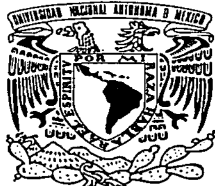
14



UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

TITULO DE TESIS

"ALGORITMOS GENÉRICOS: EL PROBLEMA DE FLUJO EN REDES"

T E S I S

QUE PARA OBTENER EL TITULO DE

LICENCIADA EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A

ARACELI LILIANA REYES CABELLO

DIRECTORA DE TESIS:

M. EN C. ELISA VISO GUROVICH

MÉXICO, D.F. DICIEMBRE DE 2002

DIVISION DE ESTUDIOS PROFESIONALES

FACULTAD DE CIENCIAS
SECCION ESCOLAR

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 UNAM a difundir en formato electrónico e impreso el contenido de mi trabajo recepcional.
NOMBRE: ARACELI LILIANA REYES CABELLO
FECHA: 02 DIC 2002
FIRMA: [firma]

DRA. MARÍA DE LOURDES ESTEVA PERALTA
Jefa de la División de Estudios Profesionales de la
Facultad de Ciencias
Presente

Comunicamos a usted que hemos revisado el trabajo escrito:

" Algoritmos Genéricos: el problema de Flujo en Redes."

realizado por Reyes Cabello Araceli Liliana

con número de cuenta 09423329-8 , quien cubrió los créditos de la carrera de:
Licenciado en Ciencias de la Computación.

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis Propietario	M.en C. Elisa Viso Gurovich.	[firma]
Propietario	Dr. Sergio Rajsbaum Gurodezky.	[firma]
Propietario	Dr. Criel Merino López.	Merino López Criel
Suplente	M. en I. María de Luz Gasca Soto.	[firma]
Suplente	M. en C. José de Jesús Galaviz Casas.	[firma]

Consejo Departamental de Matemáticas

Dra. Amparo López Gaona

FACULTAD DE CIENCIAS
CONSEJO DEPARTAMENTAL
DE
MATEMATICAS

Algoritmos Genéricos: el problema de Flujo en Redes

Araceli Liliana Reyes Cabello

A quien me enseño a luchar por mis sueños, y me apoyo a lo largo del camino.

A ti mamá.
Te amo.

Agradecimientos.

Todo tiene su tiempo ...

Tiempo de nacer, y de morir;

tiempo de plantar, y tiempo de arrancar lo plantado;

tiempo de destruir, y tiempo de edificar;

tiempo de llorar, y tiempo de reír;

tiempo de endechar, y tiempo de bailar;

tiempo de esparcir, piedras y tiempo de juntar piedras;

tiempo de abrazar, y tiempo de abstenerse de abrazar;

tiempo de buscar, y tiempo de perder;

tiempo de guardar, y tiempo de desechar;

tiempo de callar, y tiempo de hablar;

tiempo de amar, y tiempo de aborrecer;

tiempo de guerra, y tiempo de paz;

Ec 3:1-8

Gracias a todos aquellos que estuvieron en esta etapa junto a mi.

A Dios.
Por el regalo de poder existir, por el amor y la libertad de vivir que me diste. Por ser viento fresco en cada instante de mi vida.

A mis dos grandes tesoros, mamá y papá.
Por todo su amor y apoyo incondicional,por su guianza y por todas aquellas cosas que son inexpresables, pero sobre todo por confiar en mi y dejarme ser plenamente. Los amo con todo mi corazón.

A Ramón Bernal.
En donde quiera que estés, mil gracias por el toque tan especial que le diste a mi vida.

A Canek.
Por todo lo que he encontrado dentro de ti, por tu apoyo y amor. Por ayudarme a crecer.

A Elisa Viso.
Por cada instante de tu vida que me regalaste, tanto el tiempo que dedicaste a este trabajo como los momentos que me permitieron conocerte la gran persona que eres, además de excelente maestra. Mil gracias por tus sabios consejos, tu apoyo, tu confianza, cariño y la lista innumerable de cosas que diste.

A mi familia.
A mi tía Angelina, Amparito, Carmen, Víctor, Vicky, Margarita M., Clemente L. P., Carlos L. y a mis sobrinitos (hoy día sobrinotes) Carlos y Abraham. Creo que a lo largo de la vida uno va formando una familia a la cual escoge y ama, algunos de ustedes ya eran mi familia cuando nací y a los otros los conocí en el camino, pero a todos los quiero con toda mi alma. Gracias por todo lo que han aportado a mi vida.

A Favio.
Por ser mi ángel de la guarda y una de las más grandes razones por las cual sonreír ;)[1].

A Pily.
Porque si fuera cierto que los amigos es una misma alma habitando en diferentes personas ¿adivina que pasaría?. Gracias por todo tu apoyo y cariño.

A Lula.
Por ser un lago donde es fácil reflejarse, gracias porque siempre logras que la vida se vea más sencilla.

A Karla.
Por tu amistad, por tu nobleza, por escucharme y por muchas cosas más.

[1] Ahhh!!! y por el inolvidable tequila.

A Ci.
Por tu amistad incondicional, sin ti esta etapa no hubieran sido tan divertida y especial.

A Gaby.
Por compartir esta aventura conmigo, por la amistad y el cariño que me brindaste.

A La Banda y anexos.
A Karina por toda la complicidad y amistad. A Yaz por estar ahí siempre, gracias chiquitina por todo lo que en silencio has hecho por mi[2]. A Erick por toda las paz y ternura que siempre he encontrado en tu mirada, gracias por cuidarme y apoyarme. A Gustavo, por tu enorme corazón. A Rafa, Edgar y Oscar[3] por todos los momentos que compartimos. A Monty por todas tus locas ocurrencias que tanto me hacen reír. A Isma por aguantar mi histeria al final de este trabajo. A todos, mil gracias por su amistad, apoyo y cariño. Sepan que todos ustedes son "muy pero muy especiales". Los quiero.

A los cuatro mosqueteros y un colado.
A Juan, Ramiro, Enrique y Omar por su amistad, y por todos los buenos momento que me hacen pasar. Especialmente a Juan, por apoyarme y ayudarme en los días duros y por toda la ternura que guardas, eres una persona super especial. Y finalmente[4], pero no menos importante, a Rogelio por tus consejos y ánimos.

A mis maestros.
Sergio Rajsbaum, José Galaviz, Lucy Gasca y Criel Merino por el tiempo que emplearon en revisar y corregir este trabajo. A Francisco Hernández, por la confianza y apoyo que me ha brindado. Y todos los maestros que dejaran una huella en mi vida.

A La Universidad.
Por convertirse en mi segundo hogar, en el cual conocí a muchas personas a las cuales nunca olvidaré.

A Valeria.
Gracias por estar y escuchar.

A Chiquito.
Por haber formado parte de mi vida.

---

[2] Por las charlas después de las chelas...:)
[3] Aunque nunca te dejes abrazar.
[4] En esta lista, porque todavía me falta muchos agradecimientos ehh!!.

A una persona muy especial, Clemente.

Por todo el cariño, la paciencia, por todos los momentos especiales (mmm... creo que todos.), por tu amor ... Por existir y ser tu mismo, por todo lo que es imposible describir con sólo palabras.

Mil gracias por convertir cada instante en eternidad.

Te ...

Índice General

Introducción v

1 Definiciones 1
- 1.1 Gráficas Dirigidas 1
- 1.2 Redes 2
  - 1.2.1 Flujo entre dos subconjuntos de vértices 7
- 1.3 Red Residual 9
- 1.4 Cortes 13

2 Algoritmo Genérico de Flujo Máximo 19
- 2.1 El algoritmo 19
  - 2.1.1 Precondiciones y postcondiciones de los métodos de AGFM 20
  - 2.1.2 Correctez de AGFM 22
  - 2.1.3 Complejidad de AGFM 22

3 Algoritmos de Trayectoria Aumentante 25
- 3.1 Algoritmo Genérico de Trayectoria Aumentante 25
  - 3.1.1 Precondiciones y postcondiciones de los métodos de AGTA 28
  - 3.1.2 Correctez de AGTA 31
  - 3.1.3 Complejidad de AGTA 33
- 3.2 Algoritmo de Flujo Máximo Etiquetado 34
  - 3.2.1 Precondiciones y postcondiciones de los métodos de AFME 36
  - 3.2.2 Correctez de AFME 37
  - 3.2.3 Complejidad de AFME 39
- 3.3 Implementaciones concretas polinomiales 40
- 3.4 Algoritmo de Escalamiento de Capacidad 41
  - 3.4.1 Precondiciones y postcondiciones de los métodos de AEC 43
  - 3.4.2 Correctez de AEC 44
  - 3.4.3 Complejidad de AEC 46
- 3.5 Distancias Etiquetadas 49
- 3.6 Arcos y Trayectorias Admisibles 50
- 3.7 Algoritmo de Trayectoria Aumentante de Longitud Mínima 50
  - 3.7.1 Precondiciones y postcondiciones de los métodos de ATALM 52
  - 3.7.2 Correctez de ATALM 54
  - 3.7.3 Complejidad de ATALM 56

