola componente no separable.*

Demostración:

De la Definición C.7 G no cumple con ser separable ya que no tiene vértices de corte. Y por la Definición C.9, se cumple que no existe ningún subconjunto de vértices V', con $V \subset V' \subseteq V$ que sea no separable. ❑

Lema C.4 *Si* v *es un vértice de corte, entonces se pueden construir $V_1, V_2, \ldots, V_k$ tales que*

$$V_1 \cup V_2 \cup \ldots \cup V_k = V - \{v\} \qquad y \qquad V_i \cap V_j = \varnothing, \quad i \neq j.$$

Demostración:

Construimos las clases V_i de la siguiente forma: como v es vértice de corte, existe al menos una pareja de vértices (a, b), distintos de v y tales que todo camino entre a y b pasa por v. Entonces, al quitar a v, ya no existe ningún camino entre a y b. V_1 contiene a a y a todos los vértices alcanzables desde a en $G - \{\mathsf{v}\}$. Definimos $\overline{\mathsf{V}_k}$ de la siguiente manera:

$$\begin{aligned}
\overline{V_0} &= V \\
\overline{V_1} &= V - V_1 \\
&\ldots \\
\overline{\mathsf{V}_k} &= V - \left(\bigcup_{1 \leq i \leq k} V_i \right)
\end{aligned}$$

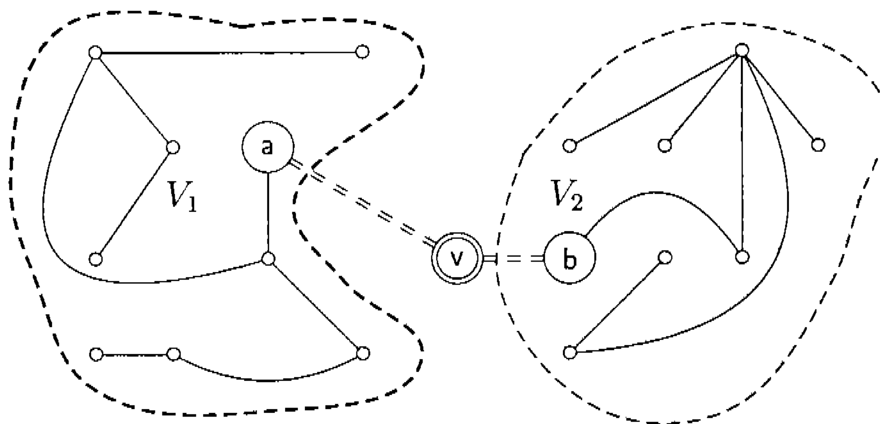
Para construir V_2 observamos si $\overline{V_1}$ contiene a algún vértice de corte. Si es así, sea v_1 ese vértice. Nuevamente existe al menos una pareja de vértices tales que todos sus caminos pasan por v_1. Tomamos a uno de esos vértices, y en V_2 quedan ese vértice y todos los alcanzables desde él en la gráfica inducida por $\overline{V_1} - \{\mathsf{v}_2\}$. Para construir a V_k vemos si existe v_{k-1} vértice de corte en $\overline{V_{k-1}}$. Si existe, repetimos el procedimiento que utilizamos para construir V_{k-1}. Si no lo es, $V_k = \overline{V_{k-1}} - \{\mathsf{v}_{k-1}\}$ y ya terminamos. Por construcción,

$$V = \bigcup_{1 \leq i < k} (V_i - \{\mathsf{v}_i)\}$$

En la Figura C.6, se muestra el caso para $k = 2$.

Ahora mostraremos que esta construcción nos da, en efecto, una partición, en la que no hay ningún vértice en común (excepto por v) entre cualesquiera dos clases de la partición. Supongamos, por contradicción, que existe un vértice x tal que $\mathsf{x} \in V_\ell \cap V_j$. Entonces, existe un camino $P_1 = \mathsf{v}_{\ell_i} \rightsquigarrow \mathsf{x}$ y un camino $P_2 = \mathsf{v}_{j_i} \rightsquigarrow \mathsf{x}$. Pero entonces existe un camino $P = \mathsf{v}_{\ell_i} \rightsquigarrow \mathsf{x} \rightsquigarrow \mathsf{v}_{j_i}$. Por construcción, v_{ℓ_i} y v_{j_i} debieron quedar en la misma componente, ya que cada uno es alcanzable desde el otro, contradiciendo la hipótesis de que V_ℓ y V_j son clases distintas; o bien, $\mathsf{x} = \mathsf{v}$, pero como v no fue incluido en ninguno de los conjuntos, esto no puede ser. Por lo tanto no puede haber ningún vértice distinto de v que sea adyacente a dos vértices, cada uno de ellos en conjuntos distintos de la partición. ❏

Figura C.6: Componentes que resultan al quitar a un vértice de corte v

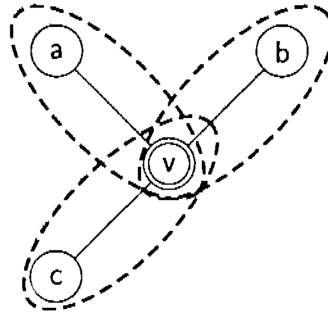


Es claro que de este lema se desprende el Corolario C.5.

Corolario C.5 *Todo camino entre componentes ajenos incluye a un vértice de corte* v.

Para que v sea vértice de corte le exigimos que se encuentre en todos los caminos entre dos vértices, pero pudiera ser vértice de corte para más de una pareja de vértices, como se muestra en la Figura C.7.

Figura C.7: Vértice de corte v para más de una pareja



Sea $G = (V, E)$ una gráfica conexa, donde v es un vértice de corte, y sean $\{v_i\}$ las componentes del Lema C.4. Si $V_i \cap V_j = \varnothing,\ i \neq j$, entonces se cumplen las siguientes propiedades:

Lema C.6 *Dos vértices* a *y* b *están en la misma componente* $V_i \iff$ *existe un camino entre ellos que no incluye a* v.

Demostración:

$\Longrightarrow$ (Por contrapositivo) suponemos que <u>todos</u> los caminos entre a y b incluyen a v. Pero entonces, al quitar a v, a ya no sería alcanzable desde b, por lo que no podría quedar en la misma componente que b.

$\Longleftarrow$ Como existe un camino que no contiene a v, no importa que quitemos a v, a seguirá siendo alcanzable desde b (y viceversa), por lo que estarán en una misma componente. ❑

Lema C.7 *Todo ciclo está dentro de una sola componente.*

Demostración:

Por contradicción, supongamos que tenemos un ciclo $C = \{v_0, v_1, \ldots, v_k, v_0\}$ que se encuentra partido entre dos componentes. Digamos que

$$C_1 = \{v_0, \ldots, v_j\} \in V_1 \quad \text{y} \quad C_2 = \{v_{j+1}, \ldots, v_k\} \in V_2$$

Pero como C es un ciclo, para cualquier pareja de vértices en el ciclo existen dos caminos entre ellos. Tomemos dos vértices adyacentes en el ciclo, v_j y v_{j+1}, cada uno de ellos en distinta componente. Por el Lema C.4, y como están en componentes distintas, todo camino entre v_j y v_{j+1} incluye a v, el vértice de corte de la partición. Por lo que la arista $v_j - v_{j+1}$, que es una arista del ciclo, es un camino entre v_j y v_{j+1}, que para contener a v se requiere que uno de los vértices sea precisamente v. Pero v no pertenece a ninguno de los V_i's, por lo que no puede estar en el ciclo considerado.

$\therefore$ este ciclo no puede existir, y todo ciclo deberá estar contenido en una única componente.

❑

Deseamos ahora obtener las componentes no separables de la gráfica G. Para ello procedemos de la siguiente manera: determinamos un vértice de corte v. Una vez que tenemos una partición dada por v en $V_1, \ldots, V_k$ que cumple las condiciones dadas arriba, tomamos cada una de las componentes $V_i \cup \{v\}$, y continuamos particionándola si es que es separable. Eventualmente vamos a terminar con una partición de componentes no separables. Para esta partición tenemos la propiedad enunciada en el Lema C.8:

Lema C.8 *Las componentes no separables pueden tener nada más vértices de corte en común.*

Demostración:

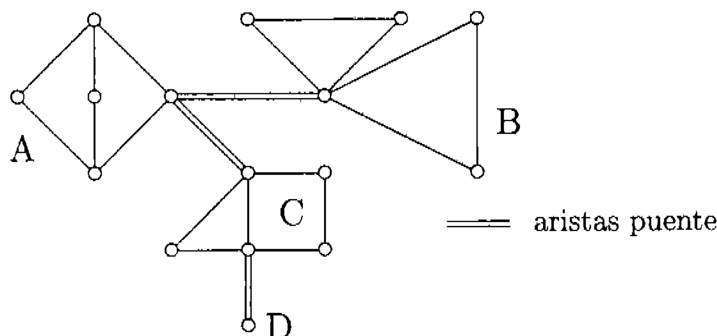
Supongamos que V_i y V_j, $i \neq j$, son componentes no separables que tienen un vértice en común u, que no es de corte, y quitemos de la gráfica a todos los vértices de corte. Como u $\in V_i$, existe un camino desde cualquier vértice x $\in V_i$ a u. Pero como también u $\in V_j$ existe un camino desde cualquier vértice w $\in V_j$ a u. Pero entonces existe un camino desde x a w que pasa por u, contradiciendo la hipótesis de que x y w están en distintas componentes no separables. Como u no es de corte, no lo quitamos de la gráfica. De todo esto, V_i y V_j no pueden tener ningún vértice en común que no sea de corte. ❏

Veamos cómo caracterizar un concepto similar al de vértice de corte, el de *arista puente*:

Definición C.10 (Arista puente) Un *puente* de G es una arista e $\in E$, tal que $G = (V, E)$ es conexa pero $G - \{e\}$ es la gráfica trivial K_1, o es no conexa. Es una arista que al quitarla incrementa el número de componentes de una gráfica.

En la Figura C.8, las aristas con doble línea corresponden a puentes.

Figura C.8: Gráfica con puentes



C.4. Gráficas con peso en las aristas

Cuando estamos trabajando con gráficas con pesos heterogéneos en las aristas, debemos precisar de mejor manera algunas de las definiciones que dimos antes.

Definición C.11 (Función de pesos) Sea $G = (V, E)$ una digráfica con una *función de pesos en las aristas*

$$w : E \rightarrow \mathbb{R}$$

Esta función asigna a cada arista un peso que, en general, puede ser un número real y puede ser negativo.

Definición C.12 (Peso de un camino) El *peso de un camino* $P = \mathsf{v}_0 \rightarrow \mathsf{v}_1 \rightarrow \ldots \rightarrow \mathsf{v}_k$, está dado por:

$$w(P) = \sum_{i=1}^{k} w(\mathsf{v}_{i-1} \rightarrow \mathsf{v}_i)$$

Definición C.13 (Distancia) El *peso de un camino más corto* o *distancia* $\delta(\mathsf{u}, \mathsf{v})$ es

$$\delta(u, v) = \begin{cases} \text{mín}\{w(P) \mid \mathsf{u} \overset{P}{\rightsquigarrow} \mathsf{v}\} & \text{si existe un camino de u a v,} \\ \infty & \text{si no hay ningún camino de u a v.} \end{cases}$$

Definición C.14 (Árbol de caminos más cortos) $G_\pi = (V_\pi, E_\pi)$ es un *árbol de caminos más cortos* si cumple con:

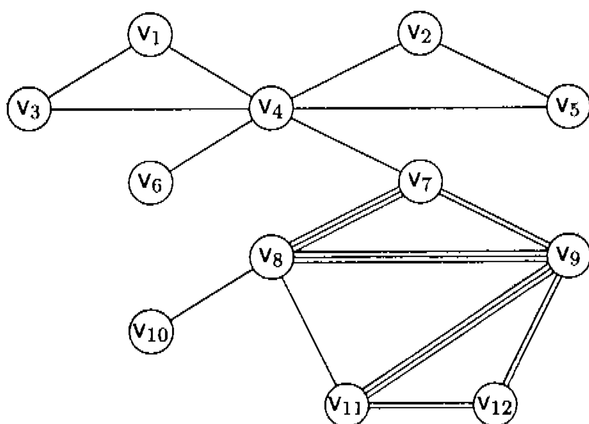
i. $V_\pi \subseteq V$ son los vértices para los cuales $\exists\ \mathsf{s} \rightsquigarrow \mathsf{v}$.
ii. G_π es un árbol dirigido con raíz en s.
iii. $\forall\ \mathsf{v} \in V_\pi$, $P = \mathsf{s} \rightsquigarrow \mathsf{v}$ en G_π es tal que $|P| = \delta(\mathsf{s}, \mathsf{v})$.

Distinguiremos ahora entre dos niveles de conexidad, la que se da entre una pareja particular de vértices, a la que llamamos *conexidad local*, y la que podemos asociar con toda la gráfica, aunque sea en términos de las distintas parejas de vértices, a la que llamamos *conexidad global* o simplemente *conexidad*.

Trabajaremos primero con la conexidad local, que es la que se da entre parejas de vértices, y se refiere, básicamente, al número de caminos distintos que hay entre dos vértices. Cuando decimos que los caminos deben ser distintos, debemos aclarar qué queremos decir con esto, ya que pueden ser distintos en cuanto a las aristas que lo forman, o pueden ser distintos en cuanto a los vértices por los que pasa ese camino o una combinación de ambos. Así, tenemos las siguientes definiciones:

Definición C.15 ($\lambda_\ell(G; u, v)$) La conexidad local por aristas entre dos vértices u y v en la gráfica G es el número de caminos ajenos por aristas que hay entre u y v.

Figura C.9: Conexidad local por aristas



Por ejemplo, en la gráfica de la Figura C.9, $\lambda_\ell(G; v_7, v_{11}) = 2$, porque aunque hay varias maneras de llegar desde v_7 hasta v_{11}, sólo dos de esos caminos son ajenos por aristas. Uno de los caminos está dado por $v_7—v_8—v_9—v_{11}$ (marcado con arista triple) y el otro camino está dado por $v_7—v_9—v_{11}$ (marcado con arista doble). Cualquier otro camino que exista entre v_7 y v_{11} tiene que usar a alguna de las aristas $v_7—v_8$ o bien $v_7—v_9$ para salir de (o llegar a) v_7, por lo que no será ajeno por aristas con estos dos caminos.

Similarmente podemos definir la conexidad local por vértices. Lo hacemos en la Definición C.16.

Definición C.16 ($\kappa_\ell(G; u, v)$) La conexidad local por vértices entre los vértices u y v, denotada por $\kappa_\ell(G; u, v)$, es el número de caminos ajenos por vértices que hay entre u y v en la gráfica G.

En la gráfica de la Figura C.9, $\kappa_\ell(G; v_7, v_{11}) = 2$, lo mismo que $\lambda_\ell(G; v_7, v_{11})$, excepto que los caminos que dimos para el caso de la conexidad local por aristas no funcionan para la conexidad local por vértices, ya que el vértice v_9 aparece en ambos caminos. Sin embargo, los caminos $v_7—v_9—v_{11}$ y $v_7—v_8—v_{11}$ son caminos entre v_7 y v_{11} que, excepto por los extremos de los caminos, no comparten ningún vértice. Cualquier otro camino que haya entre v_7 y v_{11} tiene que pasar por v_8 o por v_9, por lo que ya no serán ajenos por vértices con estos dos.

Por supuesto que si dos caminos son ajenos por vértices también lo serán por aristas, pero no al revés. Esto sugiere una relación, que demostraremos en su momento:

$$\kappa_\ell(G; u, v) \leq \lambda_\ell(G; u, v)$$

Cuando sea claro la gráfica de la que se habla, se podrá omitir la G de la notación, tanto en conexidad local por vértices como por aristas.

Podemos definir fácilmente la conexidad global – a la que nos referiremos simplemente como conexidad – por vértices y por aristas en las siguientes definiciones:

Definición C.17 ($\lambda(G)$) La conexidad por aristas de una gráfica, denota por $\lambda(G)$, está definida como

$$\lambda(G) = \min_{u,v \in V} \{\lambda_\ell(G; u, v)\}$$

Similarmente definimos la conexidad por vértices:

Definición C.18 ($\kappa(G)$) La conexidad por vértices de una gráfica, denotada por $\kappa(G)$, está definida como

$$\kappa(G) = \min_{u,v \in V} \{\kappa_\ell(G; u, v)\}$$

Realmente no necesitamos marcar a las conexidades locales con el subíndice ℓ, ya que queda claro si se está hablando de conexidad local o global por los parámetros listados, en el primer caso los dos vértices de que se trata, y en el segundo caso únicamente la gráfica.

Podemos utilizar un enfoque distinto para averiguar la conexidad de una gráfica que tiene que ver, de alguna manera, con las condiciones bajo las cuales una gráfica se parte y deja de ser conexa. Veamos primero el papel que puede jugar una determinada arista en la conexidad de una gráfica.

Trabajemos un poco con el concepto de arista puente dado en la Definición C.10. En la Figura C.10 se muestra una gráfica con varios puentes. Por ejemplo, las aristas v_4—v_6 y v_4—v_7 son puentes en esta gráfica.

El resultado de quitar el puente v_4—v_7 se muestra en la Figura C.11, con la arista punteada. La gráfica original se parte en dos componentes, con $V_1 = \{\mathsf{v}_1, \mathsf{v}_2, \mathsf{v}_3, \mathsf{v}_4, \mathsf{v}_5, \mathsf{v}_6\}$ y V_2 el resto de los vértices.

Figura C.10: Gráfica con puentes

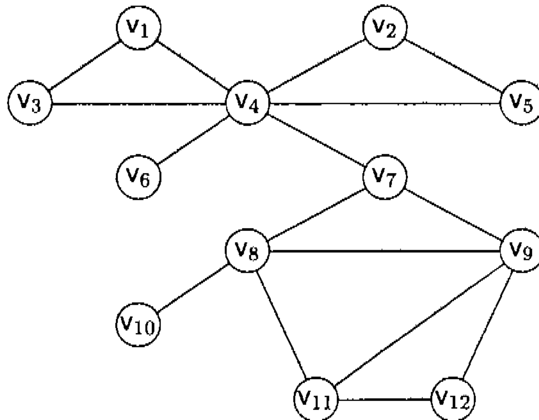
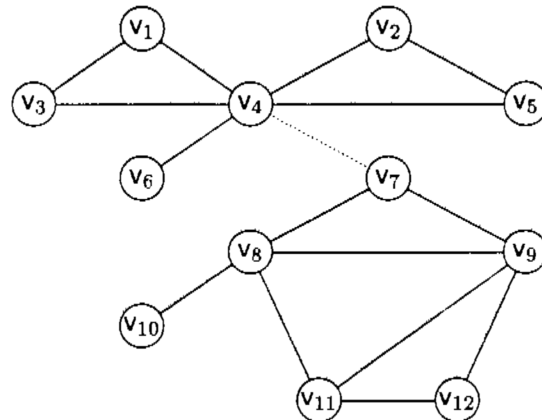


Figura C.11: Gráfica sin el puente v_4—v_7



Una manera de identificar un puente es cuando entre dos vértices el único camino está dado por una arista. Al quitarla, como dice la definición, los dos vértices para los cuales la arista es puente quedan desconectados, por lo que se tendrán dos o más componentes de la gráfica original sin esa arista. Sin embargo, por la definición que dimos, pudiéramos decir también que v_4—v_7 es un puente porque todos los caminos entre v_2 y v_9, por ejemplo, usan esa arista.

El concepto dual respecto a vértices se refiere a aquellos vértices que al quitarlos de la gráfica la parten en dos o más componentes o la gráfica se convierte en K_1, como se dijo en la Definición C.8.

Por ejemplo, en la gráfica de la Figura C.11, tanto v_4 como v_7 son vértices de corte, ya que al quitar a v_4, se parte la gráfica en cuatro componentes, como se muestra en la Figura C.12. Uno de los componentes es $K_1 = \{\mathsf{v}_6\}$, la gráfica trivial.

Figura C.12: Gráfica sin el vértice de corte v_4

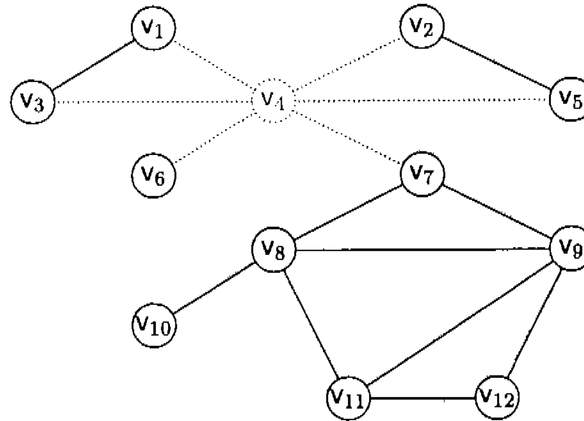
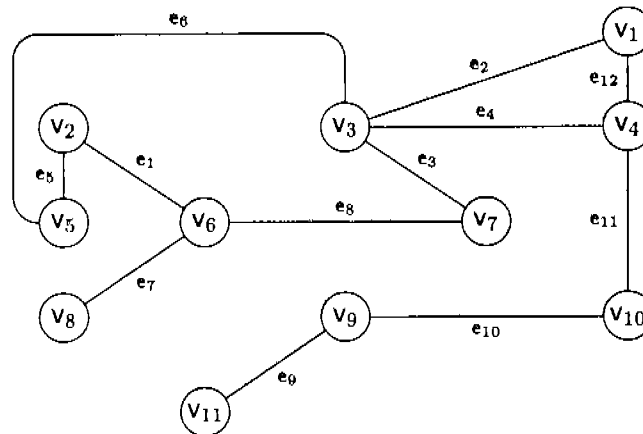


Figura C.13: Gráfica conexa



Puentes: e_2,e_4,e_9, e_i10, e_i11.
Vértices de corte: v_i3, v_i4, v_i6, v_i9, v_i10.

En la Figura C.13 podemos observar una gráfica conexa con puentes y vértices de corte. Como se puede observar, ningún vértice con grado 1 es vértice de corte, ya que al quitarlo la subgráfica que queda sigue siendo conexa. Sin embargo, una arista que es la única que incide en un vértice dado sí constituye un puente.

En ocasiones podemos considerar no a una única arista sino a un conjunto de aristas para desconectar a una gráfica, como se expresa en la Definición C.19.

Definición C.19 (Corte de aristas) Un *corte de aristas* U es un conjunto $U \subseteq E$ tal que $G = (V, E)$ es conexa pero $G - U$ es trivial o no conexa.

Un puente por sí solo constituye un corte de aristas, ya que con esa única arista se desconecta a la gráfica. Pero pudiera ser que se desconectara a la gráfica usando más de una arista. Esquemáticamente un corte de aristas se puede ver como en la Figura C.14.

Veamos un ejemplo concreto de un corte de aristas en la gráfica de la Figura C.15. En esta gráfica, las aristas e_7, e_9 y e_{11} constituyen un corte de aristas, desconectando a la componente formada por v_2, v_5, v_6 y v_8 como se muestra en esta figura. Es importante notar que ninguna de estas aristas es puente.

Figura C.14: Gráfica con corte de aristas

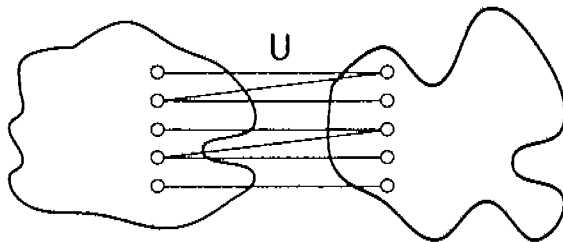
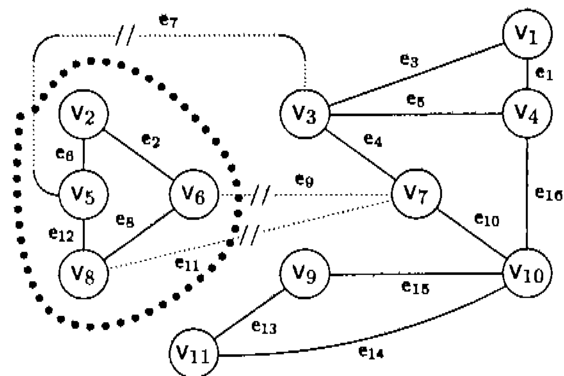


Figura C.15: Gráfica conexa con cortes de aristas



Otro corte posible serían las aristas e_{10} y e_{16}, separándose la componente dada por los vértices v_9, v_{10} y v_{11}, como se muestra en la Figura C.16.

Figura C.16: Gráfica conexa con cortes de aristas

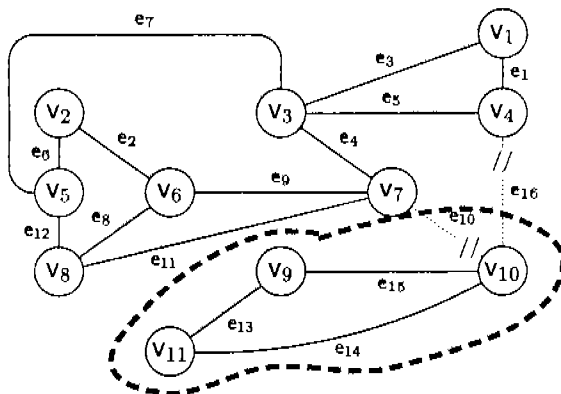
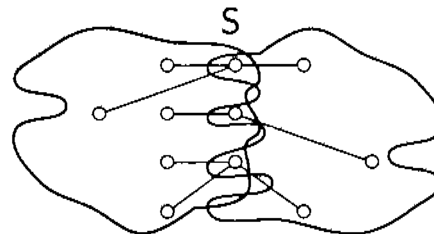


Figura C.17: Gráfica con corte de vértices



En forma dual podemos considerar a un conjunto de vértices que desconecten a una gráfica bajo la Definición C.20.

Definición C.20 (Corte de vértices) Un *corte de vértices* S es un conjunto $S \subset V$ tal que $G = (V, E)$ es conexa pero $G - S$ es trivial o no conexa.

En la Figura C.17 vemos un esquema de lo que sería un corte de vértices. En la gráfica de la Figura C.13, el vértice v_{10} solo también es un corte de vértices.

Utilizando nuevamente a la gráfica de la Figura C.9, podemos identificar cortes de vértices en ella: v_3 y v_7 conforman un corte de vértices, quedando la gráfica como se muestra en la Figura C.18.

Figura C.18: Gráfica conexa con cortes de vértices

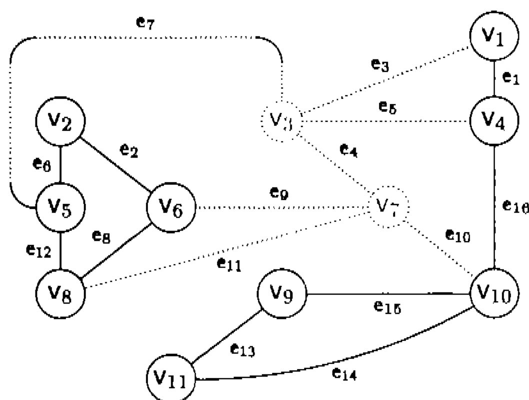
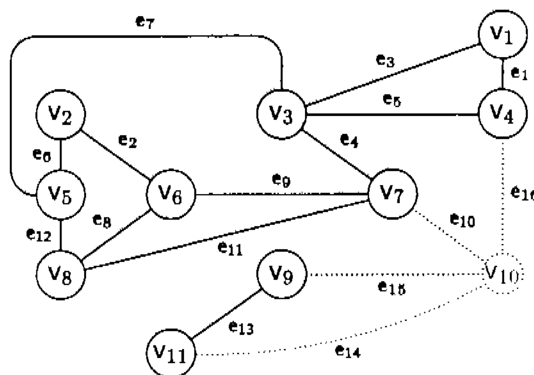


Figura C.19: Gráfica conexa con cortes de vértices alternativos



En este caso, v_{10} es asimismo un vértice de corte, como se puede observar en la Figura C.19. La subgráfica formada por los vértices v_{11} y v_9, junto con la arista e_{13}, queda desconectada del resto de la gráfica original.

La cardinalidad de un corte se refiere al número de elementos del corte, sean éstos vértices o aristas. Nos interesa la cardinalidad de los conjuntos de corte porque en términos de las aplicaciones esta cardinalidad corresponde al número máximo de vértices o aristas que pueden "desaparecer" de la gráfica sin que la conexidad se vea afectada. Qué tan pequeño o grande es este número dependerá, en general, de la cardinalidad de los distintos cortes posibles y de la relación entre ellos.

Definición C.21 (Corte minimal) Un corte de vértices (o aristas) es *minimal* si no contiene a ningún otro corte de vértices (o aristas) de cardinalidad menor.

Definición C.22 (Corte mínimo) Un corte de vértices o aristas es *mínimo* si $\nexists$ ningún otro corte respectivamente de vértices o aristas de cardinalidad menor.

Un corte puede ser minimal y sin embargo no ser mínimo. En la gráfica de la Figura C.20, $\{e_3, e_5, e_{10}\}$ conforman un corte minimal de aristas, ya que ningún subconjunto de este corte es corte. Sin embargo, éste no es un corte mínimo, ya que el conjunto $\{e_{10}, e_{16}\}$ también es un corte y tienen cardinalidad menor. Los vértices $\{v_3, v_7\}$, a su vez, conforman un corte minimal de vértices, ya que ninguno de los dos, por sí solo, separa a la gráfica. Tampoco en este caso este corte es mínimo, ya que el corte dado por $\{v_{10}\}$ tiene cardinalidad menor.

Figura C.20: Gráfica conexa con cortes de aristas

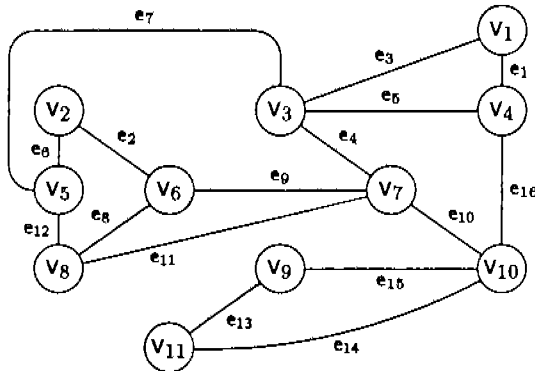
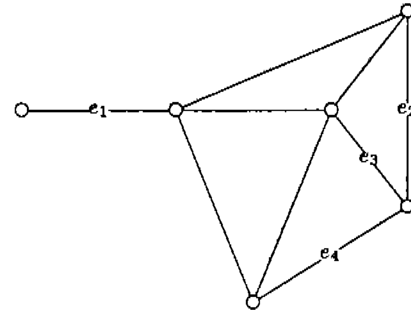


Figura C.21: Cortes de aristas minimal y mínimo



Tenemos otro ejemplo de cortes en la Figura C.21. En esta gráfica, tenemos que U={e_1} es mínimo, mientras que U={e_2, e_3, e_4 } es minimal.

C.4.1. Conexidad en términos de cortes

Los cortes, de cierta manera, definen la conexidad local de una gráfica, ya que determinan la conexidad entre determinados vértices. Pero como la conexidad global está relacionada con la local, tendremos que explorar el papel que juegan los cortes en este concepto, definiendo la conexidad global de una gráfica también en términos de cortes. Como tenemos dos tipos de cortes, los de aristas y los de vértices, la conexidad también tendrá que tomar estas dos formas, por aristas y por vértices.

Definición C.23 ($\lambda(G)$) La *conexidad por aristas* de una gráfica $G = (V, E)$ es $|U|$, con $U \subseteq E$ un corte de aristas mínimo tal que $G - U$ es trivial o no conexa.

Como se ve en esta definición, no cualquier corte de aristas sirve para determinar la conexidad global por aristas de una gráfica, sino que se exige que el corte sea mínimo.

Relacionado con la conexidad global por aristas, podemos observar lo siguiente:

Para $G = K_1$,	$\lambda(K_1) = 0$,	ya que no tiene aristas, por lo que sin quitarle ninguna arista la gráfica ya es la trivial.
Para $G \neq K_1$,	$\lambda(G) > 0$,	si G es conexa, ya que habrá que quitarle aristas para conseguir partirla.
	$\lambda(G) = 0$,	si G es disconexa, ya que sin quitar ninguna arista se cumple que tenemos al menos dos componentes en la gráfica.

Si consideramos los cortes de vértices, tenemos la Definición C.24 de lo que es la conexidad global en términos de vértices.

Definición C.24 **($\kappa(G)$)** La *conexidad por vértices* de $G = (V, E)$ es $|S|$, con $S \subset V$ un corte de vértices mínimo de acuerdo a la Definición C.8, tal que $G - S$ es trivial o disconexa.

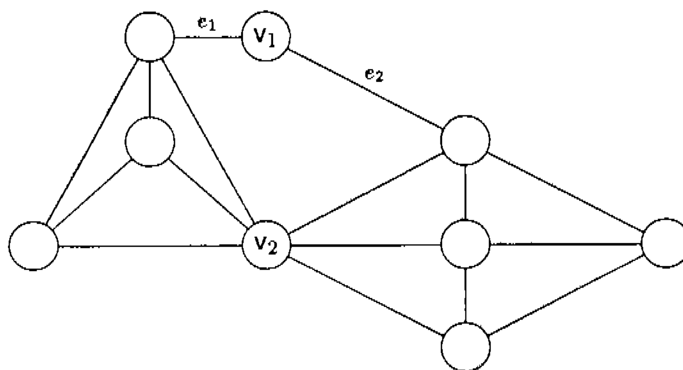
También en la conexidad por vértices podemos observar que:

$\kappa(K_p) = p - 1$; dado que cada vértice es adyacente a cualquier otro, la única forma de desconectar la gráfica es dejando un único vértice, cumpliéndose entonces que la gráfica resultante es K_1.

$\kappa(G) = 0 \iff G$ es disconexa o trivial.

Veamos un ejemplo de ambas cantidades en la gráfica de la Figura C.22, en la que se presenta lo siguiente:

Figura C.22: Conexidad por vértices y aristas

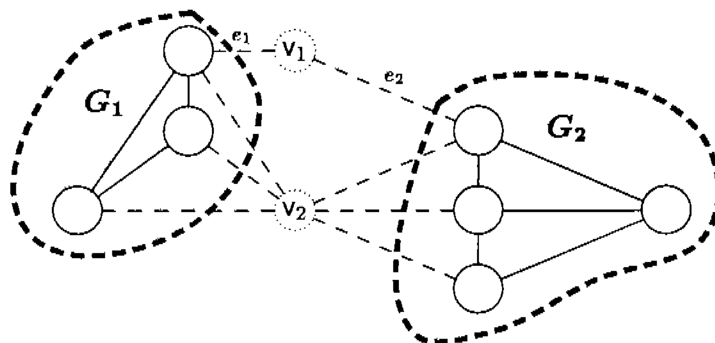


$\kappa(G) = 2$, con $S = \{v_1, v_2\}$. S es un corte porque la gráfica queda partida en los dos componentes G_1 y G_2 que se muestran en la Figura C.22. Ningún vértice por sí solo parte a la gráfica en dos o más componentes, por lo que S es mínimo.

$\lambda(G) = 2$, con $U = \{e_1, e_2\}$. Al quitar estas dos aristas, G queda partida en los componentes G_1 y G_2 como se muestra en la Figura C.23. Al quitar únicamente a una arista, cualquiera que ésta sea, la gráfica no se desconecta, por lo que U es mínimo.

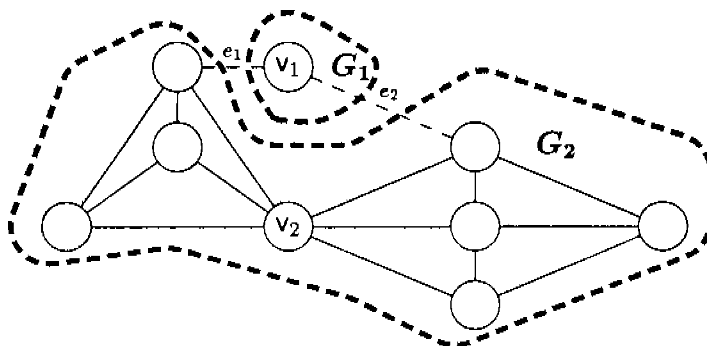
Al eliminar a los vértices v_1 y v_2, la gráfica se descompone en dos componentes conexas, como se muestra en la Figura C.23.

Figura C.23: Ejemplo de conexidad por vértices



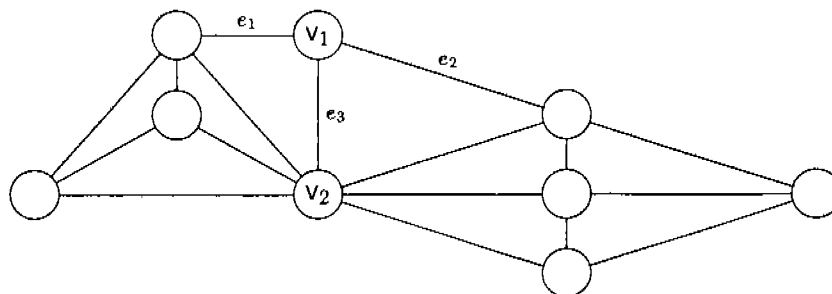
Al eliminar a las aristas e_1 y e_2, el vértice v_1 queda desconectado de la gráfica. Además, no hay ninguna arista que quitándola únicamente a ella se desconecte algún vértice. La gráfica que corresponde a $G - \{e_1, e_2\}$ se muestra en la Figura C.24.

Figura C.24: Ejemplo de conexidad por aristas



Pero si añadimos la arista $e_3 = v_1 - v_2$ para tener G' – ver Figura C.25 – obtenemos $\kappa(G') = 2$ y $\lambda(G') = 3$.

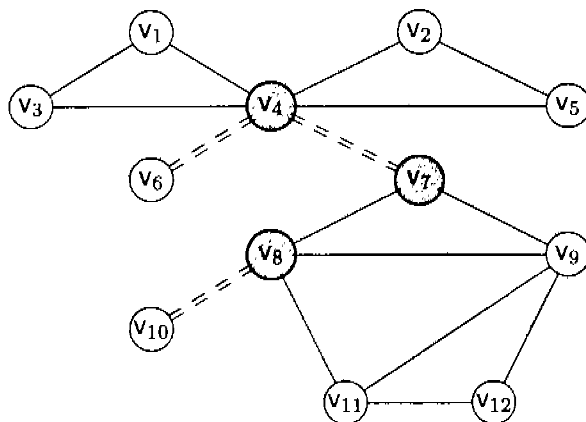
Figura C.25: Ejemplo de conexidad por vértices y por aristas, agregando una arista



Como se puede observar, aún con la nueva arista e_3, al quitar a v_1 y v_2 se parte la gráfica en las mismas dos componentes que sin esta nueva arista, ya que la arista e_3 lo único que hace es conectar a los vértices v_1 y v_2 y "desaparece" al quitar a cualquiera de estos dos vértices. Sin embargo, la conexidad por aristas sí se incrementa, ya que ahora tenemos que quitar también a e_3 para lograr desconectar al vértice v_1.

Es importante notar que puede haber más de un corte mínimo en una misma gráfica. Por ejemplo, en la gráfica de la Figura C.10, que reproducimos en la Figura C.26, hay tres cortes de aristas de cardinalidad 1 – $\{v_4—v_6\}$, $\{v_4—v_7\}$ y $\{v_8—v_{10}\}$ – ya que cada una de estas aristas, por sí sola, desconecta a la gráfica. También tenemos tres cortes distintos de vértices de cardinalidad 1 – $\{v_4\}$, $\{v_7\}$ y $\{v_8\}$ – ya que al quitar a cualquiera de estos tres vértices la gráfica se desconecta. Es casualidad en esta gráfica particular que $\lambda(G) = \kappa(G) = 1$ o que el número de cortes mínimos sea el mismo en ambos casos.

Figura C.26: Cortes de vértices y aristas mínimos



C.5. Propiedades de conexidad

Veamos ahora cuáles aristas o vértices juegan un papel importante para mantener la conexidad en una gráfica. Vimos que la conexidad se rompe en términos de vértices de corte o puentes, por lo que es natural buscar propiedades que relacionen de alguna manera el tamaño de una gráfica en términos de $|V|$ o $|E|$ con la conexidad de la misma. Es más fácil obtener información de una de estas cantidades, o bien es más fácil garantizar alguna propiedad en términos de uno de estos parámetros. También es importante saber la relación que existe entre la conexidad global y la conexidad local. En esta sección exploramos algunos resultados pertinentes.

El primer parámetro importante que relaciona a un vértice dado con sus aristas es el *grado* de un vértice v, denotado por $\delta(v)$, que es, precisamente, el número de aristas que inciden en v. Podemos definir el grado de una gráfica en términos de los grados de los vértices que la componen de la siguiente manera:

Definición C.25 ($\delta(G)$) El *grado de una gráfica G*, $\delta(G)$ se define como

$$\delta(G) = \min_{\mathsf{v}\in V}\{\delta(\mathsf{v})\}.$$

El Teorema C.9 relaciona la conexidad por vértices y por aristas con el grado de una gráfica.

Teorema C.9 *Para toda gráfica* $G = (V, E)$,

$$\kappa(G) \leq \lambda(G) \leq \delta(G).$$

Demostración:

Demostraremos por partes la desigualdad. Empezaremos por la primera mitad.

i. Por demostrar: $\kappa(G) \leq \lambda(G)$.

– Si $\lambda(G) = 0 \implies G$ es disconexa o trivial
$\implies$ por la Definición C.15, $\kappa(G) = 0$.

$\therefore \kappa(G) \leq \lambda(G)$

– Supongamos que $\lambda(G) > 0$ y sea $U \subseteq E$ tal que $G - U$ es disconexa y $|U| = \lambda(G)$. Es decir, $G - U$ consiste de varias componentes conexas, al menos 2. Agrupemos a los vértices de $G - U$ en dos conjuntos,

$$V_1, V_2 \subseteq V, \quad V_1 \cap V_2 = \varnothing, \quad V_1 \cup V_2 = V,$$

tal que $\{V_1, V_2\}$ sea una partición en la que cualquier arista $\mathsf{e} = \mathsf{u}–\mathsf{v} \in U$ tenga a uno de sus extremos en V_1 y al otro extremo en V_2.

Sea $S = \{\mathsf{v} \in V \mid \mathsf{e} = \mathsf{v}–\mathsf{x} \in U, \mathsf{v} \in V_1\}$.

Notar que:

- $|S| \leq |U|$, ya que una o más aristas de U pueden ser incidentes en un mismo vértice de S.
- $|U| = \lambda(G)$ por construcción.
- $G - S$ es disconexa o trivial, ya que al eliminar a los vértices de S se eliminan a todas las aristas incidentes en ellos, de las que U es un subconjunto.

Por lo tanto $\kappa(G) \leq |S| \leq |U| = \lambda(G)$ cuando $\lambda(G) > 0$

$\therefore$ para $\lambda(G) \geq 0$, $\kappa(G) \leq \lambda(G)$.

ii. Por demostrar: $\lambda(G) \leq \delta(G)$.

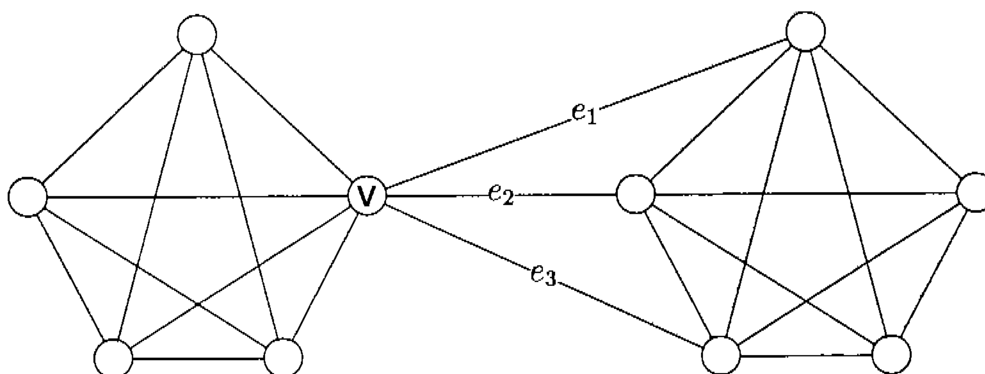
Sea v un vértice tal que $\delta(\mathsf{v}) = \delta(G)$, esto es, el grado de v es menor o igual al grado de cualquier otro vértice en G. Al quitar todas las aristas incidentes en v – posiblemente ninguna si $\delta(G) = 0$ – el vértice v se desconecta, por lo que nos queda una gráfica disconexa (o trivial, si $|V| = 1$).