3.8 Algoritmo Dinic 58
3.8.1 Precondiciones y postcondiciones de los métodos de ADinic 62
3.8.2 Correctez de ADinic 64
3.8.3 Complejidad de ADinic 66

4 Algoritmos de Preflujo **69**
4.1 Algoritmo Genérico de Preflujo 70
4.1.1 Precondiciones y postcondiciones de los métodos de AGP 75
4.1.2 Correctez de AGP 78
4.1.3 Complejidad de AGP 81
4.2 Implementaciones concretas del Algoritmo Genérico de Preflujo 84
4.3 Algoritmo de Preflujo FIFO 84
4.3.1 Precondiciones y postcondiciones de los métodos de APFIFO 85
4.3.2 Correctez de APFIFO 87
4.3.3 Complejidad de APFIFO 88
4.4 Algoritmo de Preflujo por Distancia Máxima 89
4.4.1 Precondiciones y postcondiciones de los métodos de APDM . . . . [illegible]
4.4.2 Correctez de APDM 93
4.4.3 Complejidad de APDM 94
4.5 Algoritmo de Preflujo por Escalabilidad de Exceso 97
4.5.1 Precondiciones y postcondiciones de los métodos de APEE 101
4.5.2 Correctez de APEE 103
4.5.3 Complejidad de APEE 104

Conclusiones **109**

Listados

2.1.1 Algoritmo Genérico de Flujo Máximo 20
3.1.1 Algoritmo Genérico de Trayectoria Aumentante 27
3.1.1 Algoritmo Genérico de Trayectoria Aumentante (continuación) 28
3.2.1 Algoritmo de Flujo Máximo Etiquetado 35
3.2.1 Algoritmo de Flujo Máximo Etiquetado (continuación). 36
3.4.1 Algoritmo de Escalamiento de Capacidad 41
3.4.1 Algoritmo de Escalamiento de Capacidad (continuación) 42
3.4.1 Algoritmo de Escalamiento de Capacidad (continuación) 43
3.7.1 Algoritmo de Trayectoria Aumentante de Longitud Mínima 51
3.7.1 Algoritmo de Trayectoria Aumentante de Longitud Mínima (continuación) . 52
3.8.1 Algoritmo Dinic 60
3.8.1 Algoritmo Dinic (continuación) 61
3.8.1 Algoritmo Dinic (continuación) 62
4.1.1 Algoritmo Genérico de Preflujo 72
4.1.1 Algoritmo Genérico de Preflujo (continuación) 73
4.1.1 Algoritmo Genérico de Preflujo (continuación) 74
4.1.1 Algoritmo Genérico de Preflujo (continuación) 75
4.3.1 Algoritmo de Preflujo FIFO 85
4.4.1 Algoritmo de Preflujo de Distancia Máxima 90
4.4.1 Algoritmo de Preflujo de Distancia Máxima (continuación) 91
4.5.1 Algoritmo de Preflujo por Escalamiento de Exceso 99
4.5.1 Algoritmo de Preflujo por Escalamiento de Exceso (continuación) 100
4.5.1 Algoritmo de Preflujo por Escalamiento de Exceso (continuación) 101

Introducción

Una manera natural de resolver problemas pertenecientes a una misma clase, es la de tratar de abstraer características comunes a los problemas y soluciones, que aunque no contengan todo el detalle necesario, nos dan un marco de referencia adecuado para intentar la solución de problemas concretos en esta clase. Un *algoritmo genérico* nos proporciona este marco de referencia, y proporciona de manera abstracta una forma genérica de resolver todos los problemas de la clase dada. Este algoritmo se refina o especializa dando una implementación particular para los métodos utilizados, pero manteniendo fijo el esquema general de la solución. Este enfoque para el diseño y análisis de algoritmos permite tener una visión más general de la solución antes de involucrarse con la idiosincracia de un método particular, facilitando de manera sensible las demostraciones de correctez y los cálculos de complejidad de las especializaciones.

Una vez definido el esquema general con el que se va a trabajar, que incluye el uso de métodos abstractos, se procede a definir invariantes que deberán cumplir las extensiones de los métodos. Con estas invariantes se demuestra la correctez del algoritmo genérico y se calcula la complejidad en función de la complejidad que tengan las especializaciones. Una vez hecho esto, el trabajo a realizar para demostrar la correctez de una especialización es, simplemente, demostrar que la especialización del método cumple con las invariantes requeridas en el algoritmo genérico. De la misma manera, se calcula la complejidad de la especialización del método y se inserta eso en la fórmula derivada para el algoritmo genérico. Esta metodología la aplicamos, en este trabajo, al problema de flujo máximo en redes.

Las redes son un modelo matemático que nos permite representar situaciones de la vida diaria, como es el caso de instalaciones eléctricas, redes telefónicas, redes hidráulicas, autopistas, redes de computadoras, entre otras.

Una red consiste de un conjunto de puntos llamados *vértices*, y un conjunto de ligas, llamadas *arcos*, donde cada arco conecta a dos vértices. Tenemos como ejemplos:

Redes	Vértices	Arcos	Flujo
Sistemas de comunicaciones.	Teléfonos, computadoras, satélites.	Cables, fibras ópticas, ligas de transmisión de microondas.	Mensajes de voz, video, datos.
Operaciones financieras.	Acciones, tipos de cambio.	Transacciones.	Capital.
Transportes.	Aereopuertos, cruces viales, estaciones de tren.	Carreteras, vías del tren, rutas aéreas.	Carga, pasajeros, vehículos.
Sistemas hidráulicos.	Estaciones de bombeo, reservas, lagos.	Tubería.	Agua, gas, petróleo.

Una red es una gráfica dirigida en donde los vértices concentran el flujo en cada momento, y éste se mueve a través de los arcos de un vértice a otro. Cada arco tiene una capacidad, que representa la máxima cantidad de flujo que puede pasar por el arco cuando viaja de un vértice a otro. En este trabajo asumiremos que la capacidad es siempre mayor o igual a cero.

El problema de flujo máximo consiste en enviar la mayor cantidad de flujo posible entre dos vértices distinguidos a través de los arcos de la red. Al vértice desde el cual enviamos inicialmente el flujo le llamamos *vértice origen*, y el vértice al cual deseamos que llegue finalmente todo el flujo que enviamos desde el vértice origen le llamamos *vértice destino*.

Cualquier flujo en una red siempre debe contar con las siguientes propiedades: el flujo que atraviesa un arco nunca debe exceder la capacidad del mismo, y para cualquier vértice que no sea ni el vértice origen ni el vértice destino, las unidades de flujo que recibe deben ser las mismas que envía. A esta última propiedad se le conoce como la ley *conservación del flujo*.

La idea principal de los algoritmos que resuelven el problema de flujo máximo es la siguiente: dado un flujo y una red, se verifica si es posible enviar flujo desde el vértice origen al vértice destino sin exceder ninguna de las capacidades de los arcos. Podemos pensar, entonces, en un algoritmo genérico que resuelva el problema del flujo máximo.

Aun a este nivel de abstracción podemos reconocer dos enfoques distintos, que generan, cada uno de ellos algoritmos con un poco más de especialización que el del nivel superior. En el primero de ellos se incrementa el flujo a través de trayectorias aumentantes del vértice origen al vértice destino, y mientras exista alguna trayectoria aumentante se repite el proceso. La idea principal del segundo enfoque es empujar flujo inicialmente a través de los arcos adyacentes al vértice origen. Diremos que un vértice está activo si la cantidad de flujo que recibe es mayor que la que envía. Después de que el algoritmo envíe flujo del vértice origen a sus vértices adyacentes, éstos estarán activos. En cada paso el algoritmo elige un vértice activo con el propósito de deshacerse del exceso de flujo correspondiente. Con este método no es posible incrementar el flujo cuando el único vértice que está activo es el vértice destino. Podemos observar que este algoritmo cumple con la propiedad de conservación de flujo únicamente hasta que no exista ningún vértice que esté activo, con excepción del vértice destino. Al flujo que circula por la red antes de que la conservación de flujo se satisfaga se le conoce como *preflujo*.

En el primer capítulo presentamos los conceptos que serán requeridos durante este trabajo así como algunos resultados que nos serán de utilidad en las demostraciones de correctez y complejidad de los algoritmos de flujo máximo. Concluimos este capítulo demostrando el *Teorema de Flujo Máximo Corte Mínimo*, que es tal vez el resultado más importante de la teoría de redes y del problema de flujo máximo.

En el capítulo dos introducimos la implementación del *Algoritmo Genérico de Flujo Máximo* (AGFM). Como mencionamos antes, un algoritmo genérico es aquel que abstrae las operaciones que tienen en común una serie de algoritmos. Dando por hecho que dichas operaciones son correctas, es decir, cada uno de los algoritmos concretos resuelven las abstracciones correctamente. Asumiendo lo anterior, y partiendo del hecho que AGFM abstrae las operaciones principales que son necesarias para resolver el problema de flujo máximo, únicamente debemos ocuparnos en demostrar que el algoritmo genérico resuelve el problema si las invariantes dadas se cumplen. En el caso de AGFM las abstracciones son las siguientes: verificar si es posible incrementar el flujo, y en tal caso incrementarlo, mientras esto sea posible. Una vez que ya no es posible enviar más unidades de flujo del vértice origen al vértice destino, definimos el valor del flujo actual, el cual debe ser un flujo máximo.

Como anteriormente mencionamos existen dos formas principales para resolver estas operaciones. Por lo tanto, del Algoritmo Genérico de Flujo Máximo se desprenden dos algoritmos genéricos que abstraen las operaciones básicas de cada método, así como las características principales de cada uno de ellos.

En el capítulo tres se presenta el *Algoritmo Genérico de Trayectoria Aumentante* (AGTA). Este algoritmo abstrae todas las operaciones básicas que comparten los algoritmos basados en la idea de encontrar el flujo máximo a través de trayectorias aumentantes. Entonces, verificar que sea posible incrementar el flujo en el Algoritmo Genérico de Flujo Máximo equivale a encontrar una trayectoria aumentante del vértice origen al vértice destino, y la operación de incrementar el flujo se reduce a aumentar el flujo a través de dicha trayectoria. En el algoritmo AGTA no nos ocupamos por cómo resolver la operación de encontrar una trayectoria aumentante; eso se resuelve de forma distinta en cada una de las versiones concretas de este algoritmo. Además, presentamos cuatro algoritmos concretos basados en el AGTA: el *Algoritmo de Flujo Máximo Etiquetado* (AFME); el *Algoritmo de Trayectoria Aumentante de Longitud Mínima* (ATALM); el *Algoritmo de Escalamiento de Capacidad* (AEC); y el *Algoritmo Dinic* (ADinic). Los últimos tres son polinomiales, mientras que el primero es exponencial.

Después de presentar el algoritmo AGTA y sus versiones concretas, en el siguiente capítulo presentamos el algoritmo genérico basado en la idea de preflujo; el *Algoritmo Genérico de Preflujo* (AGP), en donde la operación de verificar si es posible incrementar el flujo se traduce en revisar si existe un vértice que esté activo, y la operación de incrementar el flujo se traduce a empujar flujo desde el vértice activo hacia sus vértices vecinos. Nuevamente, como este algoritmo también es genérico, el cómo seleccione el vértice activo desde el cual va a realizar el empuje, así como el elegir el vértice al cuál va a enviar el flujo, será resuelto de manera distinta por cada una de las versiones concretas del algoritmo. Los algoritmos concretos de AGP que se presentan en este trabajo son: el *Algoritmo de Preflujo FIFO* (APFIFO); el *Algoritmo de Preflujo de Distancia Mínima* (APDM); y el *Algoritmo de Preflujo de Escalamiento de Exceso* (APEE).

Decidimos presentar cada algoritmo implementándolo en un lenguaje de programación concreto para poder probar la correctez y complejidad en un ambiente de ejecución real. El inconveniente que tiene demostrar la correctez y complejidad de un algoritmo que está en pseudo código es que no podemos garantizar que sea posible realizar todas las operaciones en un ambiente real respetando la complejidad del algoritmo.

Para realizar la implementación de los algoritmos presentados en este trabajo el lenguaje elegido fue Java básicamente por la manera en cómo este lenguaje maneja la herencia, la cual nos permite modelar de manera natural tanto los algoritmos genéricos como cada uno de los algoritmos concretos.

Cada uno de los algoritmos que se presentan está implementado en el lenguaje de programación Java, ninguno de ellos se presenta en pseudo código. Y para probar su funcionamiento se implementó la representación de una red, y cada una de las clases requeridas por ésta.

La figura 1 (pag. x) muestra la jerarquía de herencia que se utilizó para la implementación de los algoritmos.

Trabajos Relacionados con Algoritmos Genéricos

Hoy día existen varios algoritmos genéricos, además de herramientas que nos permiten implementarlos. A continuación se mencionan algunos lenguajes de programación que proveen bibliotecas que nos permiten implementar de una manera natural un algoritmo genérico.

El *diseño de patrones* tiene la idea de caracterizar interacciones comunes entre objetos que son frecuentemente utilizados, con el objetivo de reutilizar código orientado a objetos (OO). En otras palabras, la finalidad de diseñar un patrón es describir la forma en la que un grupo de objetos interactúan de tal forma que tanto los métodos como el tipo de datos de cada objeto es independiente de los otros. Esta separación ha sido siempre el objetivo de la programación orientada a objetos (OO). El diseño de patrones puede existir en diferentes niveles. Para los niveles más altos (abstractos), las soluciones son más generales; mientras que para los niveles mas bajos (particulares), las soluciones son más específicas. Actualmente, existen lenguajes de programación que proporcionan herramientas (bibliotecas) que nos permiten implementar de manera natural un patrón, o bien, un algoritmo genérico. A continuación se mencionan algunos de ellos.

La Fundación de Clases de Java *Java Fundation Clases (JFC)* introduce 23 patrones los cuales son presentados y estudiados en [7]. Ésta es una de las razones principales por la cual los algoritmo presentados en este trabajos son implementados en Java.

Otro lenguaje de programación que provee bibliotecas para realizar algoritmos genéricos es el lenguaje de programación C++, el cual contiene la Biblioteca Estándar de Patrones (Genérica) (Standar Template Library (STL)). Esta biblioteca incluye algoritmos genéricos que realizan operaciones comunes de manipulación de datos, tales como buscar, ordenar y mezclar. STL es la reencarnación de años de investigación en cuanto a los temas relacionados con algoritmos genéricos [31].

El lenguaje Ada95 también provee de una biblioteca de estructura de datos similares a STL, llamada Charles [23]. Esta biblioteca también nos permite implementar algoritmos genéricos en términos de iteradores, los cuales nos hacen posibles utilizar cualquier estructura de datos siempre y cuando tengamos un iterador con las operaciones requeridas.

Susan M. Merritt presenta un algoritmo genérico [28], que fue resultado de invertir la taxonomía existente de los algoritmos de ordenamiento, la cual clasifica a dichos algoritmos en tres categorías: inserción, selección e intercambio. Con esta clasificación el diseño de los algoritmos es realizado de abajo hacia arriba (bottom-up). La taxonomía propuesta por Merrit clasifica los algoritmos en dos categorías: hardsplit/easyjoin y easysplit/hardjoin. Esta clasificación origina que puedan abstraerse las características comunes de los algoritmos de ordenamiento, permitiendo así diseñar de arriba hacia abajo(top-down), en otras palabras, es posible desarrollar un algoritmo genérico.

David R. Musser y Gor V. Nishanov en marzo de 1998 presentan un algoritmo genérico para cazar secuencias sobre un tipo arbitrario [30]. Utilizaron para la implementación de este algoritmo biblioteca STL de C++.

Mehryar Mohri propone un algoritmo genérico [29], el cual construye un autómata con peso en las transiciones, sin $\epsilon-$transiciones equivalente a un autómata con peso que tiene $\epsilon-$transiciones. Para desarrollar este algoritmo utilizaron el algoritmo genérico de trayectoria más corta. Este algoritmo utiliza una cola de prioridades para encontrar una trayectoria con longitud mínima.

Los ejemplos antes mencionados son algoritmos genéricos en el sentido de los tipos de datos que pueden manejar. Tanto las herramientas que nos brindan los lenguajes de programación como Java, C++ y Ada, están enfocados más que a una solución abstracta a un tipo de datos abstracto, permitiéndonos que la solución no dependa del tipo de datos que el algoritmo reciba.

En este trabajo el concepto "genérico" está orientado a la solución de una misma clase de problemas, en este caso encontrar el flujo máximo en una red y no al tipo de datos que el algoritmo puede manejar, aunque también no está atado a un estructura de datos en particular.