$\therefore \kappa(G) \leq \lambda(G) \leq \delta(G)$. ❑

Ejemplo

Veamos un ejemplo de la relación que hay entre estos tres valores en la gráfica de la Figura C.27.

Figura C.27: Relación entre $\kappa(G)$, $\lambda(G)$ y $\delta(G)$



$$\kappa(G) = 1, \quad S = \{\text{v}\}.$$
$$\lambda(G) = 3, \quad U = \{e_1, e_2, e_3\}.$$
$$\delta(G) = 4.$$

Definamos de manera más precisa la conexidad (por vértices y aristas) en términos de la cardinalidad de los cortes mínimos (de vértices y aristas respectivamente).

Definición C.26 (k−conexidad) Una gráfica $G = (V, E)$ es *k-conexa por aristas*, para $k \geq 1$ si $\lambda(G) \geq k$. Y es *k−conexa (por vértices)* si $\kappa(G) \geq k$.

La Definición C.22 establece de cierta manera la necesidad de que la conexidad esté dada en términos de cortes mínimos, lo que demostramos en el Lema C.10.

Lema C.10 *Una gráfica $G = (V, E)$ es k−conexa ($k \geq 1$) $\iff$ al quitar menos de k vértices nos queda una gráfica no trivial conexa.*

Demostración:

$\Longrightarrow$ Supongamos que G es k−conexa, con $k \geq 1$ y queremos demostrar que podemos quitar menos de k vértices y la gráfica resultante sigue siendo no trivial y conexa.

Como G es k−conexa,

i. Por la Definición C.22, $\kappa(G) \geq k$.

ii. Por la Definición C.20, $\exists S \subset V$, $|S| = k$ con $|S|$ mínimo y para todo $S' \subset V$ con $|S'| < k$, S' no es corte, por lo que $G - S'$ sigue siendo no trivial y conexa.

$\therefore$ al quitar menos de k vértices, la gráfica resultante sigue siendo no trivial y conexa.

$\Longleftarrow$ Por demostrar que G es $k-$conexa. Supongamos ahora que al quitar menos de k vértices de G, G sigue siendo no trivial y conexa.

i. $\exists$ un subconjunto $S \subset V$ tal que $G - S$ es trivial o no conexa, con $|S| = k$.

ii. Ningún subconjunto de menos de k vértices hace que al quitarle ese subconjunto a G, ésta se vuelva trivial o no conexa.

Y ésta es precisamente la definición de que $\kappa(G) = k$. ❑

C.6. Conexidad S-mixta

Muchas veces ponemos atención a la conexidad por aristas, donde dos o más caminos comparten ciertos vértices. Pero si hablamos por ejemplo de una red de computadoras, y algunas de ellas son más confiables que otras, quisiéramos tener caminos ajenos respecto a las computadoras no confiables, aunque no nos importara que un procesador altamente confiable sea compartido por más de un camino. Para ello, regresamos a las definiciones de conexidad local que dimos, y relacionamos la conexidad de una gráfica con la existente entre sus vértices en el Lema C.11.

Lema C.11 $\kappa(G)$=mín$\{\kappa_\ell(\mathsf{u},\mathsf{v})\}$, $\forall \mathsf{u},\mathsf{v} \in V, \mathsf{u} \neq \mathsf{v}$. *Esto es*

$\kappa(G) \leq \kappa_\ell(\mathsf{u},\mathsf{v})$.

Demostración:

Supongamos que $\kappa(G)$=k. Deberemos demostrar que $\kappa_\ell(\mathsf{u},\mathsf{v}) \geq k$, $\forall \mathsf{u},\mathsf{v} \in V, \mathsf{u} \neq \mathsf{v}$.

Como $\kappa(G)$=k, $\exists S$ corte mínimo de vértices tal que $|S| = k$. Por contradicción, supondremos que existen u, v$\in V, \mathsf{u} \neq \mathsf{v}$ tales que $\kappa_\ell(\mathsf{u},\mathsf{v}) < k$. Esto quiere decir que entre u y v hay $j < k$ caminos ajenos por vértices. Supongamos que u$-$v no es uno de ellos. Cada camino tiene al menos un vértice distinto de u y v. Sea x_i, $i = 1, \ldots, j$, un vértice en el i-ésimo camino de u a v, y sea $S_{\mathsf{u},\mathsf{v}} = \{\mathsf{x}_i \mid 1 \leq i \leq j\}$. Observemos que $|S_{\mathsf{u},\mathsf{v}}| = j < k$, y consideremos $G - S_{\mathsf{u},\mathsf{v}}$.

Como ya no hay caminos entre u y v, y no hay arista u$-$v, no hay ningún camino entre u y v en $G - S_{\mathsf{u},\mathsf{v}}$. De esto $G - S_{\mathsf{u},\mathsf{v}}$ no es conexa, por lo que $S_{\mathsf{u},\mathsf{v}}$ es un corte de vértices también para G con cardinalidad menor a k, lo que contradice $\kappa(G)$=k. ❑

De manera similar podemos demostrar que $\lambda(G)\leq \lambda(\mathsf{u},\mathsf{v})$:

Lema C.12 $\lambda(G)$=mín$\{\lambda(\mathsf{u},\mathsf{v})\}$, $\forall \mathsf{u},\mathsf{v} \in V, \mathsf{u} \neq \mathsf{v}$. *Esto es*

$\lambda(G) \leq \lambda(\mathsf{u},\mathsf{v})$

Demostración:

Se omite para evitar aburrir al lector. ❏

Deseamos generalizar el concepto de k-conexidad, relacionando $\kappa(G)$ con $\lambda(G)$ de la siguiente manera.

Definición C.27 Sea $S \subseteq V$. Decimos que una familia de caminos que conectan a u y v en G es *S-independiente* si los caminos son ajenos por aristas y cada $\mathsf{v} \in S$ aparece como vértice interno en a lo más un camino.

De la Definición C.27 tenemos la definición que buscamos:

Definición C.28 (Conexidad S-mixta) La *conexidad S-mixta* de u y v en G, denotada por $\lambda_S(\mathsf{u}, \mathsf{v}; G)$, es el máximo número de caminos S-independientes que conectan a u y v en G.

Cuando sea claro por el contexto de cuál G se habla, se omitirá a ésta de la notación.

De las Definiciones C.27 y C.28 es claro que cuando $S = \varnothing$ la conexidad S-mixta se refiere a la conexidad por aristas, mientras que si $S = V$ estaremos hablando de conexidad por vértices.

A la conexidad S-mixta entre vértices la llamamos *conexidad local S-mixta*. La *conexidad global S-mixta* se define como:

Definición C.29 La *conexidad global S-mixta* denotada por $\lambda_S(G)$ está definida como

$$\lambda_S(G) = \min_{\mathsf{u},\mathsf{v} \in V} \{\lambda_S(\mathsf{u}, \mathsf{v}; G)\}$$

Similarmente a la definición de k-conexidad por vértices y por aristas, tenemos la *k-conexidad S-mixta* que se define de la siguiente manera:

Definición C.30 Dos vértice u y v presentan *k-conexidad S-mixta* si $\lambda_S(\mathsf{u}, \mathsf{v}) \geq k$.

Para terminar con las definiciones relacionadas con la conexidad S-mixta, presentamos la definición correspondiente en una gráfica G:

Definición C.31 Una gráfica G presenta *k-conexidad S-mixta* si $\lambda_S(G) \geq k$.

C.6.1. Propiedades de la $k-$conexidad

Podemos ver la conexidad por aristas o vértices, como ya lo mencionamos, como casos particulares de la conexidad S-mixta. Dejaremos por el momento este último término para concentrarnos en la relación que existe entre el tamaño de una gráfica (su número de vértices y de aristas) y condiciones necesarias o suficientes para la conexidad.

Teorema C.13 *Sea $G = (V, E)$ y sea k, $1 \leq k \leq |V| - 1$. Si*

$$\delta(G) \geq \frac{|V| + k - 2}{2}$$

entonces G es k-conexa.

Demostración:

Veamos la demostración revisando posibles casos de gráficas.

i. Si G es una gráfica completa, entonces

$$\kappa(G) = |V| - 1 \qquad \text{y} \qquad \delta(G) = |V| - 1.$$

Si $k = |V| - 1$, veamos que se cumple

$$\delta(G) \geq \frac{|V| + k - 2}{2}$$

con las siguientes manipulaciones algebraicas:

$$\begin{aligned}
\delta(G) &= |V| - 1 = \frac{2(|V| - 1)}{2} \\
&\geq \frac{|V| + |V| - 2 - 1}{2} \\
&= \frac{|V| + k - 1}{2} \\
&= |V| - \frac{3}{2}.
\end{aligned}$$

Sabemos que, como G es completa, $\kappa(G) = |V| - 1$; y si $|V| - 1$ es el valor de k que usamos para el antecedente, demostramos que, en efecto, G es $k-$conexa.

ii. Supongamos que G no es completa y elijamos una k para la que se cumple

$$\delta(G) \geq \frac{|V| + k - 2}{2}$$

y por contradicción supongamos que G no es $k-$conexa. Entonces $\exists$ un corte de vértices S, $|S| = i < k$. Sea G_1 una componente de tamaño mínimo de $G - S$. Como $G - S$ tiene menos de $|V| - i$ vértices, G_1 tiene menos de $(|V| - i)/2$. Si v$\in G_1$, v es adyacente a vértices de S o a otros vértices de G_1 solamente (si fuera adyacente a vértices en $G - S - G_1$ no se habría "desconectado" y no sería una componente conexa por sí sola). De esto,

$$\begin{aligned}
\delta(G) &\leq \delta(\text{v}) \\
&\leq i + \frac{|V| - i}{2} - 1 \\
&= \frac{|V| + i - 2}{2} \\
&< \frac{|V| + k - 2}{2},
\end{aligned}$$

lo que contradice nuestra hipótesis. ❑

Relacionamos ahora $|E|$ con $\kappa(G)$.

Lema C.14 *Si* $G = (V, E)$ *es* $k-$*conexa por vértices o aristas, entonces*

$$|E| \geq \frac{k \cdot |V|}{2}.$$

Demostración:

Tenemos los siguientes resultados:

$$\begin{aligned}
|E| &= \frac{1}{2}\sum_{v \in V} \delta(v) && \text{Lema 1.1} \\
&\geq \frac{|V| \cdot \delta(G)}{2} && \text{Definición C.25} \\
&\geq \frac{|V| \cdot \kappa(G)}{2} && \text{Lema C.9} \\
&= \frac{|V| \cdot k}{2} && \text{Definición C.26.}
\end{aligned}$$

❑

Con esto damos por terminado los conceptos requeridos para el desarrollo de las distintas especializaciones para la exploración de gráficas.

Apéndice D

Colofón de exploración en gráficas

En este apéndice revisaremos aspectos adicionales de las distintas exploraciones que revisamos, que están relacionados con teoría de gráficas pero que no dependen directamente de la especialización dada en este trabajo para los distintos algoritmos.

D.1. Exploración en amplitud

Una manera de obtener el diámetro de un árbol es tomando las distancias entre todas las parejas de vértices y determinando cuál es la mayor. Esto se puede lograr ejecutando ExploracionBFS colocando como origen a cada uno de los vértices de G, y en cada ejecución anotamos la distancia entre el elegido como origen y el último vértice explorado. Al terminar las $|V|$ ejecuciones, determinamos cuál es la distancia mayor. La complejidad de este algoritmo es $O(|V|^2)$.

Sin embargo, se puede obtener el diámetro de un árbol ejecutando ExploracionBFS dos veces, de la siguiente manera: se ejecuta una primera vez desde cualquier origen, y se anota cuál es el último vértice descubierto; a continuación se ejecuta nuevamente ExploracionBFS usando como origen al último vértice descubierto en la primera ejecución. El diámetro del árbol estará dado por la distancia entre el segundo origen y el último vértice visitado en la segunda ejecución. La complejidad de este algoritmo es $O(|V|)$. La demostración de que este algoritmo es correcto aparece en [DSL]; la reproducimos a continuación.

Lema D.1 *Sea G un árbol. Sea* s *el vértice desde el que se ejecuta* explorationGenerica *de* ExploracionBFS *por primera vez;* u *el último vértice visitado y el origen de una segunda ejecución de* exploracionGenerica *de* ExploracionBFS*; y* v *el último vértice visitado en esta segunda ejecución. Entonces*

$$\text{diam}(G) = \delta(\mathsf{u}, \mathsf{v})$$

Demostración:

Como G es originalmente un árbol, existe un único camino entre cualesquiera dos vértices.

Sean x y y dos vértices cualesquiera tales que $\delta(\mathsf{x}, \mathsf{y})$ es el diámetro del árbol. Existe un camino único de x a y. Sea t el primer vértice en ese camino descubierto por exploracionGenerica.

Si los caminos $p_1 = \mathsf{s} \rightsquigarrow \mathsf{u}$ y $p_2 = \mathsf{x} \rightsquigarrow \mathsf{y}$ no tienen aristas en común, entonces el camino de t a u incluye a s. Por lo tanto,

$$\begin{aligned}
\delta(\mathsf{t}, \mathsf{u}) &\geq \delta(\mathsf{s}, \mathsf{u}) \\
\delta(\mathsf{t}, \mathsf{u}) &\geq \delta(\mathsf{s}, \mathsf{x}) \\
\delta(\mathsf{t}, \mathsf{u}) &\geq \delta(\mathsf{t}, \mathsf{x}) \\
\delta(\mathsf{y}, \mathsf{u}) &\geq \delta(\mathsf{y}, \mathsf{x}).
\end{aligned}$$

Como $\delta(\mathsf{x},\mathsf{y}) \geq \delta(\mathsf{u},\mathsf{y})$, entonces $\delta(\mathsf{x},\mathsf{y}) = \delta(\mathsf{u},\mathsf{y})$.

Si los caminos p_1 y p_2 comparten aristas, entonces t está en p_1. Como u fue el último nodo encontrado por exploracionGenerica de ExploracionBFS, $\delta(\mathsf{t},\mathsf{u}) \geq \delta(\mathsf{t},\mathsf{x})$. Como p_2 es el camino más largo, $\delta(\mathsf{t},\mathsf{x}) \geq \delta(\mathsf{t},\mathsf{u})$. De donde $\delta(\mathsf{t},\mathsf{x}) = \delta(\mathsf{t},\mathsf{u})$ y $\delta(\mathsf{u},\mathsf{y}) = \delta(\mathsf{x},\mathsf{y})$.

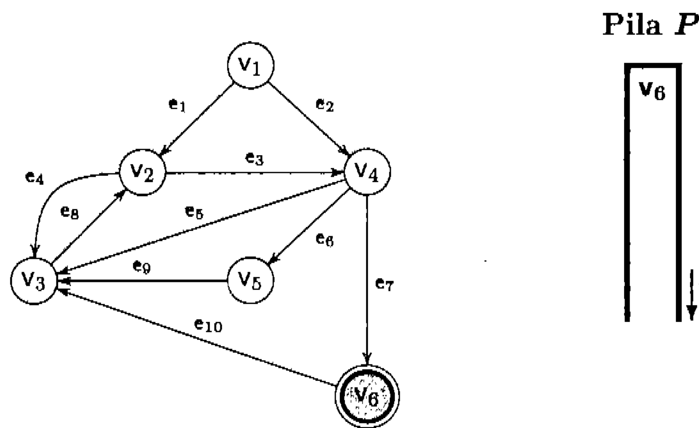
$\delta(\mathsf{x},\mathsf{y}) \geq \delta(\mathsf{u},\mathsf{v})$ y $\delta(\mathsf{u},\mathsf{v}) \geq \delta(\mathsf{u},\mathsf{y})$, por lo que los tres son iguales. Entonces $\delta(\mathsf{u},\mathsf{v})$ es el diámetro del árbol. ❏

D.2. Exploración a profundidad

D.2.1. Ejecución de ExploracionDFS con origen alternativo

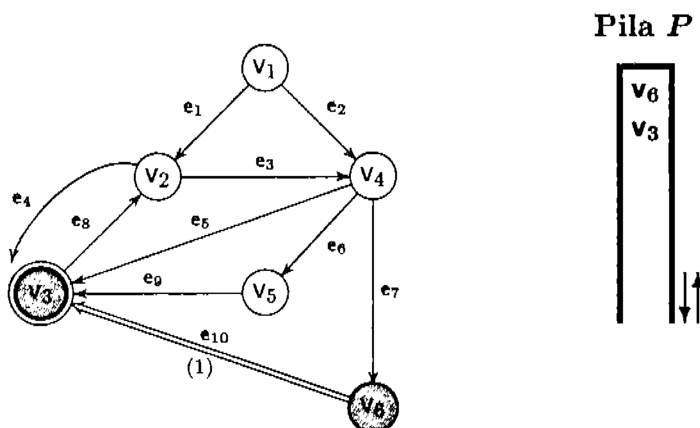
Cuando se ejecuta DFS sobre una gráfica dirigida que no sea fuertemente conexa, la elección del origen puede provocar que no se pueda explorar toda la gráfica. En la exposición de la especialización para DFS vimos un ejemplo de exploración eligiendo a un vértice determinado. Por completez, presentamos acá un ejemplo con una elección alternativa para el vértice origen, a la dada en la página 137 del Capítulo 5, tomando a v_6 como origen de la exploración.

Figura D.1: Estado inicial de DFS con v_6 como origen



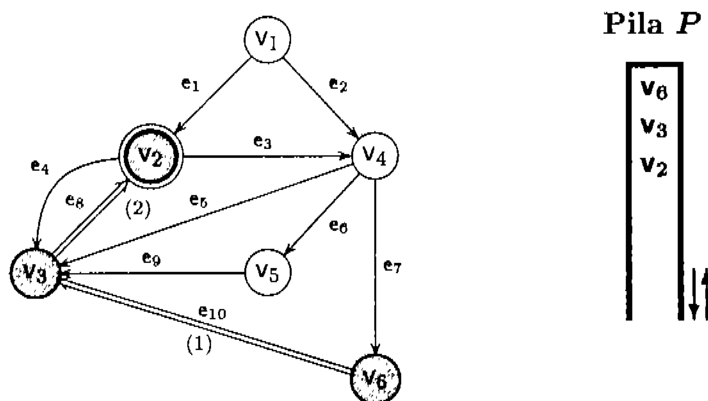
Desde v_6 DFS recorre el arco e_{10} descubriendo a v_3, a quien empuja a la pila, quedando v_3 como el centro de acción para la siguiente iteración, como se muestra en la Figura D.2.

Figura D.2: Primera iteración con origen alternativo



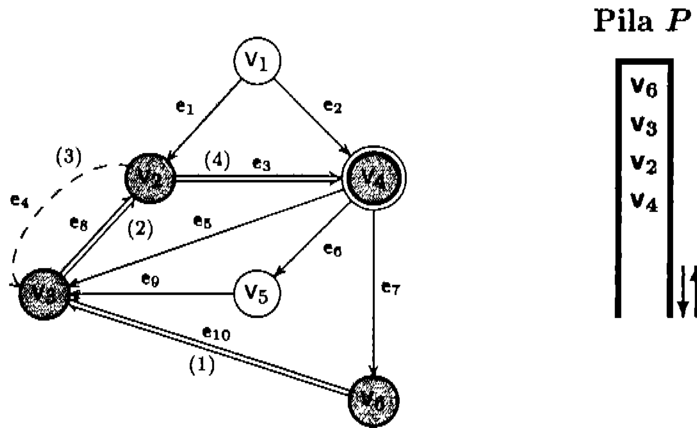
Desde v_3 se descubre a v_2 y se le empuja a la pila, quedando v_2 como el centro de acción para la siguiente iteración, como se muestra en la Figura D.3.

Figura D.3: Segunda iteración con $s = v_6$



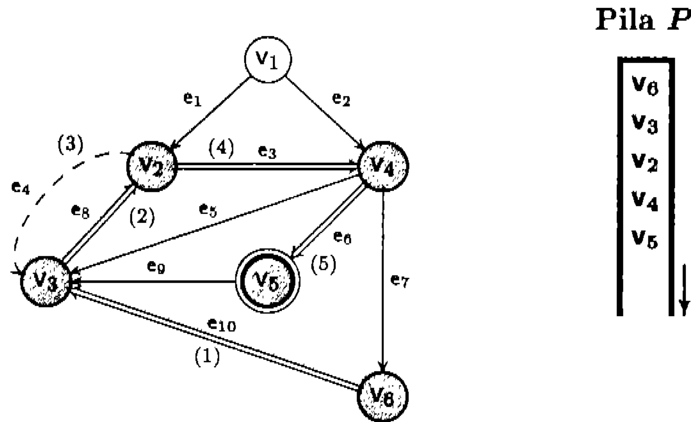
A continuación DFS recorre el arco e_4 desde v_2, pero como el destino del arco es un vértice GRIS, no hace nada y vuelve a iterar. En la siguiente iteración, descubre a v_4 y lo empuja a la pila, dejándolo como siguiente centro de acción. Esto se puede ver en la Figura D.4.

Figura D.4: Dos iteraciones con $v_a = v_2$



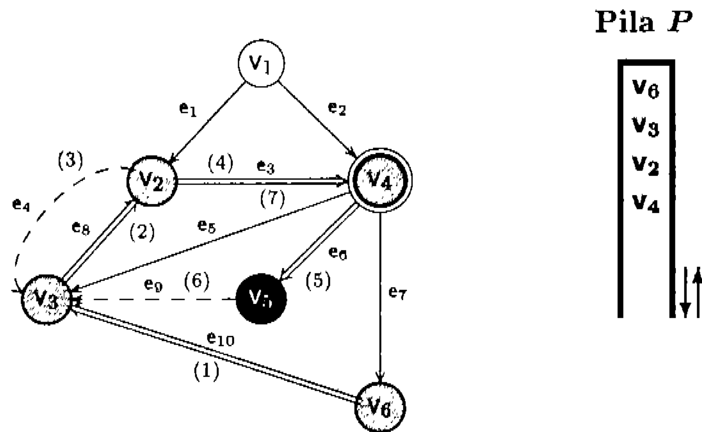
En seguida, con v_4 como centro de acción, descubre a v_5 y lo empuja a la pila, dejándolo como centro de acción para la próxima iteración, como se puede observar en la Figura D.5.

Figura D.5: Una iteración con $v_a = v_4$



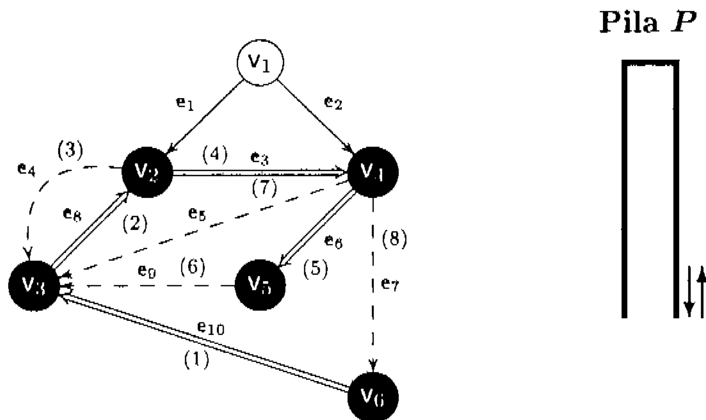
Con v_5 como centro de acción se recorre el arco e_9, pero como su destino ya fue descubierto, en esa iteración únicamente se procesa el arco. En la siguiente iteración, con v_5 nuevamente como centro de acción, y como ya no tiene arcos disponibles que salgan de él, se le pinta de NEGRO y se le saca de la pila, dejando a v_4 como el centro de acción para la siguiente iteración. Esto se observa en la Figura D.6.

Figura D.6: Iteraciones con v_5 como centro de acción



Con v_4 como centro de acción se recorren los arcos e_5 y e_7, pero como para ambos su extremo ya fue descubierto, no se incluyen en G_π. Al ya no haber más arcos disponibles, se saca a v_4 de la pila y se coloca a v_2 como centro de acción, que tampoco tiene ya arcos disponibles, por lo que se le saca de la pila, dejando a v_3 como centro de acción. Nuevamente, como v_3 ya no tiene arcos disponibles, se le saca de la pila, lo mismo que a v_6 que es el siguiente centro de acción. Con esto se vacía la pila y termina el algoritmo.

Figura D.7: Situación al vaciarse la pila



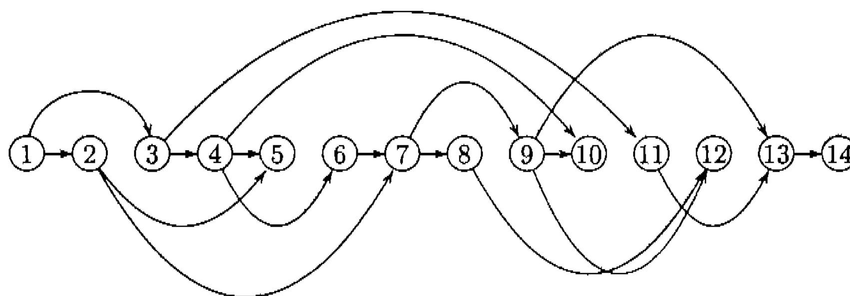
Como el vértice v_1 no es alcanzable desde v_6, al terminar la ejecución quedó pintado de blanco.

D.2.2. Ordenamiento topológico

Decimos que tenemos un *orden topológico* entre los vértices de una gráfica dirigida si podemos dibujarla de tal manera que todos los vértices estén sobre una línea horizontal y

todos los arcos vayan de izquierda a derecha. Más formalmente, si podemos asignar un índice $i = 1, \ldots, n$ a cada vértice de la gráfica dirigida de manera que $\nexists\, \mathsf{v}_k \to \mathsf{v}_\ell$ tal que $\ell < k$. En la Figura D.8 los vértices presentan un orden topológico, donde el índice asignado aparece como etiqueta de cada vértice.

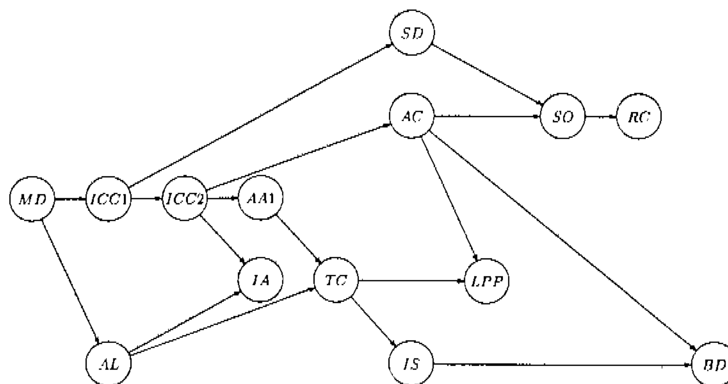
Figura D.8: Gráfica con ordenamiento topológico



Ejemplo

Las asignaturas que hay que cubrir para terminar la licenciatura en Ciencias de la Computación están relacionadas entre sí por la seriación. Por ejemplo, para llevar Cálculo Diferencial e Integral II debe haberse cubierto Cálculo Diferencial e Integral I, mientras que para cursar Álgebra Superior II se debe haber cursado Álgebra Superior I. Veamos la digráfica que identifica la seriación para las materias directamente relacionadas con computación en la Figura D.9 (en ella no mostramos la seriación con las materias de Matemáticas y por claridad no se dibuja en una sola línea). Se puede observar que todos los arcos van hacia la derecha, y que en el caso de esta relación de orden tenemos un único origen. Asimismo vale la pena notar que la digráfica no corresponde a un árbol.

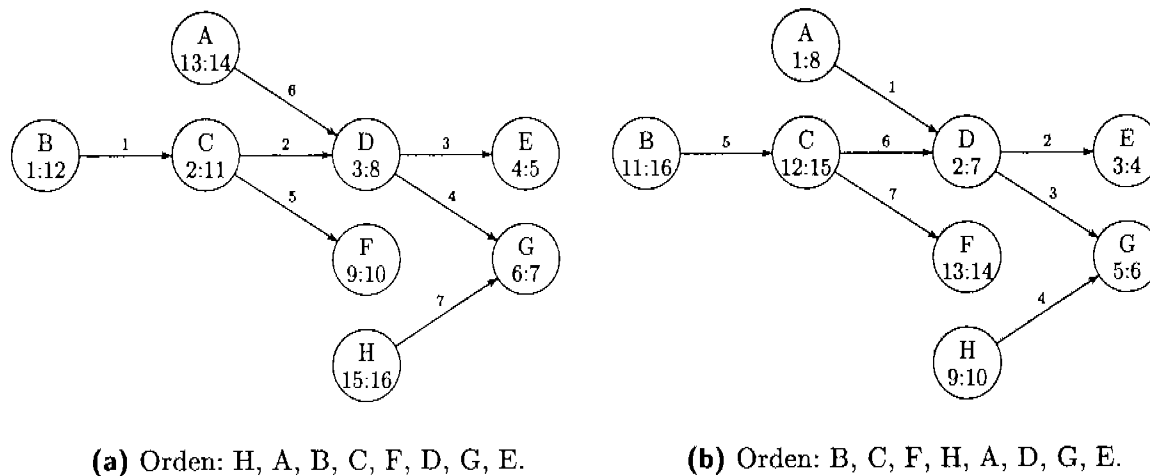
Figura D.9: Seriación en la carrera de Ciencias de la Computación



Es decir, se puede pensar en un orden topológico como una serialización de los eventos, cada uno de ellos representados por un vértice, consistente con el orden parcial. Las posibles serializaciones corresponden a las posibilidades que un alumno tiene para llevar las materias una tras otra sin violar prerrequisitos.

Para obtener un orden topológico de las actividades, basta ejecutar exploracionGenerica de ExploracionDFS sobre la gráfica, marcando los tiempos en los vértices. El orden topológico estará dado por los tiempos fin[v] asignados por ExploracionDFS, ordenados éstos de manera descendente: esto es, el último vértice que se termina de explorar es la primera actividad que se debe realizar. Observemos la gráfica de la Figura D.10, que tiene tres posibles orígenes (de los que mostramos únicamente dos). Es claro que el origen de la exploración debe ser algún vértice con ingrado 0, puesto que la primera actividad a realizarse deberá ser alguna que no tenga prerrequisitos.

Figura D.10: Ordenamiento topológico usando ExploracionDFS



(a) Orden: H, A, B, C, F, D, G, E. **(b)** Orden: B, C, F, H, A, D, G, E.

De no existir ningún vértice con esta característica, sabemos que la digráfica tiene algún ciclo y el ordenamiento topológico no es posible. Veamos cómo, en efecto, el orden inverso de los tiempos fin proporcionan el orden adecuado para que, cuando se va a llevar a cabo cierta actividad, los prerrequisitos ya están cubiertos. En la Figura D.10 etiquetamos los arcos con un entero que corresponde al orden en que fueron recorridos. Los vértices están etiquetados con un identificador, y bajo él las marcas de reloj con el formato desc : fin.

Si se observan las dos ejecuciones que se muestran (hay muchas otras posibles), la actividad correspondiente a cada vértice debe ejecutarse antes que las actividades de los vértices alcanzables desde él. Éste es el orden parcial al que nos referimos.

D.2.3. DFS en el direccionamiento de calles

Tomando en cuenta las definiciones y propiedades para componentes no separables dadas en el Apéndice C, en el Lema D.2 mostramos en qué condiciones ExploracionDFS puede asignar dirección a las calles de una ciudad. Se desea que con la dirección asignada por ExploracionDFS se pueda llegar de cualquier punto a cualquier punto de la ciudad.

Lema D.2 *La gráfica $\vec{G} = (V, \vec{E})$, resultado de ejecutar explorar exploracionGenerica de ExploracionDFS sobre $G = (V, E)$ es fuertemente conexa $\iff$ G es conexa y no tiene puentes.*

Demostración:

$\Longrightarrow$ Suponemos que $\vec{G} = (V, \vec{E})$ es fuertemente conexa. Debemos demostrar que G es conexa y no tiene puentes. La primera parte se cumple, ya que si existe un camino dirigido entre cualesquiera dos vértices de $\vec{G}$, si le quitamos la dirección a los arcos a alguno de esos caminos, cumple con ser un camino en G, por lo que G es conexa.

Para la segunda parte, por contrapositivo supongamos que G tiene un puente e $=$ x—y. Ese puente, en $\vec{G}$, tiene una única dirección. Supongamos que el camino entre dos vértices u y v incluye a ese puente y queda orientado desde u hacia v, u$\rightsquigarrow$x$\rightarrow$y$\rightsquigarrow$v. Como x—y es un puente en G, todo camino entre u y v en G debe incluir a ese puente – si hubiera algún camino entre u y v que no incluyera a e, entonces al quitar a e no se separaría a u y v, contradiciendo el que e es puente. Pero si la dirección de x—y es x$\rightarrow$y, no existe ningún camino de v a u en $\vec{G}$, lo que dice que $\vec{G}$ no es fuertemente conexa.

$\Longleftarrow$ Suponemos ahora que G es conexa y que no tiene puentes. Observemos lo siguiente:

i. Como G es conexa, ExploracionDFS explora a todos los vértices de G y recorre a todas las aristas, no importando el vértice origen s elegido.

ii. $\exists$ un camino dirigido s$\rightsquigarrow$v, $\forall$v $\in V$.

iii. Tenemos que demostrar ahora que $\exists$ un camino dirigido u$\rightsquigarrow$s, $\forall$u $\in V$. Con esto, tendremos que $\forall$u, v $\in V$ existe un camino dirigido u$\rightsquigarrow$s $\rightsquigarrow$v.

Si u $=$ s, queda demostrado. Supongamos entonces que u $\neq$ s. Tomemos el camino que va de s a u, y digamos que es s—v_1—v_2— ... —v_k—u con $k \geq 0$. Por inducción sobre k, demostraremos que siempre hay un camino de u a s.

<u>Base:</u>: Si $k = 0$ el arco s $\rightarrow$ u $\in G_\pi$, de donde la arista s—u $\in G$. Pero como G no tiene puentes, debe existir algún otro camino entre s y u además del dado por esta arista. Como G_π es un árbol, ese otro camino no puede consistir únicamente de aristas de G_π, pues entonces habría más de un camino en G_π entre dos vértices. Contiene entonces aristas hacia atrás. Como hay al menos dos caminos entre s y u, s debe tener alguna arista hacia atrás en $\vec{G}$, que llegue desde u o desde algún descendiente de u. Un segundo camino entre s y u en $\vec{G}$ es salir de u y bajar por el árbol G_π hasta llegar a aquel descendiente que tenga la arista hacia atrás a s.

<u>Inducción:</u>: Supongamos que si el camino entre s y u en G_π es de longitud k, existe un camino dirigido en $\vec{G}$ u$\rightsquigarrow$s. Veamos qué pasa con un camino de longitud $k+1$. Sea ese camino s$\rightsquigarrow$u.π $\rightarrow$ u. Por la hipótesis de inducción, existe un camino dirigido en $\vec{G}$ u.$\pi$$\rightsquigarrow$s. Como G no tiene puentes, existe otro camino entre u.π y u y este camino contiene alguna arista hacia atrás en $\vec{G}$ que termina en u.π y sale de u o algún

descendiente de u. Por lo que el camino

$$\mathsf{u} \rightsquigarrow \mathsf{u}.\pi \rightsquigarrow \mathsf{s}$$

es un camino dirigido en $\vec{G}$ de u a s.

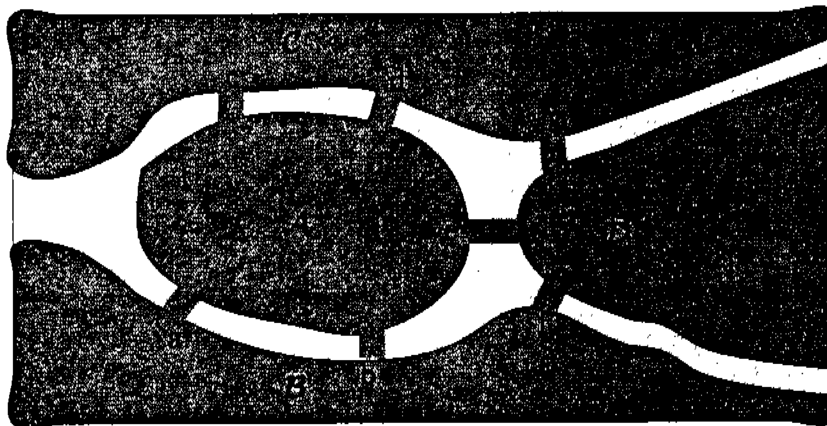
$\therefore\ \forall \mathsf{u}, \mathsf{v} \in V, \exists \mathsf{u} \rightsquigarrow \mathsf{v} \quad \text{y} \quad \mathsf{v} \rightsquigarrow \mathsf{u} \in \vec{G}.$ ❏

D.3. Gráficas eulerianas

El problema que dio origen a la teoría de gráficas y a la topología es el conocido con el nombre de *problema de los puentes de Königsberg*, que fue resuelto por el matemático suizo Leonhard Euler(1707-1782) en 1736. Este era un *acertijo* de los habitantes de la ciudad prusa de Königsberg (hoy con el nombre de Kaliningrad), que consistía de lo siguiente:

> En Königsberg había dos islas en el río Pregel, comunicadas entre sí y con las riberas del río por siete puentes (como se muestra en la Figura D.13). El paseo por los puentes era uno de los entretenimientos favoritos de los habitantes de la ciudad, y de manera natural surgía la pregunta de si era posible planear un paseo de tal manera de cruzar cada puente exactamente una vez, empezando y terminando en el mismo lugar.

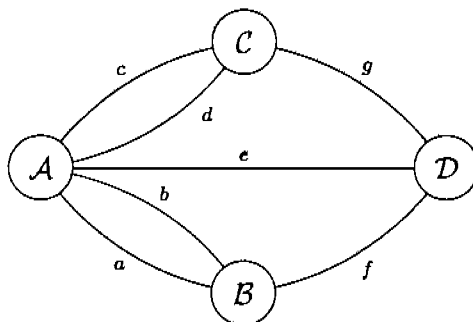
Figura D.11: Los puentes de Königsberg



Euler encontró un principio matemático escondido en el problema. En la solución que presentó a la Academia Rusa en San Petersburgo en 1735 sustituyó las áreas de tierra por puntos y los puentes por líneas que conectaban estos puntos; a los puntos les llamó *vértices*, a las líneas *aristas* y a la configuración le llamó una *gráfica*. La gráfica que representa a los puentes de Königsberg está en la Figura D.12.

Con esta abstracción para el problema, éste se reduce a recorrer la gráfica utilizando cada arista exactamente una vez. El resultado que presentó Euler fue en sentido negativo, ya que demostró que realizar el paseo de la manera indicada no era posible.

Figura D.12: Gráfica correspondiente a los puentes de Königsberg



El problema que atacó Euler es conocido hoy en día como el problema del *circuito euleriano*. Si se quita la restricción de empezar y terminar en el mismo vértice, entonces estaremos hablando de una *trayectoria euleriana*. Las definiciones formales son como sigue:

Definición D.1 (Trayectoria euleriana) Sea $G = (V, E)$ una gráfica. Una *trayectoria euleriana de* G es un camino $\langle e_{i_1}, e_{i_2}, \ldots, e_{i_m} \rangle$ tal que cada arista de G aparece exactamente una vez. De donde $m = |E|$ es la longitud del camino.

Definición D.2 (Circuito euleriano:) Un *circuito euleriano* de $G = (V, E)$ es una trayectoria euleriana de G tal que empieza y termina en el mismo vértice.

Este problema tiene muchas aplicaciones, de las que mencionaremos por el momento dos.

- Supongamos que tenemos una red de comunicaciones y deseamos probar que todas las líneas de comunicación están funcionando correctamente. Para minimizar el costo de esta prueba es deseable diseñar una estrategia que pruebe cada una de las líneas exactamente una vez. Esta estrategia deberá ser, de hecho, una que encuentre una trayectoria euleriana en la gráfica dada por la red de comunicaciones.
- Tenemos un servicio de recoger la basura y queremos que el camión de la basura pase por cada calle exactamente una vez. Si representamos al mapa de la ciudad con vértices para las intersecciones y aristas para las calles, lo que deseamos es encontrar un circuito euleriano que, saliendo de la terminal de los camiones de basura, pase por cada calle exactamente una vez y regrese a la terminal.

El caso de los Puentes de Königsberg es el de una gráfica no dirigida, ya que cualquiera de los puentes podía recorrerse en cualquier dirección. Pero en el segundo ejemplo que dimos es posible que las calles sean en un solo sentido, por lo que las aristas son dirigidas, y el camión sólo las podrá recorrer en la dirección marcada. El problema de la trayectoria o circuito eulerianos, entonces, es importante también en gráficas dirigidas.

Euler logró caracterizar a aquellas gráficas que tienen un circuito (trayectoria) euleriano dando condiciones necesarias y suficientes, que se exponen en el siguiente teorema.

Teorema D.3 *Sea $G = (V, E)$ una gráfica conexa. G tiene una trayectoria euleriana* $\iff$

tiene exactamente dos vértices de grado impar. G tiene un circuito euleriano $\iff$ *todos sus vértices son de grado par.*

Demostración:

$\Longrightarrow$ Supongamos que G tiene una trayectoria euleriana y sea esta trayectoria

$$P = \langle e_{i_1}, e_{i_2}, \ldots, e_{i_m} \rangle$$

y los vértices por los que pasa el camino $\{v_{j_0}, v_{j_1}, \ldots, v_{j_m}\}$ (el camino tiene $m+1$ vértices y m aristas, y uno o más vértices pudieran repetirse). Excepto por el primero y último vértices del camino, cada vez que aparece un vértice tiene una arista por la que se llega a él y otra por la que se sale de él. Como el camino es una trayectoria euleriana, en este camino están listadas **todas** las aristas de la gráfica. De esto, todos los vértices en el camino, excepto por el primero y el último, tienen grado par.

Si el primero y último vértice son el mismo, tenemos un circuito euleriano, y el grado de ese vértice se incrementa en 2, uno por la arista por la que se sale la primera vez, y otra unidad por la arista por la que se entra la última vez. Si el primero y último vértice no son el mismo, el grado de cada uno de ellos se incrementa en 1, haciéndolos de grado impar.

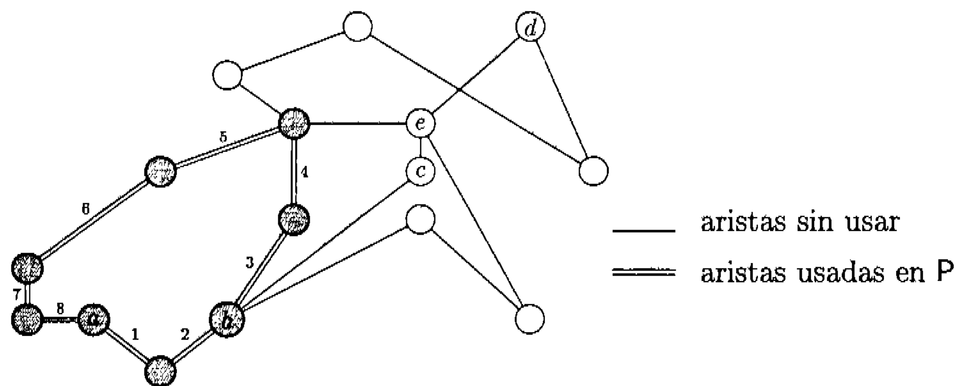
De esto, si se trata de una trayectoria euleriana, el grado de todos los vértices excepto 2 es par, y si se trata de un circuito euleriano, el grado de todos los vértices es par.

$\Longleftarrow$ Suponemos que todos los vértices tienen grado par. Procederemos a describir informalmente un algoritmo que demuestra el teorema.

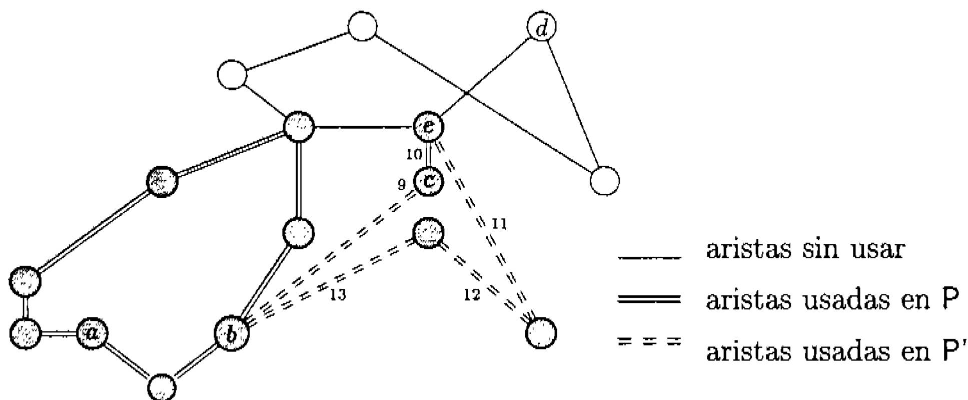
Empezamos en un vértice cualquiera a. Tomamos cualquier arista incidente en a que no haya sido recorrida, la agregamos al camino y la recorremos (nos trasladamos al vértice del otro extremo de la arista elegida). En general, al llegar a algún vértice, tratamos de salir por alguna de sus aristas que no hayamos usado. Si el vértice no es a, cada vez que llegamos a él podemos salir, ya que el grado es par, y por lo tanto usamos dos de las aristas incidentes en él. Como originalmente su grado es par, después de salir de él, el número de aristas incidentes en él, sin usar, sigue siendo par. Entonces, si llegamos a él, existe al menos una arista por la cual podamos salir. Todo esto implica que el único vértice en el que nos podemos atorar es el mismo a. Cuando esto sucede tenemos un circuito que empieza y termina en a, como se puede ver en la Figura D.13(a).

Si todavía hay alguna arista e que no esté en este circuito, como sucede en la gráfica de la Figura D.13(b) con la arista e = d—e, como la gráfica es conexa, hay alguna arista e′ – en la gráfica que estamos trabajando e′ = b—c – no usada incidente en el circuito (debe haber un camino entre alguno de los vértices en los que incide la arista e y algún vértice en el circuito). La arista b—c es una arista que no está en el camino e incidente en el vértice b que sí está en el camino. Por lo que podemos volver a ejecutar el algoritmo saliendo de b, y obteniendo un nuevo circuito P' que empieza y termina en b, como se puede observar en la Figura D.13(b). Se procede a incorporar P' a P.

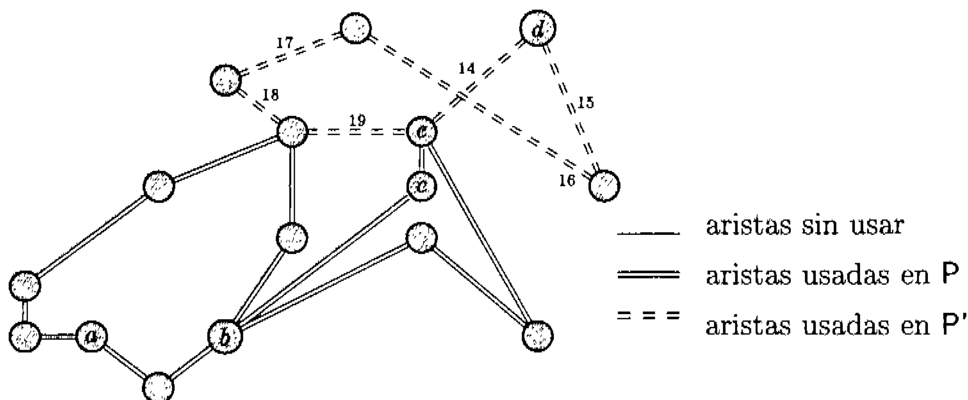
Figura D.13: Algoritmo de construcción de circuito euleriano



(a) Se incorpora el primer circuito



(b) Se incorpora un segundo circuito



(c) Se incorpora un tercer circuito

Sigue habiendo aristas sin usar. La arista d−e es incidente a $P \cup P'$, por lo que el vértice e se usa para armar otro nuevo circuito P'', como se puede ver en la Figura D.13(c). La manera de incorporar cada circuito nuevo al que ya se tiene es "colgando" al nuevo circuito del viejo circuito precisamente en el vértice en el que empieza el nuevo circuito.

Continuamos de esta manera hasta que ya no haya aristas sin usar.

El caso de que exactamente dos vértices sean de grado impar se resuelve de manera similar, con la salvedad de que al construir el primer camino se empezará en uno de los vértices de grado impar y deberá terminar en el otro vértice de grado impar. Todos los caminos que se vayan construyendo, excepto por el primero, serán circuitos. ❑

Si se trata de gráficas dirigidas, ¿cuál será la condición que permita construir la trayectoria (circuito) euleriana(o)? Si observamos la demostración del Teorema D.3 podemos determinar que la característica fundamental para que se puedan recorrer todas las aristas, es que cada vez que se llegue a un vértice haya alguna arista por donde salir. En el caso de digráficas, tenemos que traducir esto a la siguiente condición:

$$\forall \mathsf{v} \in V, \quad exgrado(\mathsf{v}) = ingrado(\mathsf{v})$$

y esto nos garantizaría que tenemos un circuito euleriano:

Teorema D.4 *Sea $G = (V, E)$ una digráfica cuya gráfica subyacente es conexa. G tiene un circuito euleriano dirigido $\iff \forall \mathsf{v} \in V, \quad exgrado(\mathsf{v}) = ingrado(\mathsf{v})$. G tiene una trayectoria euleriana dirigido $\iff \exists\ \mathsf{v}_i,\ \mathsf{v}_j \in V, \quad \mathsf{v}_i \neq \mathsf{v}_j$ tales que*

$$\begin{aligned} exgrado(\mathsf{v}_i) &= \ ingrado(\mathsf{v}_i) + 1 \qquad e \\ ingrado(\mathsf{v}_j) &= \ exgrado(\mathsf{v}_j) + 1. \end{aligned}$$

Esto para exactamente dos vértices v_i y v_j que son los extremos de la trayectoria euleriana dirigida, con v_i el primer vértice y v_j el último en ella. Y $\forall\ \mathsf{v} \in V$ tales que $\mathsf{v} \neq \mathsf{v}_i \neq \mathsf{v}_j$, $ingrado(\mathsf{v}) = exgrado(\mathsf{v})$.

Demostración:

La demostración es prácticamente igual a la del Teorema D.3 para el caso de gráficas no dirigidas, por lo que se omite. ❑

Definición D.3 (Gráfica euleriana) Una *gráfica euleriana* es aquélla que tiene un circuito (o trayectoria) euleriano(a).

D.4. Propiedades de gráficas eulerianas

Abusando un poco de la formalidad, diremos que G es conexa si la gráfica subyacente no dirigida es conexa, aún cuando sabemos que este término no se aplica a digráficas.

Lema D.5 *Sea $G = (V, E)$ una gráfica euleriana. Entonces $\forall\ \mathsf{v} \in V$, v es alcanzable desde s, i.e. $\exists\ \mathsf{s} \rightsquigarrow \mathsf{v} \in G$.*

Demostración:

Observemos lo siguiente:

i. Como G es euleriana, existe un circuito que empieza en s y termina en s que incluye a todos los arcos de la gráfica exactamente una vez.
ii. Esto quiere decir que cada vértice aparece en ese circuito al menos una vez.
iii. Por lo que existe un camino dirigido desde s a cada $\mathsf{v} \in V$.

$\therefore\ \forall \mathsf{v} \in V, \quad \exists\, \mathsf{s} \rightsquigarrow \mathsf{v}.$

D.5. Sucesiones de de Bruijn

Una de las aplicaciones importantes del algoritmo de exploración de Euler es el de la construcción de *sucesiones de de Bruijn*, que tienen que ver con codificación de cadenas de manera lo más económicamente posible. Veamos qué son las sucesiones de de Bruijn.

Sea $\Sigma = \{0, 1, \ldots, \sigma - 1\}$ un *alfabeto* de σ *símbolos* o *letras*. Tenemos entonces las siguientes definiciones:

Definición D.4 (Palabra) Una *palabra* $\boldsymbol{w}$ es una sucesión de símbolos de Σ,

$$w = a_1 a_2 \ldots a_k, \qquad a_i \in \Sigma.$$

Definición D.5 (Longitud) La *longitud* de $\boldsymbol{w}$ es $\boldsymbol{k}$, el número de símbolos en w y lo denotamos de la manera usual $|\boldsymbol{w}| = \boldsymbol{k}$.

Veamos el siguiente ejemplo. Si $\Sigma = \{0, 1, 2\}$, podemos formar, entre otras, las siguientes palabras:

$$w_1 = "201102", \qquad w_2 = "0000", \qquad w_3 = "2", \qquad w_4 = "".$$

Observación: El número de palabras con n letras sobre Σ es exactamente σ^n.

Definición D.6 (Sucesión de de Bruijn) Una *sucesión de de Bruijn* (o sucesión de Bruijn) es una sucesión $a_0 a_1 \ldots a_{\ell-1}$ sobre Σ tal que para cada palabra w de longitud n sobre Σ, existe una única i tal que

$$w = a_i a_{i+1} \ldots a_{i+n-1},$$

donde los índices se toman módulo ℓ. Pensamos entonces en la cadena que representa a la sucesión como circular, esto es, al símbolo en la posición $\ell - 1$ le sigue el de la posición 0.

En las sucesiones de de Bruijn, $\ell = \sigma^n$. Si tuviéramos que codificar cada palabra completa por separado, el número de posiciones que requeriríamos sería $n \cdot \sigma^n$, que puede ser un factor muy significativo.

El caso más relevante para computación es con $\sigma = 2$, i.e. $\Sigma = \{0, 1\}$.

Por ejemplo, si queremos construir una sucesión de de Bruijn para las palabras de longitud $n = 3$ tomadas del alfabeto $\sigma = \{0, 1\}$, una sucesión adecuada sería la que sigue:

sucesión →	1	0	0	0	1	0	1	1
posición →	0	1	2	3	4	5	6	7

La palabra 001, por ejemplo, ocupa las posiciones 2, 3 y 4, y no hay otras tres posiciones consecutivas (cíclicamente) donde esta palabra se encuentre. Similarmente, la palabra 111 ocupa las posiciones consecutivas 6, 7 y 0, y no se puede armar a partir de ninguna otra posición.

El objetivo de esta sección es probar la existencia de sucesiones de de Bruijn para cualquier alfabeto con dos o más símbolos ($\sigma \geq 2$) y para toda n. Pero hagamos primero un intento de construir una sucesión de de Bruijn, para ver que no es tarea trivial si lo tratamos de hacer por prueba y error.

Ejemplo

Construir una sucesión de de Bruijn para $\sigma = 2$ y $n = 3$, esto es, para todas las posibles cadenas binarias de longitud 3. La sucesión de de Bruijn deberá tener una longitud total de $2^3 = 8$.

Podemos empezar de la siguiente manera:

Vamos construyendo:	Cadenas incluidas
000	000
0001	001
00010	010
000100	100

Pero a partir de este momento, ya no podemos continuar, porque si agregamos un 0, tenemos 0001000 y la palabra 000 empieza a partir del lugar 0 y también a partir del lugar 4, contradiciendo la definición dada. Y si agregamos un 1, entonces tendríamos 0001001, con lo que la palabra 001 también empieza en dos posiciones, la 1 y la 4. De esto, podemos ver que la construcción de sucesiones de de Bruijn no es tan sencilla. Una sucesión de de Bruijn para $\sigma = 2$ y $n = 3$ es "00011101". Otra más es la que dimos arriba.

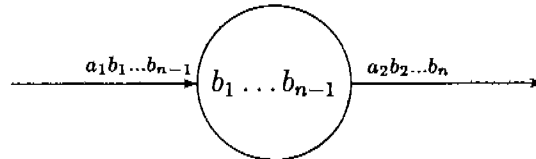
D.5.1. Construcción de sucesiones de de Bruijn

Consideremos la gráfica dirigida $G_{\sigma,n}(V, E)$, donde:

a. Tenemos un vértice por cada palabra de longitud $n - 1$ que se puede formar con los símbolos del alfabeto ($|V| = \sigma^{n-1}$). Etiquetamos a los vértices con palabras de longitud $n - 1$, $b_1 b_2 \ldots b_{n-1}$, sin repetir etiquetas.

b. Tenemos una arista por cada palabra de longitud n que podemos formar con los símbolos del alfabeto ($|E| = \sigma^n$). Etiquetamos a las aristas con estas palabras sin repetir etiquetas.
c. La arista etiquetada con $b_1b_2 \dots b_n$ corresponde a $b_1b_2 \dots b_{n-1} \rightarrow b_2 \dots b_{n-1}b_n$; sale del vértice etiquetado con $b_1b_2 \dots b_{n-1}$ y llega al vértice etiquetado con $b_2 \dots b_{n-1}b_n$ – ver Figura D.14.

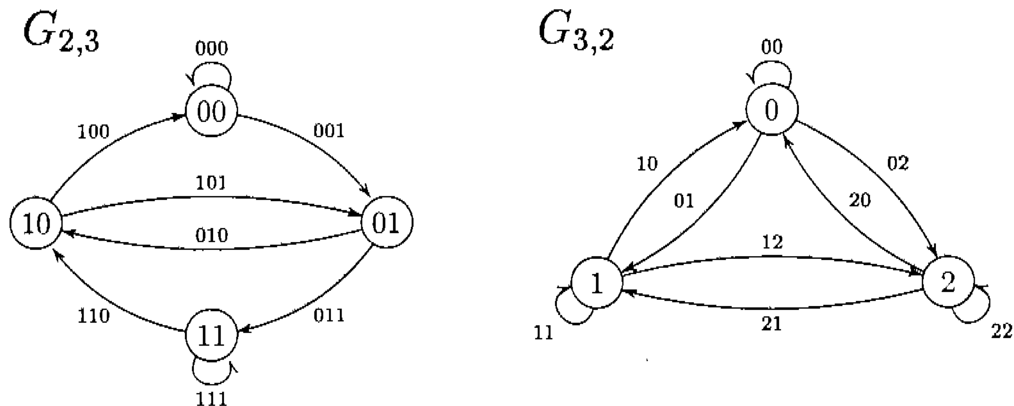
Figura D.14: Estructura de las etiquetas de los vértices y aristas



Del etiquetado de los vértices y aristas podemos ver que para etiquetar las aristas se toman los $n - 1$ símbolos que corresponden al vértice origen y como n-ésimo símbolo se le asigna el último símbolo del vértice destino.

Veamos algunas gráficas $G_{\sigma,n}$ en la Figura D.15. A estas gráficas se les llama *gráficas de de Bruijn*, y su estructura es tal que *el símbolo a_2 puede seguir al símbolo a_1 en una sucesión de de Bruijn* solamente si existe una arista etiquetada con $a_1a_1 \dots b_{n-1}$ que llega al vértice etiquetado con $a_2 \dots b_{n-1}$, y de ahí sale una arista etiquetada con $a_2b_2 \dots b_n$.

Figura D.15: Gráficas para la construcción de las sucesiones de de Bruijn

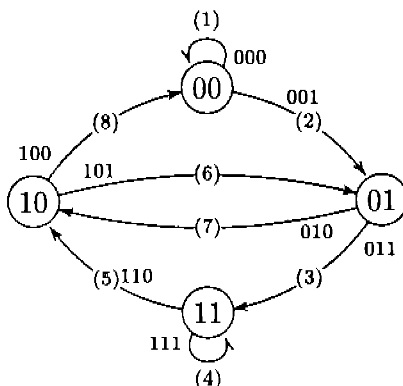


Si dada una gráfica de de Bruijn construimos un circuito dirigido euleriano, podemos a partir de este circuito construir una sucesión de de Bruijn. Para construir esta última se toma el primer símbolo de la etiqueta de cada arista recorrida con el algoritmo de Euler para circuitos eulerianos, en el orden en que se recorren las aristas, y se van concatenando estos símbolos para formar la sucesión de de Bruijn.

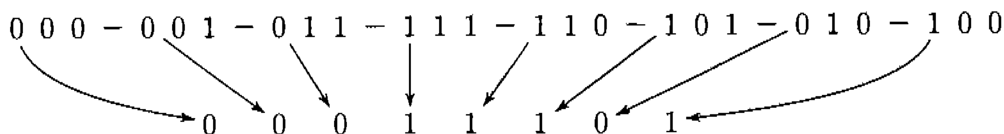
Ejemplos

a. Veamos la construcción de un circuito euleriano de $G_{2,3}$ en la Figura D.16, donde el orden en que tomamos las aristas se encuentra entre paréntesis etiquetando a cada arista.

Figura D.16: Circuito euleriano en $G_{2,3}$

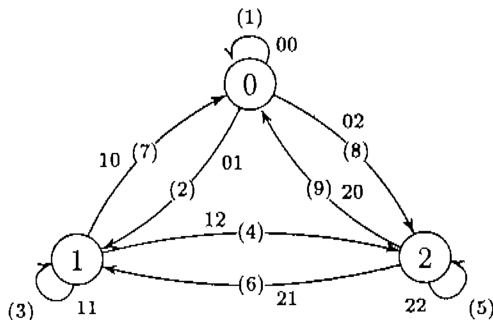


Sucesión de de Bruijn:

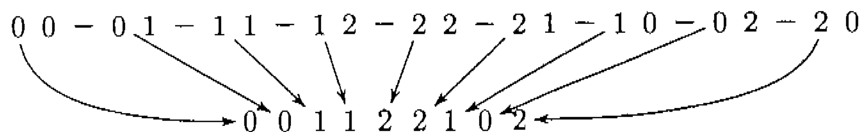


b. Ahora presentamos el circuito euleriano en $G_{3,2}$ en la Figura D.17.

Figura D.17: Circuito euleriano para $G_{3,2}$



Sucesión de de Bruijn



De los ejemplos anteriores debe quedar claro que *la existencia de sucesiones de de Bruijn es equivalente a la existencia de circuitos dirigidos eulerianos en las gráficas de de Bruijn correspondientes.* Esto es así porque para que la sucesión cumpla con ser de de Bruijn, cada arista debe aparecer en el camino exactamente una vez, y este camino existe cuando la gráfica tiene una trayectoria euleriana.

Teorema D.6 *Para toda pareja de enteros positivos σ y n, $G_{\sigma,n}$ tiene un circuito dirigido euleriano.*

Demostración:

Debemos demostrar que la gráfica $G_{\sigma,n}(V, E)$ cumple con las condiciones que se enunciaron para que una digráfica tenga un camino euleriano, y que son:

i. La digráfica debe ser fuertemente conexa.
ii. Para cada vértice de la digráfica $ingrado(v) = exgrado(v)$.

Pasamos a demostrar cada uno de estos incisos.

i. Por demostrar: $G_{\sigma,n}$ es fuertemente conexa.
La gráfica no dirigida subyacente es conexa, pues si ignoramos la dirección de los arcos, entre cada par de vértices siempre hay una arista. Probaremos ahora que es fuertemente conexa:
Sean $\mathsf{v} = b_1b_2\ldots b_{n-1}$ y $\mathsf{u} = c_1c_2\ldots c_{n-1}$ dos vértices cualesquiera en $G_{\sigma,n}$ con etiquetas distintas de $n-1$ símbolos. A continuación damos un camino compuesto de arcos, cada uno de ellos etiquetado con cadenas distintas de n símbolos, y que va del vértice etiquetado con v al vértice etiquetado con u. Como $G_{\sigma,n}$ contiene un arco por cada cadena sobre Σ de longitud n, la existencia de los arcos que mostramos está garantizada y la dirección es la correcta, por como se construye $G_{\sigma,n}$:
$$b_1b_2\ldots b_{n-1}c_1, \quad b_2\ldots b_{n-1}c_1c_2, \quad \ldots\ldots, \quad b_{n-2}b_{n-1}c_1\ldots c_{n-2}, \quad b_{n-1}c_1c_2\ldots c_{n-2}c_{n-1}$$

ii. Por demostrar: $ingrado(\mathsf{v}) = exgrado(\mathsf{v}) \quad \forall \mathsf{v} \in V$.
Sea $\mathsf{v} = b_1b_2\ldots b_{n-1}$ un vértice cualquiera de $G_{\sigma,n}$. Cada arista de la forma $\mathsf{x}b_1b_2\ldots b_{n-1}$ entra al vértice v. El número de estas aristas es precisamente σ, pues éste es el número de símbolos distintos que podemos colocar en la posición marcada con x, $\mathsf{x} \in \Sigma$.
De la misma manera, el número de arcos que salen de este vértice tienen la forma $b_1b_2\ldots b_{n-1}\mathsf{y}$, donde $\mathsf{y} \in \Sigma$ puede ser uno de los σ símbolos que tiene el alfabeto. Por lo que del vértice v salen exactamente σ arcos.

$\therefore\ ingrado(\mathsf{v}) = exgrado(\mathsf{v}) \quad \forall \mathsf{v} \in V$. ❑

Corolario D.7 *Para todo σ, $n > 0$ existe una sucesión de de Bruijn y se puede hallar en tiempo $\Theta(\sigma^n)$.*

Demostración:

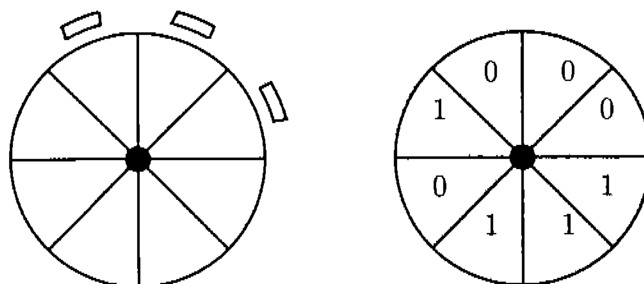
Aplicamos el algoritmo de Euler para encontrar una trayectoria euleriana en la gráfica $G_{\sigma,n}$. Dado que $G_{\sigma,n}$ tiene exactamente σ^n arcos, el tiempo que se va a tardar el algoritmo de Euler es $\Theta(|E|)$ donde $|E| = \sigma^n$. ❑

Veamos una aplicación de las sucesiones de de Bruijn, la que tiene que ver con posicionamiento de la cabeza lectora en tambores o discos magnéticos.

D.5.2. Tambores o discos magnéticos

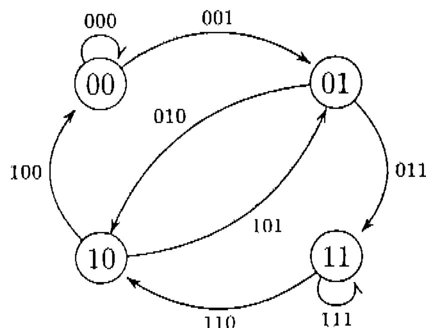
Supongamos que tenemos un disco magnético que está dividido en ocho sectores. Deseamos conocer el sector en el que se encuentra la cabeza lectora del mismo sin necesidad de verlo. Por supuesto que deseamos hacer esto con el menor costo posible. Una posibilidad es colocar material conductor en algunos sectores (1 binario) y material no conductor en otros (0 binario). Colocamos tres terminales en el disco, de tal manera que las tres terminales queden en sectores consecutivos, empezando con aquél en el que la cabeza lectora se encuentre.

Figura D.18: Un disco magnético con ocho sectores y tres terminales



El problema a resolver es el de asignar 0's y 1's a los distintos sectores de tal manera que cada cadena de tres dígitos binarios (que corresponden a la numeración de los ocho sectores) empiece en un único lugar. Como cada dígito del 0 al 7 empieza en una única posición, al detenerse la cabeza lectora en un sector, se transmitirán los tres dígitos binarios correspondientes al sector, conformado este número al concatenar el bit del sector en el que se encuentra con los dos bits consecutivos a éste. Para resolver este problema, como ya lo demostramos, debemos construir la digráfica de de Bruijn $G_{2,3}$ correspondiente y asignar los valores de 0 y 1 como lo indique dicha digráfica. La gráfica de de Bruijn correspondiente se encuentra en la Figura D.19.

Figura D.19: Gráfica de de Bruijn para el problema de los sectores de disco



Al construir el circuito euleriano en esta gráfica determinamos el orden en que deben ir

los ocho bits, uno por cada arista de la gráfica, eligiendo el primero (o último, pero siempre el mismo) de cada una de las aristas para determinar la sucesión de de Bruijn correspondiente.

En la Figura D.18 se encuentra la asignación correspondiente a cada uno de los sectores del disco, de acuerdo al circuito euleriano 000, 001, 011, 111, 110, 101, 010, 100, quedando la cadena circular de de Bruijn 00011101, que sirve para nuestro propósito.

D.5.3. Secuencias para ADN

Uno de los campos de las ciencias naturales en las que las ciencias de la computación han aportado de manera importante es al diseñar algoritmos para identificar la estructura de la célula a nivel molecular, y especialmente en lo que se refiere a las proteínas y el ADN.

El ADN se presenta como repeticiones de cadenas en el genoma humano. El genoma contiene los genes (que codifican las proteínas) que están "escritos" en el ADN. Adicionalmente, el ADN de organismos superiores (seres humanos, ratones, monos, etc.) contiene muchísimas estructuras que también se pueden abstraer como repeticiones de cadenas, las que juegan un papel importante en la explicación que podamos dar del funcionamiento de la herencia y duplicación de genes. Resulta además que en muchos casos este patrón o número de repeticiones del ADN es distinto de un ser humano a otro, lo que se puede utilizar para identificar individuos. Estas repeticiones se presentan en diversos patrones, como pueden ser repeticiones locales, intercaladas o de pequeña o gran escala. No para todos estos tipos de repeticiones se entiende el papel que juegan las repeticiones.

Como acabamos de mencionar, la identificación de patrones de repetición para distinguir entre distintas especies, o entre distintos individuos de la misma especie, constituye un problema cuya solución es de mucho interés. Éste ha sido uno de los campos de acción importante de la algorítmica, con un interés muy fuerte por encontrar métodos para determinar, identificar y almacenar estos patrones de manera eficiente. Esta identificación es importante para campos muy variados que van desde la criminología hasta la paleontología.

Otro problema interesante es el de lograr encontrar, dada una subcadena, la supercadena que la contiene. En este caso se trabaja a partir de una muestra del material genético y se busca reconocer a la estructura a la que representa esta muestra. Se dice que se tiene un problema de *secuenciación por hibridación* (SBH en inglés) si se busca determinar tanto como sea posible una cadena S de ADN, la cadena objetivo, a partir de la lista $\mathfrak{L}$ de todas las subcadenas de longitud k que aparecen en $\mathfrak{L}$. Esta cadena S deberá contener a todas las cadenas de $\mathfrak{L}$ como subcadenas, pero no podrá aparecer en S ninguna subcadena que no esté en $\mathfrak{L}$.

P. Pevzner, en [Pev89], redujo el problema SBH al de encontrar caminos eulerianos en una digráfica, de la siguiente manera:

Problema SBH

Trataremos de construir una gráfica dirigida similar a la de de Bruijn de la siguiente manera:

- Nuestro alfabeto corresponde a cada una de las cuatro proteínas que pueden aparecer en el ADN, $\sigma = \{A, T, C, G\}$. Por lo tanto, $\sigma = 4$.
- La longitud de las subcadenas que estamos considerando es k, de donde en la digráfica de de Bruijn $n = k$.

Dada una lista $\mathfrak{L}$ de todas las subcadenas de longitud k en la cadena objetivo S, construye una digráfica $G(\mathfrak{L}) \subseteq G_{4,k}$ como sigue:

i. La gráfica consiste de 4^{k-1} vértices, cada uno de ellos etiquetado con una cadena distinta de ADN de longitud $k-1$.

ii. Para cada cadena $\chi \in \mathfrak{L}$, existe un arco desde el vértice etiquetado con los $k-1$ primeros carácteres de χ al vértice etiquetado con los últimos $k-1$ carácteres de χ. El arco se etiqueta con χ, como lo hicimos en las digráficas de de Bruijn.

Nótese que algunos de los vértices en $G(\mathfrak{L})$ no van a tener arcos incidentes, por no existir en el ADN la secuencia que corresponde a una arista que termine o empiece en esos vértices. Estos vértices pueden eliminarse.

Una trayectoria euleriana en $G(\mathfrak{L})$ *especifica* a la cadena S de la siguiente manera: la cadena S empieza con la etiqueta del primer vértice en la trayectoria y prosigue a partir de ese momento, con la concatenación, en orden, del último símbolo de las etiquetas sobre los arcos que va incluyendo en el camino. Veamos un ejemplo.

Supongamos que tenemos subcadenas de longitud 3 formadas de proteínas:

$$\begin{aligned}\mathfrak{L} = \{ \; & AAA, AAC, ACA, CAC, CAA, ACG, CGC, \\ & GCA, ACT, CTT, TTA, TAA\}\end{aligned}$$

a la que corresponde la digráfica de la Figura D.20. Una posible cadena S para esta digráfica es $S = ACAGCGCAACTTAAA$, que corresponde a la trayectoria euleriana que se construye si se visitan los vértices en el siguiente orden:

$$AC, \quad CA, \quad AC, \quad CG, \quad GC, \quad CA, \quad AA, \quad AC, \quad CT, \quad TT, \quad TA, \quad AA, \quad AA$$

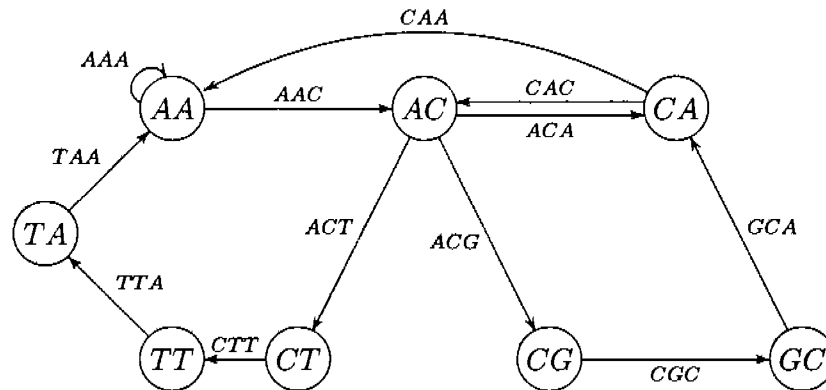
utilizando los arcos correspondientes. En general, puede haber más de una trayectoria euleriana en una misma gráfica. Nos interesan, sin embargo, aquellas que corresponden *exactamente* a S, lo que formalizamos en la Definición D.7.

Definición D.7 Una cadena S es *compatible* con $\mathfrak{L} \iff S$ contiene a cada cadena en $\mathfrak{L}$ y, suponiendo que $\mathfrak{L}$ contiene subcadenas de longitud k, S no contiene a ninguna otra subcadena de longitud k que no esté en $\mathfrak{L}$.

Debemos notar que este resultado es ligeramente distinto al que se obtuvo, por ejemplo, para las posiciones en el disco, ya que una cadena de $\mathfrak{L}$ puede aparecer en más de una posición en la cadena resultado. La restricción fuerte en este caso es que no aparezca como subcadena de S ninguna cadena de longitud k que no estuviere en $\mathfrak{L}$.

Con esta definición podemos enunciar el Teorema D.8.

Figura D.20: Gráfica $G(\mathfrak{L})$ correspondiente a $\mathfrak{L}$



$$\mathfrak{L} = \{ AAA,\ AAC,\ ACA,\ CAC,\ CAA,\ ACG,\ CGC,\ GCA,\ ACT,\ CTT,\ TTA,\ TAA \}$$

Teorema D.8 *Una cadena S es compatible con $\mathfrak{L} \iff S$ puede ser especificada por una trayectoria euleriana en $G(\mathfrak{L})$.*

Demostración:

Este teorema es fácil de demostrar a partir de las propiedades de una trayectoria euleriana y de la Definición D.7. ❏

Corolario D.9 *Supongamos que cada subcadena de $\mathfrak{L}$ aparece exactamente una vez en la cadena de ADN objetivo. Entonces, el conjunto de cadenas $\mathfrak{L}$ determina unívocamente la cadena de ADN objetivo $\iff G(\mathfrak{L})$ tiene una única trayectoria euleriana.*

Corolario D.10 *Existe una correspondencia uno-a-uno entre caminos eulerianos en $G(\mathfrak{L})$ y cadenas que son compatibles con $\mathfrak{L}$.*

Si los datos fueron recolectados de manera correcta y, en efecto, ninguna cadena de tamaño k aparece más de una vez en el objetivo, entonces $G(\mathfrak{L})$ debe tener al menos una trayectoria euleriana que especifique a la cadena de ADN. Aun cuando haya más de un camino euleriano en la digráfica, puede ser posible extraer suficiente información del conjunto de todos los posibles caminos eulerianos. De esto, el problema de encontrar la cadena objetivo de ADN se puede reducir a la de encontrar una trayectoria euleriana.

Desafortunadamente, no es frecuente que las condiciones que se suponen para esta solución se presenten. Muchas veces se cometen errores al recolectar los datos, o el suponer que no hay repeticiones en el ADN tampoco es una suposición fácil de hacer. Pero, finalmente, todas estas observaciones llevan, en el mediano y largo plazo, a una mejor comprensión de los organismos vivos y lo que determina su estructura.

Con esto terminamos este apéndice, donde vimos algunos aspectos de las distintas exploraciones que no estaban directamente relacionadas con la manera como hicimos las especializaciones. Incluimos acá aplicaciones de las distintas especializaciones, que si bien se pueden realizar a partir de nuestro planteamiento, no son intrínsecas a él.

Apéndice E

Árboles

En este apéndice revisaremos con más detenimiento a los árboles, gráficas con características específicas que aparecen constantemente y son muy útiles en las ciencias de la computación. El primer registro del uso de este concepto es de 1847, cuando Gustav Kirchhoff los utilizo para sus trabajos en circuitos eléctricos. Posteriormente fueron utilizados por Arthur Cayley en el estudio de la química.

Las propiedades de este tipo de gráficas favorecen la elaboración de algoritmos eficientes que resuelven un gran conjunto de problemas, no sólo de ciencias de la computación sino de las más variadas disciplinas. Proveen maneras eficientes para organizar información que favorecen su acceso y manipulación.

E.1. Caracterización de árboles

Empezamos esta sección definiendo de manera precisa qué es un árbol:

Definición E.1 Sea $G = (V, E)$ una gráfica. G es un *árbol* si es conexa y no tiene ciclos.

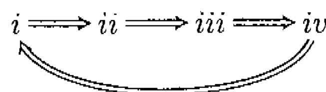
Por el momento trabajaremos con gráficas no dirigidas, para posteriormente extendernos a digráficas. Es conveniente dar caracterizaciones alternas para los árboles, de tal manera de usar éstas para demostrar de mejor manera las propiedades de algunos algoritmos. Hacemos esto en forma de teoremas:

Teorema E.1 *Sea $G = (V, E)$ una gráfica. Entonces, las siguientes cuatro propiedades son equivalentes:*

i. *G es un árbol: es conexa y acíclica*
ii. *G es maximal acíclica: no tiene ciclos, pero al agregarle una arista a G se forma un ciclo*
iii. *G no contiene lazos y para cualesquiera dos vértices existe un único camino simple que los conecta*
iv. *G es minimal conexa: es conexa, pero al quitarle cualquier arista deja de ser conexa*

Demostración:

Realizaremos la siguiente demostración:

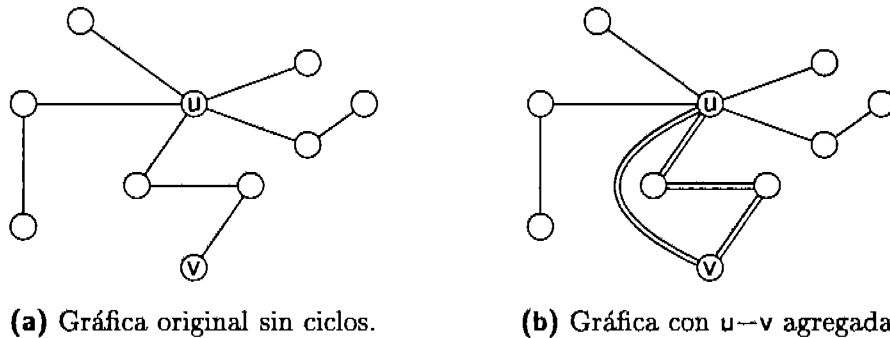


Supondremos que la gráfica tiene al menos 2 vértices.

$\boldsymbol{i \Rightarrow ii}$. Suponemos que $G = (V, E)$ es un árbol, esto es, por la Definición E.1 que es conexa y no tiene ciclos, y debemos demostrar que al agregarle cualquier arista se forma un ciclo.

Sean u y v dos vértices cualesquiera en V tales que u—v $\notin E$. Como G es conexa, existe un camino entre u y v, digamos $P = \mathsf{u} \overset{P}{\rightsquigarrow} \mathsf{v}$. Agreguemos la arista u—v. Entonces, el camino $P' = \mathsf{v}-\mathsf{u} \overset{P}{\rightsquigarrow} \mathsf{v}$ forma un ciclo que empieza y termina en v, como se puede observar en la Figura E.1:

Figura E.1: Al agregar una arista a un árbol



(a) Gráfica original sin ciclos. **(b)** Gráfica con u—v agregada.

***ii* ⇒ *iii*.** Suponemos ahora que la gráfica es conexa y no tiene ciclos, y que al agregarle cualquier arista se forma un ciclo; queremos demostrar que si la gráfica no tiene lazos, para cualesquiera dos vértices existe un único camino simple que los conecta.

Sean u y v dos vértices arbitrarios en G. Si no existe un camino entre u y v, entonces al agregar la arista u—v **no** se forma un ciclo, contradiciendo la hipótesis; por lo que G debe ser conexa. Como G es conexa, existe un camino, digamos $P = \mathsf{u} \overset{P}{\rightsquigarrow} \mathsf{v}$, entre u y v.

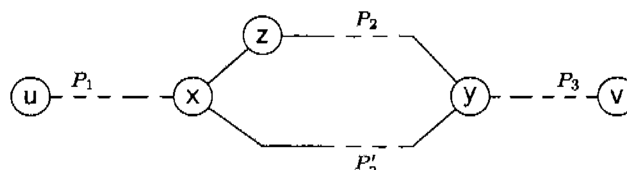
Por contrapositivo, supongamos que existen dos caminos simples P y P', entre u y v y veamos como esto nos lleva a una contradicción. Como P y P' son caminos simples, uno no puede contener al otro, aunque sí tienen vértices en común (al menos a u y v). Sea $P_1 = \mathsf{u} \overset{P_1}{\rightsquigarrow} \mathsf{x}$ la porción inicial del camino que comparten P y P' inmediatamente al salir de u, con x el último vértice que comparten. Sea $P_3 = \mathsf{y} \overset{P_3}{\rightsquigarrow} \mathsf{v}$ la porción final del camino que comparten P y P' para llegar a v, con y el vértice en el que los dos caminos se reencuentran. De esto,

$$P = \mathsf{u} \overset{P_1}{\rightsquigarrow} \mathsf{x} \overset{P_2}{\rightsquigarrow} \mathsf{y} \overset{P_3}{\rightsquigarrow} \mathsf{v}$$

$$P' = \mathsf{u} \overset{P_1}{\rightsquigarrow} \mathsf{x} \overset{P_2'}{\rightsquigarrow} \mathsf{y} \overset{P_3}{\rightsquigarrow} \mathsf{v}$$

como se muestra en la Figura E.2.

Figura E.2: Dos caminos simples en una gráfica conexa



Si $|P_1| = 0$ quiere decir que P y P' se separan inmediatamente al salir de u, y si $|P_3| = 0$ quiere decir que P y P' se reencuentran en v.

Como $P \neq P'$, $P \not\subset P'$ y $P' \not\subset P$, existe al menos un vértice, digamos z, que está en uno de los caminos y no en el otro. Supongamos sin pérdida de generalidad que está en P. Tenemos la situación que se muestra en la Figura E.2. Pero si éste es el caso, tenemos un ciclo $\mathsf{x}{-}\mathsf{z} \overset{P_2}{\rightsquigarrow} \mathsf{y} \overset{P_2'}{\rightsquigarrow} \mathsf{x}$, lo que contradice la hipótesis de que G no tiene ciclos.

***iii* ⇒ *iv*.** Suponemos ahora que G no tiene lazos y que para cualesquiera dos vértices en G existe un único camino simple que los conecta. Queremos demostrar que G es conexa pero que si quitamos cualquier arista de G, ésta se desconecta.

Por contrapositivo, elegimos arbitrariamente una arista $\mathsf{e} \in E$ y la quitamos. Como G no tiene lazos, e no puede ser uno. De esto, $\mathsf{e} = \mathsf{u}{-}\mathsf{v}$ con $\mathsf{u} \neq \mathsf{v}$. Si la gráfica no se desconecta, quiere decir que sigue habiendo un camino simple de u a v, que no utilizaba a e. Pero entonces había en G dos caminos simples de u a v, el que consistía únicamente de e y el que quedó al quitar a e, contradiciendo la hipótesis de que hay un único camino simple entre cualesquiera dos vértices.

***iv* ⇒ *i*.** Suponemos ahora que G es conexa pero al quitarle cualquier arista se desconecta y tenemos que demostrar que G es conexa y acíclica.

La demostración de que G es conexa está dada por las hipótesis. Por contrapositivo, suponemos que G tiene algún ciclo. Pero entonces cualquier arista en este ciclo puede quitarse sin desconectar a la gráfica.

$\therefore$ G es acíclica. ❏

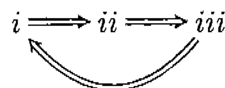
Nos interesa caracterizar a los árboles también en términos de la relación que existe entre el número de vértices y el número de aristas. El Teorema E.2 establece dos relaciones.

Teorema E.2 *Sea $G = (V, E)$ una gráfica finita con $|V| = n$. Las tres propiedades que siguen son equivalentes:*

i. G es un árbol.
ii. G es acíclica y tiene $|V| - 1$ aristas.
iii. G es conexa y tiene $|V| - 1$ aristas.

Demostración:

Nuevamente pretendemos demostrar que



Para el caso en que la gráfica sólo contenga un vértice, estas implicaciones se cumplen trivialmente. Por lo que nos fijaremos únicamente en gráficas con $|V| \geq 2$.

***i* ⇒ *ii*:** Demostraremos por inducción sobre $n = |V|$, el número de vértices de la gráfica, que si G es un árbol, entonces G es acíclica y tiene $n - 1$ aristas. Que G es acíclica se sigue de la definición de árbol, por lo que únicamente tenemos que demostrar que tiene $n - 1$ aristas.

<u>Base:</u> **($n = 1$).** Si la gráfica tiene únicamente un vértice, el único tipo de aristas que podría tener son lazos. Pero como es acíclica, no puede tener lazos, por lo que el número de aristas es $n - 1 = 1 - 1 = 0$.

Veamos también el caso para $n = 2$, que no es el caso trivial. Como G no tiene lazos y es acíclica no puede tener aristas múltiples;

$\therefore$ el número de aristas entre estos dos vértices es $n - 1 = 2 - 1 = 1$.

<u>Inducción:</u> **($n = k < m$).** Supongamos que $i \Rightarrow ii$ se cumple para todas las gráficas con $n = k < m$ vértices; sea G un árbol con $n = m$ vértices y quitemos una arista arbitraria e de G. Por la propiedad iv del Teorema E.1, G se desconecta, y presenta ahora dos componentes conexas $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ con $|V_1| \leq m$ y $|V_2| \leq m$ y tal que $|V_1|+|V_2| = m$. Por la hipótesis de inducción, cada una de estas componentes cumple que el número de aristas es, respectivamente, $|V_1| - 1$ y $|V_2| - 1$. De esto, la gráfica original tenía $(k_1 - 1) + (k_2 - 1) + 1$ aristas, y haciendo un poco de aritmética tenemos

$$\begin{aligned}(k_1 - 1) + (k_2 - 1) + 1 &= k_1 + k_2 - 1 - 1 + 1\\ &= k_1 + k_2 - 1\\ &= m - 1\end{aligned}$$

***ii* $\Rightarrow$ *iii*:** Suponemos que G es acíclica y tiene $|V| - 1$ aristas y debemos demostrar que G es conexa con $|V| - 1$ aristas; lo haremos por inducción en el número de vértices.

Nuevamente, para $n = 1$ se cumple trivialmente.

Supongamos $n \geq 2$. Demostraremos primero que G tiene al menos dos vértices con grado 1. Elijamos arbitrariamente una arista e (debe existir al menos una, ya que $n \geq 2$; por hipótesis tiene entonces al menos una arista). Extendemos esa arista a un camino simple, agregando en sus extremos nuevas aristas, en la medida en que sea posible. Cada vez que agregamos una arista en el extremo del camino simple, se agrega un vértice al camino o se cierra un ciclo. Pero como estamos suponiendo que G es acíclica, esto último no puede suceder. Así que nuestro camino continúa siendo un camino simple. Como la gráfica es finita, llegará el momento en que ya no podamos agregar aristas al camino. Entonces, los dos vértices en el extremo tienen grado 1.

La demostración de que G es conexa es por inducción en el número de vértices $|V|$.

<u>Base:</u> **$|V| = 2$.** Como la gráfica no tiene ciclos y tiene una arista, ésta tiene que conectar a los dos vértices, de donde G es conexa.