Trabajos Relacionados con Flujo en Redes

Los primeros en estudiar el problema de flujo máximo fueron Dantzing [9] y, Ford y Furkelson [15] en los años 50's, proponiendo algoritmos psuedopolinomiales basados en la idea de envía flujo a través de trayectorias aumentante. En los años $70's$ Dinic [11] y Edmonds y Karp [14], de manera independiente, sugieren algoritmos polinomiales, los cuales aumentan el flujo por medio de trayectorias de longitud mínima. Durante las siguientes décadas, se desarrollaron algoritmos más eficientes para resolver el problema de flujo máximo. Dinic [11] presenta un algoritmo basado en la idea de bloquear el flujo. Hasta este momento todos los algoritmos propuestos aumentan el flujo a través de trayectorias aumentante. El primero en proponer un algoritmo utilizando preflujos es Karzanov [25], propone el primer algoritmo basado en preflujos utilizando una red de capas. Shiloach y Vishkin [32] describen un algoritmo algoritmo de preflujo que es antecesor de el Algoritmo de Preflujo FIFO, el cual es presentado en el capítulo 3 de este trabajo. Ahuja y Orlin [1] sugieren el Algoritmo de Escalamiento de Capacidad. El Algoritmo Genérico de Preflujo es desarrollado en los $80's$ por Tarjan y Goldberg [19].

Estos son solo algunos de los algoritmo desarrollados hasta hoy para resolver el problema de flujo máximo (para más información ver [21] y [4]).

En los $60's$ Dantzing [10] y Jewell [24], estudian el problema de flujo máximo generalizado, el cual, a diferencia del problema tradicional de flujo máximo no exige que el flujo cumpla con la ley de conservación de flujo. Dado que existen muchos problemas que pueden ser modelados con una red, donde las unidades de flujo que se desea enviar del vértice destino al vértice origen sufren alguna daño durante la transmisión, por ejemplo, supongamos que tenemos una tubería por la cual fluye un gas que puede evaporarse, en este caso, existe una pérdida de las unidades de flujo que se enviaron, violándose la ley de conservación de flujo. En 1999 Kevin D. Wayne [36] propone un algoritmo para el problema de flujo máximo generalizado.

En [4] se presenta el Algoritmo Genérico de Trayectoria Aumentante y el Algoritmo Genérico de Preflujo, así como las especializaciones presentadas en este trabajo, con la diferencia que únicamente son mostrados en pseudocódigo o bien de manera intuitiva. Mientras, que en este trabajo cada uno de los algoritmo ha sido implementado utilizando orientado a objetos (OO), además, de demostrar la complejidad y correctez de cada uno de ellos. Incluyendo también las invariante que deben cumplir cada uno de los métodos abstractos. En [4] también se presentan diferente aplicaciones del problema de flujo máximo.

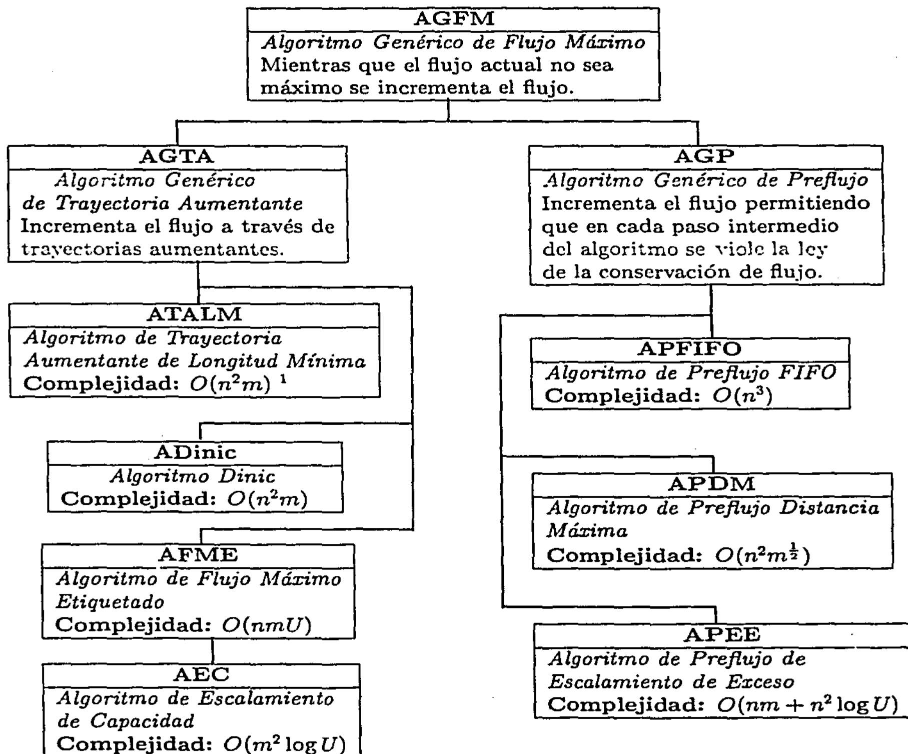


Figura 1: Jerarquía de herencia.

Capítulo 1

Definiciones

En este capítulo revisaremos algunos conceptos básicos que están relacionados con las *redes* y *el problema de flujo máximo.*

Antes de pasar a definir lo que es una red es necesario tener claro el concepto de *gráfica dirigida*, dado que una red no es más que una gráfica dirigida con ciertas propiedades especiales, tales como dos vértices distinguidos y capacidad en los arcos, entre otras.

1.1 Gráficas Dirigidas

En todos los algoritmos que veremos consideraremos una red como una gráfica dirigida $G = (V, A)$ donde V es un par ordenado no vacío de puntos llamados *vértices.* A es un conjunto de *arcos*, donde cada arco es una pareja ordenada (u, v), con u y $v \in V$.

Definición 1.1 *Una* gráfica dirigida $G = (V, A)$ *consiste de un conjunto* V *finito y no vacío de vértices y un conjunto (posiblemente vacío)* A *de parejas ordenadas* (u, v) *con* $u, v \in V$.

Cada vértice $u \in V$ tiene asociada una lista de adyacencias, que es el conjunto de vértices v para los cuales existe un arco $(u, v) \in A$. Denotamos con $Ad(u)$ la lista de adyacencias del vértice u. Formalmente:

$$Ad(u) = \{v|(u, v) \in A\}$$

El *exgrado* de un vértice $u \in V$ es el número total de arcos tales que u es el primer componente de la pareja ordenada, esto es, el número total de arcos que salen de u. El número de arcos donde u es el segundo componente de la pareja ordena corresponde a el *ingrado* del vértice u, es decir, es el número de arcos que llegan al vértice u).

Un *camino dirigido* es una secuencia de vértices y arcos —lo formalizaremos a continuación. Otro concepto que utilizaremos frecuentemente es el de *trayectoria dirigida* que es simplemente un camino dirigido en el que un vértice no puede aparecer dos veces. Formalmente:

Definición 1.2 *Un* camino dirigido *de un gráfica* $G = (V, A)$ *es una secuencia orientada de vértices y arcos*

$$v_1 - a_1 - v_2 - a_2 - \ldots v_{r-1} - a_{r-1} - v_r$$

tal que, $\forall k$ con $1 \leq k \leq r-1$, cada arco $a_k = (v_k, v_{k+1}) \in A$.

Definición 1.3 *Una* trayectoria dirigida *de una gráfica $G = (V, A)$ es un camino dirigido simple, en otras palabras, en el que no se repite ningún vértice.*

1.2 Redes

Una red es una gráfica dirigida G con atributos asociados a cada arco de la misma.

En una red se pueden estudiar tres problemas:

1. Si el atributo asociado a los arcos es un costo, entonces, el problema de trayectoria más corta en una red G, consiste en encontrar un camino simple entre dos vértices dados, cuyo costo sea mínimo.

2. Si cada arco tienen asociada una capacidad, el problema de flujo máximo consiste en encontrar el flujo máximo que puede ser enviado entre un par de vértices distinguidos en una red.

3. El problema de costo mínimo en una red, a la cuál se le ha asignado costo y capacidad a cada arco; consiste en encontrar el flujo máximo que puede ser enviado desde el vértice origen hasta el destino con el menor costo posible.

En este trabajo nos enfocaremos en el problema de flujo máximo: dada una red encontrar el valor del flujo máximo que puede ser enviado desde un vértice origen a un vértice destino.

Intuitivamente el problema de flujo máximo lo podemos expresar de la siguiente manera: dada una red, que es una gráfica dirigida $G = (V, A)$ donde cada arco $(u, v) \in A$ tiene una capacidad $c(u, v)$ —la cuál limita el flujo que puede circular desde el vértice u al vértice v— debemos encontrar cuál es el máximo flujo que puede ser enviado desde un vértice origen, denotado por s, hasta un vértice destino, denotado por t, con $s, t \in V$. Encontrar un flujo máximo en la red G será posible siempre y cuando G cumpla con las siguientes condiciones:

$R1$ La red debe ser una gráfica dirigida.