<u>Inducción:</u> $|V| = k - 1$. Supongamos cierto para gráficas con $m < k$ vértices y $m-1$ aristas. Acabamos de demostrar que entonces G tiene al menos dos vértices de grado 1. Eliminamos de G a uno de estos vértices, y a la arista incidente en él. Nos quedamos entonces con una gráfica acíclica que tiene $m - 1$ vértices y $m - 2$ aristas, que por la hipótesis de inducción es conexa. De donde G es también conexa.

***iii* $\Rightarrow$ *i*:** Suponemos que G es conexa y tiene $|V| - 1$ aristas y debemos demostrar que entonces G es un árbol. Por contradicción, si G contiene algún ciclo, podemos eliminar aristas (sin quitar vértices) de tal manera que G siga siendo conexa. Cuando este proceso termina, la gráfica resultante es un árbol (conexa y acíclica) y por lo tanto tiene $|V| - 1$ aristas (i$\Rightarrow$ii). ¡Pero empezamos con $|V|-1$ aristas, por lo que no pudimos eliminar ninguna arista! Entonces, la gráfica original no tiene ciclos. ❏

Definición E.2 (Hoja) Una *hoja* es aquel vértice en un árbol cuyo grado es 1.

De esta definición y la demostración del Teorema E.2 tenemos el siguiente corolario:

Corolario E.3 *Todo árbol finito con más de un vértice tiene al menos 2 hojas.*

Una propiedad que surge en el contexto de exploración de DFS es la de diámetro de un árbol:

Definición E.3 (Diámetro) El diámetro ∂ de un árbol $G = (V, E)$, $\partial(G)$, está definido como la mayor distancia entre cualesquiera dos vértices de G.

$$\partial(G) = \text{máx}\{\delta(\mathsf{u}, \mathsf{v}) \mid \mathsf{u}, \mathsf{v} \in V\}$$

En este mismo contexto es importante precisar la relación entre los vértices de un árbol.

Definición E.4 (Descendiente) Un vértice v es *descendiente* de un vértice u en el árbol G, denotado por $\mathsf{u} \prec_G \mathsf{v} \iff \exists\, \mathsf{s} \rightsquigarrow \mathsf{u} \rightsquigarrow \mathsf{v} \in G$ (i.e., si u precede en el árbol a v, desde la raíz). v es *descendiente propio* de u si $\mathsf{u} \neq \mathsf{v}$.

E.2. Árboles generadores

De los teoremas de la sección anterior, sobre todo los que describen la relación entre el número de aristas y el número de vértices de un árbol, podemos ver que los árboles son "económicos" en términos del número de aristas que se requieren para tener gráficas conexas. Es conveniente, en muchas ocasiones, encontrar un árbol que sea subgráfica de una gráfica y que contenga a todos los vértices de la gráfica original, ya que hemos demostrado que este tipo de gráficas son óptimas en términos del número de aristas. Ésta es una idea que aparece en muchas aplicaciones, como lo son líneas de distribución de energía, redes de distribución de agua, circuitos eléctricos y, en general, recorrido de los vértices de una gráfica. Pasamos entonces a revisar el tema de árboles que son subgráficas de otras gráficas. Recordemos la definición de una subgráfica:

Definición E.5 (Subgráfica) Sea $G = (V, E)$ una gráfica. Decimos que $G' = (V', E')$ es una subgráfica de G si $E' \subseteq E$ y $V' \subseteq V$.

Definición E.6 (Árbol Generador) Un *árbol generador* de una gráfica $G = (V, E)$ es una subgráfica de G que es un árbol y que contiene a todos los vértices de V.

Si la gráfica original es un árbol, entonces su único árbol generador es ella misma.

Existen varias maneras de encontrar el árbol generador de una gráfica. Una manera simple de hacerlo consiste en localizar en la gráfica los ciclos, e ir quitando aristas para que estos ciclos desaparezcan. Cuando la gráfica no tiene más ciclos, se ha llegado a un árbol generador.

Es importante mencionar que para una misma gráfica pueden existir varios árboles generadores.

E.3. Árboles dirigidos

Nos fijamos ahora en gráficas dirigidas; tenemos la siguientes definiciones:

Definición E.7 (Raíz) Una digráfica $G = (V, E)$ tiene una *raíz* si existe un vértice distinguido, $\mathsf{r} \in V$, desde el cual se puede llegar a todos los demás vértices, esto es, si existe un camino dirigido desde r a cualquier vértice de G.

Definición E.8 (Árbol dirigido) Un *árbol dirigido* es una digráfica no forzosamente finita cuya gráfica subyacente no dirigida es un árbol, que además tiene un vértice raíz.

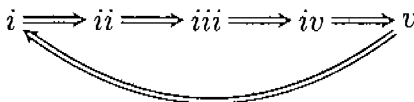
De la misma manera en que lo hicimos con árboles (no dirigidos) pasamos a dar las propiedades que caracterizan a los árboles dirigidos en el siguiente teorema.

Teorema E.4 *Sea $G = (V, E)$ una gráfica dirigida. Las siguientes cinco caracterizaciones son equivalentes:*

i. *G es un árbol dirigido.*
ii. *G tiene una raíz $\mathsf{r} \in V$ desde la cual se puede llegar a todos los demás vértices, por medio de un único camino dirigido.*
iii. *G tiene una raíz $\mathsf{r} \in V$ tal que $ingrado(\mathsf{r}) = 0$ y para cualquier otro vértice $\mathsf{v} \neq \mathsf{r}$, $ingrado(\mathsf{v}) = 1$.*
iv. *G tiene una raíz r, y al eliminar cualquier arista, G deja de tener raíz.*
v. *La gráfica no dirigida de G es conexa y G tiene un vértice r para el cual $ingrado(\mathsf{r}) = 0$, mientras que para cualquier otro vértice $\mathsf{v} \neq \mathsf{r}$, $ingrado(\mathsf{v}) = 1$.*

Demostración:

La demostración se hará de manera idéntica a como se hizo para árboles no dirigidos. Demostraremos

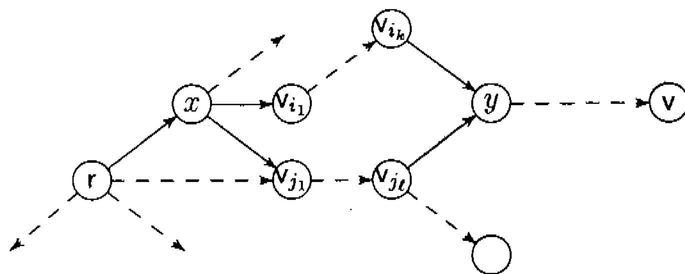


Sea G' la gráfica subyacente no dirigida de G.

i* $\Rightarrow$ *ii: De la Definición E.8, G' es un árbol y G tiene un vértice raíz. Lo único que nos queda por demostrar es que el camino dirigido entre r y v, $\forall \mathsf{v} \in V$, es único.

Por contrapositivo, supongamos que existe un vértice v para el cual existen dos caminos simples dirigidos de r a v, como se puede observar en la Figura E.3.

Figura E.3: Dos caminos dirigidos de r a v



Pero entonces, G' tiene un ciclo ($x - v_{i_1} \rightsquigarrow v_{i_k} - y - v_{j_\ell} \rightsquigarrow v_{j_1} - x$), negando con esto que G' sea un árbol. De esto, no puede existir más que un único camino de r a v.

ii* ⇒ *iii: Suponemos ahora que G tiene una raíz desde la cual se puede llegar a cualquier otro vértice de la gráfica por un único camino, y demostraremos que G tiene una raíz r tal que *ingrado*(r) $= 0$; para cualquier otro vértice $v \in V$, $v \neq r$, *ingrado*(v) $= 1$.

Haremos la demostración por contradicción. Supongamos que *ingrado*(r) $\neq 0$, esto es, existe un arco incidente en r ($v \rightarrow r$). $v \neq r$, pues $v = r$ implica que en G' hay un lazo, y no sería árbol (inciso iii del Teorema E.1). Pero entonces existen dos caminos de r a r, uno de longitud 1, y el otro $r \rightsquigarrow v \rightarrow r$, contradiciendo el que hay un único camino entre cualesquiera dos vértices.

Probaremos ahora que *ingrado*(v) $= 1$ $\forall v \in V$, $v \neq r$. Claramente *ingrado*(v) > 0, porque si no fuera así, ningún camino llegaría a v y en particular, ningún camino que saliera de r, contradiciendo la hipótesis. Si *ingrado*(v) > 1, existen al menos dos arcos, $e_1 = u_1 \rightarrow v$ y $e_2 = u_2 \rightarrow v$. Como existen caminos de r a u_1 y de r a u_2, los caminos $P = r \rightsquigarrow u_1 \rightarrow v$ y $P' = r \rightsquigarrow u_2 \rightarrow v$ son dos caminos distintos de r a v, contradiciendo la hipótesis de que existe un único camino de r a v(¡!).

iii* ⇒ *iv: Suponemos ahora que G tiene una raíz tal que *ingrado*(r) $= 0$;y $\forall v \in V$, $v \neq r$, *ingrado*(v) $= 1$; queremos demostrar que al quitar cualquier arco de G, ésta deja de tener raíz.

Como *ingrado*(r) $= 0$ ningún arco que quitemos es incidente en r. Elijamos un arco arbitrario $e = u \rightarrow v$. $v \neq r$, ya que *ingrado*(v) $= 1$. Al quitar e, *ingrado*(v) $= 0$, lo que dice que ningún camino llega a v, en particular, ninguno que salga de r. Por lo que r ya no es raíz, ya que no hay camino de r a v.

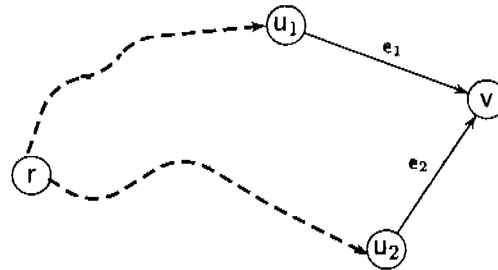
iv* ⇒ *v: Suponemos ahora que G tiene una raíz y al quitar cualquier arco, G deja de tener raíz; queremos demostrar que G' es conexa y tiene un vértice r para el cual *ingrado*(r) $= 0$, mientras que $\forall v \in V$, $v \neq r$, *ingrado*(v) $= 1$.

Es claro que *ingrado*(r) $= 0$, pues si no fuera así, podríamos borrar arcos que entraran a r sin que r dejara de ser raíz (quedarían intactos los caminos que *salen* de r).

Sea $v \in V$, $v \neq r$ un vértice arbitrario en G. Claramente, *ingrado*(v) > 0, porque si no fuera así no se podría llegar a v desde r, y eso contradice el que r sea raíz. Supongamos entonces que *ingrado*(v) > 1, digamos 2. En este caso tenemos dos arcos, $e_1 = u_1 \rightarrow v$ y $e_2 = u_2 \rightarrow v$, tales que ambos inciden en v, como se observa en la Figura E.4. Pero como r es raíz, existe un camino de r a u_1 y un camino de r a u_2. Y podemos quitar cualquiera de

e_1 o e_2, sin que deje de haber un camino dirigido de r a v, y por lo tanto sin que r deje de ser raíz, lo que contradice la hipótesis. De esto, $ingrado(v) = 1$.

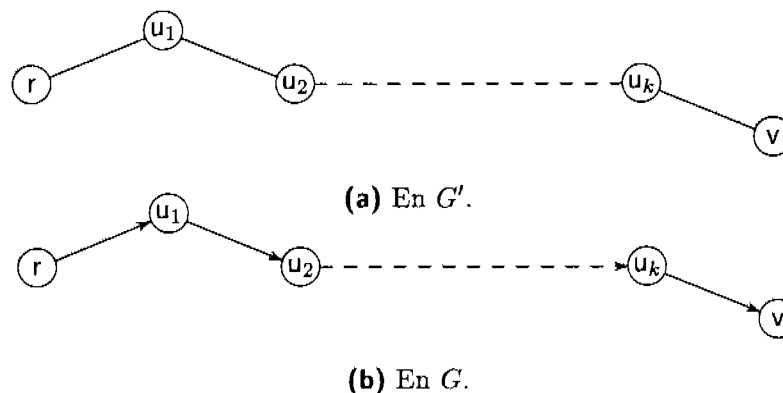
Figura E.4: Digráfica con $ingrado(v) = 2$



v ⇒ i: Nuestra hipótesis es que G' es conexa, G tiene un vértice r tal que $ingrado(r) = 0$ y $\forall v \in V$, $v \neq r$, $ingrado(v) = 1$; queremos demostrar que G tiene una raíz y que G' es un árbol.

Demostraremos primero que G tiene una raíz. Como G' es conexa, entre cualesquiera dos vértices en G' existe un camino simple que los conecta. Sea $P = r \rightsquigarrow v$ un camino simple que conecta a r con v en G'. Sean $u_1, u_2, \ldots, u_k$ vértices distintos en ese camino simple. Como $ingrado(r) = 0$, el arco que corresponde a la arista $r - u_1 \in G'$ debe estar orientada $r \rightarrow u_1$, pues ningún arco llega a r. Tomemos después el vértice u_2 y la arista $u_1 - u_2 \in G'$. Como $ingrado(u_1) = 1$, el arco $r \rightarrow u_1$ es el único que llega a u_1, por lo que la arista $u_1 - u_2$ está orientada $u_1 \rightarrow u_2$. Seguimos con este razonamiento hasta llegar a la arista $u_k - v \in G'$ que corresponde al arco $u_k \rightarrow v \in G$. De lo anterior, $\exists$ un camino dirigido $r \rightsquigarrow v$ de r a cualquier vértice de G, por lo que r es raíz.

Figura E.5: Construcción de camino dirigido de r a v



(a) En G'.

(b) En G.

Demostramos ahora que G' es un árbol. G' es acíclica, pues si hubiera un ciclo en G', éste correspondería a un ciclo dirigido en G (construido de la misma manera que en el párrafo anterior usando la hipótesis de que $ingrado(r) = 0$ e $ingrado(v) = 1, v \neq r$) y eso haría que

algún vértice de G tuviera *ingrado* > 1 (o bien que *ingrado*(r) > 0), lo que contradice la hipótesis. ❑

Las propiedades que hemos presentado se aplican tanto a digráficas finitas como infinitas. A continuación mostramos algunas propiedades que presentan únicamente las digráficas finitas.

Teorema E.5 *Una digráfica finita G es un árbol dirigido si y sólo si:*

i. Su gráfica subyacente no dirigida G' no tiene ciclos.
ii. Uno de los vértices de G, r, tiene ingrado(r) $= 0$, *mientras que* $\forall$v $\in V$, v $\neq$ r, *ingrado*(v) $= 1$.

Demostración:

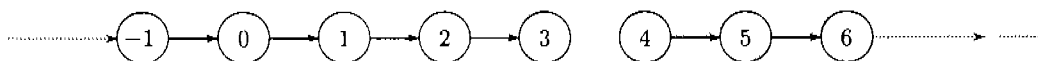
$\Longrightarrow$ Por el Teorema E.4 inciso *iii*.

$\Longleftarrow$ Como *ingrado*(v) $= 1$ $\forall$v $\in V$, v $\neq$ r e *ingrado*(r) $= 0$, el número de aristas en G' debe ser $|V| - 1$ (a cada vértice llega exactamente una arista, menos a r al que no llega ninguna). Entonces, G' no tiene ciclos (por hipótesis) y tiene $|V| - 1$ aristas, de donde por el Teorema E.2 ($ii \Rightarrow iii$) G' es conexa. Y por el Teorema E.4 ($v \Rightarrow i$) G es un árbol dirigido. ❑

Es posible que no se vea claramente el porqué este resultado se aplica únicamente a digráficas finitas. Para convencer veamos el siguiente contraejemplo.

Contraejemplo: Supongamos que tenemos una digráfica $G = (\mathbb{N}, E)$ donde $E = \{\mathsf{e} = i \rightarrow j \mid i = j - 1\} - 3 \rightarrow 4$, como se puede ver en la Figura E.6.

Figura E.6: Digráfica infinita que cumple con (v) sin ser árbol

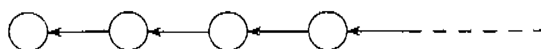


Esta gráfica cumple con las condiciones de no tener ciclos y las relativas a los ingrados de los vértices, pero no es un árbol porque no es conexa (no hay arista entre el nodo 3 y el 4).

Definición E.9 (Digráfica arbitrada) Una digráfica es *arbitrada* si para cualesquiera dos vértices v_1 y v_2, $\exists$ un vértice v, llamado *árbitro de* v_1 y v_2, tal que $\exists$n caminos dirigidos $\mathsf{v} \rightsquigarrow \mathsf{v}_1$ y $\mathsf{v} \rightsquigarrow \mathsf{v}_2$.

Existen gráficas infinitas arbitradas sin raíz, como se puede ver en la Figura E.7.

Figura E.7: Digráfica infinita arbitrada, sin raíz



En la Figura E.7, como la digráfica es infinita, dados cualesquiera dos vértices v_1 y v_2, con v_2 a la derecha de v_1, siempre podemos tomar el vértice a la derecha de v_2 (siempre existe, ya que la digráfica es infinita) y hay un camino desde él tanto a v_1 como a v_2.

Sin embargo, en el caso de digráficas finitas, el hecho de que la digráfica sea arbitrada implica un resultado más fuerte:

Teorema E.6 *Si una digráfica finita es arbitrada, entonces tiene una raíz.*

Demostración:

Sea $G = (V, E)$ una digráfica finita arbitrada, con $V = \{v_1, v_2, \ldots, v_n\}$. Demostraremos por inducción que todo subconjunto de vértices $V' = \{v_{i_1}, v_{i_2}, \ldots, v_{i_m}\}$, con $m \leq n$, tiene un árbitro, esto es, que para todo vértice v_{i_j}, con $1 \leq i \leq m$, $\exists\ v_{i_m} \rightsquigarrow v_{i_j}$.

Base: Por definición y como G es arbitrada, dado v_{i_1} existe v_{i_2} tal que $v_{i_2} \rightsquigarrow v_{i_1}$.

Inducción: Supongamos que $v_{i_{m-1}}$ es un árbitro para $\{v_{i_1}, v_{i_2}, \ldots, v_{i_{m-2}}\}$. Sea v_{i_m} el árbitro de $v_{i_{m-1}}$ y v_{i_m}. Como $v_{i_{m-1}}$ es alcanzable desde v_{i_m}; y v_{i_j}, con $1 \leq j \leq m-1$, es alcanzable desde $v_{i_{m-1}}$, se sigue que v_{i_j}, con $1 \leq j \leq m-1$, es alcanzable desde v_{i_m}. De esto, v_{i_n} es la raíz de G. ❏

Dado que para digráficas finitas es equivalente decir que tengan raíz a que sean arbitradas, en el Teorema E.4 incisos *i* a *iv* la condición de que tengan raíz puede ser sustituida por la condición de que sean arbitradas.

E.4. Digráficas infinitas

Supongamos que tenemos una digráfica infinita, lo que se puede deber a que sea infinita en el número de vértices adyacentes a un vértice dado, o bien a que siempre haya al menos un vértice desde el cual se pueda seguir extendiendo un camino simple, de manera infinita. En la gráfica de la subfigura E.8(a) el rango de uno o más vértices es infinito, mientras que en la gráfica de la subfigura E.8(b), existe un camino simple que se puede extender de manera infinita. Por supuesto que podemos tener una gráfica infinita con ambas características.

Si bien es difícil en muchas ocasiones determinar propiedades de gráficas infinitas, König en 1936 publicó lo que se conoce como el Lema del Infinito de König, que se enuncia y demuestra a continuación.

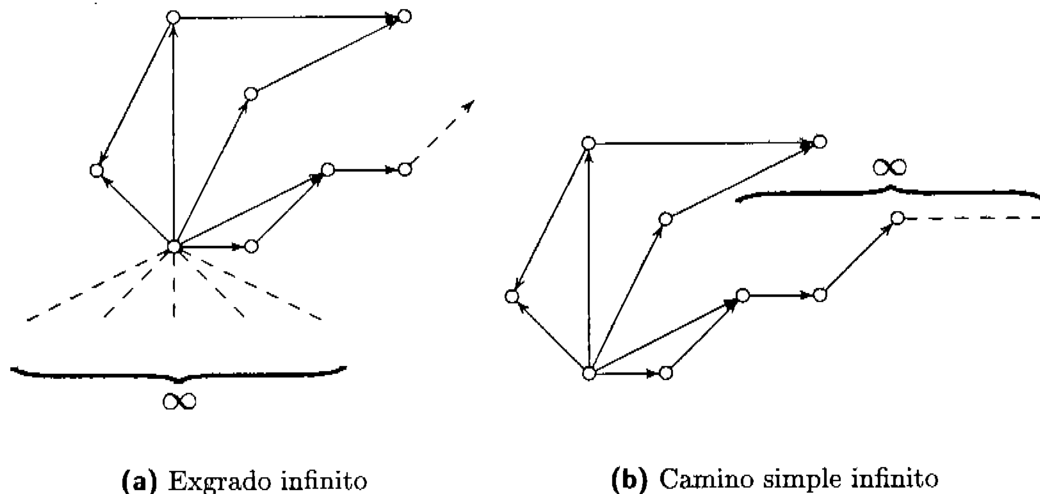
Lema E.7 (Lema del infinito de König) *Si G es una digráfica infinita, con raíz en r y ex-grado finito para todos sus vértices, entonces G tiene un camino dirigido simple infinito que empieza en r.*

Demostración:

Primero, demostraremos que existe $T \subseteq G$, tal que T es un árbol dirigido con raíz en r. T consiste de lo siguiente:

i. Seleccionamos a r como raíz de T.

Figura E.8: Tipos de digráficas infinitas



(a) Exgrado infinito **(b)** Camino simple infinito

ii. Seleccionamos a todos los vértices que están a distancia 1 de r en G. Obviamente, estos vértices estarán a distancia 1 de r en T.
iii. Seleccionamos a los vértices a distancia ℓ de r en G. Obviamente estos vértices estarán a distancia ℓ de r en T.
iv. En general, si un vértice v está a distancia ℓ de r en G, también está a distancia ℓ de r en T. Quitamos ahora todos los arcos que entran a v, excepto uno que conecte a v con algún vértice a distancia $\ell - 1$.

T cumple con las propiedades de ser un árbol dirigido, ya que por construcción $ingrado(r) = 0$ en T, mientras que $ingrado(\mathsf{v}) = 1$ en T. Es suficiente mostrar que en T existe un camino dirigido de longitud infinita, ya que éste estará contenido en G.

Claramente, como T es subgráfica de G, todos los vértices de T son de exgrado finito.

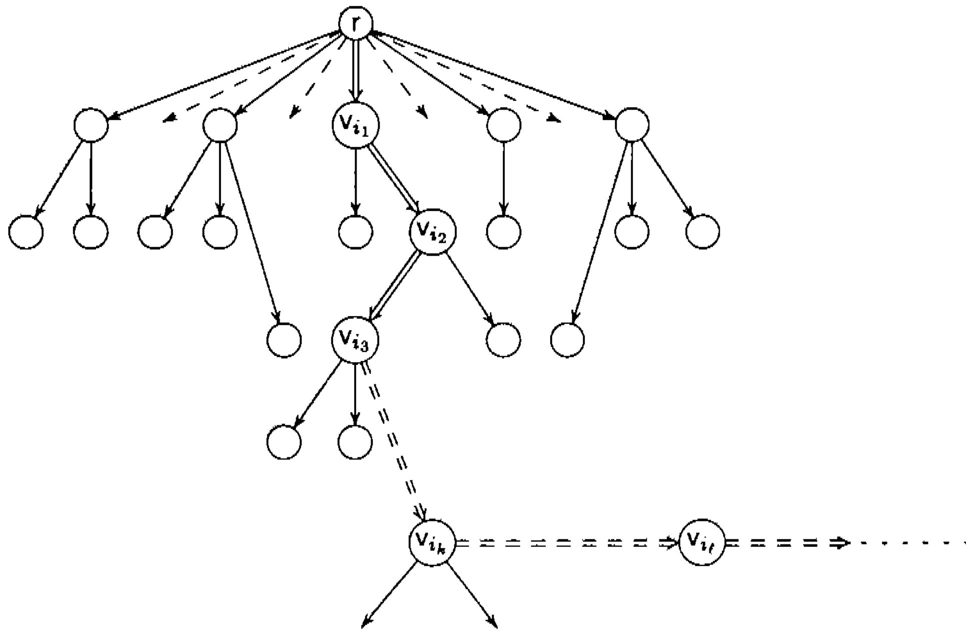
Construimos el camino dirigido infinito en T de la siguiente manera:

- Como G es infinito, r tiene un número infinito de descendientes. Pero como el exgrado de r es finito, debe tener al menos un vértice adyacente, digamos v_{i_1}, con un número infinito de descendientes, por lo que elegimos a ese vértice para extender el camino.
- Supongamos que tenemos ya un camino de longitud k, $\mathsf{r} \rightarrow \mathsf{v}_{i_1} \rightarrow \mathsf{v}_{i_2} \rightarrow \ldots \rightarrow \mathsf{v}_{i_k}$, habiendo elegido sucesivamente, a distancia ℓ al descendiente directo con un número infinito de descendientes. Para el vértice $k+1$ simplemente elegimos, de entre los descendientes directos de v_{i_k} a alguno que tenga un número infinito de descendientes.

Esta construcción se muestra en la Figura E.9. El camino infinito está dado por

$$\mathsf{r} \rightarrow \mathsf{v}_{i_1} \rightarrow \mathsf{v}_{i_2} \rightarrow \ldots \rightarrow \mathsf{v}_{i_k} \rightarrow \ldots \rightarrow \mathsf{v}_{i_\ell} \rightarrow \ldots\ldots\ldots$$

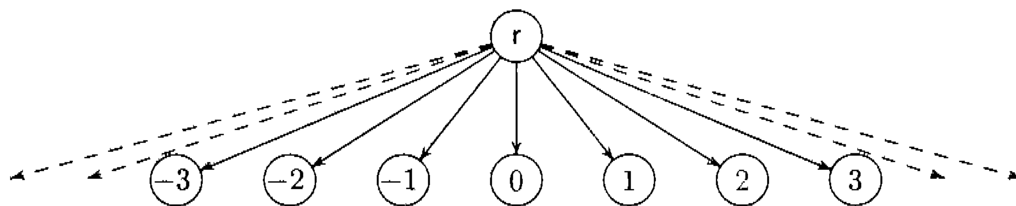
Figura E.9: Construcción de camino dirigido infinito



Es importante hacer notar que la restricción en la finitud del exgrado de cada vértice es una condición necesaria para que el Lema E.7 se cumpla, ya que es una condición necesaria a la hora de elegir a alguno de los hijos y garantizar que existe uno con un número infinito de descendientes. Para mostrar esta necesidad, observemos la gráfica de la Figura E.10, donde la raíz tiene un número infinito de descendientes directos, pero no tiene ningún camino de longitud infinita.

Como se puede observar en esta figura, cada uno de los descendientes directos de r tienen un número finito de descendientes, cero. Lo infinito de la gráfica se debe a que r tiene un número infinito de descendientes directos. De esto, si no garantizamos que cada vértice tiene un exgrado finito, no siempre vamos a poder extender un camino de manera infinita.

Figura E.10: Árbol infinito sin camino infinito



A pesar de que el Lema E.7 es muy sencillo, nos permite sacar conclusiones importantes respecto a algunas gráficas. Por ejemplo, si sabemos que una digráfica tiene exgrado finito

para todos sus vértices y que no contiene ningún camino simple infinito, podemos concluir que la gráfica es finita. Esto garantiza en muchas aplicaciones que la ejecución de algoritmos, por ejemplo para explorar gráficas, puede terminar.

E.4.1. Paradoja de la bolsa de monedas

Cuando se trabaja con conjuntos infinitos se debe tener más cuidado que el usual al plantear problemas y sus soluciones. Veamos como ejemplo la paradoja de la bolsa de monedas.

> Tenemos una bolsa y un número infinito de monedas. Repetimos el siguiente proceso *ad infinitum*:
>
> Echamos dos monedas a la bolsa y sacamos una.
>
> La pregunta que nos hacemos es ¿cuántas monedas quedan al final? La respuesta depende de la manera de echar y sacar monedas de la bolsa:

1. Echamos las monedas 1 y 2, y sacamos la 1; echamos la 3 y 4 y sacamos la 3; etc.
   Queda un número infinito de monedas en la bolsa, ya que quedan las pares.
2. En el momento de la $2 \cdot i$-ésima echada sacamos la i-ésima moneda.
   ¡Al final, no queda ninguna moneda en la bolsa!

E.4.2. Mosaicos: una aplicación del Lema del Infinito

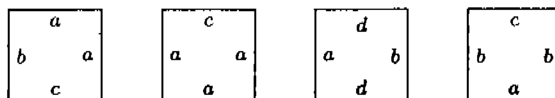
El Lema del Infinito es útil en muy diversas circunstancias prácticas. Por ejemplo, tenemos el siguiente problema de la vida diaria:

Problema: Consideremos el cubrir de mosaicos cuadrados una superficie. Los mosaicos presentan las siguientes propiedades:

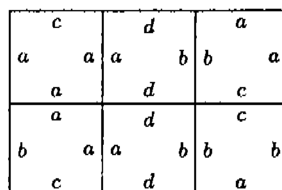
a. Todos los mosaicos son del mismo tamaño.
b. Tenemos un número finito de tipos de mosaicos distintos.
c. Cada lado del mosaico está etiquetado con letras del alfabeto y cada mosaico de un tipo dado tiene las mismas letras, por lo que son indistinguibles uno de otro (del mismo tipo).
d. No se puede rotar ni reflejar los mosaicos, y sus letras corresponden a la orilla norte, oriente, sur y occidente respectivamente.
e. Existe una oferta infinita de cada tipo de mosaico.
f. Los mosaicos se pueden poner uno junto al otro y estar lado a lado, únicamente si los lados que se juntan tienen la misma etiqueta.

Por ejemplo, si tenemos los tipos de mosaicos que se muestran en la Figura E.11, pudiéramos tapizar como se muestra en la misma figura.

Figura E.11: Tipos de mosaicos y enlosetado de 2×3



(a) Tipos de mosaicos



(b) Enlosetado de 3×2

Es fácil notar que la organización de los mosaicos en la Figura E.12(b) corresponde a un "toroide", ya que mantiene las restricciones aún si se considerara a la superficie circular tanto horizontal como verticalmente. Repitiendo este patrón tanto horizontal como verticalmente podemos cubrir a todo el plano.

Es interesante preguntarse bajo qué condiciones se puede cubrir una superficie arbitraria con un número finito de tipos de mosaicos. Y aunque no parece obvio, la solución a esto, dada por Wang en 1961, tiene una íntima relación con el Lema del Infinito, como se enuncia en el Teorema E.8.

Teorema E.8 *Si es posible cubrir el primer cuadrante del plano (de la manera como especificamos arriba) entonces es posible cubrir todo el plano.*

Demostración:

Construimos un árbol con raíz en r y donde cada nivel del árbol es como sigue:

- En el nivel 1 ponemos un vértice por cada una de las posibles maneras de cubrir un cuadrante de lado 1, esto es, por cada uno de los tipos de mosaicos. En el caso del ejemplo que dimos nos queda como se muestra en la Figura E.12.
- En el nivel 2, son descendientes directos de un vértice aquellos enlosetados que se puedan hacer teniendo a ese vértice en el centro, y rodeándolo con una "cinta" de 1 loseta. Estos enlosetados son de tamaño 3×3. Se muestra uno de ellos en la Figura E.13. En esta figura se puede observar que el vértice padre se encuentra en el centro, y tenemos todas las maneras posibles de, con ese mosaico en el centro, rodearlo con mosaicos.

Figura E.12: Nivel 0 y 1 del árbol para enlosetar

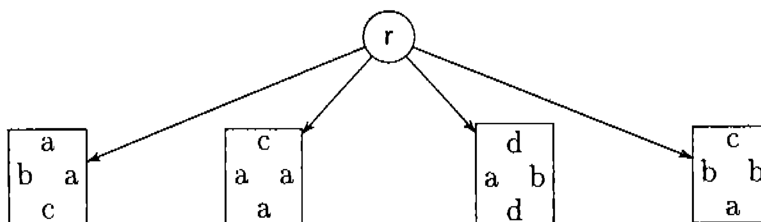
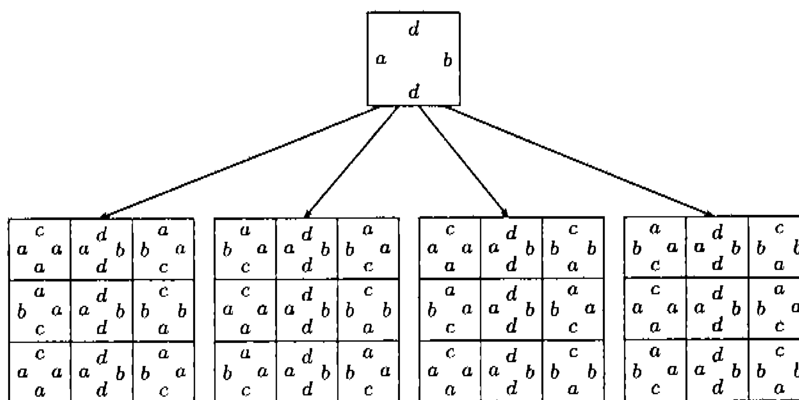


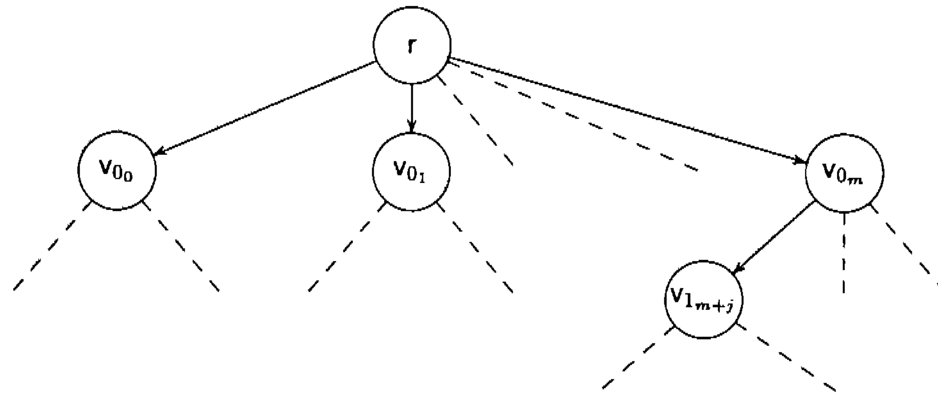
Figura E.13: Un vértice en el nivel 3 del enlosetado



– En términos generales, para cada k tenemos un vértice v_{k_i} por cada una de las formas válidas de enlosetar una superficie de $(2k+1) \times (2k+1)$ mosaicos. El vértice padre de estos vértices es el vértice $\mathsf{v}_{(k-1)_i}$ de superficie $(2k-1) \times (2k-1)$ que corresponde al centro del vértice v_{k_i}, como se muestra en la Figura E.14. En esta figura, los enlosetados $\mathsf{v}_{1_{m+j}}$ tienen como centro al enlosetado v_{0_m}.

Ahora, si para cualquier k existe una manera de enlosetar una superficie de $(2k+1)\times(2k+1)$, entonces T tiene un número infinito de vértices. Como el número de tipos de mosaicos es finito, el exgrado de cada vértice es finito (nótese que no estamos garantizando que esté acotado, sino que en cada nivel del árbol va a haber un número finito de descendientes directos). Por ello, por el Teorema E.7 existe un camino simple infinito en T. Tal camino muestra la manera de enlosetar todo el plano.

Figura E.14: Árbol que corresponde a enlosetados válidos



v_{0_i} : enlosetados de 1×1.
v_{1_i} : enlosetados de 3×3.
v_{2_i} : enlosetados de 5×5.
$\vdots$ $\vdots$
v_{k_i} : enlosetados de $(2k+1) \times (2k+1)$.

Los árboles, como ya mencionamos, juegan un papel sumamente importante en las ciencias de la computación. Si bien vimos únicamente un subconjunto de las propiedades que tienen este tipo de gráficas, introduciremos algunos más en el contexto en que sea necesario. Usaremos los resultados de este apéndice continuamente en este trabajo.

Apéndice F

Análisis amortizado

En este apéndice estudiaremos una forma alternativa de calcular el costo de la ejecución de un algoritmo, llamada análisis amortizado. En este tipo de análisis el costo de las operaciones se evalúa en conjunto en toda la ejecución, no en cada punto donde la operación aparece. Esto proporciona cotas más realistas del costo de un algoritmo, ya que evalúa al conjunto de operaciones y cómo las operaciones interactúan entre sí.

Aprovecharemos el tema de análisis amortizado para mostrar implementaciones de colas de prioridades, cuyas operaciones, bajo la óptica del análisis amortizado, resultan tener un costo más que aceptable.

F.1. Costos amortizados y estructuras de datos avanzadas

Hasta ahora hemos hecho análisis de peor caso, y si bien esto nos proporciona una cota superior para el costo de la ejecución de nuestros algoritmos, muchas veces este costo pesimista no es realista, ya que no se presentará el caso en que lo tengamos que pagar. Como vimos en la construcción del ciclo euleriano, en teoría el costo de cada operación de retroceder por el camino puede ser sumamente alto – $O(|E|)$ en una iteración dada – lo que daría al algoritmo una complejidad de peor caso de $O(|V| \cdot |E|)$.

Sin embargo, no se dará el caso en que *siempre* que saquemos aristas de la pila el número de aristas sea $|E|$, pues para ello sería necesario que hubiésemos introducido tantas aristas como pretendemos sacar, y cada arista es introducida únicamente una vez. En resumen, en ningún momento durante la ejecución del algoritmo podremos sacar más elementos de los que se encuentran en la estructura en ese momento. Es por lo tanto natural el evaluar la complejidad de un algoritmo que utiliza una estructura de datos de este estilo no en términos del peor caso, aislando la operación (en este caso de sacar), sino en términos de un análisis que contemple un periodo más largo y que calcule en base al total de las operaciones durante ese periodo, ya que en un momento dado la historia determina el estado de la estructura – en una operación en la que se pretende eliminar de una estructura de datos dada, no podemos eliminar a más elementos de los que, a lo largo de la ejecución hasta ese momento, hayamos agregado. Veamos qué queremos decir ilustrando con el siguiente ejemplo:

Tenemos una pila con las operaciones tradicionales:

pop(): Bota lo que está en el tope de la pila. Únicamente quita un símbolo en cada operación.

push(a): Coloca el símbolo a en el tope de la pila. Únicamente puede colocar un símbolo en cada operación.

y agregamos una tercera operación,

multipop(k): Ejecuta k operaciones consecutivas de pop() sobre la pila. Si la pila contiene menos de k elementos, entonces ejecutará tantos pop()'s como elementos haya en la pila.

Si hacemos un análisis de peor caso (tradicional) sobre estas operaciones, obtenemos lo que se muestra en la Tabla F.1.

Operación	**Costo**	**Justificación**
push(a)	$O(1)$	Es una única operación.
pop()	$O(1)$	Es una única operación.
multipop(k)	$\text{mín}\{k, s\}$	donde s es el número de elementos que hay en la pila. Si en la pila hay menos de k elementos, este costo se reduce.

Tabla F.1: Costos de peor caso para operaciones de una pila

Dada esta estructura de costos, surge la pregunta:

¿Cuál sería el costo de una sucesión cualquiera de n operaciones?

Como el costo de cada operación, en un análisis de peor caso, es n (el costo de multipop(n)), el análisis de peor caso nos indica que si tenemos una sucesión de n operaciones, debemos suponer el peor caso para cada una de ellas, por lo que el costo total es de n^2. Pero veamos por qué este costo no es realista. Tratemos de construir una ejecución que presente este costo de peor caso. Supongamos que la ejecución presenta una sucesión de operaciones que primero ingresa a n elementos a la pila y después continúa únicamente con multipop's – trataremos de evaluar únicamente las sucesiones de multipop's – y que en el momento de ejecutarse el primer multipop(n) la pila contiene exactamente n elementos. El primer multipop(n) saca n elementos de la pila y la deja vacía, así que cuando se ejecuta el segundo multipop(n), el costo es 0, ya que no quedan elementos por sacar de la pila. Esto muestra que nuestro cálculo de peor caso puede darnos un costo que resulte exagerado, aún como cota superior.

Observemos lo siguiente: el costo de la operación multipop(n), que es el más alto, en realidad depende de cuántos elementos haya en la pila en el momento en que es invocada. No es un costo *intrínseco* a la operación. Por ello, podríamos tratar de cargar el costo de la operación multipop( ) a la operación que coloca elementos en la pila, push(a), que es realmente la que ocasiona que multipop( ) cueste. Este tipo de cálculo es a lo que llamamos *costos amortizados*, ya que el costo de la operación multipop(k) se pretende *amortizar* al efectuar las operaciones que lo determinan, esto es, por cada operación de push(a) que se ejecute, se *deposita* el costo de, finalmente, sacar a ese elemento de la pila. Se puede ver como si cada elemento que se coloca en la pila lleva consigo el "boleto" que tiene que pagar para que lo saquen de la pila. En este caso, la operación de multipop() todo lo que tiene que hacer es "recoger" esos boletos, pero ya no tiene que desembolsar nada por la operación, ya que

ésta fue pagada (*amortizada*) por adelantado. Con este tipo de cálculo de costos, nuestra tabla queda como se muestra en la Tabla F.2.

Operación	**Costo amortizado**	**Justificación**
push(a)	2	Paga 1 por la operación de push(a) y 1 que *deposita* para pagar cuando lo saquen.
pop()	0	El costo ya fue pagado por adelantado, cuando push(a) depositó (*amortizó*) lo que costaría más adelante el sacar a a.
multipop(k)	0	El costo ya fue pagado por adelantado, por lo que esta operación ya no cuesta.

Tabla F.2: Costos amortizados para operaciones de una pila

Esta manera de razonar resulta muy adecuada para aquellos problemas donde el costo de una operación depende de cuantas veces se hayan ejecutado otras operaciones relacionadas. Por ejemplo, si se trata de una búsqueda en un árbol y estamos considerando el costo de insertar nodos en el árbol y de hacer una búsqueda posterior, el costo de la búsqueda va a ser muy bajo si hay muy pocos nodos en el árbol, y va a ser alto si hay muchos. Por ello resulta razonable pedirle a la operación de inserción que asuma el costo posterior de la búsqueda, ya que del número de veces que la operación de inserción se haya ejecutado va a depender el costo de la búsqueda. También se utiliza cuando no es conveniente evaluar el costo de una operación individualmente, ya que la operación analizada aisladamente puede ser muy costosa, pero si se observan sucesiones de operaciones, donde la sucesión particular es la que define el costo total, en esta sucesión no puede suceder que todas las operaciones sean de las costosas, por lo que se "reparte" el costo entre todas las operaciones de la sucesión. Por ejemplo, regresando a la operación de multipop( ), ésta es una operación, en el peor caso, muy costosa, pero su costo real estará dado por la sucesión de operaciones en la cual aparezca.

Al hacer el análisis amortizado de un algoritmo la parte más importante es la decisión de a cuál o cuáles operaciones se les carga el costo de las operaciones más *complejas*, de tal manera que se garantice que cuando la operación compleja es invocada, todo su costo esté ya amortizado.

Cuando se asignan costos amortizados, habrá que demostrar que el costo amortizado debe corresponder a una cota superior del costo real, esto es:

Sea $\{op_i\}$ una sucesión de operaciones en un algoritmo. Entonces, para poder utilizar costo amortizado se debe cumplir que

$$\sum_i costo(op_i) \;\geq\; costo\ real.$$

Recordemos que el análisis de peor caso también corresponde a una cota superior, pero

se pretende que el análisis de costos amortizados dé una cota superior más ajustada, más cercana a la realidad.

Si el costo amortizado, en efecto, representa una cota superior del costo real, podemos utilizar los costos amortizados como si tuviéramos los costos reales en el análisis de las estructuras de datos y sus operaciones.

Si analizamos nuevamente, usando costos amortizados, una sucesión de n operaciones (algunas de las cuáles pudieran ser multipop's) bajo este esquema su costo es de $O(n)$, no de $O(n^2)$.

Nos falta demostrar que en el caso que nos ocupa se cumple que con el costo amortizado en efecto podemos "pagar" por cualquier sucesión de n operaciones. Pero a estas alturas esto es ya obvio:

- Por cada operación de push(a) pagamos el costo de la operación (una unidad) y amortizamos o depositamos otra unidad para pagar por sacar a ese nodo en su momento.
- Cada pop() cuesta en realidad una unidad, pero como ya está depositado ese costo, su costo amortizado es de 0 unidades.
- Lo mismo sucede con multipop(k).

Como se puede observar, el costo amortizado nos ofrece una visión más realista y precisa del costo de la ejecución de un determinado algoritmo, siempre y cuando los costos amortizados estén bien asignados para que se cumpla la condición de que son una cota superior al costo real.

Dado que el análisis de costo amortizado tiene sentido con estructuras de datos donde alguna operación puede tener un costo que depende de lo que se haya hecho hasta ese momento con ellas, presentaremos a continuación algunas estructuras de datos avanzadas en las que este tipo de análisis no sólo es natural, sino que es el único realista. Examinaremos aquellas estructuras de datos que se conocen con el nombre genérico de *colas de prioridades*. Las colas de prioridades son muy útiles cuando tenemos una situación donde hay varios registros compitiendo por ser procesados – en nuestro caso, los vértices o arcos de un gráfica cuando el criterio para su elección no es aleatorio. Estos registros pueden referirse a procesos listos para ser ejecutados, solicitudes de impresión en un dispositivo, personas que desean se les otorgue un préstamo, vértices en una gráfica a los que se les tiene que determinar su distancia al origen. Cada uno de estos registros contiene información pertinente al individuo que desea ser procesado, además de una *llave* que es la que determina su *prioridad*. En el caso de algoritmos de exploración, por ejemplo, esa llave puede ser el peso de una arista o la estimación de la distancia al origen cuando el peso de los arcos no es homogéneo, y que cambia conforme se va desarrollando el algoritmo. En el caso de los procesos listos a ejecutarse, la llave puede ser una combinación de la prioridad que el sistema operativo le asignó al proceso, junto con el tiempo que el proceso lleva esperando y el tiempo total que ya ha sido atendido. Para la solicitud de un préstamo, la llave puede ser una combinación de la antigüedad de la persona en su empleo, su nivel de ingresos, el grupo social al que pertenece. Procedemos a revisar algunas estructuras que permiten implementar eficientemente colas de prioridades.

F.2. Colas de prioridades

Una *cola de prioridades* es, como su nombre lo indica, una estructura de cola (*queue*), pero donde el siguiente a ser atendido es el que presenta la mayor prioridad (la llave con mayor (menor) valor). Los registros se forman en la cola con una cierta prioridad, y durante el proceso de la estructura de datos esta prioridad puede cambiar, moviendo a alguno de los elementos hacia el frente de la cola. La prioridad puede estar dada en términos de mayor valor, como acabamos de decir, o también pudiera tener mayor prioridad el elemento que tenga como llave al menor valor de entre los presentes en la cola. Trabajaremos por el momento con una cola de prioridades donde el elemento con la menor llave es el que debe encontrarse al frente de la misma.

Una cola de prioridades presenta las siguientes operaciones:

i. Agregar a la cola, verificando la posición que le corresponde con respecto al frente de la cola.
ii. Eliminar al mínimo.
iii. Obtener el mínimo.
iv. Incrementar (decrementar) la llave de alguno de los elementos, reorganizando la cola de prioridades.

Una manera directa, pero costosa, de mantener una cola de prioridades es manteniendo ordenados en todo momento a los elementos de la cola. Es costoso porque requiere de un ordenamiento de toda la cola con tres de las cuatro operaciones que acabamos de mencionar. Y en realidad lo único que nos interesa es que el elemento menor se encuentre al frente de la cola, no el orden relativo del resto de los elementos. Si no los tenemos ordenados, debemos también elegir, con cada operación, al menor elemento de la cola, lo que implica revisar a toda la cola en cada ocasión. Ninguna de estas dos opciones es satisfactoria, máxime que la primera solución proporciona demasiada estructura mientras que la segunda demasiada poca. Con colas de prioridades nos interesa que las cuatro operaciones que mencionamos tengan un costo razonable, en términos de análisis amortizado. Esto lo conseguimos con lo que conocemos como *árboles parcialmente ordenados*.

Definición F.1 (Árbol parcialmente ordenado) Un *árbol parcialmente ordenado*[1] es un árbol en el cual cada nodo padre tiene asignado un valor menor o igual que cualquiera de sus hijos, como se ve en la gráfica de la Figura F.1 – se puede tener también que el valor de la llave en un nodo sea *mayor o igual* que el valor de la llave de cualquiera de sus descendientes.

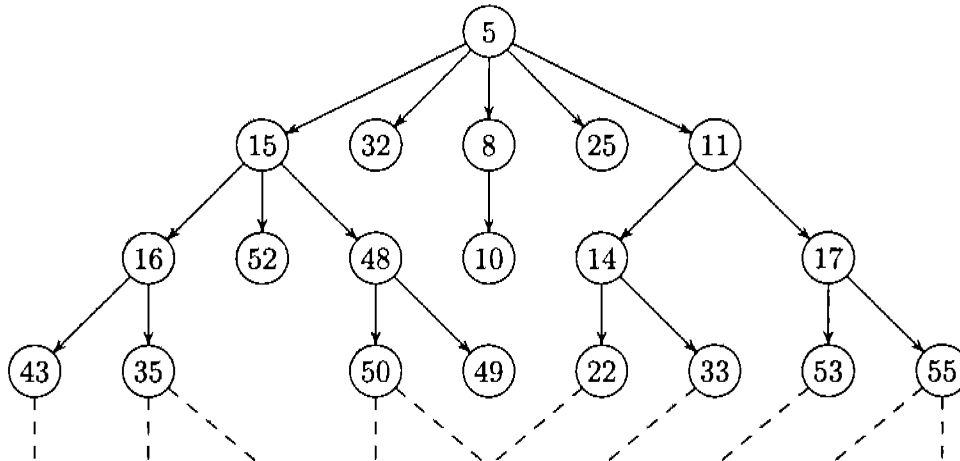
A la propiedad de que que cada nodo tenga un valor menor al de cualquiera de sus hijos se le denomina *propiedad de heap*. Un heap no forzosamente tiene que ser binario o equilibrado. Veremos distintos tipos de heaps en lo que resta de este material. Veremos también colas de prioridades estructuradas con bosques de heaps de distintos tipos.

Los heaps proveen maneras eficientes de implementar colas de prioridades, necesarias para muchos algoritmos como el de caminos más cortos de Dijkstra o el algoritmo para árboles

[1] En inglés *Partially Ordered Trees, POT*. Utilizaremos el término *heap* por carecer de una traducción mejor.

generadores de costo mínimo de Prim. Utilizaremos este tipo de estructuras para mantener colas de prioridades.

Figura F.1: Árbol parcialmente ordenado



Para procesar los registros en una cola de prioridades, no siempre es conveniente que el registro completo participe en el proceso, ya que muchas veces la información adicional a la llave no se requiere para determinar la prioridad. Para los efectos de mostrar las estructuras de datos en este capítulo supondremos que cada registro está representado por un *nodo* que contiene la información pertinente para definir las prioridades, y que mantiene en todo momento ya sea una referencia al resto de la información, o a la información misma en el caso de que sea más económico. Los *nodos* los colocaremos en gráficas, y jugarán el papel que hasta ahora han jugado los vértices. Sin embargo, abandonamos por el momento el término *vértice* ya que éste no involucra información adicional a la estrictamente relacionada con la gráfica. Este tipo de estructura se presta particularmente bien para análisis de costos amortizados, ya que análogamente a como sucede en las pilas, la profundidad de un árbol con estructura rígida depende de la historia de inserciones y remociones que se hayan hecho sobre el árbol. Empezaremos con un análisis de los heaps binarios, para proceder después a presentar sofisticaciones de estas estructuras que permiten un análisis amortizado que, como ya vimos, es más apegado a la realidad.

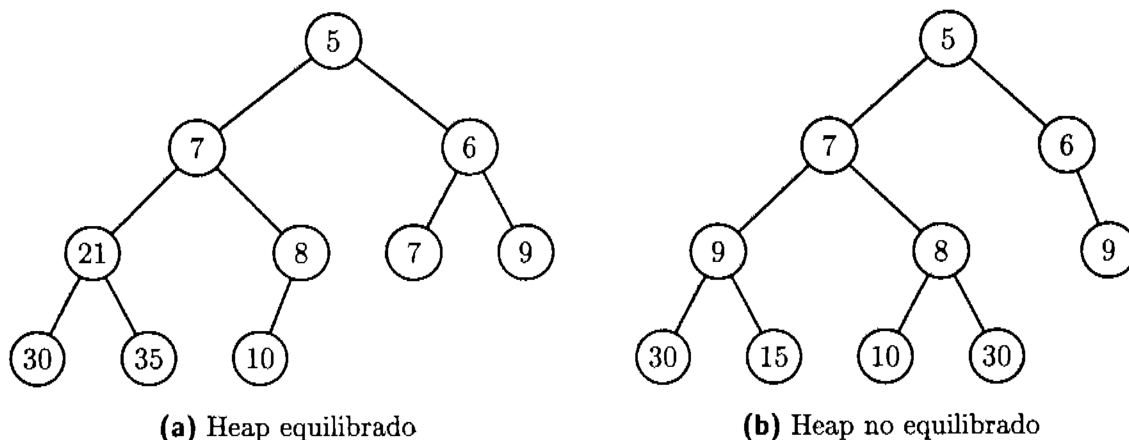
En las listas de prioridades, lo que realmente nos interesa es quién es el que tiene la mayor prioridad en un momento dado, que estaría colocado en la raíz del heap. Un heap lo utilizaremos, entonces, como una estructura de datos para almacenar *nodos*, cada uno de ellos con *valor* asociado (a lo que hemos llamado la llave).

Sin pérdida de generalidad supondremos que el valor que se encuentra en la raíz del heap es el que corresponde al mínimo.

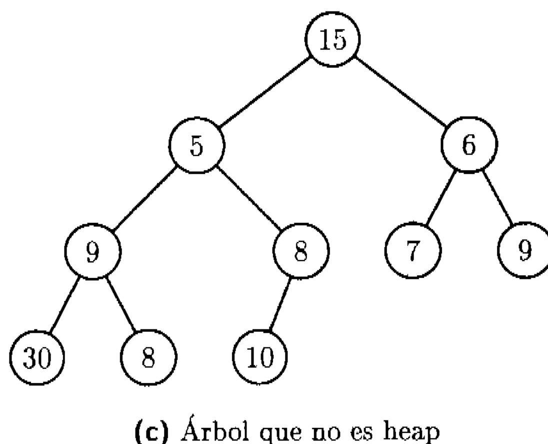
Supongamos que tenemos un heap donde cada nodo tiene a lo más n hijos; decimos entonces que el heap es $n-ario$. Veamos las siguientes definiciones que describen la estructura del heap y determinan cotas relevantes para costos amortizados.

Definición F.2 (Heap equilibrado) Decimos que un heap es *equilibrado* si todos los posibles nodos están presentes en todos los niveles (ver Figura F.2), excepto por el último nivel, el de las hojas. También exigimos que en el nivel de las hojas, todos los nodos presentes se encuentren tan a la izquierda como sea posible.

Figura F.2: Propiedades de un heap



(a) Heap equilibrado **(b)** Heap no equilibrado



(c) Árbol que no es heap

En un heap n-ario equilibrado con k nodos, el camino más largo posible entre la raíz y cualquiera de las hojas es, a lo más, $\log_n k$.

Definición F.3 (Heap completo) Decimos que un heap n-ario de profundidad k es *completo* si el número de nodos en el nivel k es n^k.

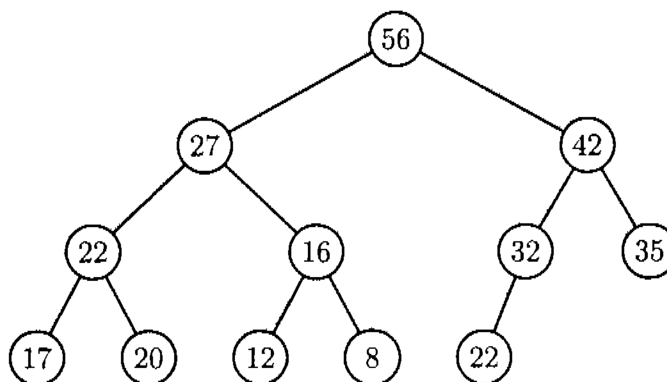
Definición F.4 (Heap binario) Un heap es *binario* si $n = 2$.

Hablamos del *nivel* de un nodo en un heap como la distancia desde él hasta la raíz. El nivel de la raíz es 0, de sus descendientes directos 1 y así sucesivamente. Decimos que la

profundidad de un heap es el máximo nivel entre las hojas del heap. También nos referimos a la *altura* de un nodo como el número de aristas entre él y la hoja más cercana.

Podemos ver un ejemplo de heap, con valores enteros, en la Figura F.3. En ella, podemos observar varios aspectos en el maxheap ilustrado. Antes que nada, se trata de un heap *binario*, pues el máximo número de hijos para cada nodo es dos. Se cumple la definición de heap, pues cada valor en la raíz de un subárbol es mayor o igual a los valores en los nodos de ese subárbol. Sin embargo, no podemos decir nada de la relación entre nodos que son raíces de diferentes subárboles. Por ejemplo, en el segundo nivel la raíz del subárbol izquierdo tiene un valor menor que la raíz del subárbol derecho; pero si observamos en el tercer nivel del subárbol izquierdo sucede lo contrario: el valor de la raíz en el subárbol izquierdo es mayor que el de la raíz del subárbol derecho. También podemos observar que si se repite algún valor, los nodos que lo contienen no tienen por qué estar relacionados entre sí en un mismo subárbol. Tal es el caso del valor 22, por ejemplo.

Figura F.3: Ejemplo de un maxheap



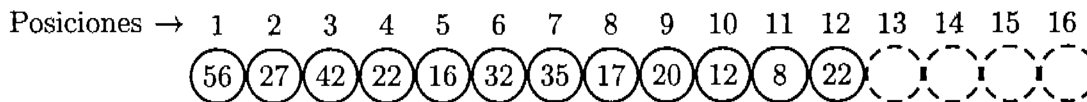
Otro aspecto importante es delimitar el significado de "equilibrado" y "completo". En el caso de un heap, estos términos se refieren a que no puede haber más de un nivel de diferencia entre el subárbol izquierdo y el derecho, y que si lo hay, el subárbol izquierdo deberá ser el más grande. Esta característica hace que el heap pueda ser almacenado en un arreglo con acceso directo, en posiciones consecutivas del arreglo.

Manejo del heap

Como acabamos de mencionar, una de las grandes ventajas de un heap binario, equilibrado y casi completo es que se puede almacenar en una lista con acceso directo, ocupando posiciones consecutivas, a partir de la posición 1. Los nodos se guardan por niveles, con todos los nodos de un mismo nivel ocupando posiciones consecutivas. De esta manera, se puede calcular fácilmente la posición de un descendiente desde la posición del padre y, viceversa, calcular la posición del padre sabiendo la de cualquiera de sus hijos. Si el padre se encuentra en la posición i, su hijo izquierdo se encuentra en la posición $2i$ y su hijo derecho en la posición $2i + 1$. Similarmente, si el hijo se encuentra en la posición j, el padre se encuentra en la

posición $\lfloor j/2 \rfloor$. Por ejemplo, el heap de la Figura F.3 estaría guardado en la lista que aparece en la Figura F.4.

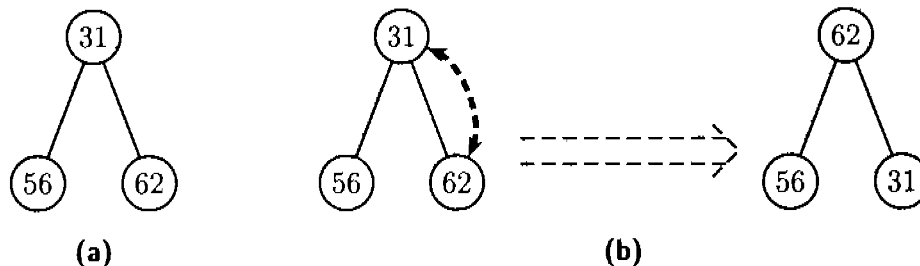
Figura F.4: Organización de un heap en una lista con acceso directo



Recordemos que un árbol binario tiene la propiedad de ser un heap si dado cualquier nodo y su llave correspondiente, las llaves de sus hijos son menores o iguales a aquélla. Un subárbol que consiste únicamente de la raíz tiene la propiedad de heap.

Es fácil verificar que un árbol con dos niveles tenga la propiedad de heap, y si no la tiene, manipularlo para que se convierta en un heap. Si tenemos, por ejemplo, el heap de la subfigura F.5(a), lo que debemos hacer es lo que se muestra en la subfigura F.5(b).

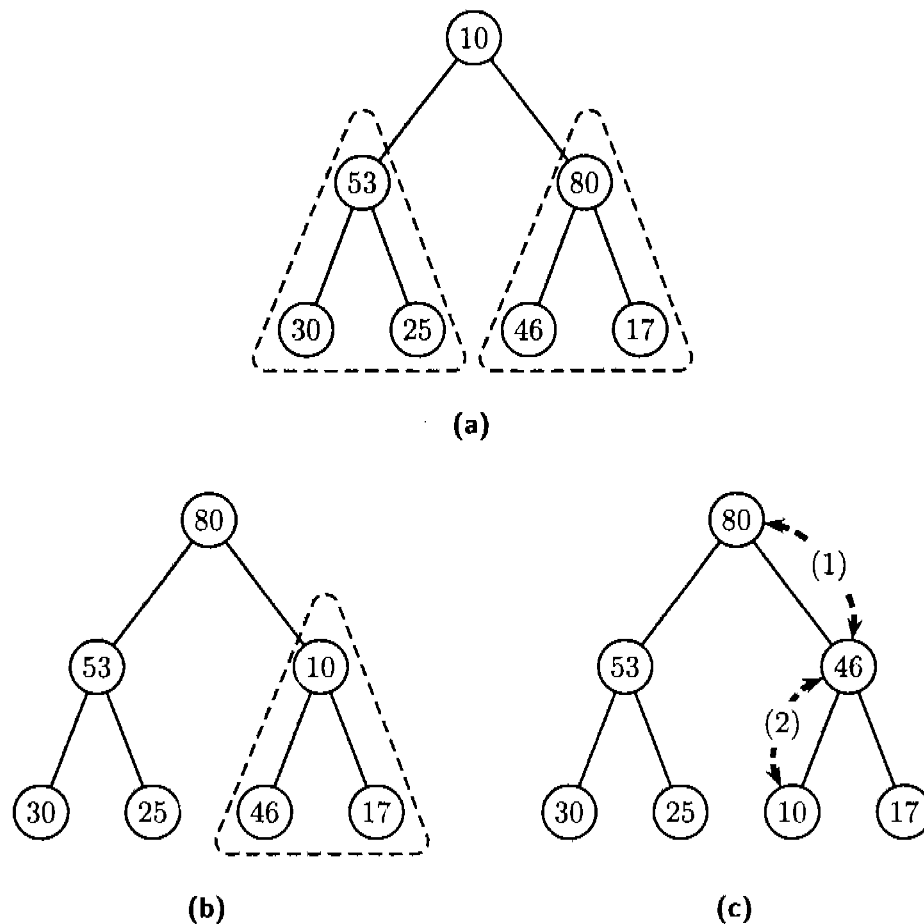
Figura F.5: Propiedad de heap en subárbol de dos niveles



Una vez intercambiado el padre con el mayor de sus hijos, si es que alguno de sus hijos es mayor a él, se recupera la propiedad de heap. Pero veamos, por ejemplo, un maxheap con profundidad $k > 2$, como el que se muestra en la Figura F.6(a), y en el que estamos insertando a un elemento nuevo en la raíz. Podemos contar con que los subárboles de profundidad $k - 1$ tienen la propiedad de heap antes de llevar a cabo el intercambio.

En este caso, como se puede observar en la Figura F.6(b), no basta con intercambiar a la raíz con el mayor de sus hijos, porque el resultado sería un árbol que no forzosamente presentaría la propiedad de heap, el subárbol derecho. Se debe continuar hacia las hojas del heap para restaurar la propiedad de heap, como si el nodo a insertar se encontrara en la raíz del subárbol afectado. Los intercambios que se deben llevar a cabo se pueden ver en la Figura F.6(c). Por supuesto que el subárbol izquierdo en este caso, como antes del intercambio ya presentaba la propiedad de heap y el intercambio no lo tocó, permanece exactamente igual.

Figura F.6: Garantizando propiedad de heap en árbol con profundidad mayor a 2



En resumen, el intercambio de un nodo en la raíz de un subárbol por alguno de sus hijos tiene que propagarse hacia las hojas del subárbol, por la rama del hijo que fue intercambiando, hasta que se encuentre con una raíz que presenta la propiedad de heap. Llamamos a este proceso (método) heapify, y se muestra en el Listado F.1.

Listado F.1: Método que inserta un elemento nuevo en un heap con propiedad de heap 1/2

```
101      /* Verifica la condición de heap del subárbol almacenado en  *
102       * [i..n].                                                   */
103    private void heapify(int i, int n) {
104        int l = 2 * i;            // Referencia a hijo izquierdo.
105        int r = 2 * i + 1;        // Referencia a hijo derecho.
106        int largest;              // Referencia al de mayor valor.
107      /* Si existe hijo izquierdo y es mayor, referirlo.          */
108        if (l <= n && comp.gtr(objs.get(l), objs.get(i)))
109            largest = l;
110        else
111            largest = i;
```

Listado F.1: Método que inserta un elemento nuevo en un heap con propiedad de heap 2/2

```
112        /* Si existe hijo derecho y es mayor, referirlo.                  */
113          if (r <= n && comp.gtr(objs.get(r), objs.get(largest)))
114              largest = r;
115        /* Si el mayor valor no está en la raíz, intercambiarlos         *
116         * y proseguir hacia abajo del árbol.                            */
117          if (i != largest) {
118              swap(i, largest);
119              heapify(largest, n);
120          }
121      }
```

Tomemos el caso de recibir una lista de datos con una permutación arbitraria, y organizarlo en un maxheap binario. La primera idea que viene a la mente es la de ir construyendo el heap tomando a los elementos de la entrada en el orden en que vienen, e insertar a cada uno desde la raíz. Pero con la representación que hemos dado de una lista con acceso directo, esto implicaría cada vez que se inserta un nuevo elemento recorrer al resto para hacerle lugar, o bien intercambiarlo con el menor elemento; lo primero resulta muy costoso y lo segundo haría que perdiéramos la garantía de que los subárboles presentan la propiedad de heap.

No hay forma realmente eficiente de construir un heap a partir de una permutación arbitraria, pero tenemos una forma relativamente económica de hacerlo, que usa el método heapify, pero en condiciones tales en que puede garantizar la propiedad de heap en los subárboles. Justificaremos después por qué decimos que es relativamente económica.

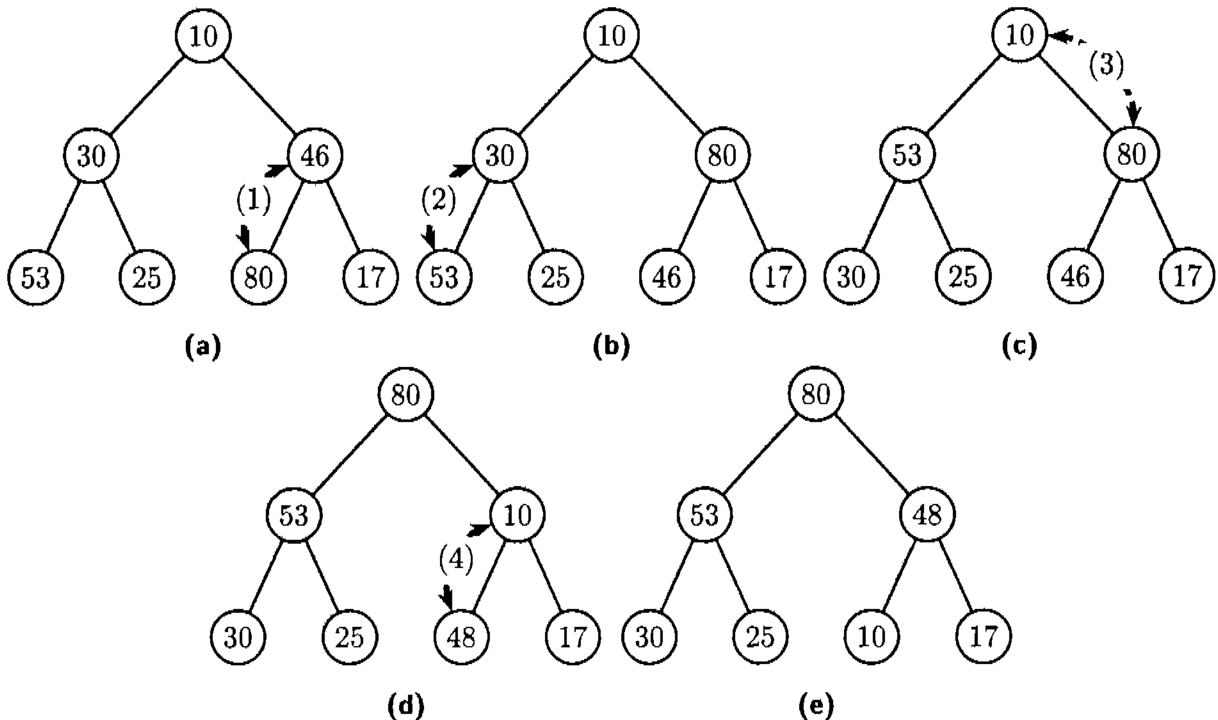
Dadas las consideraciones anteriores, la conversión de un árbol binario cualquiera en un heap deberá recorrer los subárboles desde el penúltimo nivel hacia arriba, revisando cada pareja de hijos y obteniendo la referencia al mayor de ellos; una vez hecho esto, se compara contra el valor de la raíz de ese subárbol; si es necesario un intercambio, se lleva a cabo, y se baja por el subárbol con el que se hizo el intercambio, verificando que se mantenga la propiedad de heap. El algoritmo se encuentra en el Listado F.2, desde donde se invoca al método que se encarga de garantizar la propiedad de heap a partir de una raíz – recordemos que el árbol binario está guardado en una lista con acceso directo, por lo que los hijos directos del nodo i se encuentran en las posiciones $2i$ y $2i+1$.

Listado F.2: Método que establece la propiedad de heap en una permutación arbitraria

```
201        /* Organiza el recorrido del heap desde el último subárbol       *
202         * hacia el primero.                                             */
203      private void buildHeap(int N) {
204          for (int i = ((int) Math.floor(N / 2)); i > 0; i--)
205              heapify(i, N);
206      }
```

La idea en este caso es ir garantizando que los subárboles cumplen la propiedad de heap de la misma manera en que lo hicimos en el caso anterior, e ir subiendo en los niveles de los árboles. Si hacemos únicamente eso, el árbol nos quedaría como se muestra en la Figura F.7, con los nodos que fueron intercambiados señalados.

Figura F.7: Intercambios necesarios hasta garantizar un maxheap



Como se puede ver de los intercambios hechos al árbol en la Figura F.7(a) para producir el árbol de la Figura F.7(b), si bien se obtuvo al elemento mayor en la raíz del árbol, el árbol no es un heap, ya que para el nodo con valor 10, su valor no es mayor que el de cualquiera de sus hijos. No es suficiente "emerger" con el valor mayor. Cada vez que se hace un intercambio, se debe descender en el árbol, por la rama del hijo intercambiado, para verificar que se mantiene la propiedad de heap hacia abajo del nodo recién intercambiado. Si partimos nuevamente del árbol en la Figura F.7(a), podemos observar en la Figura F.7 todos los intercambios que debemos hacer hasta garantizar que tenemos un heap.

Lo primero que deberemos establecer es que el algoritmo mostrado en los listados F.1 y F.2 en efecto terminan con un maxheap en el arreglo.

Lema F.1 (Construcción correcta de un maxheap) *Si se ejecuta el método buildHeap sobre una lista de acceso directo con n elementos, el resultado es un maxheap, que corresponde a una permutación de la lista original, y que ocupa las posiciones 1 a n de la nueva lista.*

Demostración:

Haremos la demostración por inducción sobre n, el número de elementos de la lista.

Base: Para $n = 1$, si el árbol consiste de un único elemento, el ciclo de la línea [204] no se ejecuta ni una sola vez, por lo que la propiedad de maxheap se cumple trivialmente para un árbol que consiste únicamente de la raíz.

Inducción: Como el proceso de heapify se lleva a cabo de abajo hacia arriba, cuando estamos revisando a un árbol con raíz en i, sabemos que los árboles con raíz en $2i$ y $2i + 1$ ya

cumplen con la propiedad de heap, esto es, que su valor no es menor que el de cualquiera de sus descendientes, en particular de sus hijos directos, de existir éstos. Ésta será nuestra hipótesis de inducción. Veamos qué pasa entonces al aplicar al nodo en la posición i el método heapify. Revisemos cada uno de los casos posibles:

(i) Se cumple la condición en la línea [108], esto es, existe un hijo izquierdo y éste tiene un valor mayor al del padre: largest queda refiriéndose al hijo izquierdo.

(ii) Se cumple la condición de la línea [113] que dice que existe un hijo derecho y éste es mayor que el mayor entre el padre y el hermano izquierdo: largest queda refiriéndose al hijo derecho.

(iii) O bien no existen hijos de este nodo, o ninguno de los hijos tiene un valor mayor. largest queda refiriéndose al padre.

En la línea [117] se verifica que se tenga el caso (i) o (ii), y si es así es necesario el intercambio. Después de este intercambio tenemos que se cumple:

$$objs[i] \geq objs[2i] \qquad \text{y} \qquad objs[i] \geq objs[2i+1]. \tag{1}$$

Por esta relación y por la hipótesis de inducción, sabemos que todos los elementos en los subárboles enraizados en $2i$ y en $2i+1$ cumplen con la propiedad de heap respecto a la nueva raíz en i, pero no sabemos nada de la relación de esos nodos con la nueva raíz en $2i$ o en $2i+1$, dependiendo de con cuál de ellos se haya hecho el intercambio.

Supongamos que se intercambió a la raíz original con el de la posición $2i$ – el argumento es idéntico si se trata del nodo en $2i+1$. Por la hipótesis de inducción, heapify sabe garantizar la propiedad de heap en el subárbol con raíz en $2i$, ya que tiene menos nodos que el original. Además, cualquiera que sea el resultado, la propiedad de maxheap nos dice que en ese subárbol todos los nodos tienen valores no mayores que el de la raíz que estaba en $2i$; como hicimos el intercambio entre el nodo en i y el nodo en $2i$, sabemos que el nodo en $2i$ era menor que el nodo en i, y sólo pudo sumergirse en el subárbol. De esto, todos los nodos en el subárbol enraizado en $2i$ son no mayores que el nodo en i. Con el árbol enraizado en $2i+1$ no hay que hacer nada, pues ya se cumplía la propiedad de maxheap para el subárbol con respecto al nodo en $2i+1$, y además no se cumplió la condición en la línea [113], ya sea porque no existe subárbol derecho o porque la raíz de ese subárbol contiene un valor no mayor que el del nodo en i. Como este subárbol no se modificó de ninguna manera, por transitividad tenemos que todos los nodos en el subárbol derecho del árbol enraizado en i cumplen con la propiedad de maxheap respecto a i.

$\therefore$ si heapify es llamado a trabajar sobre un árbol enraizado en i, y los subárboles de ese árbol presentan la propiedad de maxheap, entonces, al terminar de ejecutarse heapify el árbol enraizado en i presenta la propiedad de maxheap. ❏

Complejidad del método heapify. Este método es invocado indirectamente desde el constructor, y después en cada iteración desde el método select. Analicemos su complejidad.

heapify(int i, int n) (Líneas [103] a [121] del Listado F.1).

- Inicialización – líneas [104] y [105]:

Número de pasos: 2 en: $O(1)$.

- Determinar la referencia al nodo con mayor valor – líneas [108] a [114](*):
  Número de pasos: 4 en: $O(1)$.
- Condición en la línea [117]:
  Número de pasos: 1 en: $O(1)$.
- Intercambio en la línea [118](**):
  Número de pasos: 3 en: $O(1)$.
- Llamada recursiva a heapify(i*2, N)(***)
  Número de pasos: $T(3n/2) + 9$.

Nota(*): Si la condición en la línea [113] se cumple tenemos 4 operaciones que se ejecutan, mientras que si esta condición no se cumple, entonces únicamente tendremos 3 operaciones. La diferencia no amerita hacer la diferencia.

Nota(**): Este intercambio únicamente se llevará a cabo si es que la condición en la línea [117] es verdadera, en cuyo caso se procederá a la llamada recursiva de la línea [119].

Nota(***): Cada vez que se llama a heapify en la recursión, el número de nodos en el árbol sobre el que se hace la llamada es a lo más $3/2$ del número original y al menos $1/2$. En el primer caso tenemos un árbol donde el último nivel del subárbol izquierdo está completo, pero el subárbol derecho tiene un nivel menos, y la recursión se hace precisamente sobre el lado izquierdo. En el segundo caso, el árbol es completo y perfectamente equilibrado, y entonces hay $n/2$ nodos en cada subárbol.

Peor caso: Cuando se tiene que bajar todo el subárbol restituyendo la propiedad de heap. En este caso, como el subárbol por el que se baja tiene a lo más $(2/3)n$, la recurrencia está dada por $T(2n/3)+9$ y en cada llamada recursiva se reduce en 1 la altura del árbol sobre el que se trabaja. Si en cada nivel se llevan a cabo a lo más dos comparaciones, una entre los hermanos y otra con el padre, además del intercambio, se llevarán a cabo $9\log_2 n$ comparaciones e intercambios, donde $\log_2 n$ es la altura de un árbol binario, completo y equilibrado con n nodos.
Número de pasos: $9 \cdot \log_2 n$ en: $O(\log_2 n)$.

Mejor caso: Cuando la raíz del subárbol tiene la propiedad de heap.
Número de pasos: 7 en: $O(1)$.

Caso promedio: La recurrencia es la misma que para el peor caso, excepto que se recorre la mitad de la altura del árbol.
Número de pasos: $\frac{9}{2}\log_2 n$ en: $O(\log_2 n)$.

Total para heapify:

Peor caso:	Número de pasos:	$9\log_2 n$	en:	$O(\log_2 n)$.
Mejor caso:	Número de pasos:	7	en:	$O(1)$.
Caso promedio:	Número de pasos:	$\frac{9}{2}\log_2 n$	en:	$O(\log_2(n))$.

De lo anterior, el costo del método heapify queda acotado de la siguiente manera:

$$7 \le \mathsf{f}_{\mathsf{heapify}}(\mathsf{n}-\mathsf{i}) \le 9\log_2 n,$$

y por lo tanto en la clase

$$O(\log_2 n).$$

Conociendo la complejidad de heapify no es muy complicado calcular la complejidad de buildHeap.

buildHeap(int N) (Líneas [203] a [206] del Listado F.2)

- Control de la iteración en la línea [204]
  Número de pasos: $\lfloor n/2 \rfloor$ en: $O(n)$.
- Ejecución de heapify$^{(*)}$.
  Número de pasos: $9(\log_2 n - \log_2 i)$ en: $O(\log_2 n)$.

Nota$^{(*)}$: Tomaremos en este caso los tres casos presentados en el análisis de **heapify**.

Peor caso: Si los datos vienen en el orden deseado, cada construcción de un subheap tendrá que sumergir a cada raíz, ya que la permutación original da un minheap.

Número de pasos: $\sum_{k=1}^{\lfloor n/2 \rfloor} 9 \log_2 k \leq 5(n \log_2 n - 1.433n)$ en: $O(n \log_2 n)$.

Mejor caso: Este caso se da cuando la permutación original viene en orden inverso; tenemos entonces un maxheap desde el principio. En este caso, cada vez que se toca la raíz de cada subheap, lo único que se hace es comparar a la raíz con sus dos hijos.

Número de pasos: $\sum_{i=1}^{\lfloor n/2 \rfloor} 4 \leq 2n.$ en: $O(n)$

Caso promedio: Este caso se presenta si cada vez que se verifica si una de las raíces presenta o no la propiedad de heap, la profundidad a la que tiene que desplazar a la raíz es aproximadamente:

$3(n \log_2 n - 1.433n)$ $O(n \log_2 n)$,

la mitad de la profundidad del subárbol. Tenemos entonces la mitad de la complejidad de peor caso.

De esto, las cotas para la complejidad de heapify quedan de la siguiente manera:

Total para buildHeap:

Peor caso:	Número de pasos:	$5(n \log_2 n - 1.433n)$	en:	$O(n \log_2 n)$.
Mejor caso:	Número de pasos:	$2n$	en:	$O(n)$.
Caso promedio:	Número de pasos:	$3(n \log_2 n - 1.433n)$	en:	$O(n \log_2 n)$.

Del análisis anterior, tenemos que:

$$\mathbf{2n} \leq \mathsf{f}_{\mathsf{buildHeap}}(\mathsf{n}) \leq \mathbf{5(n \log_2 n - 1.433n)},$$

quedando este método en

$$O(n \log_2 n).$$

Con estos dos métodos definidos podemos pensar en un heap binario como una clase, donde cada heap es un ejemplar (objeto) y lo identificamos con h. La clase debe soportar las siguientes operaciones:

h.inserta(k):	Acomoda al nodo con llave k en el heap h, de tal manera que se mantenga la propiedad de heap en h.
h.daMinimo():	Regresa el nodo cuya llave se encuentre en la raíz del heap, sin modificar al heap.
h.eliminaMin():	Quita al nodo de la raíz del heap, y reacomoda al árbol para que siga teniendo la propiedad de heap.
h.reduceLlave(k, Δ):	Reduce el valor de la llave del nodo en la posición k en Δ unidades y reacomoda al árbol para mantener la propiedad de heap.

Seguiremos por el momento trabajando con heaps binarios. En la Figura F.2 presentamos distintos estados para heaps binarios.

Aunque no hemos discutido el costo de cada una de estas operaciones en el marco de la complejidad amortizada, avanzaremos en la definición de la clase que corresponde a una cola de prioridades. Pasemos a revisar cada uno de los métodos que requiere esta clase, presentadas en una interfaz de Java en el Listado F.3. Revisaremos uno por uno la implementación de cada uno de estos métodos. En lo que sigue trabajaremos con un minheap.

F.2.1. Insertar un nodo en un heap binario

En el Listado F.3 presentamos la interfaz que define los servicios que debe proporcionar un heap.Las estructuras de datos necesarias para implementar esta interfaz se muestran en el Listado F.4, donde se elabora ya la implementación de la interfaz del Listado F.3.

Listado F.3: Interfaz para un heap binario 1/2

```
 1 public interface Heap {
 2         /* Agrega a un elemento al final del heap, sin verificar   *
 3          * que la propiedad de heap se mantenga.                   */
 4     Object inserta(Object v);
 5         /* Garantiza la propiedad de heap para todo el arreglo,    *
 6          * suponiendo que no hay ningún orden previo.              */
 7     void heapifica();
 8         /* Si el heap no está vacío, regresa una referencia al    *
 9          * elemento al frente de la cola de prioridades.           */
10     Object selecciona();
11         /* Elimina al elemento del heap, intercambiándolo con el  *
12          * último, y luego descartando la última posición.         */
13     void elimina(Object obj);
```

Listado F.3: Interfaz para un heap binario 2/2

```
14         /* Sumerge a un elemento desde su posición hasta la que     *
15          * debe ocupar para preservar la propiedad de heap.         */
16     void sumerge(int i);
17         /* Reduce la llave de un elemento y le busca el nuevo       *
18          * lugar para garantizar que la propiedad de heap se        *
19          * sigue cumpliendo después de reducir la llave.            */
20     void reduceLlave(int i, int delta);
21         /* Dice si el heap está vacío.                              */
22     boolean esVacia();
23 }
```

Listado F.4: Estructuras de datos para la implementación de un heap binario

```
1 public class BinMinHeap implements Heap {
2         /* El heap va a ser implementado en un arreglo.              */
3     private ElementoDeMiLista[] elementos;
4         /* Marca la última posición que pertenece al heap.           */
5     private int              ultimo = 0;
6         /* El tamaño del arreglo que contiene al heap.               */
7     private int              tamHeap;
8         /* Número de elementos agregado al heap.                     */
9     private int              cont;
10        /* Comparador que define el orden entre los elementos        *
11         * del heap.                                                 */
12    private Comparator comp;
13        /* Constructor del heap con n elementos. Establece el        *
14         * comparador.                                               */
15    public BinMinHeap(int n, Comparator comp) {
16        elementos = new ElementoDeMiLista[n + 1];
17        ultimo = 0;
18        tamHeap = n;
19        this.comp = comp;
20    }
```

El primer método que revisaremos es el que inserta (o acomoda) a un nodo en el heap. El método de inserción coloca al nodo nuevo al final del arreglo (en el primer lugar disponible en el arreglo, donde los nodos se encuentran ocupando lugares consecutivos desde la posición 1). Si hay en ese momento n nodos en el heap, los lugares ocupados son del 1 al n. Mantenemos una variable que nos dice cuántos nodos hay en el heap (contador). Al agregar el nodo, contador se incrementa en 1.

Después procedemos a *flotarlo*[2] hacia la raíz del heap. Esta operación de flotar consiste en ir comparando la llave del nodo nuevo con la de su padre, y si no cumplen la propiedad de heap (la llave del padre debe ser mayor o igual a la del nodo en cuestión), son intercambiados

[2] En inglés, *bubble up*

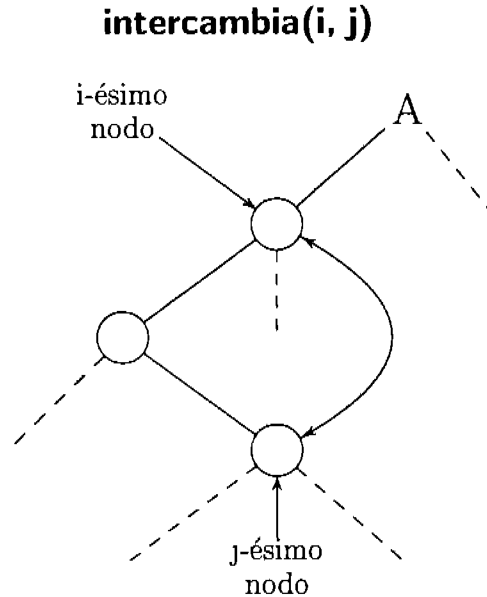
entre sí. Se continúa de esta manera hasta que, o bien ya se cumple la propiedad de heap, o el nodo nuevo se coloca en la raíz del heap. Veamos el método que se encuentra en el Listado F.5. Omitimos el código del método intercambia ya que éste es el usual.

Listado F.5: Procedimiento que inserta a un elemento en su lugar

```
21         /* Agrega un elemento al heap, suponiendo que el resto      *
22          * del heap cumple con la propiedad de heap.                */
23     public Object inserta(Object v) {
24         ultimo++;
25         cont++;
26         if (ultimo > tamHeap) {
27             throw new IllegalArgumentException
28                 (¡"No más espacio en el heap!");
29         }
30
31         ElementoDeMiLista el = new ElementoDeMiLista(v, ultimo);
32         elementos[ultimo] = el;
33         if (ultimo == 1)  { // El primero en ser agregado.
34             return el;
35         }
36
37         /* se procede a ''flotar'' al elemento.                     */
38         int locH = ultimo;             // Posición del hijo.
39         int locP = (int) (locH / 2); // Posición del padre.
40         while (locP != 0)  {
41             if (comp.geq(elementos[locH].getInfo(),
42                          elementos[locP].getInfo())) {
43                 return el;
44             } else  {      // Tenemos que subirlo por el heap.
45                 intercambia(locH, locP);
46                 locH = locP;
47                 locP = (int) (locP / 2);
48             }
49         }   /* Si la ejecución llega a este enunciado quiere        *
50              * decir que locP es 0 y su posición es en el           *
51              * último lugar.                                         */
52         return el;
53     }
```

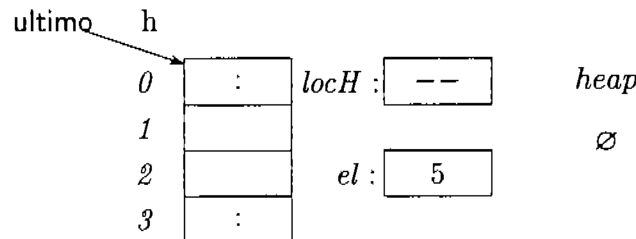
Es importante mencionar que dos nodos se pueden intercambiar sólo si uno de ellos es antecesor en el árbol del otro, y esto se cumple cuando se puede llegar de uno al otro sin pasar por un antecesor común. Uno de los nodos puede ser la raíz, en cuyo caso se puede intercambiar con cualquiera de los nodos en el árbol. En el caso de inserta, los nodos que se intercambian son siempre un padre y su hijo, por lo que no hay necesidad de vigilar esto en el código.