$R2$ La capacidad de todos los arcos siempre debe ser un número entero no negativo.

$R3$ La red G no debe tener ninguna trayectoria en la cual todos los arcos que la forman tengan capacidad infinita.

$R4$ La red no contiene arcos paralelos, esto es, dos arcos donde sus componentes son iguales.

$R5$ La red no debe contener arcos que sean lazos, es decir, los dos componentes de la pareja ordenada son el mismo vértice.

Formalmente una red se define de la siguiente manera:

Definición 1.4 *Una* red *es una gráfica dirigida $G = (V, A, s, t, c)$ donde V es el conjunto de vértices, A el conjunto de arcos, s y t son dos vértices distinguidos que están en V, s el origen y t el destino, y por último $c : A \to \mathbb{N}$. Cada arco $(u, v) \in A$ tiene una capacidad no negativa $c(u, v) \geq 0$. Si el arco $(u, v) \notin A$ asumimos que su capacidad es igual a cero.*

Denotamos con U a la máxima capacidad de los arcos (u, v) que pertenecen a A.

$$U = \max\{c(u, v) | c(u, v) \geq 0 \text{ y } (u, v) \in A\}$$

El *flujo* en una red es una función f que dado un arco $(u, v) \in A$ devuelve un número real que representa el flujo que es enviado desde u hasta v.

1. El flujo de un arco siempre debe ser menor o igual que su capacidad.
2. El flujo total que puede ser enviado de un vértice u es igual al flujo que recibe este vértice, excepto para los vértices origen y destino.

Formalizando la idea intuitiva del flujo en redes, tenemos las siguientes definiciones:

Definición 1.5 *Sea $G = (V, A, s, t, c)$ una red. Un* flujo *es una función $f : A \to \mathbb{R}$ que cumple con las siguientes condiciones:*

1. Restricción de la capacidad: *El flujo de un arco $(u, v) \in A$ nunca debe exceder su capacidad:*

$$\forall \ (u, v) \in A \quad f(u, v) \leq c(u, v)$$

2. Conservación del flujo: *Para cualquier vértice $u \in V - \{s, t\}$ se cumple:*

$$\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) = 0$$

Si además se cumple que para cualquier arco (u, v) el flujo es un número no negativo, entonces, es un *flujo factible*.

Cualquier flujo en una red debe ser un flujo factible, esto es, el flujo que es enviado entre cualesquiera dos vértices no puede tomar un valor negativo. A lo largo de este trabajo siempre que hablemos del flujo f que circula en una red, f es un flujo factible. En caso de que estemos hablando de un flujo f en la red residual, f no es necesariamente un flujo factible, por lo tanto, solo debe cumplir con las propiedades de flujo.

El *flujo total que entra a un vértice $u \in V$* está definido como:

$$\sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v)$$

Y el *flujo total que es enviado desde u* está definido como:

$$\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u)$$

El valor del flujo f, denotado por $|f|$, es igual al flujo total que puede ser enviado desde s.

$$|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s)$$

Una de las maneras de incrementar el valor del flujo en una red G es haciéndolo directamente en ésta, encontrando una trayectoria desde el vértice s que es el origen hasta el vértice t que es el destino. Esta trayectoria no necesariamente debe ser dirigida —por esta razón la llamaremos $s \rightsquigarrow t$ *semitrayectoria*. La definición formal se presenta a continuación:

Definición 1.6 *Una $s \rightsquigarrow t$* semitrayectoria *de una red $G = (V, A, s, t, c)$ es una sucesión de vértices y arcos:*

$$v_1 - a_1 - v_2 - a_2 - \ldots v_{n-1} - a_{n-1} - v_n$$

tal que $v_1 = s$, $v_n = t$ y cada arco a_i pertenece a A y es de la forma $a_i = (v_i, v_{i+1})$ o $a_i = (v_{i+1}, v_i)$ con $i = 1, \ldots, n-1$.

Sea p una $s \rightsquigarrow t$ semitrayectoria. Decimos que un arco es de *avance* si es de la forma $a_i = (v_i, v_{i+1})$; en caso contrario decimos que el arco es de *retroceso.*

Intuitivamente, una $s \rightsquigarrow t$ semitrayectoria p es *no saturada* si en todos los arcos de avance de p el flujo que atraviesa el arco es menor que su capacidad, y para todos los arcos de retroceso el flujo es mayor que cero. En otras palabras, si el arco es de avance, incrementaremos el flujo a través de él; por eso es que necesitamos que la capacidad sea mayor que el flujo actual. En cambio, si el arco es de retroceso el flujo se decrementa. Por lo tanto, si queremos incrementar el flujo actual, se requiere que para poder decrementar la cantidad que sale del vértice, el flujo de las aristas de retroceso sea positivo. Formalmente:

Definición 1.7 *Sea p una $s \rightsquigarrow t$ semitrayectoria con*

$$p = v_1 - a_1 - v_2 - a_2 - \ldots v_{n-1} - a_{n-1} - v_n$$

decimos que p es no saturada *si para cada arco a_i con $i = 1 \ldots n-1$ se cumple:*

$$\begin{array}{lll} a. & f(a_i) < c(a_i) & \text{si } a_i = (v_i, v_{i+1}) \\ b. & f(a_i) > 0 & \text{si } a_i = (v_{i+1}, v_i) \end{array}$$

Si p es una $s \rightsquigarrow t$ semitrayectoria no saturada decimos que p es una semitrayectoria aumentante.

Luego entonces, el aumento del flujo en una red se puede hacer mediante una semitrayectoria aumentante, mientras exista alguna. Una vez que no podemos encontrar ninguna semitrayectoria aumentante ya no tenemos forma de seguir incrementando el flujo y por lo tanto habremos encontrado un flujo máximo. El siguiente resultado nos muestra la relación que hay entre las semitrayectorias aumentantes y el flujo máximo de una red G.

Lema 1.1 *Sea $G = (V, A, s, t, c)$ una red y f un flujo en G. Si G no contiene una semitrayectoria aumentante, entonces f es un flujo máximo.*

Demostración:

La demostración se hará por contradicción. Supongamos que $G = (V, A, s, t, c)$ contiene una semitrayectoria aumentante Q, y f es un flujo máximo. Sea Q de la siguiente forma:

$$Q = u_0, a_1, u_1, \ldots, u_{n-1}, a_n, u_n.$$

Por definición de Q sabemos que cada arco a_i, con $i = 1, \ldots, n$, cumple con alguna de las siguientes condiciones:

$$\begin{array}{lll} \text{a.} & f(a_i) < c(a_i) & \text{si } a_i = (v_i, v_{i+1}) \\ \text{b.} & f(a_i) > 0 & \text{si } a_i = (v_{i+1}, v_i) \end{array}$$

Para cada i definimos $\Delta(a_i)$ como:

$$\Delta(a_i) = \begin{cases} c(a_i) - f(a_i) & \text{si } a_i = (u_{i-1}, u_i). \\ f(a_i) & \text{si } a_i = (u_i, u_{i-1}). \end{cases}$$

Sea $\Delta = \min\{\Delta(a_i) | a_i \in Q\}$. Definimos el flujo $f' : A \to \mathbb{R}$ como:

$$f'(u,v) = \begin{cases} f(u,v) + \Delta & \text{si } (u,v) \in Q \text{ y es una arista de avance} \\ f(u,v) - \Delta & \text{si } (u,v) \in Q \text{ y es una arista de retroceso} \\ -(f(u,v) - \Delta) & \text{si } (v,u) \in Q \text{ y es una arista de retoceso} \\ -(f(u,v) + \Delta) & \text{si } (v,u) \in Q \text{ es una arista de avance} \\ f(u,v) & \text{en cualquier otro caso} \end{cases}$$

Ahora tenemos que probar que f' es un flujo en G y $|f'| > |f|$. Notemos que $f'(a) \leq c(a)$ para cada $a \in A$. Por lo tanto cumple con la restricción de la capacidad. Veamos ahora que también cumple con la ley de conservación de flujo.