Figura F.8: Intercambio de dos nodos en un heap



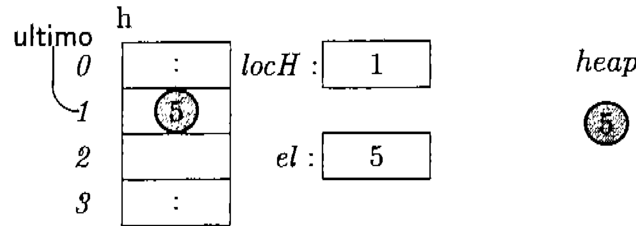
Observemos cómo se arma el heap para los valores $\{5, 3, 8, 4, 2, 6, 7\}$. En la parte izquierda de las figuras mostraremos el estado de las estructuras de datos, mientras que en la parte derecha mostraremos conceptualmente al heap. En la discusión que sigue, cuando mencionemos líneas de código nos referiremos al Listado F.5.

Figura F.9: Al inicio de las operaciones en el heap



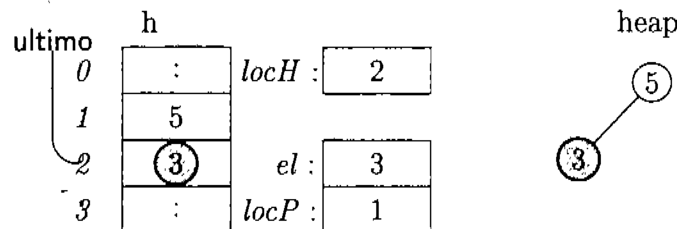
Al insertar al nodo con llave 5, como el lugar en el que estamos metiendo al primer valor es el 1, la condición en la línea [33] se cumple, por lo que después de acomodar al elemento en el heap – línea [32] – la ejecución va a la línea [34] y regresa. El efecto se puede observar en la Figura F.10.

Figura F.10: (1) Al insertar el nodo con llave=5



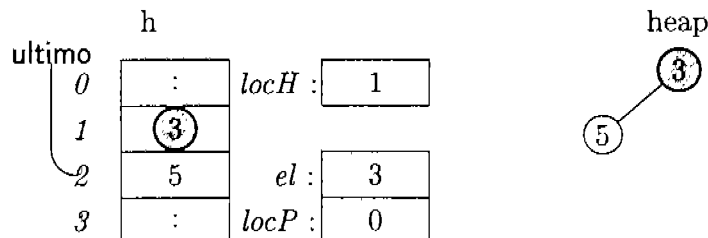
La segunda llamada es inserta(3) y se presenta como se ve en la Figura F.11. Como no se cumple ultimo == 1 en la línea [33], ejecutamos las líneas [38–39] y llegamos al inicio de la iteración en la línea [40].

Figura F.11: (2) Al insertar el nodo con llave=3



Como en la línea [41] no se cumple que la llave del padre sea menor que la llave del hijo, hay que reacomodarlos, invocando al método intercambia en la línea [45] y reubicando la posición del nodo a comparar en las líneas [46–47]. Regresamos a la línea [40], y como no se cumple que la llave del hijo sea mayor o igual que la del padre, pasamos a la línea [43] donde se termina la invocación de esta llamada. Esta invocación se puede observar en la Figura F.12.

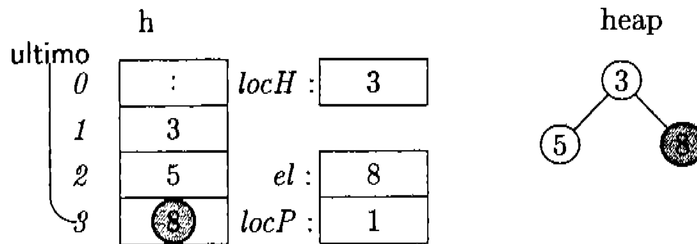
Figura F.12: (3) "Flotamos" al valor 3 hacia la raíz del heap



Volvemos a llamar a inserta con 8 en la llave. Como el nodo cuya llave es 8, al ser insertado

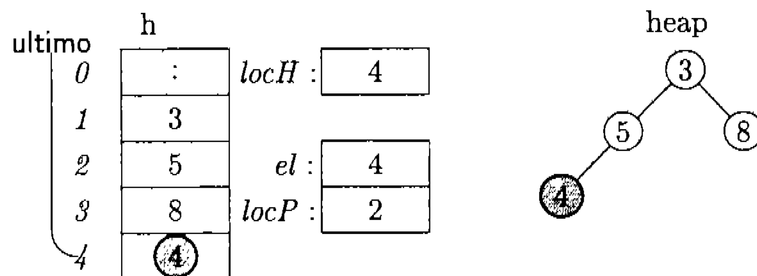
mantiene la propiedad de heap (es mayor o igual a su padre, cuya llave es 3), no hay necesidad de flotarlo, y la ejecución de este método termina saliendo por la línea [43]. Las estructuras quedan como se muestra en la Figura F.13.

Figura F.13: (4) Inserción del nodo con llave=8



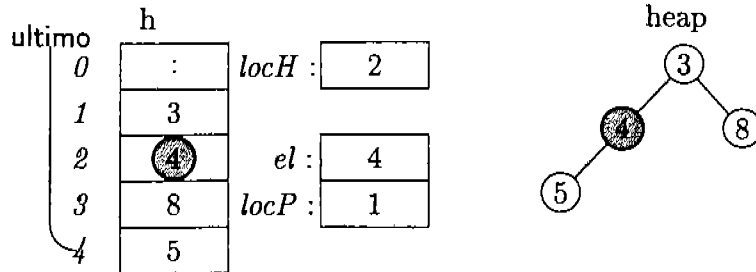
Llamamos nuevamente a inserta con 4 como argumento. Como se cumple, en la línea [41], que la llave del hijo es menor que la del padre, tenemos que intercambiarlos. Esta invocación se ve en la Figura F.14.

Figura F.14: (5) Insertamos al nodo con llave 4



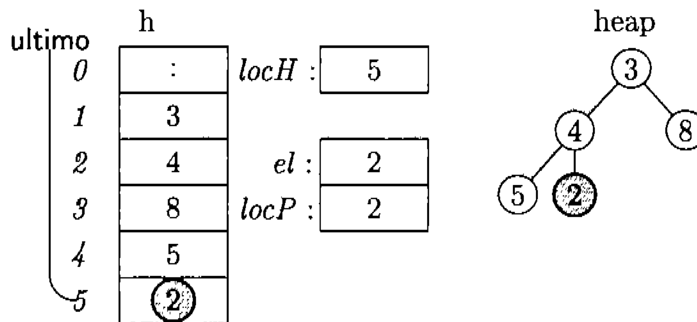
Como al insertar al nodo con 4 como llave se deja de cumplir la propiedad de heap, tenemos que flotar a ese nodo hacia la raíz. Lo intercambiamos con su padre, ya que éste tiene una llave mayor que la suya, flotándolo un nivel hacia arriba en el árbol. En la línea [41] ya no se cumple que la llave del hijo sea mayor o igual que la del padre, por lo que salimos de la invocación en la línea [43]. Al intercambiar los valores, quedan las estructuras como se ve en la Figura F.15.

Figura F.15: (6) El nuevo nodo "flotado" un nivel hacia arriba



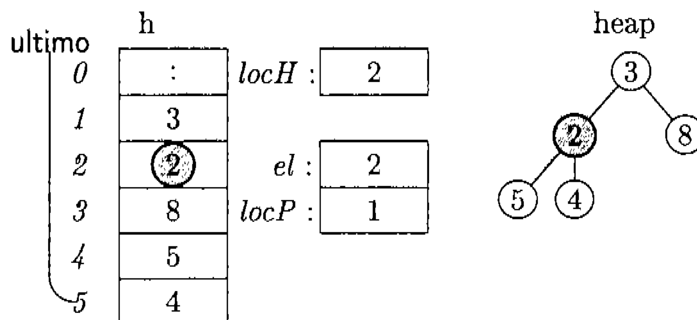
Insertamos ahora al nodo cuya llave es 2. En la línea [40], al llegar a la iteración, las estructuras de datos están como se muestran en la Figura F.16.

Figura F.16: (7) Inserción del nodo con llave=2



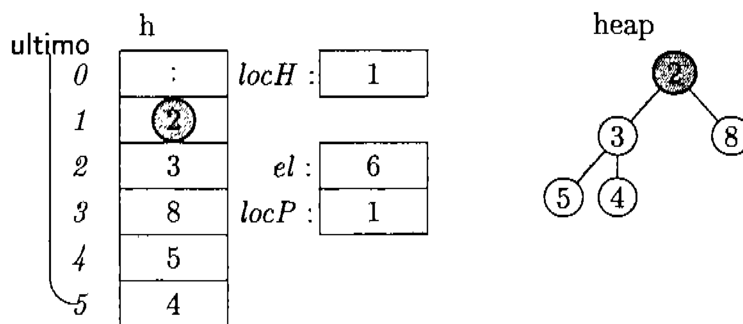
Al insertar al nodo con llave 2 se deja de cumplir la propiedad de heap, por lo que tenemos que flotar a este nodo hacia la raíz. Como locP $\neq 0$ y la llave del padre es mayor que la del hijo, ejecutamos las líneas [45–47] y regresamos a la línea [40] con locP $= 2$. Estas acciones se pueden observar en la Figura F.17.

Figura F.17: (8) Intercambio del nuevo nodo con su padre



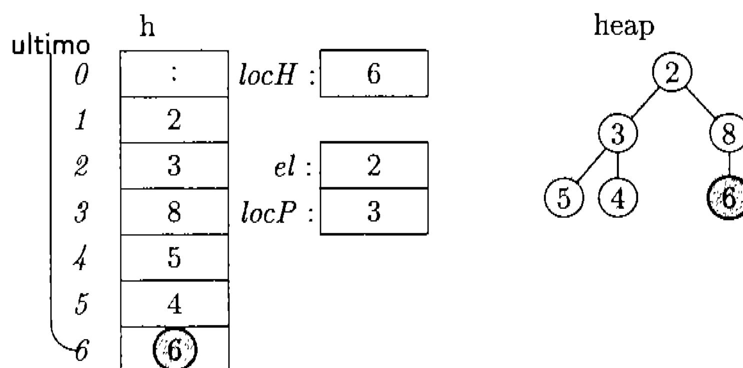
Como el nuevo nodo, cuya llave es 2, sigue violando la propiedad de heap con respecto a su padre, que tiene llave 3, debemos seguir flotándolo. Regresamos a la línea [40] después de intercambiar al padre con el hijo en las líneas [45–47], y como locP $\neq 0$, seguimos adelante. El estado de las estructuras de datos se puede ver en la Figura F.18.

Figura F.18: (9) Sigue la flotación del nuevo nodo hacia la raíz



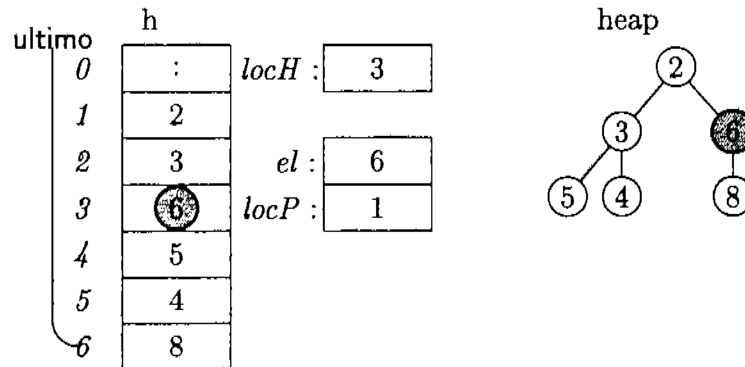
Como ya se cumple, en la línea [40], que locP $== 0$, pasamos a la línea [52] y termina la invocación de esta llamada. Nuevamente invocamos a inserta(6) y ejecutamos las líneas [38–39]. Como en la línea [40] locP $\neq 0$, entramos a la iteración. Como no se cumple en la línea [41] que la llave del hijo sea mayor o igual que la llave del padre, entramos a ejecutar el bloque que empieza en la línea [45]. El estado de las estructuras de datos se puede observar en la Figura F.19.

Figura F.19: (10) Inserción del nodo con llave=6



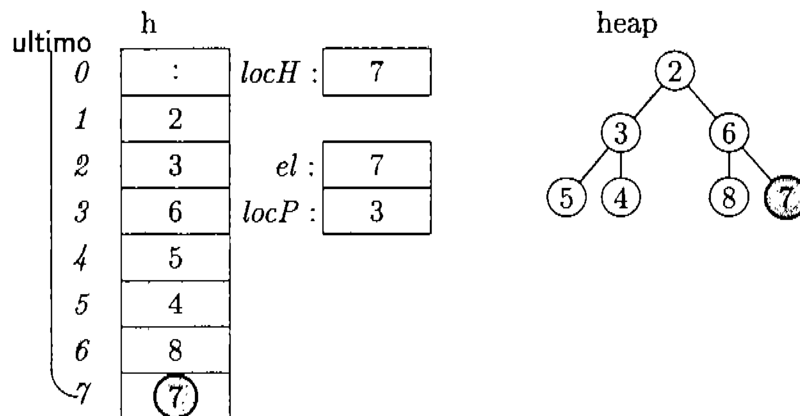
Intercambiamos a padre e hijo y regresamos a la línea [40]. Como se cumple locP $\neq 0$, seguimos adelante con las líneas [38–39]; como no se cumple en la línea [41] que la llave del hijo sea mayor o igual que la del padre, entramos a ejecutar el bloque que empieza en la línea [45], donde intercambiamos los valores y regresamos a la línea [40]. En este punto, la condición locP $\neq 0$ ya no se cumple, por lo que salimos de esta invocación por la línea [52]. El estado de las estructuras de datos se puede observar en la Figura F.20.

Figura F.20: (11) El nuevo nodo ya cumple la propiedad de heap



Volvemos a invocar inserta con el valor de 7 en la llave. Como la llave del hijo es mayor o igual que la del padre en la línea [41], se termina con esta invocación en la línea [43]. Las estructuras de datos quedan como se observa en la Figura F.21.

Figura F.21: (12) Inserción del nodo con llave=7



Terminamos de insertar a los nodos de la lista $(5, 3, 8, 4, 2, 6, 7)$ y nos queda un árbol parcialmente ordenado (el orden es parcial porque sólo se respeta entre el padre y cada uno de sus hijos, pero no podemos garantizar el orden entre hermanos o "primos"). La ventaja es que, al terminar cada inserción, tenemos al elemento con la llave menor en la raíz del heap.

Complejidad para el método inserta

Al observar la ejecución del algoritmo, para insertar un nodo debemos calcular cuántas veces regresa la ejecución a la línea [40], pues eso nos da el número de veces que se va a ejecutar el código correspondiente a acomodar a un nuevo nodo recién insertado en el heap:

1. Las líneas [24–39] se llevan $O(1)$.
2. El bloque de líneas [45–47] se lleva $O(1)$.
3. Veamos cuántas veces se ejecuta el ciclo de las líneas [40–49]:
   a) El primer valor de locP para el cual llegamos a la línea [40] es el que corresponde al número de nodos que se encuentran en el heap, o sea ultimo (línea [38]).
   b) Cada vez que regresamos a la línea [40], es con el valor de locP a la mitad de lo que tenía en la iteración anterior, ya que dividimos entre 2 el valor de ultimo al entrar por primera vez, y en cada iteración dividimos su valor entre 2 en la línea [47]. Por lo que vamos a regresar a la línea [40], en el peor caso cuando tengamos que "flotar" el valor que acabamos de insertar hasta la raíz del heap, $O(\log n)$.
   c) La ejecución del ciclo en las líneas [40–49] corresponde, en realidad, al proceso de "flotar" a un cierto nodo desde el lugar que ocupa hacia el que debe ocupar (***bubble up***).

   El ciclo se ejecuta, en el peor de los casos, $O(\log n)$, donde n es la posición que ocupa el nodo en el heap.

Complejidad para inserta: $\boldsymbol{O(\log n)}$

F.2.2. Obtener el nodo con el valor mínimo

Con esta estructura de datos, el proceso de obtener el nodo con el valor mínimo es de $O(1)$, pues todo lo que hay que hacer es regresar el nodo que se encuentre en la posición 1 del arreglo, o sea en la raíz del heap. Este proceso, obviamente, se lleva $O(1)$, y su código se muestra en el Listado F.6.

Listado F.6: Método que devuelve el mínimo del heap

```
54          /* Si el heap no está vacío, regresa al elemento que se      *
55           * encuentra al frente de la cola de prioridades.             */
56      public Object elige() {
57          if (ultimo <= 0) {
58              throw new EmptyCollectionException
59                  (¡¡¡¡"El heap está vacío!!!!");
60          }
61          return elementos[1].getInfo();
62      }
```

Complejidad para daMinimo(h): $\boldsymbol{O(1)}$

F.2.3. Reducción del valor de la llave de un nodo en el heap

Cuando lo que deseamos hacer es reducir el valor de la llave de algún nodo del heap, al ejecutar esta reducción podemos encontrarnos con que se pierde la propiedad de heap, pues

el valor resultante es menor que el valor de la llave de su padre. Si esto sucede, debemos flotar a este nodo hacia arriba en el árbol, hasta que o bien se recupere la propiedad de heap, o el nodo al que se le cambió el valor se coloque en la raíz.

Esta operación de flotado es exactamente la misma que ejecutamos al insertar un nodo nuevo, cuando nos dábamos cuenta de que el nodo no estaba en el lugar adecuado, de tal manera que no se violara la propiedad de heap. Por lo que si separamos del algoritmo de inserción la rutina que flota al nodo hacia arriba, el método que flota a un nodo hacia la raíz queda como se ve en el Listado F.7:

Listado F.7: Flotado de un nodo hacia la raíz del heap

```
63         /* Reacciona frente a la reducción de la llave de un       *
64          * elemento. Como la llave se redujo, la única posibilidad *
65          * para el elemento es que migre hacia el frente de la     *
66          * cola de prioridades.                                    */
67     protected void flotaLlave(int i) {
68         int actual = i;
69         int locPadre = (int) (actual / 2);
70         Object u = elementos[i].getInfo();
71         while (locPadre > 0) {
72             if (comp.less(u, elementos[locPadre].getInfo())) {
73                 intercambia(actual, locPadre);
74                 actual = locPadre;
75                 locPadre = (int) (locPadre / 2);
76             }
77             else { // Ya encontró su lugar
78                 return;
79             }
80         }
81     }
```

Con el método flota, el método que reduce una llave y reacomoda el heap queda como se ve en el Listado F.8.

Listado F.8: Reducción de llave y reacomodo en el heap

```
82         /* Reduce la llave y después flota al nodo hacia la raíz.    */
83     public void reduceLlave(Nodos v, int delta) {
84         elementos[v.getPos()].setLlave(elementos[v.getPos()].
85                                             getLlave() - delta);
86         flota(k);
87     }
```

Como ya analizamos detenidamente la complejidad del código que corresponde a flotar un nodo, sabemos que en el peor de los casos le va a llevar $O(\log n)$ para acomodar a un nodo en el lugar que le corresponde. La reducción de la llave lleva $O(1)$, por lo que la complejidad de esta rutina es, en total, la misma que para la que inserta un nodo nuevo, $O(\log n)$:

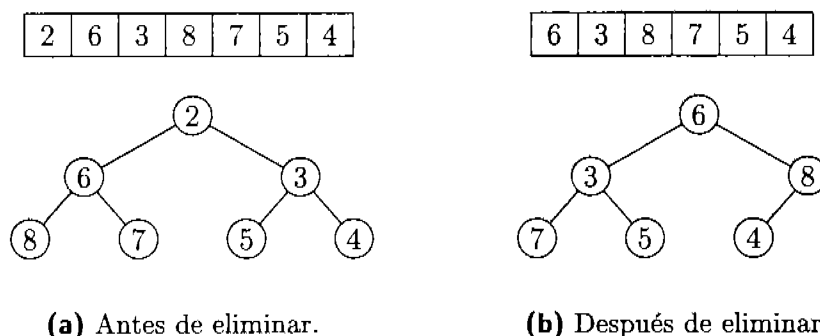
Complejidad para reduceLlave(Nodos v, int delta): $O(\log n)$

F.2.4. Eliminación de la raíz de un heap

El proceso de eliminar la raíz de un heap no es trivial. Lo primero que se nos ocurre es, simplemente, quitar al nodo que se encuentra en la posición 1 del arreglo, y tal vez recorrer a los $n-1$ nodos restantes a la izquierda, de tal manera que mantengamos la propiedad de que si hay $n-1$ nodos en el heap, éstos se encuentren en las posiciones $1 \ldots n-1$ del arreglo. Es obvio que esta opción es demasiado costosa y no funciona por las siguientes razones:

i. El recorrer a los nodos en el arreglo tiene un costo de $O(n)$, muy alto pues sólo representa una parte del proceso.
ii. Como el árbol estaba *parcialmente ordenado*, la raíz del subárbol izquierdo, que estaba en la posición 2 y lo recorrimos a la posición 1, no contiene forzosamente, al mínimo de todo el heap una vez quitada la raíz, sino únicamente un mínimo local relativo al subárbol con raíz en ese nodo – ver Figura F.22. Pero además, el nodo que estaba en la posición 3 era hermano del nodo en la posición 2, lo que no establece una relación de orden entre sus llaves. Pero ahora, como se recorrieron, tienen una relación de padre-hijo donde los valores sí deben estar ordenados. En pocas palabras, se destruye totalmente la estructura que tenía el árbol, por lo que tenemos que volver a construir el heap, lo que tiene un costo de $O(n \log n)$.
iii. Todo el procedimiento tendría un costo, entonces, de $O(n + n \log n)) = O(n \log n)$

Figura F.22: Eliminando simplísticamente al mínimo de un heap binario



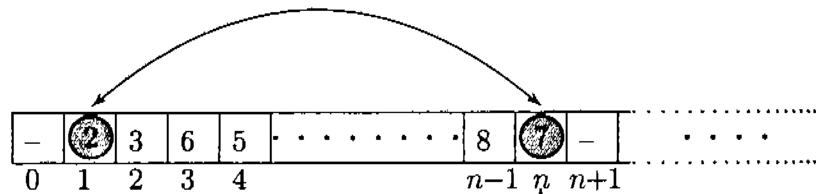
(a) Antes de eliminar. **(b)** Después de eliminar.

Tenemos una manera lo más eficiente posible de resolver este problema, aprovechando la implementación que se tiene del heap en un arreglo, donde una variable ultimo nos dice el número de nodos en el heap, y esos nodos se encuentran, como acabamos de mencionar, ocupando posiciones consecutivas en el arreglo de $1 \ldots$ ultimo. Lo que vamos a hacer es lo siguiente:

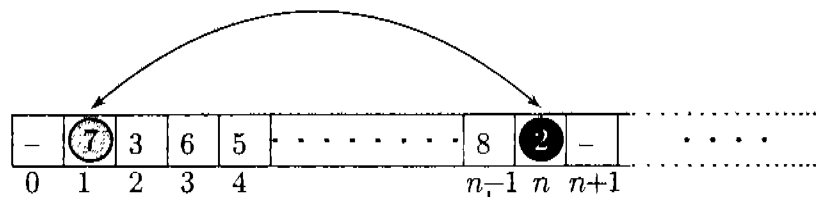
i. Intercambiamos al nodo que se encuentra en la raíz del árbol (que es a quien queremos eliminar, ya que corresponde al valor mínimo en el heap) con el nodo que se encuentre más a la derecha (el que esté en la posición ultimo), posición que eliminaremos del arreglo, como se muestra en la Figura F.23.

Figura F.23: Eliminación de la raíz de un heap

intercambia(1,n)



(a) Antes



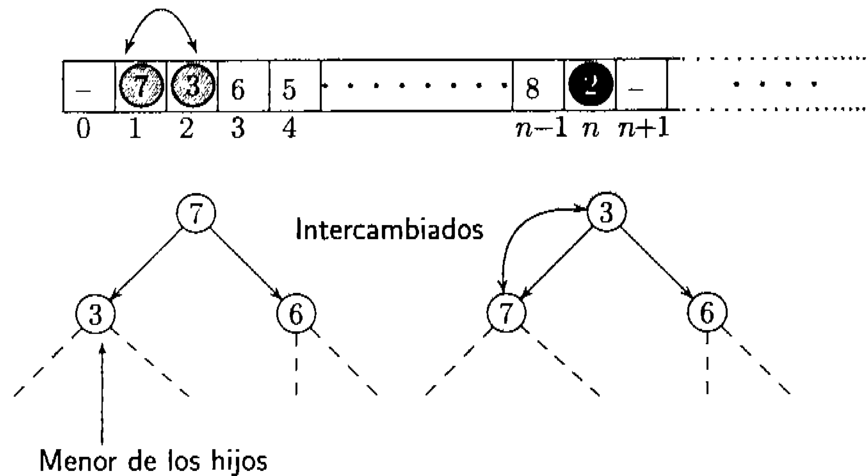
(b) Después

ii. Pero el arreglo (considerando únicamente a los nodos en las posiciones de la 1 a la ultimo−1) puede ya no presentar la propiedad de heap, como sucede en este caso (se puede apreciar en la Figura F.24(b)). De manera similar a como flotábamos un nodo desde las hojas hacia la raíz cuando lo insertábamos (o reducíamos su llave), calculando en cada momento la posición de su padre como locP $\leftarrow \lfloor i/2 \rfloor$, vamos a *sumergir*[3] a un nodo desde la raíz hacia las hojas, calculando la posición de sus hijos como $lH \leftarrow i * 2$. Lo que buscamos es que el nodo ocupe su lugar adecuado en el heap, que sólo puede ser hacia abajo. Este proceso se lleva a cabo, en una primera instancia, con el nodo en la

[3]En inglés, *bubble down*

raíz y sus dos hijos, intercambiando al nodo en la raíz con el que tiene el menor valor de entre sus dos hijos, como se puede ver en la Figura F.24.

Figura F.24: Reorganización de un heap binario



Cuando intercambiamos a un nodo padre con su hijo estamos "hundiendo" o "sumergiendo" a un nodo en el nivel inmediato inferior. Con este intercambio, logramos que el nodo en la raíz cumpla con tener una llave con valor menor o igual a las llaves de sus hijos. Hay que notar, sin embargo, que la propiedad de heap, hasta este momento, sólo se cumple entre la raíz y sus dos hijos. Como acabamos de intercambiar a dos nodos, debemos seguir validando la propiedad de heap hacia abajo (hacia las hojas) en el árbol.

iii. Este proceso de sumergir continúa en la posición en que quedó el nodo que en la iteración anterior correspondía al padre. Esto es, se considera el subárbol cuya raíz es el nodo que sumergimos, y se repite este proceso, hasta que la relación entre el nodo y sus hijos cumpla con la propiedad de heap, para todos los nodos en el árbol.

El código que sumerge a un nodo un nivel, intercambiándolo con su hijo, se encuentra en las líneas [90–113] del Listado F.9. En la línea [94] la raíz del subárbol está en la posición i. Si este nodo es hoja, las posiciones que corresponderían a sus hijos ($2*i$ y $2*i+1$) están más allá de la posición del último nodo, marcada por ultimo. En la línea [97] se verifica si los hijos del nodo existen – locH <= ultimo – y en la línea [98] que la raíz tenga dos descendientes directos, o sea, un nodo más a la derecha del hijo izquierdo (locH < ultimo), que correspondería al hijo derecho del nodo en la posición i.

Listado F.9: Método que sumerge a un nodo desde la posición i 1/2

```
88        /* Sumerge a un elemento en el heap a la posición que le    *
89         * corresponde para garantizar la propiedad de heap.         */
90    public void sumerge(int i) {
91        // heapifica desde la posición i hacia abajo.
92        if (ultimo <= 0)
93            return;
94        int locActual = i;
```

Listado F.9: Método que sumerge a un nodo desde la posición i 2/2

```
95          int locHijo = 2 * locActual;
96          Object v = elementos[i].getInfo();
97          while (locHijo <= ultimo) {
98              if (locHijo < ultimo)
99                  if (comp.less(elementos[locHijo + 1].getInfo(),
100                                 elementos[locHijo].getInfo()))
101                     locHijo++;
102             // locHijo se refiere al mayor de los hijos.
103             if (comp.less(elementos[locHijo].getInfo(), v)) {
104                 intercambia(locActual, locHijo);
105                 locActual = locHijo;
106                 locHijo = 2 * locActual;
107             } else { // El arreglo ya está heapificado.
108                 elementos[locActual] = new ElementoDeMiLista(v,
109                                                               locActual);
110                 break;
111             }
112         }
113     }
```

Una vez ubicados los hijos, se determina cuál es el que tiene la llave menor. Primero se supone que la menor llave se encuentra en el hijo izquierdo, asignando a locH la posición de este nodo ($2 * i$) en la línea [106] dentro del ciclo, y en la línea [95] la primera vez. Si el hijo derecho (posición $2 * i + 1 = \text{locH} + 1$) tiene una llave menor, entonces la variable locH tomará el valor de la posición del hijo derecho (línea [101]).

Se procede a verificar la condición de heap, aunque sólo es necesario comparar contra el hijo que tiene la menor llave (línea [103]). Si se viola la condición de heap, se sumerge a la raíz, intercambiándolo con su hijo menor, y se baja un nivel en el árbol para seguir verificando hacia abajo en el subárbol (líneas [103–106]).

Con el uso del método sumerge es muy fácil programar al método que elimina a un nodo arbitrario del heap, el que se encuentre en la posición i. Este método se encuentra en el Listado F.10.

Listado F.10: Para eliminar al elemento del heap que ocupa la posición i 1/2

```
114         /* Elimina al elemento del heap que ocupa la posición i,    *
115          * intercambiándolo con el último en el heap, y después     *
116          * heapificando nuevamente, pero un heap con un elemento    *
117          * menos.                                                   */
118     public void elimina(int i) {
119         if (ultimo <= 0)
120             throw new EmptyCollectionException
121                 ("Tratando de eliminar de un heap vacío.");
```

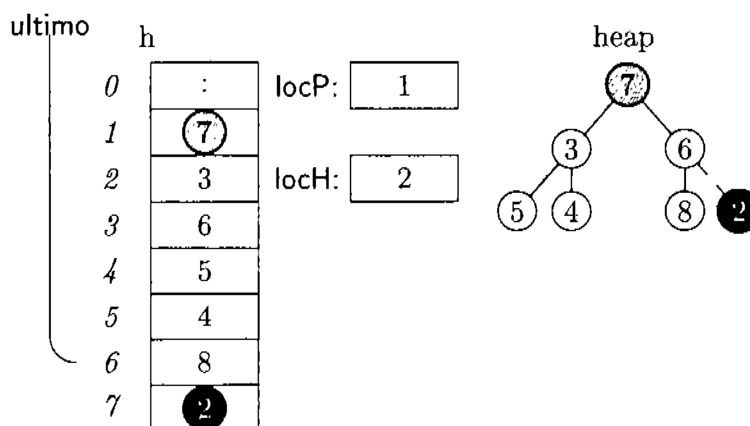
Listado F.10: Para eliminar al elemento del heap que ocupa la posición i 2/2

```
122            Object obj = elementos[i].getInfo();
123            if (i <= 0)
124                return;
125            elementos[i] = elementos[ultimo];
126            elementos[i].setPos(i);
127            contador--;
128            elementos[ultimo] = null;
129            ultimo--;
130            sumerge(i);
131        }
```

Finalmente, para eliminar al mínimo del heap, lo único que se tiene que hacer es invocar a elimina(1).

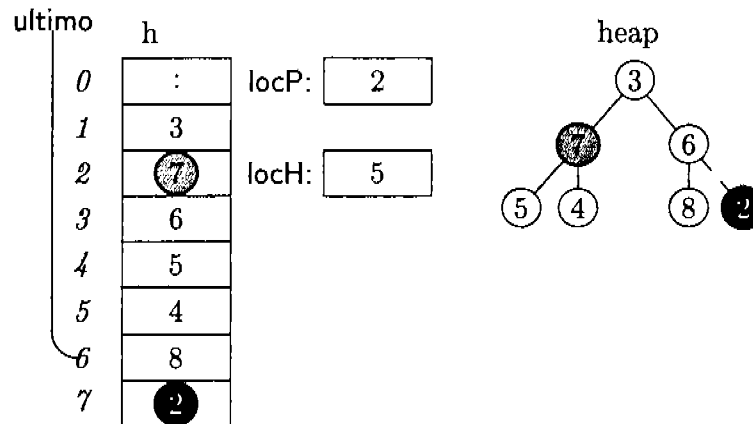
Veamos la ejecución de este algoritmo con el heap que armamos como ejemplo, eliminando al mínimo del heap.

Figura F.25: Se intercambia a la raíz con la última hoja



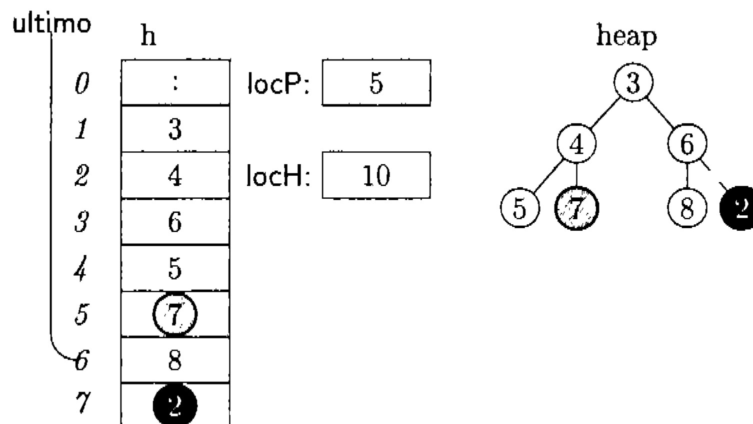
Determinamos que esta raíz tiene hijo izquierdo y derecho (líneas [97–98]) y que el menor nodo se encuentra en el hijo izquierdo (línea [98]). Como el nodo en la posición i es mayor que su hijo, debemos intercambiarlos (línea [104]) y volver a verificar con el hijo como raíz (líneas [105–106] y [112]).

Figura F.26: Se intercambia a la nueva raíz con su hijo izquierdo



Nuevamente, la raíz en locP = 2 cumple con tener ambos hijos, y el hijo que tiene el menor valor es el hijo derecho (locH = 5). Como la raíz no cumple con ser menor que ambos hijos (8) tiene que intercambiarse con el menor de sus hijos. Regresamos a verificar el subárbol con raíz en locP = 5 con locH = 10.

Figura F.27: Se intercambia al nodo en la posición 2 con su hijo derecho



Como al regresar a la línea [97] se cumple que la raíz con locP = 5 no tiene hijos (locH = 10 que es mayor que ultimo = 6), salimos de la iteración a la línea [112] y de esa invocación del método al llegar a la línea [113].

Complejidad de eliminaMin

Dado que para sumergir hacia abajo la nueva raíz, en el peor caso se recorre el árbol a profundidad, y la profundidad máxima para un árbol binario equilibrado con n nodos es de

$log_2 n$, la complejidad de esta rutina es $O(\log n)$ cada vez que es invocada.

$$\text{Complejidad para eliminaMin():} \qquad \boldsymbol{O(\log n)}.$$

F.2.5. Aplicaciones de heaps binarios: ordenamiento Heapsort

Un uso muy común de los heaps binarios es el que utiliza uno para ordenar a una lista de nodos con un valor como llave. Inicialmente se colocan todos los nodos en el arreglo y se procede a convertir al arreglo en un maxheap, colocando al nodo con el valor máximo en la raíz. Una vez hecho esto, se elimina a la raíz, de la manera que describimos anteriormente, colocándola en el último lugar del arreglo, y se procede nuevamente a recuperar la propiedad de heap. Se itera colocando al máximo de entre los que quedan en la raíz, y eliminándolo. Se termina con los nodos en la lista ordenados de menor a mayor de acuerdo al valor de su llave. La complejidad de este algoritmo es tan buena como se puede obtener. Cada eliminación del máximo toma $O(\log n)$ y como esto lo tenemos que hacer para los n nodos, tenemos una complejidad total de $O(n \log n)$.

F.3. Operaciones adicionales sobre heaps binarios

Se ocurren al menos otras tres operaciones que pudieran ser necesarias para manejar una cola de prioridades.

incrementaLlave(Nodos v, int delta): Supongamos que la prioridad de un nodo en un max-heap no decrece, sino que se incrementa cuando, por ejemplo, a un programa que es atendido por el sistema operativo y llegó al frente de la cola, el sistema operativo decide suspenderlo y volverlo a colocar en la cola, pero con la prioridad modificada para que se vuelva a formar. O bien, el usuario mismo le baja la prioridad a su proceso, ya que decide que no le urge tanto. En este caso lo que se ocurre es simplemente aumentar la llave y proceder a sumergirlo dentro de su subárbol. El proceso tiene una complejidad de $O(\log n)$ en el peor caso, ya que puede involucrar sumergir a un nodo desde la raíz hasta una hoja del árbol.

Como ya tenemos separado al código que sumerge a un nodo en el heap – utilizado para eliminar al nodo de la raíz del heap – todo lo que tenemos que hacer para incrementar la llave en un nodo es sumergirlo a partir de la posición donde se encuentra. El código se encuentra en el Listado F.11.

eliminaNodo(int k): Es posible que algún elemento en la cola de prioridades decida salirse de la misma. En este caso se debe eliminar al nodo del heap. Como en el caso de la eliminación de la raíz, y dado que el árbol es equilibrado, no podemos simplemente recorrer a los elementos. Tampoco se resuelve el problema intercambiándolo con el último descendiente del subárbol del que es raíz, ya que este último descendiente también ocasiona problemas al recorrerlo.

Listado F.11: Método que incrementa una llave

```
132         /* Incrementa la llave de un nodo y rearregla el heap    *
133          * para conservar la propiedad de heap.                   */
134     public void incrementaLlave(Nodos v, int delta)  {
135         elemento[v.getPos()].setLlave(elemento[v.getPos()].
136                                                getLlave()+delta);
137         sumerge(v);
138     }
```

Se sugiere colocar al nodo en la raíz, reduciendo su llave para que sea la menor y manteniendo en todo momento la propiedad de heap. Una vez que se encuentre en la raíz, invocar al método que elimina al mínimo. La complejidad de este método es $O(\log n)$.

Aunque no reducimos la complejidad, una manera más económica de eliminar un nodo arbitrario del heap es haciendo algo equivalente a cuando se elimina al mínimo, pero trabajando únicamente en el subárbol del que el nodo que se desea eliminar es raíz. Procedemos de la siguiente manera:

i. Se intercambia al nodo que se desea eliminar con el último del heap.

ii. Se sumerge al nuevo nodo en el subárbol en el que quedó.

Esto se consigue fácilmente utilizando el método elimina que dimos en el Listado F.10. Podemos ver los estados por los que pasa un heap al eliminar al nodo en la posición $k = 2$ en las Figuras F.28 a F.31.

Figura F.28: Incrementando el valor de una llave

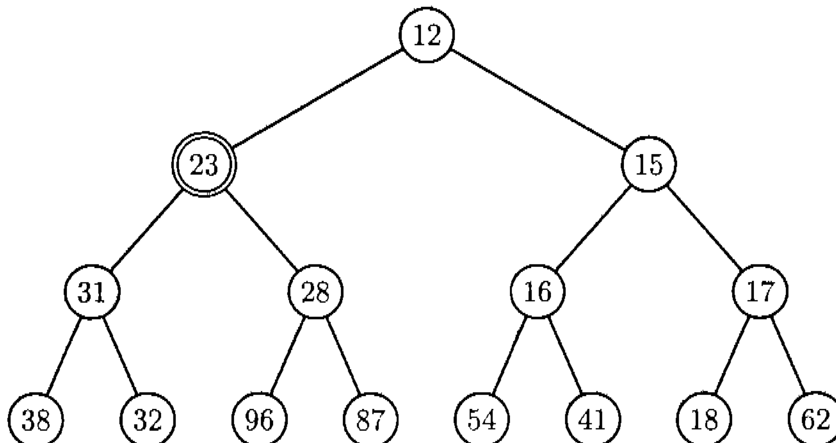


Figura F.29: Se intercambia con el último y se poda al último

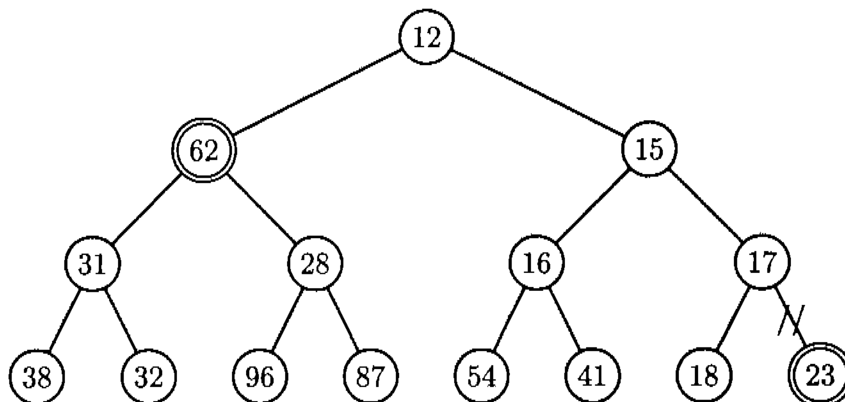


Figura F.30: Se compara con sus hijos y se sumerge un nivel

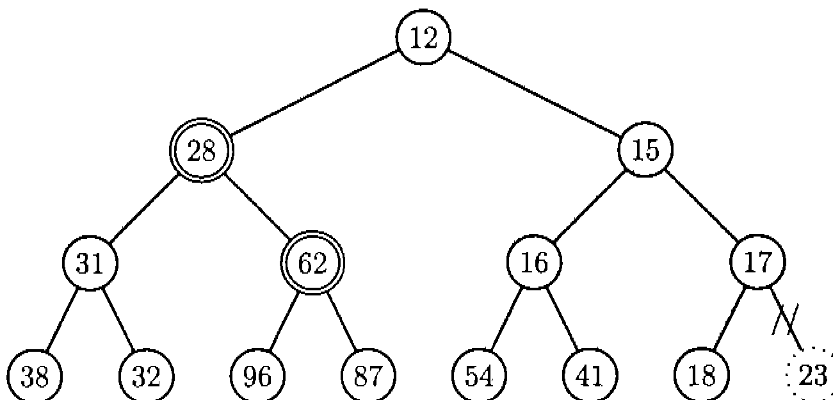
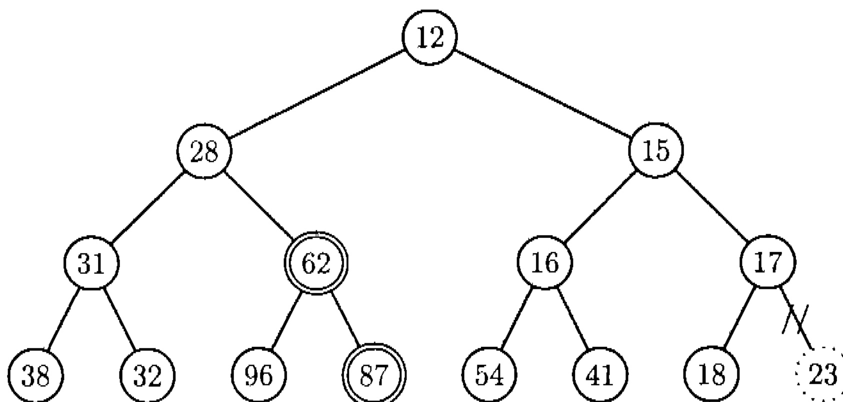


Figura F.31: Se compara con sus hijos y ya no se sumerge

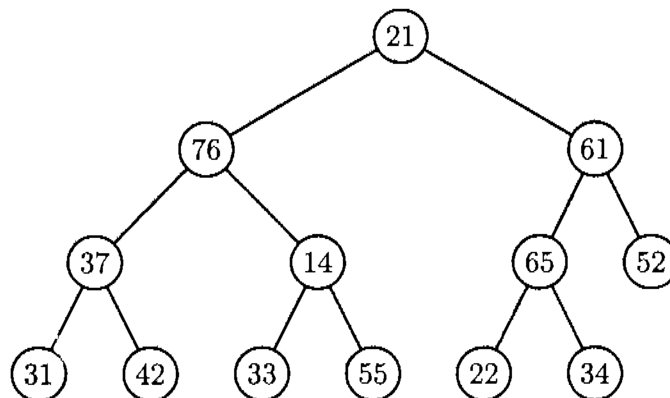


HeapBinario(ElementosDeMiLista[] A): Supongamos que queremos construir un heap con n elementos, los cuáles conocemos desde el inicio del proceso. La manera eficiente de hacerlo es como describimos en la página 449. El método en el contexto de la clase que estamos construyendo se encuentra en el Listado F.12.

Listado F.12: Método de complejidad lineal para construir un heap, dado un arreglo con las llaves

```
139         /* Heapifica al heap completo, sin suponer un orden       *
140          * parcial previo.                                        */
141     public void heapifica() {
142          /* Posición del padre de la última hoja.                 */
143         int locPadre = ultimo / 2;
144          /* Hijo izquierdo del ''último'' padre.                  */
145         int locHijo = locPadre * 2;
146         while (locPadre >= 1) {
147             if (locHijo + 1 <= ultimo
148                 && comp.less(elementos[locHijo + 1].getInfo(),
149                              elementos[locHijo].getInfo())) {
150                 locHijo++;
151             }
152             if (comp.gtr(elementos[locPadre].getInfo(),
153                          elementos[locHijo].getInfo()))     {
154                 intercambia(locHijo, locPadre);
155                 sumerge(locHijo);
156             }
157             locPadre--;
158             locHijo = locPadre * 2;
159         }
160     }
```

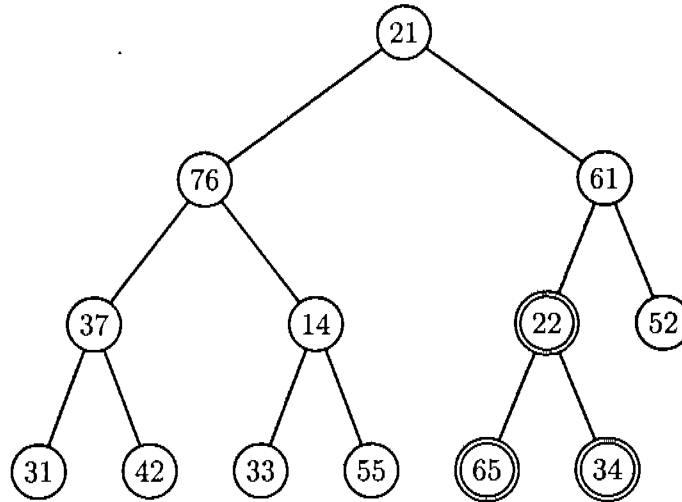
Figura F.32: Construcción de un heap binario a partir de un arreglo



A:

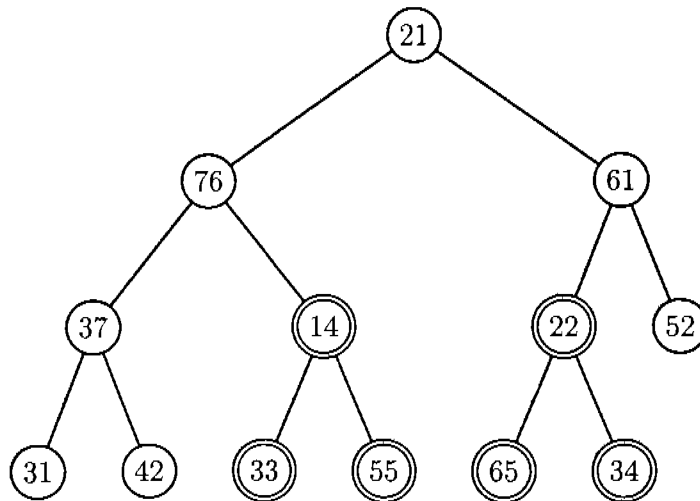
	21	76	61	37	14	65	52	31	42	33	55	22	34

Figura F.33: Propiedad de heap en el subárbol con raíz en $i = \lfloor 13/2 \rfloor = 6$



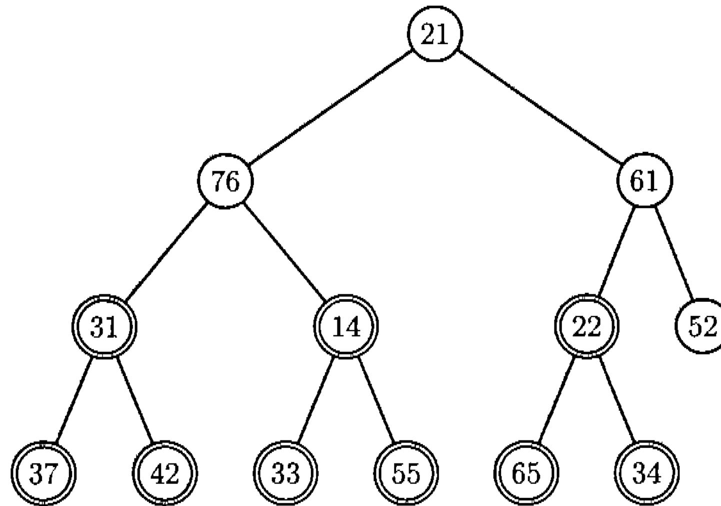
h:		21	76	61	37	14	22	52	31	42	33	55	65	34

Figura F.34: Propiedad de heap en el subárbol con raíz en $i = 5$



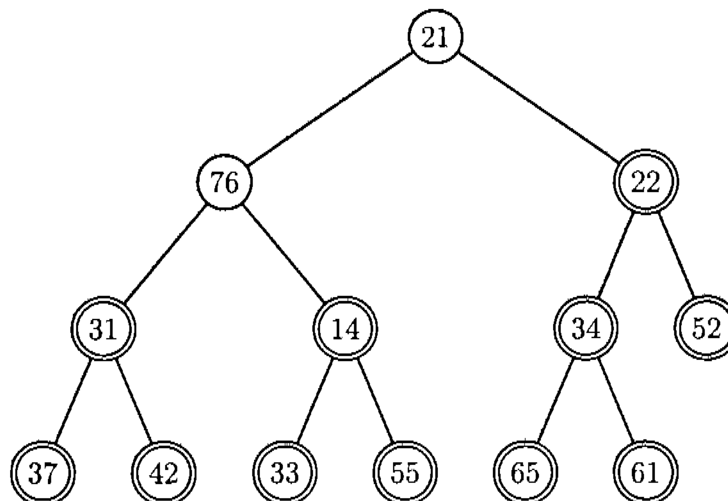
h:		21	76	61	37	14	22	52	31	42	33	55	65	34

Figura F.35: Propiedad de heap en el subárbol con raíz en $i = 4$



h:		21	76	61	31	14	22	52	37	42	33	55	65	34

Figura F.36: Propiedad de heap en el subárbol con raíz en $i = 3$



h:		21	76	22	31	14	34	52	37	42	33	55	65	61

Figura F.37: Propiedad de heap en el subárbol con raíz en $i = 2$

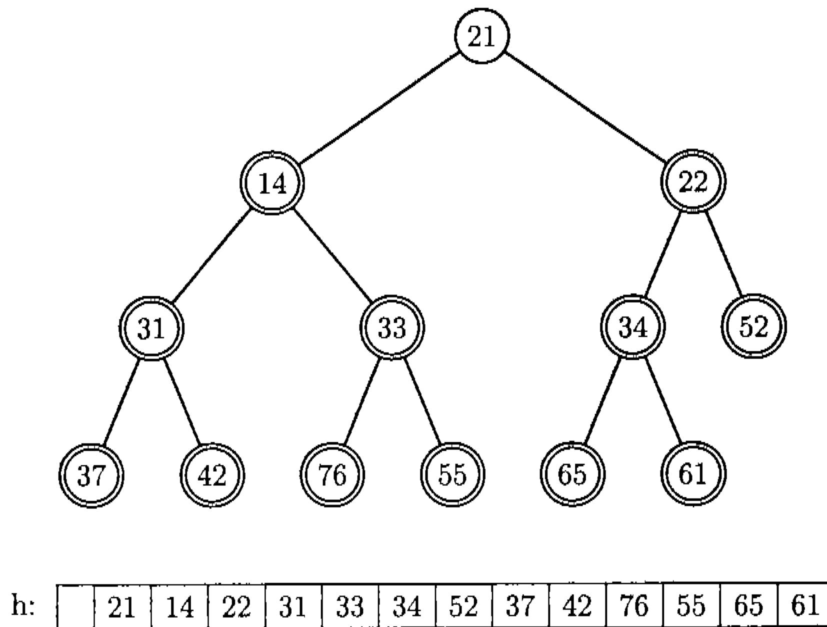
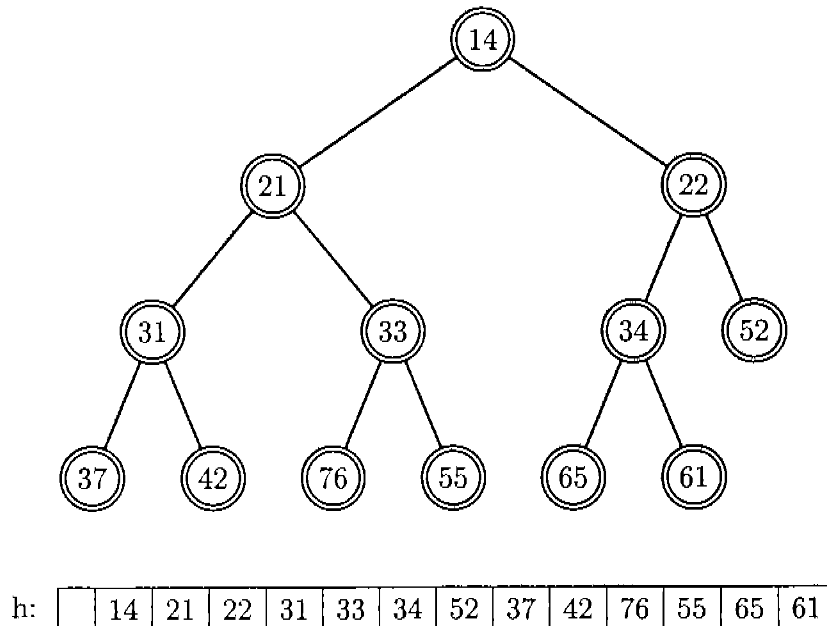


Figura F.38: Propiedad de heap en el subárbol con raíz en $i = 1$



Como se puede ver en la Figura F.38, se estableció ya la propiedad de heap en todo el árbol.

Faltaría por demostrar que la complejidad de este algoritmo es $O(n)$. Para ello, primero definiremos la *altura* de un nodo:

Definición F.5 (Altura de un nodo) Decimos que un nodo en un árbol está a altura h si la máxima distancia en el árbol de ese nodo a una hoja descendiente del nodo es h.

De la definición anterior, la altura de la raíz del árbol es, precisamente, la profundidad del árbol, mientras que la altura de una hoja es 0.

El peor caso para la ejecución de este método es que cada vez que se sumerge a un nodo se le tenga que sumergir hasta que alcance una hoja. Entonces, el peor caso para toda la ejecución estará acotado por la suma de las alturas de los nodos del árbol, lo que demostramos en el Lema F.2.

Lema F.2 *Para un árbol binario completo que contiene $2^{h+1}-1$ nodos, la suma de todas las alturas de los nodos es $2^{h+1}-1-(h+1)$.*

Demostración:

Si el árbol es completo y contiene $2^{h+1}-1$ nodos, entonces la profundidad del árbol es exactamente h. De esto, tenemos:

1 nodo de altura h, la raíz	h,
2 nodos de altura $h-1$	$2(h-1)$,
4 nodos de altura $h-2$	$4(h-2)$,
$\vdots$	$\vdots$
2^k nodos de altura $h-k$	$2^k(h-k)$.

La suma de estas alturas es, entonces

$$S=\sum_{i=0}^{h}2^i(h-i)=h+2(h-1)+4(h-2)+8(h-3)+\cdots+2^{h-1}(1).$$

Multiplicamos por 2 ambos lados de la igualdad y obtenemos

$$2S=2h+4(h-1)+8(h-2)+16(h-3)+\cdots+2^h(1).$$

Y restamos estas dos expresiones, y vemos que muchos términos "casi" se cancelan. Por ejemplo, $2h-2(h-1)=2$; $4(h-1)-4(h-2)=8$. Por lo que restando aquellos términos que tienen el mismo coeficiente en ambas expresiones prácticamente eliminamos los términos en h, quedándonos de la siguiente manera:

$$S=-h+2+4+8+\ldots+2^{h-1}+2^h=(2^{h+1}-1)-(h+1).$$

❏

Si un árbol tiene n nodos, su profundidad será $\log n$, por lo que podemos sustituir $\log n$ en lugar de h y tenemos

$$
\begin{aligned}
S &= (2^{\log n+1} - 1) + (\log n + 1) \\
&= 2^{\log n} \cdot 2 - 1) + (\log n + 1) \\
&= 2n - 1 + \log n + 1.
\end{aligned}
$$

Y como el término en n domina, podemos decir que el método que dimos para construir un heap binario a partir de un arreglo tiene complejidad $O(n)$.

Pudiéramos dar una cota superior más ajustada, pero por el momento esto no es necesario.

F.4. Árboles binomiales

Los árboles binomiales pertenecen a una categoría de estructuras de datos, conocidas como *heaps intercalables*, a la cual pertenecen los heaps binarios.

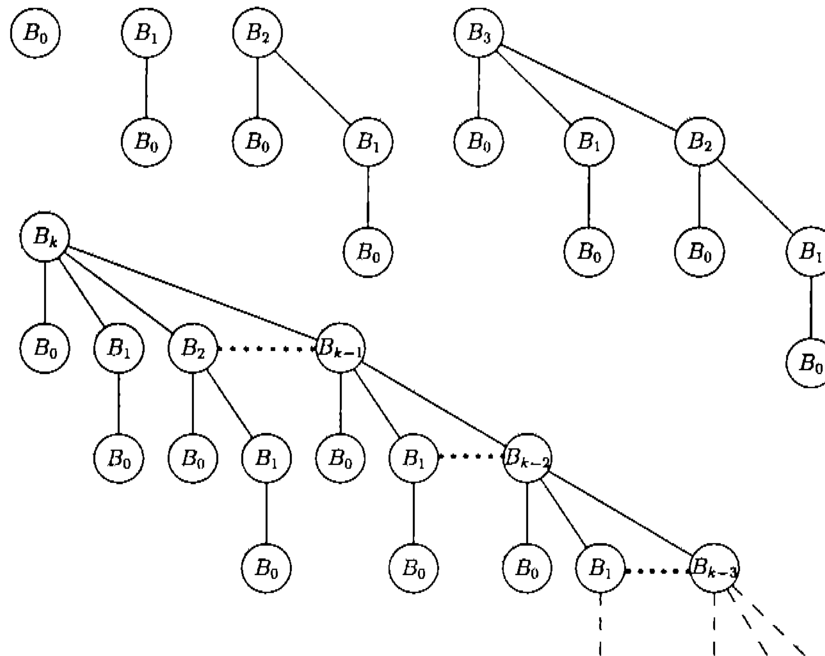
En muchos de los algoritmos que descansan en el manejo de colas de prioridades, el costo del algoritmo está dado por el costo de las operaciones de **reduceLlave** y **elige** – este último que regresa la llave menor en el heap – que se tienen que ejecutar cada vez que se elige a algún elemento de la estructura y se le "despacha", y cada vez que cambia el valor de alguna de las llaves. Estas dos operaciones, la de obtener el mínimo y la de reducir el valor de alguna de las llaves en la estructura, definen para muchos problemas el costo de resolverlos.

En la sección anterior vimos los heaps binarios, que permiten que el costo de estas operaciones se reduzca. Pero si queremos una operación que combine dos árboles binarios en uno (dos colas de prioridades en una sola), el costo de hacer esto en un heap binario es de $O(n)$, si se procede a concatenar los dos arreglos que contienen a los correspondientes heaps binarios y a continuación se ejecuta flotaLlave para cada una de las llaves del segundo arreglo. Esta es una operación muy común en una cola de prioridades, por lo que su costo puede resultar muy relevante. Estos costos se pueden reducir mediante dos mecanismos: el primero de ellos consiste en repartir a los nodos del heap en varios árboles, de cierta manera ajenos, de tal manera que las operaciones sobre la estructura no involucren más que a un subconjunto con un número limitado de nodos, como sucede en los heaps binomiales y de Fibonacci. El segundo mecanismo que ya mencionamos es el de llevar a cabo un análisis de costos amortizados, para conseguir una cota superior más ajustada a los costos reales. Empezaremos por trabajar con *heaps binomiales*.

Definición F.6 (Árbol binomial) Un *árbol binomial* es un árbol parcialmente ordenado – ver Definición F.1 – donde cada nodo del árbol, como en los heaps binarios, contiene la información de un registro y una *llave* que es el valor que define el orden parcial de los nodos en el heap. Se define, inductivamente de la siguiente manera:

– B_0 es el árbol que consiste de un único nodo.
– B_1 es un B_0 del que cuelga otro B_0.
⋮
– B_k es un B_{k-1} al que se le agrega como hijo en el extremo derecho a otro B_{k-1}.

Figura F.39: Árboles binomiales



Como se puede observar de la Figura F.39, cada árbol binomial B_k se construye a partir de dos árboles binomiales B_{k-1}. Supongamos que tenemos dos árboles binomiales B_{k-1} y B'_{k-1}. Tomamos el árbol con valor menor en la raíz y a ese le colgamos, agregando una arista en el extremo derecho, el otro árbol. A este proceso se le llama de *ligado* de dos árboles. Nótese que únicamente se pueden ligar dos árboles del mismo índice.

Pasamos a demostrar algunas propiedades de los árboles binomiales relacionadas con el índice del árbol.

Lema F.3 *La raíz de un árbol de índice k tiene exactamente k hijos (descendientes directos). Al número de hijos de un nodo en la raíz le llamamos el* rango *del nodo. Por lo tanto,* $grado(B_k.raiz) = k$.

Demostración:

Por inducción en el índice del árbol.

Base: B_0 tiene 0 hijos; B_1 tiene 1 hijo.

Inducción: Supongamos cierto que para $j < k$, B_j tiene j hijos y demostremos para B_k. Observemos:

i. B_k se construye colgando de un B_{k-1} a otro B_{k-1} como hijo en el extremo derecho.
ii. Por la hipótesis de inducción, B_{k-1} tiene exactamente $k-1$ hijos, más 1 que le estamos colgando, que es la raíz de otro B_{k-1}.

$\therefore$ B_k tiene k hijos (los subárboles con índices del 0 a $k-1$). ❑

Lema F.4 *El árbol B_k tiene exactamente 2^k nodos.*

Demostración:

Esto lo podemos demostrar por inducción sobre k.

Base: Para $k = 0$, B_0 tiene 1 nodo, y $2^0 = 1$.

Inducción: Supongamos cierto para k que el número de nodos en B_k es 2^k y veamos qué pasa con B_{k+1}.

B_{k+1} está compuesto por los árboles $B_0, B_1, \ldots, B_k$, que por la hipótesis de inducción tienen, respectivamente $2^0 = 1, 2^1 = 2, \ldots, 2^k$ nodos, por lo que el número de nodos en B_{k+1} está dado por

$$\sum_{i=0}^{k} 2^i = 2^{i+1} - 1.$$

Pero esta es la suma de los nodos en los subárboles que cuelgan de B_{k+1}, por lo que nos falta sumarle 1 que corresponde a la raíz del árbol. De esto, el número de nodos en B_{k+1} es $2^{k+1} - 1 + 1 = 2^{k+1}$. ❏

Veamos ahora la manera de almacenar n nodos con árboles binomiales. Es importante utilizar únicamente árboles binomiales completos, de tal manera que el acceso de un padre a sus hijos, o de un hijo a su padre sea mediante una función sobre la posición del nodo en el árbol. Podemos pensar que cada árbol binomial nos permite almacenar r nodos, donde $r = 2^k$, para alguna k. Esto nos sugiere la representación binaria (potencias de 2). Dado un entero positivo $n > 0$, n tiene una representación única en binario, por lo que si utilizamos árboles binomiales con $r > 0$ nodos, $r = 2^i$ para alguna i, a n la podemos representar en binario; numeramos las posiciones binarias con la potencia de 2 con la que contribuyen, y a cada posición binaria en la que aparezca un 1 le asociamos un árbol binomial con el índice de la posición. Veamos algunos ejemplos en la tabla que sigue.

n	binario	potencias de 2	árboles binomiales
5	101	$2^2 + 2^0 = 4 + 1$	B_2, B_0
32	100000	$2^5 = 32$	B_5
121	1111011	$2^6 + 2^5 + 2^4 + 2^3 + 2^1 + 2^0$ $= 64 + 32 + 16 + 8 + 2 + 1$	$B_0, B_1, B_3, B_4, B_5, B_6$

Dado esto, cada vez que queramos acomodar a n nodos usando árboles binomiales, podemos repartirlos en B_k's, donde cada k aparece a lo más una vez y va a contener exactamente a 2^k de los nodos.

Esta última observación es la que nos da la capacidad de, en un conjunto de n nodos, procesar únicamente a un subconjunto de ellos. Más adelante veremos que la cardinalidad de estos subconjuntos está dada por

$$\text{máx}\, k \;=\; \lfloor \log_2 n \rfloor + 1 \;=\; O(\log n).$$

F.4.1. Propiedades

La coincidencia entre el número de nodos que puede almacenar un árbol binomial y la representación binaria de un entero $n > 0$ sugiere varias propiedades de estos árboles, que nos van a garantizar cotas para la complejidad de las operaciones que deseemos realizar sobre ellos. Presentamos a continuación algunas de ellas.

Lema F.5 *$altura(B_k) = k$, donde la altura se refiere a la distancia de la raíz a la hoja más alejada de ella – Definición F.5.*

Demostración:

La demostración será por inducción sobre k, el índice del árbol.

Base: Para $k = 0$, B_0 tiene únicamente un nodo, la raíz, por lo que la distancia desde la raíz a la raíz es 0.

$\therefore\ altura(B_0) = 0.$

Inducción: Supongamos que $altura(B_j) = j, 0 \leq j < k$, y veamos qué pasa con B_k.

Por el Lema F.3, B_k tiene exactamente k hijos, los subárboles $B_0,\ldots,B_{k-1}$, de los cuales, por la hipótesis de inducción, el de mayor altura es B_{k-1}, cuya altura es $k-1$. Esto quiere decir que la distancia desde la raíz de B_{k-1} a la hoja más distante es $k-1$. La distancia desde la raíz de B_k a esa misma hoja, que es la más distante de la raíz de B_k, es 1 más, pues primero tiene que bajar a la raíz de B_{k-1} y desde allí llegar a esa hoja.

$\therefore\ altura(B_k) = altura(B_{k-1} + 1 = k - 1 + 1 = k.$ ❑

Lema F.6 *En cada árbol B_k hay un único vértice de $grado(v) = k$, y ese vértice es la raíz.*

Demostración:

Como cada B_k está formado con árboles $B_0,\ldots,B_{k-1}$ que cuelgan de él, y como por el Lema F.2 cada B_k tiene exactamente k hijos, fuera de la raíz de B_k, cualquier otro vértice es la raíz de un B_j, $j < k$, de donde no podrá tener más de $j < k$ hijos. ❑

Lema F.7 *Hay exactamente $\binom{k}{i}$ nodos a profundidad i del árbol, donde $\binom{k}{i}$ representa al coeficiente binomial (de acá el nombre de los árboles), para $k = 0, 1, \ldots, i$. El coeficiente binomial está definido de la siguiente manera:*

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Demostración:

Estableceremos esta propiedad por inducción sobre i, la profundidad del árbol B_k. Sea $D(k, i)$ el número de nodos en el nivel i del árbol binomial B_k, y queremos demostrar que

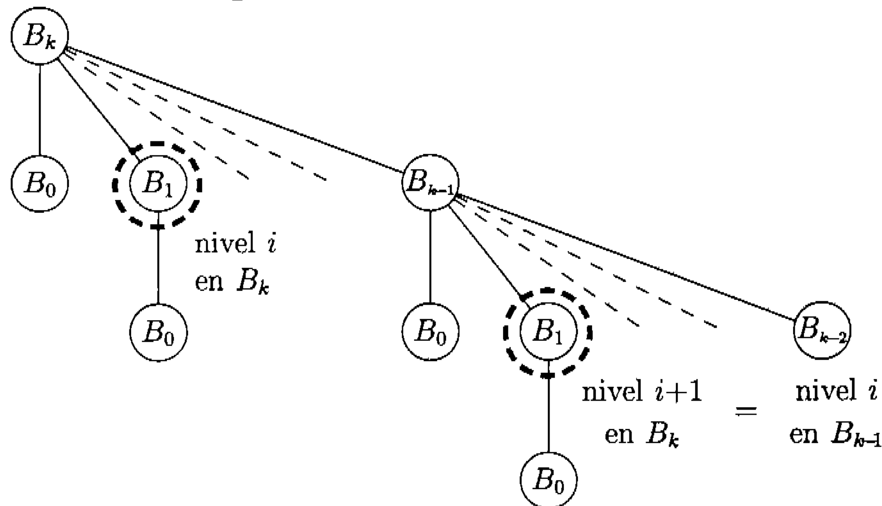
$$D(k,i) = \binom{k}{i}.$$

Base: Para $i = 0$. En el nivel 0 sólo se encuentra la raíz del árbol:

$$D(k,0) = 1 = \binom{k}{0} = \binom{k}{k} = 1.$$

<u>Inducción:</u> Supongamos cierto para valores menores que k y valores menores que i, y veamos que pasa con el nivel i en el árbol B_k. Recordemos que B_k se construye ligando dos copias de B_{k-1}. Por la forma que toma el ligado, un nodo con etiqueta j – que corresponde a la raíz del subárbol B_j – que se encontraba en el nivel i en B_{k-1} aparece como etiqueta de dos nodos distintos en B_k, una en el nivel i y otra en el nivel i+1 – nivel i del árbol B_{k-1} – como se puede observar en la Figura F.28. Esto se debe a que la raíz del árbol B_{k-1} que se colgó en el extremo derecho del otro B_{k-1} se encuentra desplazada un nivel de profundidad, y lo mismo sucede con todos sus niveles.

Figura F.40: Niveles en árboles ligados



En otras palabras, el número de nodos en el nivel i de B_k es el número de nodos en el nivel i de B_{k-1} más los nodos en el nivel $i-1$ de B_{k-1}:

$$\begin{aligned} D(k,i) = &\ D(k-1,i) + D(k-1,i-1) \\ = &\ \binom{k-1}{i} + \binom{k-1}{i-1} \\ = &\ \binom{k}{i}. \end{aligned}$$

La tercera igualdad se puede comprobar desarrollando los términos que corresponden a los coeficientes binomiales. ❑

Terminamos esta sección insistiendo sobre el hecho de que cada árbol binomial contiene *exactamente* un número de nodos que corresponde a una potencia de 2. Si queremos almacenar n nodos, donde n no corresponda a una potencia de 2, tendremos que utilizar más de un árbol binomial, ya que queremos mantener a cada árbol completo.

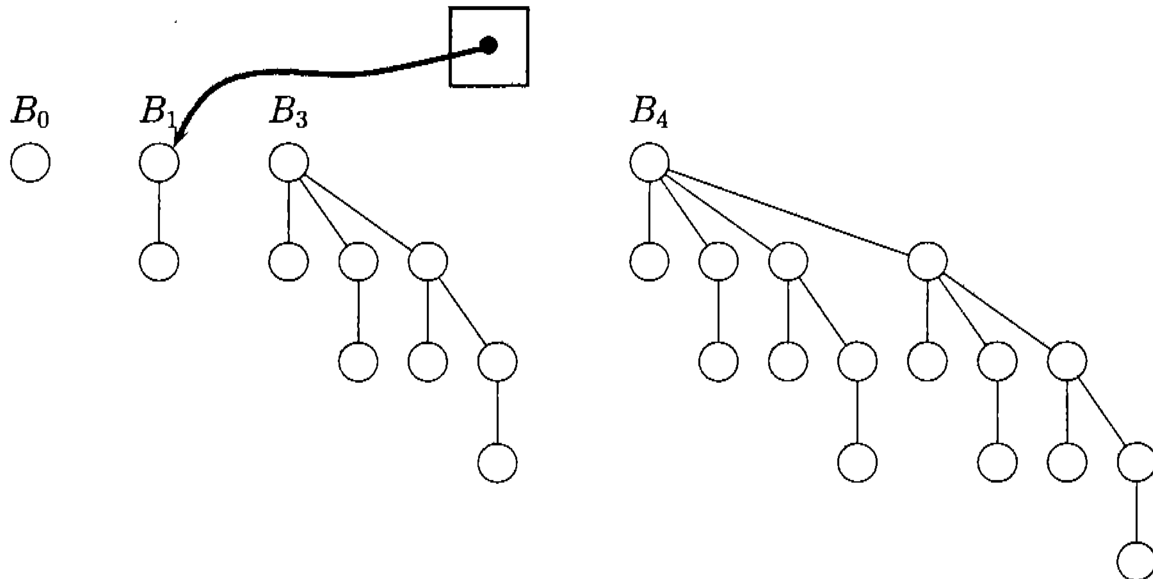
F.5. Heaps binomiales

Pasemos ahora a definir lo que es un *heap binomial*:

Definición F.7 (Heap binomial) Un *heap binomial* es una colección de *árboles binomiales* parcialmente ordenados, con un apuntador a la raíz del árbol binomial que en ese momento contiene al mínimo (está en la raíz de ese árbol binomial). El número total de nodos en el heap binomial es n, repartidos los nodos de la manera como describimos arriba.

En el transcurso de un proceso con un heap binomial puede suceder que tengamos, en un momento dado, más de un árbol binomial con índice k. Aún cuando esto no es lo deseable, nos va a permitir obtener mejores tiempos en algunos de los algoritmos. En ocasiones vamos a requerir adicionalmente la propiedad de que para cada k, en el heap haya a lo más un árbol binomial B_k. Veamos los ejemplos en las figuras F.41 y F.42. Debemos recordar que los árboles binomiales que componen un heap binomial son árboles parcialmente ordenados, por lo que la raíz de cada uno de los árboles contiene al nodo menor en ese árbol. El apuntador min apunta al árbol cuya raíz contiene al menor valor en el heap binomial.

Figura F.41: Heap binomial para acomodar a $27 = 2^4 + 2^3 + 2^1 + 2^0$ nodos



Lema F.8 *En un heap binomial con n nodos a lo más hay $\log_2 n$ árboles.*

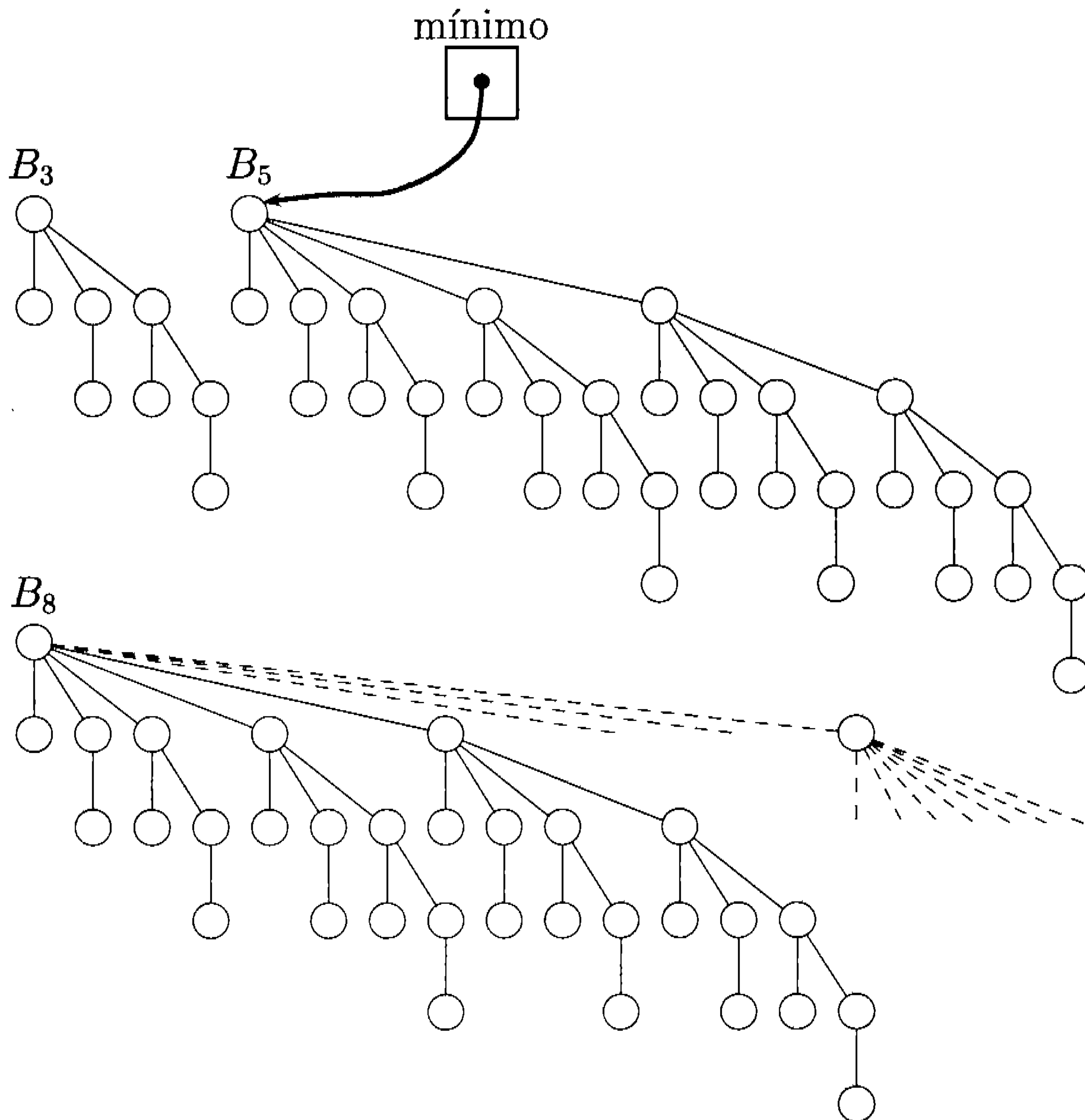
Demostración:

Vimos ya que en un heap binomial vamos a acomodar a los n nodos en árboles binomiales B_i, donde en cada B_i colocaremos exactamente 2^i nodos, similarmente a como representamos a un número n mediante la suma de potencias de 2, donde cada potencia aparece una única vez. Si interpretamos a la representación binaria de un entero de la siguiente manera:

- Si la posición i tiene un 1, el heap contiene exactamente a un B_i.
- Si la posición i tiene un 0, el heap binomial **no** tiene a B_i.

Requerimos de $\lfloor \log_2 n \rfloor + 1$ posiciones binarias para representar al entero n, por lo que un heap binomial con n nodos va a tener a lo más $\log_2 n$ árboles binomiales. ❏

Figura F.42: Heap binomial para acomodar a $296 = 2^3 + 2^5 + 2^8$ nodos



Veamos la interfaz para la clase HeapBinomial, UnHeapBinomial, en el Listado F.13, que supone la definición de una clase ArbolBinomial. Postergaremos por el momento la definición de esta última clase, suponiendo por lo pronto que los métodos invocados estarán definidos acorde.

Listado F.13: Interfaz para un heap binomial 1/2

```
1 public interface UnHeapBinomial {
2         /* Regresa la referencia al árbol binomial que tiene al    *
3          * mínimo en la raíz.                                       */
4     public ArbolBinomial daMinimo();
```

Listado F.13: Interfaz para un heap binomial 2/2

```
1        /*  Inserta al nodo con llave i en el heap binomial.        */
2     public void inserta(Nodos v);
3        /* Da el valor que corresponde al mínimo en el heap          *
4         * binomial.                                                  */
5     public Nodos daMinimo();
6        /* Elimina al vértice que contiene al mínimo en el heap.    */
7     public void eliminaMin();
8        /* Combina dos heaps binomiales en uno solo. Si heap          *
9         * contiene n nodos y otroHeap contiene m nodos, se           *
10        * debe construir un nuevo heap para n+m nodos y              *
11        * dejarlo en el viejo heap.                                  */
12    public HeapBinomial combina(HeapBinomial otroHeap);
```

Presentamos en el Listado F.14 las estructuras de datos necesarias para un heap binomial.

Listado F.14: Definición de la estructura de datos para un Heap Binomial y constructores de la clase

```
1  public class HeapBinomial implements UnHeapBinomial  {
2         /* Referencias a los árboles binomiales.                   */
3      protected Vector heap = null;
4         /* Posiciones disponibles.                                  */
5      int nMax = 0;
6         /* Máximo índice de los árboles.                            */
7      int n = 0;
8          /* Índice del árbol que contiene al mínimo.                */
9      ArbolBinomial minimo = null;
10         /* Constructor de un heap binomial con k referencias       *
11          * nulas.                                                   */
12     public HeapBinomial(int k)  {
13         heap = new Vector(k);
14     }
15           /*  Construye un heap binomial con B0 únicamente.        */
16     public HeapBinomial(Nodos v)  {
17           heap = new Vector(1);
18           heap[0] = new ArbolBinomial(1, v);
19           minimo = heap[0];
20     }
```

Vamos a codificar dos constructores para HeapBinomial, uno que construya un vector con k referencias a ejemplares de ArbolBinomial, y el segundo constructor para que construya un heap binomial que contenga únicamente al ArbolBinomial B_0, pasándole al nodo que va a constituir a ese árbol. En el Listado F.14 está el listado de los atributos que requerimos para la clase HeapBinomial, y el código correspondiente a sus constructores.

Veamos la manera en que se implementan las operaciones para insertar, encontrar y obtener el mínimo en el Listado F.15. Como se puede ver, la operación de insertar se remite

a la de combinar dos heaps binomiales, uno de los cuáles es B_0 y únicamente contiene al nodo que se desea insertar. La operación que encuentra el mínimo en el heap supone que cada uno de los árboles binomiales presenta la propiedad de heap, por lo que únicamente consulta la raíz de los árboles binomiales presentes en el heap binomial. Para obtener el nodo que contiene al valor mínimo únicamente seguimos la referencia a minimo.

Listado F.15: Código para insertar, obtener el mínimo y borrar el mínimo

```
1        /* Inserta al nodo v en este heap binomial.                    */
2      public void inserta(Nodos v)  {
3          HeapBinomial nuevoHeap = new HeapBinomial(v);
4          combina(nuevoHeap);
5      }
6          /* Da el valor que corresponde al mínimo en el heap            *
7           * binomial.                                                 */
8      public Nodos getMinimo()  {
9            return minimo.raiz();
10     }
11         /* Regresa la referencia al árbol binomial que contiene        *
12          * al mínimo, suponiendo que minimo no está actualizado.       */
13     public ArbolBinomial daMinimo()   {
14         if (heap == null)  {
15             throw new EmptyCollectionException(
16 ¡                 "El heap está vacío!");
17         }
18         ArbolBinomial minArb = heap[0];
19         int minValor = heap[0].getLlave();
20         for (int i = 1; i < heap.size(); i++)  {
21             if (heap[i] != null)  {
22                 if (minValor < heap[i].getLlave())  {
23                     minValor = heap[i].getLlave();
24                     minArb = heap[i];
25                 }
26             }
27         }
28         return minArb;
29     }
```

El método más complicado hasta el momento es el que elimina la raíz de uno de los árboles de un heap binomial. Como cada árbol binomial debe estar completo, si se le quita un nodo, el árbol no puede seguir existiendo tal cual. Nuestro caso es un poco más sencillo, pues el nodo que se le quita es la raíz. Recordemos que si estamos trabajando con B_i, de la raíz cuelgan exactamente i árboles binomiales, desde B_0 hasta B_{i-1}. Por lo tanto resulta natural al eliminar a la raíz de un árbol binomial, en el contexto de un heap binomial, tomar a cada uno de los subárboles cuya raíz es un hijo del nodo que deseamos eliminar, y construir con eso un nuevo heap binomial, que mantenga la característica de consistir de árboles binomiales completos, a lo más uno por cada índice. El código para el método eliminaMin se encuentra en el Listado F.16.

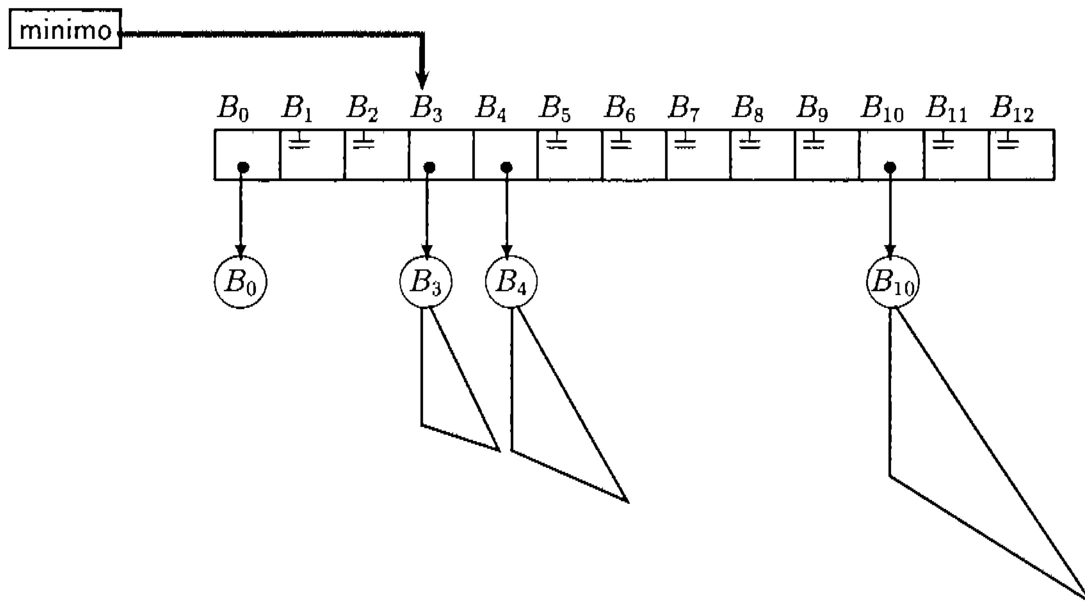
Listado F.16: Se elimina el elemento con menor valor del heap

```
30        /* Remueve a la raíz del árbol binomial que contiene al        *
31         * mínimo.                                                      */
32    public void eliminaMin() {
33        int k = minimo.getOrden();
34        HeapBinomial nuevo = new HeapBinomial(k+1);
35        for (int i = 0; i <= k; i++)
36            nuevo[i] = minimo.getHijo(i);
37        combina(nuevo);
38    }
```

Del código anterior podemos ver que, en general, el costo de las operaciones va a depender del costo que tenga combina.

Hay dos maneras de operar sobre un heap binomial: la primera de ellas consiste en que en todo momento antes y después de cada operación, vamos a mantener la invariante de que para cada k hay a lo más un árbol binomial B_k en el heap. Si trabajamos de esta forma, podemos tener un arreglo de referencias, donde en la posición i del arreglo se encuentra, o bien una referencia a B_i, o una referencia nula si B_i no está presente en heap (Figura F.43).

Figura F.43: Representación de un heap, donde se cumple que a lo más hay un único B_i para cada i



Veamos cómo se lleva a cabo la operación de combina(otroHeap) con este tipo de representación. Aprovechando la similitud entre las referencias de un heap binomial y la representación binaria de un entero, podemos pensar en que esta operación corresponde a una suma binaria, ya que tenemos lo siguiente:

- Supongamos que tenemos dos números binarios que vamos a sumar. El algoritmo para

sumar dos números binarios se encuentra en el Listado F.17.

Listado F.17: Suma(b_1, b_2)→ b_3.

```
1         /* Suma dos número binarios. Cada posición vale 0 o 1.    */
2     public byte[] suma(byte[] b1, byte[] b2)  {
3         int n = b1.length;
4         int m = b2.length;
5         int k = (n > m) ? n : m; // máximo entre n y m.
6         byte[] b3 = new byte[k + 1];
7         byte sumaParcial;
8         byte acarreo = 0;
9         for (int i = 0; i < k; i++)  {
10            sumaParcial = acarreo + b2[i] + b1[i];
11            b3[i] = sumaParcial % 2;
12            acarreo = sumaParcial / 2;
13        }
14        b3[k] = acarreo;
15        return b3;
16    }
```

Podemos observar lo siguiente en la suma binaria. Supongamos que en la posición 4 tanto en b_1 como en b_2 tenemos 1. Eso quiere decir que esa posición aporta $2^4+2^4 = 16+16 = 32 = 2^5$ nodos a la suma; por lo tanto, en b_3 la posición 4 estaría en 0 y la posición 5 en 1, ya que $010000+010000 = 100000$ $(16+16 = 32)$. Pero si estuviéramos sumando $011000+011000$ (que en base 10 sería $24 + 24$), en la posición 3 tendríamos un acarreo, porque $2^3 = 8$ y $8+8 = 16$. En ese caso, el resultado de la suma en la posición 4 seria de 48 $(16+16+16)$ cuya representación binaria es 110000.