Sea $u \in V - \{s, t\}$. Si $u \notin Q$ entonces $f'(u,v) = f(u,v)$ y $f'(v,u) = f(v,u)$ para cada $v \in V$. Entonces

$$\sum_{v \in V} f'(u,v) - \sum_{v \in V} f'(v,u) = 0.$$

Si $u \in Q$, sea $u = u_i$ para alguna i, con $1 \leq i \leq n - 1$. Entonces tenemos tres casos:

- **Caso 1:** Suponemos que $a_i = (u_i, u_{i-1})$ y $a_{i+1} = (u_i, u_{i+1})$. Sea $V' = V - \{u_{i-1}, u_{i+1}\}$ tenemos, entonces

$$\begin{aligned} \sum_{v \in V} f'(u_i, v) = & \quad f'(u_i, u_{i-1}) + f'(u_i, u_{i+1}) + \sum_{v \in V'} f(u_i, v) \\ = & \quad (f(u_i, u_{i-1}) - \Delta) + (f(u_i, u_{i+1}) + \Delta) + \sum_{v \in V'} f(u_i, v) \\ = & \quad \sum_{v \in V} f(u_i, v) \end{aligned}$$

y con una descomposición similar

$$\sum_{v\in V} f'(v,u_i) = \sum_{v\in V} f(u_i,v)$$

En este caso se cumple la ley de conservación del flujo, es decir

$$\sum_{v\in V} f'(u_i,v) - \sum_{v\in V} f'(v,u_i) = 0$$

- **Caso 2:** Suponemos que $a_i = (u_{i-1}, u_i)$ y $a_{i+1} = (u_{i+1}, u_i)$. La demostración es análoga al caso anterior.

- **Caso 3:** Suponemos que tenemos alguno de los siguientes casos $a_i = (u_{i-1}, u_i)$ y $a_{i+1} = (u_i, u_{i+1})$, o bien $a_i = (u_i, u_{i-1})$ y $a_{i+1} = (u_{i+1}, u_i)$. Veamos qué sucede en el primer caso. Sea $V' = V - \{u_{i+1}\}$, entonces:

$$\begin{aligned}
\sum_{v\in V} f'(u_i,v) &= f'(u_i,u_{i+1}) + \sum_{v\in V'} f(u_i,v) \\
&= (f(u_i,u_{i+1}) + \Delta) + \sum_{v\in V'} f(u_i,v) \\
&= \sum_{v\in V} f(u_i,v) + \Delta
\end{aligned}$$

y sea $V'' = V - \{u_{i-1}\}$,

$$\begin{aligned}
\sum_{v\in V} f'(v,u_i) &= f'(u_{i-1},u_i) + \sum_{v\in V''} f(v,u_i) \\
&= (f(u_{i-1},u_i) + \Delta) + \sum_{v\in V''} f(v,u_i) \\
&= \sum_{v\in V} f(v,u_i) + \Delta
\end{aligned}$$

de donde

$$\sum_{v\in V} f'(u_i,v) = \sum_{v\in V} f'(v,u_i)$$

Por lo tanto, en este caso también se cumple la conservación del flujo, es decir

$$\sum_{v\in V} f'(u_i,v) - \sum_{v\in V} f'(v,u_i) = 0$$

El otro caso es análogo a éste.

Podemos concluir que el flujo f' también cumple con la conservación del flujo para cualquier vértice $u \in V - \{s,t\}$.

Ahora sólo nos falta demostrar que $|f'| > |f|$. Por definición,

$$|f'| = \sum_{v \in V} f'(s,v) - \sum_{v \in V} f'(v,s).$$

Si $a_1 = (s, u_1)$ entonces $f'(s, u_1) = f(s, u_1) + \Delta$; entonces

$$\sum_{v \in V} f'(s,v) = \sum_{v \in V} f(s,v) + \Delta$$

y

$$\sum_{v \in V} f'(v,s) = \sum_{v \in V} f(s,v).$$

Por lo tanto

$$|f'| = |f| + \Delta \text{ lo cual implica } |f'| > |f|.$$

En el otro caso. Si $a_1 = (u_1, s)$ entonces $f'(u_1, s) = f(u_1, s) - \Delta$, entonces

$$\sum_{v \in V} f'(v,s) = \sum_{v \in V} f(s,v) - \Delta$$

y

$$\sum_{v \in V} f'(s,v) = \sum_{v \in V} f(s,v)$$

Por lo tanto

$$|f'| = \sum_{v \in V} f(s,v) - (\sum_{v \in V} f(v,s) - \Delta)$$

Lo cual implica que $|f'| > |f|$.

Por lo tanto, contradice que f fuera un flujo máximo.

■

1.2.1 Flujo entre dos subconjuntos de vértices

Sea $G = (V, A, s, t, c)$ una red, X y Y dos subconjuntos del conjunto de vértices V; mencionaremos las propiedades del flujo que es enviado a través de ellos. Definimos el flujo que es enviado desde el conjunto X hasta el conjunto Y, con X y Y subconjuntos de V, como el flujo total que es enviado desde los vértices de X hasta los vértices de Y menos el flujo que es enviado desde los vértices de Y hasta los vértices de X. Formalmente tenemos:

Definición 1.8 *Sean $X, Y \subseteq V$, el flujo total que puede ser enviado desde X hasta Y, denotado por $f(X,Y)$, se define como:*

$$f(X,Y) = \sum_{\substack{\{(u,v)|u\in X \\ y\ v\in Y\}}} f(u,v) - \sum_{\substack{\{(u,v)|u\in Y \\ y\ v\in X\}}} f(u,v)$$

Lema 1.2 *Sea $G = (V, A, s, t, c)$ una red y f un flujo en G. Entonces*

1. $\forall X \subseteq V,\ f(X,X) = 0$
2. $\forall X, Y \subseteq V$ *tenemos* $f(X,Y) = -f(Y,X)$.
3. $\forall X, Y, Z \subseteq V$ *con* $X \cap Y = \emptyset$*, se cumple lo siguiente:*
   - $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$
   - $f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$

Demostración:

1. Por demostrar $f(X,X) = 0$

$$f(X,X) = \sum_{\{(u,v)|u,v\in X\}} f(u,v) - \sum_{\{(u,v)|u,v\in X\}} f(v,u)$$

   Cada arco (u,v) que aparece en la primera suma, vuelve a aparecer en la segunda.

$$\sum_{\{(u,v)|u,v\in X\}} f(u,v) = \sum_{\{(u,v)|v,u\in X\}} f(u,v)$$

   Por lo tanto $f(X,X) = 0$.

2. Por demostrar $f(X,Y) = -f(Y,X)$.

$$\begin{aligned}
\forall X, Y \subseteq V,\ f(X,Y) &= \textstyle\sum_{\{(u,v)|u\in X \text{ y } v\in Y\}} f(u,v) - \sum_{\{(u,v)|u\in Y \text{ y } v\in X\}} f(u,v) \\
&= -\left(\textstyle\sum_{\{(u,v)|u\in Y \text{ y } v\in X\}} f(u,v) - \sum_{\{(u,v)|u\in X \text{ y } v\in Y\}} f(u,v)\right) \\
&= -f(Y,X)
\end{aligned}$$

3. Por demostrar $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$

$$
\begin{aligned}
\forall X, Y, Z \subseteq V, \quad f(X \cup Y, Z) = \; & \textstyle\sum_{\{(u,v)|u \in X \cup Y \text{ y } v \in Z\}} f(u,v) - \sum_{\{(u,v)|u \in Z \text{ y } v \in X \cup Y\}} f(u,v) \\
= \; & \Big(\textstyle\sum_{\{(u,v)|u \in X \text{ y } v \in Z\}} f(u,v) + \sum_{\{(u,v)|u \in Y \text{ y } v \in Z\}} f(u,v) \Big) \\
& - \Big(\textstyle\sum_{\{(u,v)|u \in Z \text{ y } v \in X\}} f(u,v) + \sum_{\{(u,v)|u \in Z \text{ y } v \in Y\}} f(u,v) \Big) \\
= \; & \Big(\textstyle\sum_{\{(u,v)|u \in X \text{ y } v \in Z\}} f(u,v) - (\sum_{\{(u,v)|u \in Z \text{ y } v \in X\}} f(u,v) \Big) \\
& + \Big(\textstyle\sum_{\{(u,v)|u \in Y \text{ y } v \in Z\}} f(u,v) + \sum_{\{(u,v)|u \in Z \text{ y } v \in Y\}} f(u,v) \Big) \\
= \; & f(X,Z) + f(Y,Z)
\end{aligned}
$$

4. **Por demostrar** $f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$. La demostración es análoga al inciso anterior.

■

1.3 Red Residual

Una red residual es aquella que determina las capacidades sin usar de una red, dado un flujo f. Esta capacidad residual se puede calcular con base en las capacidades de los arcos y el flujo actual de la red.

Definición 1.9 *La* capacidad residual *de un arco* $(u,v) \in A$ *inducida por un flujo* f, $c_f(u,v)$, *es el flujo máximo adicional que puede ser enviado desde* u *hasta* v *usando los arcos* (u,v) *o* (v,u). *En otras palabras la capacidad residual es la capacidad no utilizada del arco* (u,v). *Formalmente, la capacidad residual de un vértice* u *a un vértice* v *inducida por un flujo* f *está dada por:*

$$c_f(u,v) = c(u,v) - f(u,v)$$

Explicando de manera intuitiva el concepto de red residual tenemos: dada una red G la red residual inducida por un flujo f es una red, denotada por G_f, en la cual el conjunto de vértices es el mismo que el de G, y el conjunto de arcos de G_f, denotado por E_f, está formado por aquellos arcos por los cuales todavía puede ser enviado más flujo. Esto significa que en la red residual sólo estarán los arcos cuya capacidad residual es mayor que cero. La definición formal de una red residual es la siguiente:

Definición 1.10 *Dada una red* $G = (V, A, s, t, c)$ *y un flujo* f, *la* red residual *de* G *inducida por* f *es la red* $G_f(V, A_f, s, t, c_f)$, *donde*

$$A_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$$

El problema del flujo máximo puede resolverse aumentando el flujo directamente en la red original o aumentándolo desde la red residual. Si aumentamos el flujo en la red original nos evitamos actualizar el flujo, dado que en cada aumento vamos sumándolo al valor del flujo f en G. Si aumentamos el flujo en la red residual, significa que debemos estar actualizando la capacidad residual de cada arco en la red residual en cada incremento del flujo que hagamos, y una vez que ya no podemos aumentar más el flujo debemos actualizar el flujo en la red original.

Observemos cuál es la relación que existe entre un aumento de flujo en la red residual y un aumento de flujo en la red original: sea f un flujo en la red original G y G_f la red residual inducida por G y f; dado que la capacidad residual de los arcos en la red residual se definió como:

$$c_f(u,v) = c(u,v) - f(u,v) \quad \text{y} \quad c_f(u,v) > 0.$$

Luego entonces, un aumento de flujo en la red residual de δ unidades implica alguno de los siguientes casos:

1. Que las unidades de flujo enviadas a través de un arco (u, v) en la red original, se incrementen.
2. O bien, que las unidades de flujo enviadas a través de un arco (u, v) en la red original, disminuya.

Recordemos que el flujo es distinto entre cada par de vértices; por lo tanto, un flujo puede provocar uno o más de estos efectos: quizás para algún arco el incremento del flujo en la red residual implique un aumento de flujo en la red original, y para otro el incremento implique un decremento de flujo en la red original.

El siguiente lema nos muestra cómo afecta al flujo total en la red original un aumento de flujo en la red residual:

Lema 1.3 *Sean $G = (V, A, s, t, c)$ una red y f un flujo en G. Sea G_f la red residual inducida por G y f, y sea f' un flujo en G_f. Entonces, la suma de $f + f'$ definida como:*

$$(f + f')(u, v) = f(u, v) + f'(u, v)$$

es un flujo en G con valor $|f + f'| = |f| + |f'|$.

Demostración:

Si $f + f'$ es un flujo en G debemos probar que cumple con la propiedad de restricción de capacidad, con la ley de conservación del flujo y que el valor del flujo $f + f'$ es igual al flujo que es enviado desde s y al flujo total que llega al vértice t.

Por demostrar que $f + f'$ es un flujo en G.

$$\begin{aligned} (f + f')(u, v) &= f(u, v) + f'(u, v) \\ &\leq f(u, v) + (c(u, v) - f(u, v)) \\ &= c(u, v) \end{aligned}$$

Por lo tanto $f' + f$ cumple con la restricción de la capacidad.

Como f y f' son flujos en G y G_f respectivamente, para cualquier vértice $u \in V - \{s,t\}$ se cumple:

$$
\begin{aligned}
\sum_{v\in V}(f+f')(u,v) - \sum_{v\in V}(f+f')(v,u) =& \sum_{v\in V}(f(u,v)+f'(u,v)) - \sum_{v\in V}(f(v,u)+f'(v,u)) \\
=& \left(\sum_{v\in V} f(u,v) + \sum_{v\in V} f'(u,v)\right) \\
& - \left(\sum_{v\in V} f(v,u) + \sum_{v\in V} f'(v,u)\right) \\
=& \left(\sum_{v\in V} f(u,v) - \sum_{v\in V} f(v,u)\right) \\
& + \left(\sum_{v\in V} f'(v,u) - \sum_{v\in V} f'(v,u)\right) \\
=& \; 0
\end{aligned}
$$

Por lo tanto también cumple con la conservación del flujo. Por demostrar que $|f+f'| = |f| + |f'|$.

$$
\begin{aligned}
|f+f'| =& \sum_{v\in V}(f+f')(s,v) - \sum_{v\in V}(f+f')(v,s) \\
=& \sum_{v\in V}(f(s,v)+f'(s,v)) - \sum_{v\in V}(f(v,s)+f'(v,s)) \\
=& \left(\sum_{v\in V} f(s,v) + \sum_{v\in V} f'(s,v)\right) - \left(\sum_{v\in V} f(v,s) + \sum_{v\in V} f'(s,v)\right) \\
=& \left(\sum_{v\in V} f(s,v) - \sum_{v\in V} f(v,s)\right) + \left(\sum_{v\in V} f'(s,v) + \sum_{v\in V} f'(v,s)\right) \\
=& \; |f| + |f'|
\end{aligned}
$$

Por lo tanto $f + f'$ es un flujo en G.

■

Para aumentar el flujo a través de una red residual debemos encontrar una trayectoria desde el vértice s hasta el vértice t. Esta trayectoria se llama *trayectoria aumentante.* La definición formal se presenta a continuación:

Definición 1.11 *Dada un red $G = (V, A, s, t, c)$ y un flujo f, una* trayectoria aumentante *p es una trayectoria de s a t en la red residual G_f.*

La capacidad de flujo para una trayectoria aumentante p en una red residual G_f, es la mínima de las capacidades de los arcos que la forman. A este valor le llamamos *capacidad residual de una trayectoria aumentante p*, denotado como $c_f(p)$; definimos formalmente la capacidad de una trayectoria aumentante como:

Definición 1.12 *La* capacidad residual de una trayectoria aumentante *p está definida de la siguiente manera:*

$$c_f(p) = \min\{c_f(u, v) : (u, v) \in p\}.$$

Dado que el flujo de cualquier arco nunca debe exceder la capacidad del mismo, y como el aumento de flujo en una red residual se hace mediante una trayectoria aumentante, entonces el valor del flujo que atraviesa la trayectoria siempre es menor a su capacidad. El siguiente lema nos muestra más claramente este resultado.

Lema 1.4 *Sean $G = (V, A, s, t, c)$ una red y f un flujo en G, y sea p una trayectoria aumentante en la red residual G_f. Definimos $f_p : A \to \mathbb{R}$ como:*

$$f_p(u, v) = \begin{cases} c_f(u, v) & si\ (u, v) \in p \\ -c_f(u, v) & si\ (v, u) \in p \\ 0 & en\ cualquier\ otro\ caso. \end{cases}$$

Luego entonces, f_p es un flujo en G_f con valor $|f_p| = c_f(p) > 0$.

Demostración:
Para cada arco $(u, v) \in A_f$ se cumple alguno de los siguientes casos:

- Si $(u, v) \in p$ entonces $f(u, v) = c_f(u, v)$, por lo tanto se cumple la restricción de la capacidad.
- Si $(u, v) \notin p$ entonces $f(u, v) = 0$, y también se cumple que el flujo sea menor que la capacidad.

Para todos los vértices u, v que no están en p se cumple la ley de conservación del flujo, dado que $f_p(u, v) = 0$. Lo interesante es ver qué pasa cuando un vértice u está en p, para lo cual tenemos dos casos.

Sea

$$p = u_0, a_1, u_1, \ldots, u_{n-1}, a_n, u_n$$

donde $u_0 = s$ y $u_n = t$.

Caso $u \in Q$: Sea $u = u_i$ con $0 < i < n$ (esto es, u no es ni s ni t). Observemos que los arcos (u_{i-1}, u_i) y (u_i, u_{i+1}) están en p, lo cual implica que (u_{i+1}, u_i) y (u_i, u_{i-1}) no están en p. Sea $V' = V - \{u_{i-1}, u_{i+1}\}$. Entonces tenemos

$$\sum_{v \in V} f_p(u_i, v) = f_p(u_i, u_{i-1}) + f_p(u_i, u_{i+1}) + \sum_{v \in V'} f(u_i, v)$$
$$= (c_p(u_i, u_{i-1}) + (-c_p(u_i, u_{i+1})) + \sum_{v \in V'} f(u_i, v)$$
$$= \sum_{v \in V'} f(u_i, v)$$

y

$$\sum_{v \in V} f_p(v, u_i) = f_p(u_{i-1}, u_i) + f_p(u_{i+1}, u_i) + \sum_{v \in V'} f(v, u_i)$$
$$= (c_p(u_{i-1}, u_i) + (-c_p(u_{i+1}, u_i)) + \sum_{v \in V'} f(v, u_i)$$
$$= \sum_{v \in V'} f(v, u_i)$$

Podemos concluir que para cualquier vértice $u \in V - \{s, t\}$ se cumple la conservación del flujo, esto es:

$$\sum_{v \in V} f_p(u_i, v) - \sum_{v \in V} f_p(v, u_i) = 0$$

Por lo tanto, f_p es un flujo en G_f.