Podemos razonar respecto a los árboles binomiales de manera similar. Si al combinar dos heaps tenemos 2 árboles, digamos B_4, en cada uno de esos árboles hay 16 nodos. Al combinarlos, tenemos que acomodar 32 nodos, que corresponden exactamente a un B_5. El algoritmo para combinar dos heaps binomiales se encuentra en el Listado F.18. Recuérdese que *ligar* un árbol binomial con otro es el proceso de tomar dos árboles binomiales del mismo índice y colgar uno de ellos del otro, para obtener un árbol binomial con su índice incrementado en 1.

Listado F.18: Implementación de combina(otroHeap). 1/2

```
39          /* Combina (suma) dos árboles binomiales en uno solo.    */
40    public HeapBinomial combina(HeapBinomial otroHeap)   {
41        int k = (getNMax() > otroHeap.getNMax()) ? getNMax()
42                   : otroHeap.getNMAX();
43        HeapBinomial resultado = new HeapBinomial(k + 2);
44        ArbolBinomial acarreo = new ArbolBinomial(0);
45        for (int i = 0; i <= k; i++) {
46            if (heap[i] == null)  {
47                if (otroHeap.heap[i] == null)
48                    resultado.heap[i] =  null;
```

Listado F.18: Implementación de combina(otroHeap). 2/2

```
49                  else
50                      resultado.heap[i] = otroHeap.heap[i];
51              else
52                  if (otroHeap.heap[i] == null)
53                      resultado.heap[i] = heap[i];
54                  else {
55                      resultado.heap[i] = null;
56                      acarreo = heap[i].ligado(otroHeap.heap[i]);
57                  }
58              }
59              if (acarreo != null) {
60                  if (heap[i] == null) {
61                      if (otroHeap.heap[i] == null) {
62                          resultado.heap[i] = acarreo;
63                          acarreo = null;
64                      }
65                      else {
66                          acarreo = acarreo.ligado(otroHeap.heap[i]);
67                          resultado.heap[i] = null;
68                      }
69                  }
70                  else {
71                      if (otroHeap.heap[i] == null) {
72                          acarreo = acarreo.ligado(heap[i]);
73                          resultado.heap[i] = null;
74                      }
75                      else {
76                          acarreo = acarreo.ligado(heap[i]);
77                          resultado.heap[i] = otroHeap.heap[i];
78                      }
79                  }
80              }
81          }
82          resultado.heap[k + 1] = acarreo;
83          return resultado;
84      }
```

Se puede observar que dado que no podemos "sumar módulo 2", tenemos que revisar todas las posibles combinaciones de las posiciones en los dos heaps, con el acarreo de la posición anterior. Para la posición 0 no tenemos nada acarreado. Cuando se termina de iterar, tenemos posiblemente el acarreo de la última posición.

Con este tipo de implementación, y si hacemos un análisis amortizado, los costos para un heap binomial con n nodos son como siguen:

HeapBinomial(Nodo)	$O(1)$.
encuentraMin()	$O(1)$.
inserta(i)	$O(1)$.

eliminaMin()	$O(\log n)$.
combina(h')	$O(\log n)$.

Si bien queda claro, de ver los algoritmos, el porqué de los costos de $\log n$ para eliminaMin y combina, no está claro el porqué podemos insertar un nodo en el heap en tiempo constante. Pero pensemos un poco en este problema: insertar un nodo a un heap nos puede llevar hasta $O(\log n)$ operaciones, si es que el heap está muy "lleno" (tiene B_i para casi todas las i's) y esto ocasiona muchos acarreos a la hora de combinar. Esto nos dice que el costo de una inserción estará relacionado directamente con el número de árboles binomiales que se encuentren en el heap. Por lo que podemos *repartir* el costo de la inserción entre las operaciones que fueron creando cada uno de los árboles. Para nuestro análisis amortizado del heap binomial, estableceremos una cuenta de ahorros para cada árbol en el heap. Cuando el árbol es creado, cobraremos "por adelantado" a la operación que crea el árbol. Como para esta operación el costo es una constante, no se afecta el orden de complejidad de la operación (si antes costaba una constante c, ahora cuesta $c + 1 = c'$; si antes costaba, por ejemplo, $\log n$, ahora cuesta $2 \log n$ que sigue siendo $O(\log n)$).

Mantendremos la siguiente invariante en el heap binomial:

Cada árbol en el heap tiene una unidad depositada.

Tanto las operaciones de inserta como el constructor HeapBinomial(Nodos) cuestan, cada una, una unidad. La operación eliminaMin genera en el proceso hasta $\log n$ nuevos árboles, por lo que se le cobrará $2 \cdot \log n$, la mitad para que pague lo que cuesta la operación y la otra mitad ($\log n$) para que le deposite a cada árbol que genera una unidad. Sin embargo, eliminaMin sigue costando $O(\log n)$. Las operaciones de ligado las pagamos con la unidad depositada en la raíz del árbol que ligamos (la raíz del árbol al cual ligamos mantiene su depósito de una unidad). La operación inserta puede ocasionar acarreos, pero el tiempo para ejecutar estos acarreos ya fue contabilizado, ya que los acarreos se manifiestan como ligados.

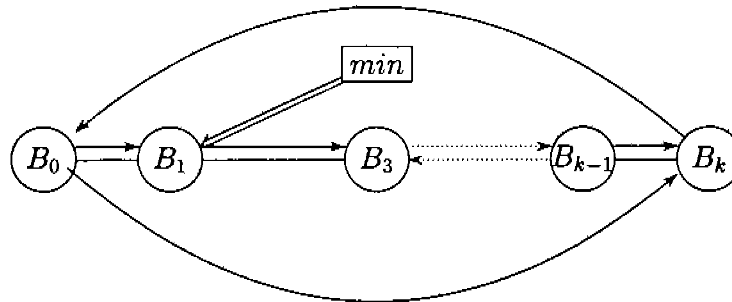
Todo esto hace que podamos realizar la operación de inserta en tiempo constante amortizado.

Podemos optar por una representación alterna para el heap binomial, que nos permitiría obtener tiempo de $O(1)$ para la operación combina, si no nos obligamos a mantener la invariante de que haya a lo más un árbol binomial para cada índice i. Si representamos al heap como una lista circular doblemente ligada en la que se encuentran todos los árboles binomiales en un momento dado, puedo tener más de un árbol binomial con el mismo índice, como se muestra en la Figura F.44.

Si mantenemos este tipo de estructura, la operación de combina(otroHeap) consistiría únicamente de mover apuntadores para concatenar las listas de los heaps que se desea combinar, y decidir cuál es el menor de los dos mínimos. Como tenemos ligas dobles, esto se lleva un número constante de operaciones, por lo que quedaríamos con un costo de $O(1)$.

Sin embargo, el costo de eliminaMin() depende de cuántos árboles tenemos en el heap (ya que debo revisar las raíces de todos y cada uno de los árboles para encontrar el nuevo

Figura F.44: Representación de un heap binomial con lista circular doblemente ligada



mínimo). Si tengo demasiados árboles, ya no vamos a poder llevar a cabo esta operación en $O(\log n)$. Lo que podemos hacer es, cada vez que se invoque a eliminaMin, reorganizar el heap de tal manera de conseguir tener a lo más un árbol binomial para cada i. Esto lo hacemos creando un arreglo como en el caso anterior, inicialmente con todos los apuntadores vacíos, y conforme nos vamos encontrando a un árbol B_i en la lista, ligamos y acarreamos de ser necesario. En el transcurso de este proceso, buscamos al mínimo.

Utilizando los costos amortizados, y mediante operaciones de ligado cuyo costo ya fue pagado (gastamos la unidad que se encuentra en la raíz de cada árbol), ejecutamos una cantidad constante de trabajo por cada árbol en la lista para colocarlos en el nuevo heap: si empezamos con m árboles y hacemos k ligados, gastamos $O(m+k)$ tiempo en todo el proceso. Para pagarlo, tenemos k unidades ahorradas de los ligados (utilizando, como acabamos de mencionar, la unidad depositada en la raíz del árbol que se cuelga), y podemos "cobrarle" a la operación de eliminaMin $\log n$ unidades, ya que esto no altera el orden de costo de eliminaMin, que sigue siendo $O(\log n)$. De lo anterior, tenemos de dónde obtener $O(k + \log n)$ unidades para pagar por esta operación, por lo que si mostramos que $m+k$ es $O(k+\log n)$, tendremos bien calculado el costo amortizado de la operación ($m + k$ es el costo real). Argumentamos que esto es cierto de la siguiente manera:

i. Si hacemos k ligados, cada ligado disminuye el número de árboles en 1, por lo que al final terminaremos con $m - k$ árboles.
ii. Estos árboles son todos de distinto índice, por lo que a lo más hay $\log n$ de ellos (es el número de árboles en un heap binomial con n nodos).
iii. Entonces,

$$\begin{aligned} m + k = &\quad 2k + (m - k) \\ \leq &\quad 2k + \log n \\ = &\quad O(k + \log n). \end{aligned}$$

F.5.1. Métodos borra(Nodo v) y reduceLlave(Nodos v, int delta)

Otras dos operaciones que resultan importantes en el contexto de listas de prioridades son las que permiten borrar un nodo arbitrario de un heap binomial, y aquella que permite

reducir el valor de una llave cualquiera, manteniendo las características del heap binomial (esta última es una de las más importantes en el algoritmo de Dijkstra para caminos más cortos).

Cuando estamos implementando nuestra cola de prioridades con un heap binomial, los costos de estas operaciones (ya sean peor caso o amortizados) van a estar dados de la siguiente manera:

borra(Nodo v): Esta operación se puede llevar a cabo de la siguiente manera:

(1)	Reduce el valor de la llave en el nodo v a $-\infty$.	$O(1)$
(2)	Reorganiza el árbol binomial para que este valor quede en la raíz.	$O(\log n)$
(3)	Pon al apuntador al mínimo del heap a apuntar a este árbol binomial.	$O(1)$
(4)	Ejecuta un eliminaMin()	$O(\log n)$.

Como se puede ver, aun con costos amortizados, esta operación resulta costosa, con un costo de $O(\log n)$.

reduceLlave(Nodos v, int delta): Es casi igual que la operación de borrado de un nodo, excepto que nos ahorramos el tener que reorganizar el heap. Pero este costo, si tenemos la representación de árboles en lista ligada circular, es constante:

(1)	Reduce el valor de la llave en el nodo i en Δ.	$O(1)$
(2)	Reorganiza el árbol binomial para que vuelva a estar parcialmente ordenado.	$O(\log n)$
(3)	Compara contra el mínimo del heap, para ver si el nuevo valor es el mínimo. Si es así, pon al apuntador a apuntar a él.	$O(1)$

Como se puede observar, también esta operación es de $O(\log n)$.

En la siguiente sección buscaremos mejorar estos tiempos, modificando la estructura de los heaps binomiales para que estas dos operaciones sean más eficientes, introduciendo el concepto de heaps de Fibonacci.

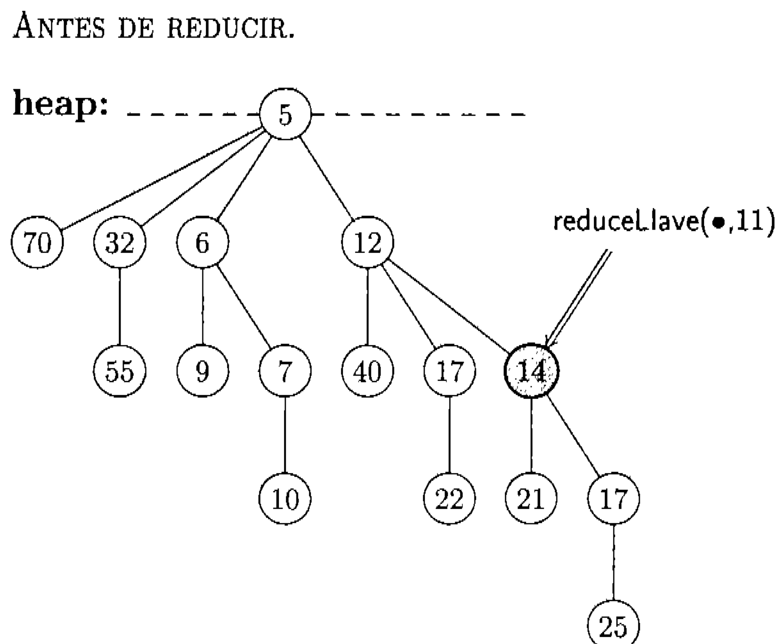
F.6. Heaps de Fibonacci

En los heaps binomiales, aun con una integración perezosa de los subárboles (hacerlo lo más tarde posible), la complejidad de realizar operaciones de eliminar a un elemento arbitrario (eliminaNodo(Nodo v)) o de reducir la llave siguen siendo costosas, y en algoritmos donde continuamente esté cambiando el valor de la llave, esto puede acrecentar desproporcionadamente la complejidad del algoritmo. Si logramos que el costo de la operación reduceLlave(Nodos v, int delta) sea $O(1)$, en algoritmos como el de Dijkstra donde repetidamente se va a llamar a esta operación, obtendremos un ahorro considerable, sobre todo cuando el número de aristas en la gráfica sea grande.

En los heaps de Fibonacci, todas las operaciones que vimos para los heaps binomiales toman $O(1)$, excepto por eliminaMin(); por lo que resultan, al menos desde el punto de vista teórico, una buena opción para implementar una cola de prioridades, como la que se requiere por ejemplo para el algoritmo de Dijkstra, bajando su complejidad de $O(|E| \cdot |V|)$ con una lista ligada simple a $O(|E| + |V| \log |V|)$. Sin embargo, los heaps binomiales no admiten las operaciones de eliminaNodo(v) para un nodo distinto de la raíz, y tampoco permiten el decremento de una llave arbitraria, al menos no a un costo razonable. Los heaps de Fibonacci agregan a los heaps binomiales estas dos operaciones a costos razonables. Veamos en detalle en qué consisten los heaps de Fibonacci.

Fredman y Tarjan presentaron por primera vez los heaps de Fibonacci en 1984 [FT84], como una generalización de los heaps binomiales. Les llevó 3 años más para presentar el trabajo completo en *Journal of the ACM*, en 1987 [FT87]. En su diseño incorporan el análisis amortizado de costos. Partimos de un heap binomial, pero no exigimos que en todo momento los árboles binomiales estén completos, sino que si queremos realizar la eliminación del nodo v, cortamos al subárbol con raíz en v y procedemos a manejarlo de la misma manera que lo hacíamos con un heap binomial, eliminando a la raíz de ese subárbol y generando tantos árboles como hijos del nodo v hubiera, para después proceder a incorporar cada uno de los árboles generados al heap original, determinando el mínimo. Este proceso lleva $O(\log n)$. Para el caso de reduceLlave(Nodos v, int delta), decrementamos la llave del nodo, y en caso de que viole la condición de heap con respecto a su padre, procederemos también a cortar el subárbol con raíz en v y proceder a manejarlo como si estuviéramos incorporando un árbol a un heap binomial. Podemos ver un ejemplos en las Figuras F.45 a F.47. En la Figura F.45 se reduce el valor de la llave del nodo sombreado de 14 a 3, por lo que deja de cumplirse la condición de heap con su padre, que tiene 12 como valor en su llave, obligando a podar ese subárbol.

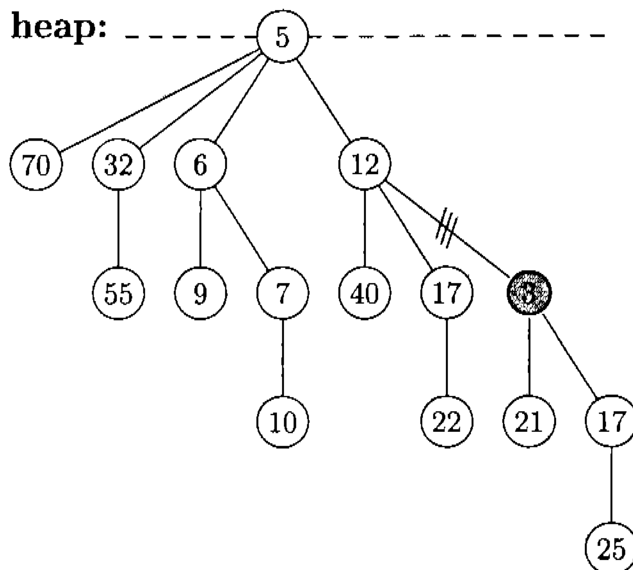
Figura F.45: Acción de podado en un árbol de Fibonacci



Se reduce la llave del nodo, y como se viola la condición de heap, se poda al subárbol con raíz en ese nodo, para no tener que reorganizar a todo el árbol. La operación de podar lleva tiempo constante, pero además amortizaremos para cuando se tenga que llevar a cabo una operación de encontrar el mínimo.

Figura F.46: Acción de podado en un árbol de Fibonacci

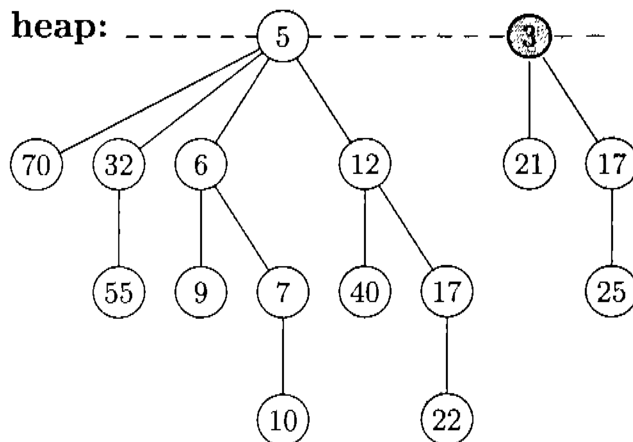
SE REDUCE NODO.



Al podar el subárbol, éste se coloca en la lista circular de los árboles. Si bien en este ejemplo el subárbol podado corresponde a un árbol binomial, no sucede así en todos los casos. Baste notar que el árbol al que se le podó esta rama ya no corresponde a un árbol binomial, puesto que no es completo, esto es, el número de hijos no es una potencia de 2.

Figura F.47: Acción de podado en un árbol de Fibonacci

SE PODA EL SUBÁRBOL Y SE ENLAZA EN EL HEAP



Si bien esto hace que el costo de reducir una llave sea $O(1)$, si no acotamos el número de podas que le podemos hacer a un árbol, podemos terminar con muchísimos árboles, con pocos nodos cada uno de ellos, de tal manera que perdamos la propiedad de que n valores se distribuyen en $\log n = k$ árboles, y como el costo de la operación que elimina al mínimo depende del número de árboles en el heap, esto nos puede encarecer de manera significativa el costo de la ejecución del algoritmo en cuestión. Para evitar la fragmentación del heap ponemos restricciones sobre el número de ramas que le podemos quitar a un nodo, y este número es 2. En cuanto le quito a un nodo su segundo hijo, a ese nodo, junto con lo que le queda de su subárbol, lo podo a su vez de su padre. Esta acción continúa hacia la raíz del árbol, como en cascada, hasta llegar a un nodo al que ya no se le esté quitando su segundo hijo.

Con esta acción buscamos que la relación entre el número de nodos en la estructura y el número de árboles que se utilizan para acomodarlos sea una relación exponencial, esto es, que para alguna base b, $n = b^k$ donde k es una cota superior para el número de árboles en la estructura. En el caso de los árboles binomiales, $b = 2$. Es obvio que en el caso de los árboles de Fibonacci, b deberá ser menor a 2, ya que el número de nodos que vamos a tener en cada árbol B_i va a estar acotado por arriba por 2^i, pues vamos a permitir quitar nodos. El meollo del asunto es que no quitemos tantos nodos que terminemos con un nodo por árbol.

El problema con este método es el siguiente: las cotas de $\log n$ en el costo de las operaciones de los heaps binomiales se deben a que mantenemos a los árboles *frondosos*, esto es, el número total de nodos en el árbol B_k – su tamaño – es exponencial en k, lo que hace que la profundidad del árbol sea precisamente $\log k$. Sin embargo, si cortamos arbitrariamente subárboles, podemos convertir al árbol en uno donde la relación entre el número total de nodos y la profundidad del árbol sea más cercana a lineal que a logarítmica, y entonces el análisis de costos que hicimos ya no es válido. Hagamos un nuevo análisis.

Como ya mencionamos, limitaremos el corte de subárboles de un nodo dado a 2. Supongamos que el nodo k es descendiente directo del nodo i. El costo de podar un subárbol es constante, pues únicamente tenemos que enlazarlo a la lista ligada del heap, y comparar la nueva raíz contra el mínimo, si es que la acción que realizamos fue la de disminuir una llave que rompió la relación de heap entre el nodo k y el nodo i. Si es que se va a podar el subárbol con raíz en k, además del costo que se paga por podarlo, se depositan 2 unidades en el padre. De esta manera se lleva cuenta de cuántos de sus hijos se han podado. Al podar el segundo hijo, esto provoca que el padre, con el subárbol restante, sea podado también. El costo de podar al padre es pagado por el depósito de cada hijo, por lo que estas podas en cascada, aun cuando se extiendan muy alto hasta la raíz, ya estarán amortizadas.

Veamos en la Figura F.48 cómo se extienden las podas de subárboles, utilizando nuevamente el árbol de la Figura F.45, reduciendo las llaves 14 y 17 que son descendientes del nodo con llave 12, a los valores 11 y 5 respectivamente.

Cuando se reduce la primera llave, simplemente se poda el subárbol, y se anota que el nodo con llave 12 ya sufrió una poda, como se muestra en la Figura F.49.

La reducción de la llave del segundo hijo hace que nuevamente se viole la condición de heap. Pero como éste es el segundo subárbol que se va a podar, esto desencadena la poda

Figura F.48: Acción de podado en un árbol de Fibonacci

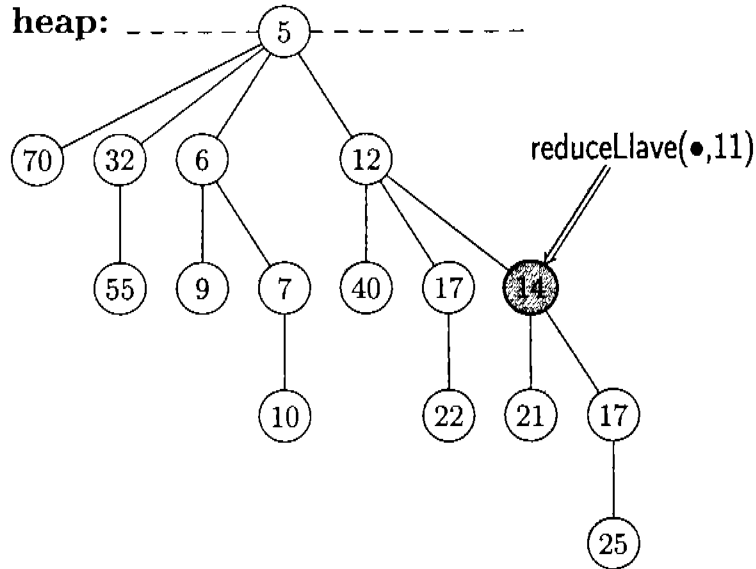
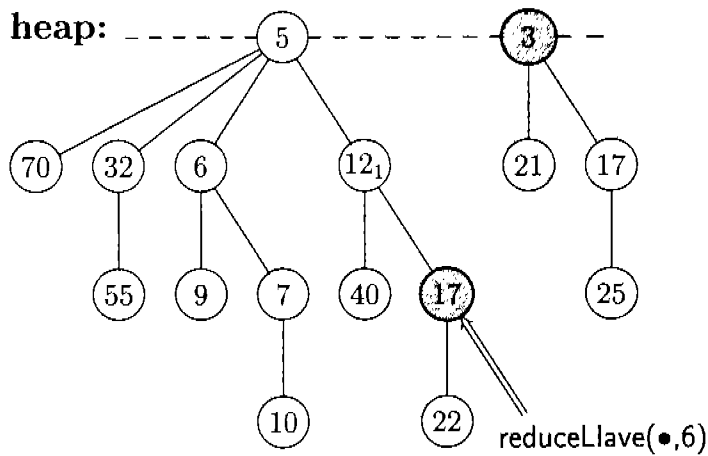


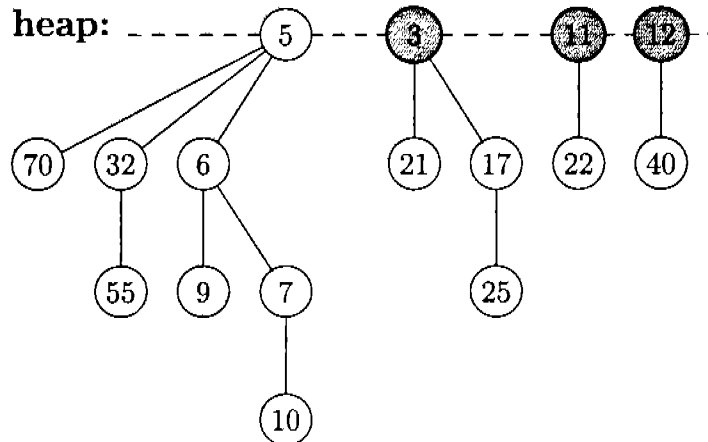
Figura F.49: Acción de podado en un árbol de Fibonacci



del subárbol con raíz en el padre, una vez que se le han podado los dos hijos. El resultado se muestra en la Figura F.50.

Figura F.50: Acción de podado en un árbol de Fibonacci

SE PODA EL SEGUNDO SUBÁRBOL.



Se puede observar en la Figura F.50 que al podar el subárbol que ya había sufrido dos podas, el número de nodos en todos los árboles vuelve a ser exponencial con el rango de la raíz, lo que demostraremos en los lemas F.9 y F.10.

Lema F.9 *Sea* x *un nodo cualquiera en un heap de Fibonacci y sea* y_i *el i-ésimo subárbol que se agregó como hijo de* x. *Entonces el rango de* y_i *es al menos* $i-2$.

Demostración:

En el momento en que se liga a y_i a x, x ya tiene a y_1, y_2, ..., y_{i-1}, ligados a él. Como nunca se ligan árboles de distintos rangos, esto quiere decir que en el momento en que y_i fue ligado a x, tenía rango $i-1$, el mismo que x. Desde ese momento, y si sigue siendo hijo de x, a lo más se le quitó un hijo, por lo que tiene rango al menos $i-2$. ❑

Pasamos ahora a demostrar que el número de descendientes es exponencial en el rango de la raíz.

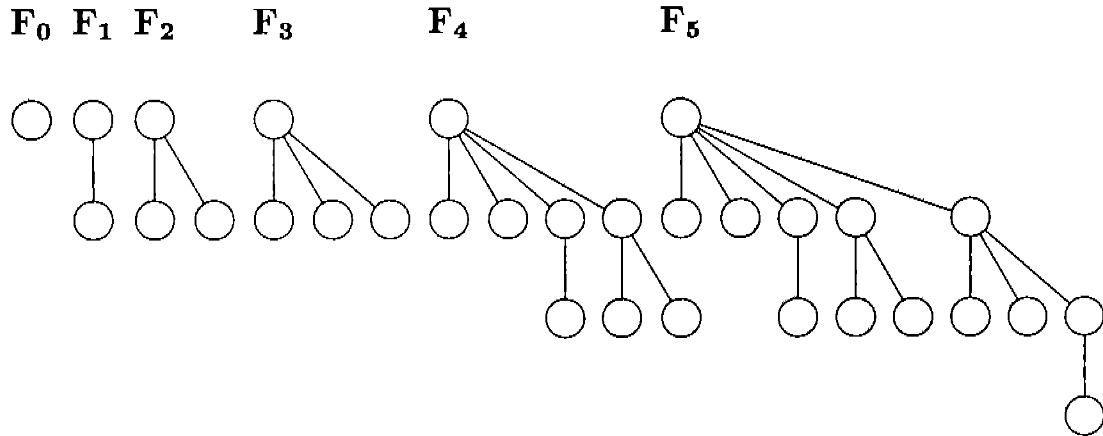
Lema F.10 *Sean* F_k *los números de Fibonacci definidos por* $F_0 = 1$, $F_1 = 1$ *y* $F_k = F_{k-2} + F_{k-1}$. *Cualquier nodo de rango* $k \geq 1$ *tiene al menos* F_{k+1} *descendientes (incluyéndose a sí mismo).*

Demostración:

La demostración se hará por inducción sobre k, el rango de la raíz.

Base: Observemos en la Figura F.51 a los árboles F_0, F_1, F_2, F_3, F_4 y F_5. El índice del árbol corresponde al rango de la raíz, y cada uno es un árbol binomial al que se le podó a lo más un subárbol con raíz en alguno de los hijos. En esta figura se puede observar que el lema se cumple para $k = 1, 2, 3$ (se omite mostrar cuáles son las otras formas que pudieran tomar árboles de Fibonacci con índice 3, 4 y 5, para no saturar al lector).

Figura F.51: Algunos árboles de Fibonacci



Inducción: Supongamos ahora que el lema se cumple para $2 \leq j < k$ (esto incluye a $k-2$ y $k-1$) y veamos que pasa con el árbol de índice k.

Sea S_k el árbol de menor tamaño de rango k. Por el Lema F.9, de la raíz de S_k deben colgar subárboles de rango $k-2, k-3, \ldots, 1$ y 0, más algún otro subárbol que tiene al menos 1 nodo. Si contamos además a la raíz de S_k, tenemos que el tamaño de S_k, $k > 1$, está dado por

$$|S_k| = \sum_{i=0}^{k-2} = F_{i+1}.$$

Esta última parte de la igualdad se demuestra fácilmente. La propiedad de que los números de Fibonacci crecen exponencialmente también se puede demostrar fácilmente, pero no es materia de este trabajo. ❏

El siguiente lema redondea el punto de que los heaps de Fibonacci tienen tamaño exponencial.

Lema F.11 *El rango de cualquier nodo en un heap de Fibonacci es* $O(\log n)$.

Demostración:

Si el número total de vértices en un subárbol es siempre exponencial con el rango, cualquiera de los nodos debe tener rango a lo más de $\log n$. ❏

La principal motivación en este capítulo es la de obtener un costo amortizado de $O(1)$ para el método reduceLlave, y como éste es el principal de los que cambia con la generalización de los heaps binomiales, tenemos la obligación de demostrar que esta cota se cumple.

El tiempo que se requiere para decrementar una llave es 1 más el número de cortes que se produzcan en cascada hacia la raíz del árbol. Como este número pudiera ser en función de la profundidad del árbol, debemos amortizar estos cortes de alguna manera. Podemos observar

que el número de árboles se incrementa con cada poda que se haga. Al mismo tiempo, el número de nodos que ya tienen un hijo podado se decrementa, pues todos los subárboles que son podados tenían ya a un hijo podado pero ahora ya son hojas. Por lo tanto, cuando podamos al primer hijo de un nodo podemos amortizar, pagando doble, por la poda del segundo hijo que se vaya a realizar en forma de cascada. Como únicamente aquellos nodos que ya están marcados, y por lo tanto tienen amortizado el costo de la poda del segundo hijo, van a ser podados, esto nos garantiza que las podas en cascada tienen un costo amortizado de $O(1)$.

Terminamos esta sección con la demostración de la correctez de esta estructura de datos en cuanto a la complejidad.

Teorema F.12 *Las cotas de costo amortizado para los heaps de Fibonacci son $O(1)$ para combina, inserta y reduceLlave, y $O(\log n)$ para eliminaMin.*

Demostración:

Pensemos que cada vez que construimos un árbol, como en el caso de los heaps binomiales, depositamos una unidad en su raíz, que va a ser utilizada cuando se le combine con otro árbol. También recordemos que cada vez que marcamos a un hijo de un árbol estamos depositando una unidad para que cuando se le pode por su segundo hijo, esta poda ya esté pagada. Dada esta situación, tenemos lo siguiente:

combina(Heap otroHeap): Se paga con la unidad depositada en la raíz del árbol que se cuelga.

inserta(int k): Se crea un F_0 al que se le deposita una unidad para su posterior combinación, más una unidad que cuesta la construcción misma.

reduceLlave(Nodos v, int delta): Como al marcar a un hijo se deposita para la poda del segundo, y marcarlo cuesta tiempo constante, las podas en cadena se pagan con este depósito y el costo total es, entonces, $O(1)$.

eliminaMin(): Como al eliminar a la raíz de uno de los árboles se producen a lo más $\log n$ nuevos árboles que hay que combinar, el costo amortizado de esta operación resulta ser precisamente $O(\log n)$. En este proceso no se usa nada del costo depositado por los marcados de la poda del primer hijo.

F.7. Implementaciones eficientes para heaps binomiales y de Fibonacci

Una pregunta relevante es el cómo implementar tanto los heaps de Fibonacci como los heaps binomiales. Dado que vamos a estar podando subárboles, ya no se ve factible o económico guardar los árboles por niveles, pues en un momento dado podemos estar eliminando un subárbol que afecte a varios niveles. Por lo tanto se tendrá que recurrir a referencias para tener un buen manejo de las estructuras de datos, donde cada hijo deberá apuntar a su padre, para poder comparar la propiedad de heap; para las referencias a los hijos se sugiere una

representación de un apuntador desde el padre al hijo mayor, y cada nodo puede tener un apuntador a su hijo mayor y a su hermano inmediato a la derecha.

Para la implementación de algoritmos de exploración, es conveniente tener en cada vértice una referencia al árbol de Fibonacci en el cual se encuentra ese vértice, para poder decrementar su llave de manera directa. Es obvio que este tipo de árboles no son árboles de búsqueda; lo mismo se puede decir para cualquier árbol parcialmente ordenado.

Utilizaremos heaps de Fibonacci para cuando tengamos una frontera en los algoritmos de exploración donde el valor de las llaves puede variar mucho, y son constantemente actualizadas. Ejemplos típicos son el algoritmo de Dijkstra para caminos más cortos con pesos positivos arbitrarios en las aristas, y el algoritmo de Prim para encontrar un árbol generador de peso mínimo.

Bibliografía

Bibliografía

[ACDM01] T. Amtoft, C. Consel, O. Danvy, and K. Malmkjr. The abstraction and instantiation of string-matching programs, 2001.

[AMO93] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice-Hall, 1993.

[ASS96] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs, Second Edition.* MIT Press, 1996.

[Ata99] M. J. Atallah, editor. *Algorithms and Theory of Computation Handbook.* CRC Press, 1999.

[Aus00] Metthew H. Austern. *Segmented Iterators and Hierarchical Algorithms*, chapter 2, pages 80–90. Lecture Notes in Computer Science. Springer, 2000.

[BBJ02] George S. Boolos, John P. Burgess, and Richard C. Jeffrey. *Computability and Logic.* Cambridge University Press, 4th. edition, 2002.

[BCD+01] T. Biedl, T. Chan, E. D. Demaine, R. Fleischer, M. Golin, and J. I. Munro. Funsort. *Proceedings of the 2nd International Conference FUN with Algorithms 2*, Proceedings in Informatics 10:15–26, 2001.

[BG00] S. Baase and A. Van Gelder. *Computer Algorithms, Introduction to Design and Analysis, Third Edition.* Addison-Wesley, 2000.

[BK02] Jérémy Barbay and Claire Kenyon. Adaptive intersection and t-threshold problems. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 390–399. Society for Industrial and Applied Mathematics, 2002.

[BM77] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[BM99] J. Boyer and W. Myrvold. Stop minding your p's and q's: A simplified o(n) planar embedding algorithm. In *10th Annual ACM-SIAM Symposium on Discrete Algorithms*, Baltimore, Maryland, USA, Jan 1999.

[Boy04] J. Boyer. The algorithm of lempel, even and cederbaum. In *Latin America 2004*, 2004.

[Bur27] A. Burloud. *La Pensée Conceptuelle*, page 163. Alcan, Paris, 1927.

[CD93] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 493–501. ACM Press, 1993.

[Cer03] Paul E. Ceruzzi. *A History of Modern Computing.* The MIT Press, 2nd. edition, 2003.

[CKT93] Joseph Cheriyan, Ming-Yang Kao, and Ramakrishna Thurimella. Scan-first search and sparse certificates: an improved parallel algorithm for k-vertex connectivity. *SIAM J. Comput.*, 22(1):157–174, 1993.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, second edition.* The MIT Press and McGraw-Hill Book Company, 2001.

[Col] George E. Collins. Algebraic algorithms. CS801 Lecture Notes, University of Wisconsin, 1971.

[Dav58] M. Davis. *Computability and Unsolvability.* McGraw-Hill, 1958.

[Din70] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Mathematics Doklady*, 11:1277–1280, 1970.

[DLOM00] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 743–752. Society for Industrial and Applied Mathematics, 2000.

[DS00] James C. Dehnert and Alexander Stepanov. *Fundamentals of Generic Programming*, chapter 1, pages 1–11. Lecture Notes in Computer Science. Springer, 2000.

[DSL] Erick Demaine, Lee Wee Sun, and Charles E. Leiserson. Problem set 9 solutions. Handout 32, Course: Introduction to Algorithms. Found in the Internet.

[ECW92] Vladmir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Comput. Surv.*, 24(4):441–476, 1992.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, April 1972.

[Eve79] S. Even. *Graph Algorithms.* ComputerSoftware Engineering Series. Computer Science Press, 1979.

[FF62] L. R. Ford and D.R. Fulkerson. *Flows in Networks.* Princeton University Press, 1962.

[FFFK01] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing.* MIT Press, 2001.

[FT84] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In IEEE, editor, *25th annual Symposium on Foundations of Computer Science, October 24–26, 1984, Singer Island, Florida*, pages 338–346, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1984. IEEE Computer Society Press.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addisson-Wesley Professional Computing Series, 1994.

[Gri89] D. Gries. *The Science of Programming.* Springer–Verlag, 1989.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, October 1988. Preliminary version in Proc. 18th Annual ACM Symposium on the Theory of Computing, pages 136–146, 1986.

[Gus97] D. Gusfield. *ALGORITHMS ON STRINGS, TREES, AND SEQUENCES, Computer Science and Computational Biology.* Cambridge University Press, 1997.

[Har72] F. Harari. *Graph Theory.* Addison-Wesley Publishing Company, 1972.

[HT74] J. Hopcroft and R. E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21:549–569, 1974.

[IHB+00] Georges Ifrah, E. F. Harding, David Bellos, Sophie Wood, and Harding E. F. *The Universal History of Computing: From the Abacus to Quantum Computing.* John Wiley & Sons, Inc., 2000.

[JLM00] M. Jazayeri, R. Loos, and D. Musser, editors. *Generic Programming: International Seminar, Dagstuhl Castle, Germany, 1998, Selected Papers*, volume 1766 of *Lecture Notes in Computer Science.* Springer-Verlag, Heidelberg, Germany, 2000.

[KM61] Donald E. Knuth and JackÑ. Merner. Algol 60 confidential. *Commun. ACM*, 4(6):268–272, 1961.

[KMP77] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977. This paper presents a fast deterministic algorithm for the problem of determining if a given pattern of m symbols occurs in a text of length n. Their well-known algorithm runs in time $O(n + m)$, making judicious use of a *prefix function*, which for a given pattern encapsulates knowledge about how the pattern matches against shifts of itself.

[Knu68] D. Knuth. THE ART OF COMPUTER PROGRAMMING, VOLUME 3: SORTING AND SEARCHING. Addison-Wesley, Reading, MA, 1968.

[Knu98a] D. Knuth. THE ART OF COMPUTER PROGRAMMING, VOLUME 1: FUNDAMENTAL ALGORITHMS. Addison-Wesley, Reading, MA, 3rd edition, 1998.

[Knu98b] D. Knuth. THE ART OF COMPUTER PROGRAMMING, VOLUME 3: SORTING AND SEARCHING. Addison-Wesley, Reading, MA, 3rd edition, 1998.

[Koz97] D. C. Kozen. *Automata and Computability.* Undergraduate Texts in Computer Science. Springer -Verlag New York Inc., 1997.

[Kül00] Dietmar Kül. *Generic Graph Algorithms*, chapter 5, pages 249–255. Lecture Notes in Computer Science. Springer, 2000.

[KV00] O. Kupferman and M. Y. Vardi. An automata-theoretic approach to modular model checking. *ACM Transactions on Programming Languages and Systems*, 22(1):87–128, January 2000.

[KW00] Ulrich Köthe and Karsten Weihe. *The STL Model in the Geometric Domain*, chapter 5, pages 232–248. Lecture Notes in Computer Science. Springer, 2000.

[Lam02] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison Wesley, 2002.

[LEC66] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing on graphs. In *Proceedings of the Intertnational Symposium on the Theory of Graphs*, Rome, Italy, 1966.

[Loc90] John Locke. *An Essay Concerning Human Understanding*, chapter XII: of Complex Ideas, Made by mind out of simple ones. 1690.

[LPS00] Karl Lieberherr and Boaz Patt-Shamir. *The Refinement Relation on Graph-Based Generic Programs*, chapter 1, pages 40–52. Lecture Notes in Computer Science. Springer, 2000.

[Mer85] S. M. Merritt. An inverted taxonomy of sorting algorithms. *Communications of the ACM*, 28(1):96–99, January 1985.

[Mer89] S. M. Merritt. A top down unification of minimum cost spanning tree algorithms. *SIGCOMM*, pages 116–127, 1989.

[MN98] D. R. Musser and G. V. Nishanov. A fast generic sequence matching algorithm. Technical report, Computer Science Dept., Rensselaer Polytechnic Institute, Troy, NY, March 1998.

[MS96] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide.* Addisson Wesley, 1996.

[MSL00] David Musser, Sibylle Schupp, and Rüdiger Loos. *Requirement Oriented Programming, Concepts, Implications, and Algorithms*, chapter 1, pages 12–24. Lecture Notes in Computer Science. Springer, 2000.

[NI92] Hiroshi Nagamochi and Toshihide Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7:583–596, 1992.

[NW01] D. Z. Nguyen and S. B. Wong. Design patterns for sorting. In *Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, pages 263–267. ACM Press, 2001.

[Pev89] P. A. Pevzner. L-tuple DNA sequencing: Computer analysis. *J. Biomol. Struct. Dyn.*, 7:63–73, 1989.

[Rey00] Liliana Araceli Reyes. Algoritmos genéricos para flujo en redes. Tesis de licenciatura, Facultad de Ciencias, 2000.

[Rib97] T. Ribot. *L'Évolution des idées générales*, pages 232–3. Alcan Paris, 1897.

[RV99] S. Rajsbaum and E. Viso. Enfoque unificador para algoritmos de exploración. In *Proceedings of the Segundo Encuentro Nacional de Computación, Pachuca, Hidalgo, México*. SMCC, Septiembre 1999.

[RV03] Sergio Rajsbaum and Elisa Viso. A case for OO – Java – in teaching algorithm analysis. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 79–83. ACM, Computer Science Press, Inc., 2003.

[RV05] Sergio Rajsbaum and Elisa Viso. Object-oriented algorithm analysis and design with Java. *Science of Computer Programming*, 54(1):25–47, Jan 2005.

[Sch00] Christoph Schwarzweller. *Mizar Correctness Proofs of Generic Fraction Field Arithmetic*, chapter 4, pages 178–191. Lecture Notes in Computer Science. Springer, 2000.

[SF96] R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley Publishing Company, 1996.

[SH99] W-K Shih and W-L Hsu. A new planarity test. *Theoretical Computer Science*, 223(1–2):179–191, 1999.

[Sha84] Mary Shaw. Abstraction techniques in modern programming languages. *IEEE Software*, 1(4):10–24, 26, October 1984.

[Sma93] J. Small. A unified approach of testing, embedding and drawing planar graphs. In *ALCOM International Workshop on Graph Drawing*, Sevre, France, 1993.

[Tay98] R. Gregory Taylor. *Models of Computation and Formal Languages*. Oxford University Press, 1998.

[Teo04] Virgina Teodosio. El algoritmo de boyer-moore y sus aplicaciones. Tesis de licenciatura, Facultad de Ciencias, 2004.

[Woo93] D. Wood. *Data Structures, Algorithms, and Performance*. Addison-Wesley Publishing Company, 1993.