■

Un resultado inmediato es el siguiente:

Corolario 1.1 *Sea $G = (V, A, s, t, c)$ una red, f un flujo en G, y sea p una trayectoria de aumento en la red residual G_f. Definimos $f' : A \to \mathbb{R}$ como $f' = f + f_p$. Entonces, f' es un flujo en G con valor $|f'| = |f| + |f_p| > |f|$.*

Demostración:
La demostración es inmediata de los lemas 1.3 y 1.4.

■

1.4 Cortes

Otro concepto importante dentro de las redes es el de *corte*, ya que el problema de flujo se puede plantear también en el marco de los cortes. Un *corte* se puede definir como una partición (recordemos que en una partición los subconjuntos que se forman son ajenos) del conjunto de vértices V en dos subconjuntos.

Definición 1.13 *Un* corte *es una partición del conjunto de los vértices* V *en dos subconjuntos* S *y* $\overline{S} = V - S$. *Cada corte define un conjunto de arcos los cuales tienen un vértice en* S *y el otro vértice en* $\overline{S}$. *Por lo tanto, nos referimos a este conjunto de arcos como* corte *y lo denotamos por* $[S, \overline{S}]$.

Decimos que una arco (u, v) es de *avance* en un corte $[S, \overline{S}]$ si el vértice u pertenece a S y el vértice v pertenece a $\overline{S}$, y el arco es de *retroceso* cuando v está en S y u en $\overline{S}$. Dicho de otra forma, y tomando en cuenta que estamos trabajando en redes podemos decir que una arco es de *avance* si a través de él se puede enviar flujo desde el conjunto S hasta el conjunto $\overline{S}$, y es de retroceso si a través de él se envía flujo desde el conjunto de vértices $\overline{S}$ hasta el conjunto S.

Los cortes que a nosotros nos interesan son aquellos en los cuales el vértice origen s y el vértice destino t están en diferentes conjuntos, para que tenga sentido hablar del flujo que es enviado a través de un corte. A este tipo de corte le llamaremos un *s-t corte*. La definición formal es la siguiente:

Definición 1.14 *Un* s-t corte $[S, T]$ *de una red* $G = (V, A, s, t, c)$ *es una partición del conjunto de vértices* V *en los conjuntos* S *y* $T = V - S$ *tal que* $s \in S$ *y* $t \in T$.

La razón por la cual nos interesan los $s-t$ cortes para resolver el problema del flujo máximo es la siguiente: el valor de un flujo siempre está acotado por la capacidad de cualquier $s - t$ corte (este resultado será demostrado más adelante); si logramos encontrar un flujo cuyo valor sea igual al de un $s - t$ corte habremos encontrado un flujo máximo. Ahora bien, la capacidad de un corte no es más que la suma de las capacidades de los arcos que lo forman. Un $s-t$ corte es mínimo cuando su capacidad es menor que la de cualquier otro $s - t$ corte. Las definiciones formales son las siguientes:

Definición 1.15 *La* capacidad *de un* $s-t$ *corte* $[S, T]$, *denotada por* $c[S, T]$, *es la suma de las capacidades de los arcos de avance. Esto es*

$$c[S, T] = \sum_{\substack{\{(u,v) | u \in S, \\ v \in T\}}} c(u, v)$$

Definición 1.16 *Un* $s-t$ *corte mínimo de una red es un* $s-t$ *corte cuya capacidad es mínima.*

Dado un $[S, T]$ corte en una red G con flujo f, la capacidad residual de $[S, T]$, denotada por $c_f[S, T]$, es el máximo flujo restante que puede ser enviado desde el conjunto S hasta T tomando en cuenta que ya enviamos un flujo f: en otras palabras, es la suma de las capacidades residuales de los arcos por las que podemos enviar flujo desde S hasta T. Formalmente la capacidad residual de un corte se define de la siguiente manera:

Definición 1.17 *La* capacidad residual de un $s - t$ corte *es la suma de las capacidades residuales de los arcos de avance en el corte* $[S, T]$. *Esto es*

$$c_f[S, T] = \sum_{\substack{\{(u,v) | u \in S, \\ v \in T\}}} c_f(u, v)$$

El siguiente resultado nos muestra que el valor del flujo que puede ser enviado a través de un $s - t$ corte es igual al valor del flujo que puede ser enviado desde el vértice s hasta el vértice t.

Lema 1.5 *Sea f un flujo en $G = (V, A, s, t, c)$ y $[S, T]$ un corte en G. Entonces, el valor del flujo neto que atraviesa el corte $[S, T]$, denotado por $f(S, T)$, es igual al valor del flujo f.*

Demostración:
Por el lema 1.2 tenemos lo siguiente:

$$\begin{aligned} f(S, V) &= f(S, S \cup T) \\ &= f(S, S) + f(S, T) \end{aligned}$$

Despejando $f(S, T)$, obtemos:

$$\begin{aligned} f(S, T) &= f(S, V) - f(S, S) \\ &= f(S, V) \\ &= f(\{s\}, V) + f(S - \{s\}, V) \\ &= f(\{s\}, S) \\ &= |f| \end{aligned}$$

■

Un resultado inmediato del lema anterior es el siguiente corolario.

Corolario 1.2 *Sea f un flujo en $G = (V, A, s, t, c)$ y $[S, T]$, el valor del flujo f es igual al flujo total enviado al vértice t.*

Demostración:
Sea $S = V - \{t\}$ y $T = \{t\}$, es claro que $[S, T]$ es un $s - t$ corte en G. Por el lema anterior sabemos:

$$\begin{aligned} |f| = &\ f(S, T) \\ = &\sum_{\substack{\{(u,v)|u \in S \\ y\ v \in T\}}} f(u, v) - \sum_{\substack{\{(u,v)|u \in T \\ y\ v \in S\}}} f(u, v) \\ = &\sum_{\{(u,t)|u \in S\}} f(u, t) - \sum_{\{(t,v)|v \in S\}} f(t, v) \\ = &\sum_{u \in V} f(u, t) - \sum_{\{(t,v)|v \in V\}} f(t, v) \end{aligned}$$

Por lo tanto, el valor de f es igual al flujo total que entra a t.

■

Ya habíamos hablado de la relación que hay entre la capacidad de un corte y el valor de un flujo. El siguiente lema nos muestra que para cualquier flujo f, el valor de éste siempre es menor o igual que la capacidad de cualquier corte, es decir, la capacidad de cualquier corte es una cota superior para el valor de un flujo.

Lema 1.6 *El valor de cualquier flujo f en una red $G = (V, A, s, t, c)$ es menor o igual que la capacidad de cualquier $s - t$ corte de G.*

Demostración:

Sea f un flujo en G y $[S, T]$ cualquier $s - t$ corte de G. Por el lema anterior tenemos:

$$\begin{aligned} |f| &= f(S,T) \\ &= \sum_{\substack{\{(u,v) \mid u \in S \\ y\ v \in T\}}} f(u,v) - \sum_{\substack{\{(u,v) \mid v \in S \\ y\ u \in T\}}} f(v,u) \end{aligned}$$

Además, sabemos que $f(u,v) \leq c(u,v)$ para cualquier arco en A, entonces

$$|f| \leq \sum_{\substack{(u,v) \mid u \in S, \\ v \in T\}}} c(u,v) = c(S,T)$$

■

Si el valor del flujo f es igual que la capacidad de un $s - t$ corte, entonces podemos decir que f es un flujo máximo, además, un flujo f es un flujo máximo si en la red residual inducida por f no existe ninguna trayectoria aumentante.

Teorema 1.1 (Teorema del Flujo Máximo y Corte Mínimo)

Si f es un flujo en una red $G = (V, A, s, t, c)$, entonces las siguientes condiciones son equivalentes:

1. *f es un flujo máximo de G.*
2. *La red residual de G_f no contiene una trayectoria aumentante.*
3. *$|f| = c(S,T)$ con $[S,T]$ un $s - t$ corte mínimo en G.*

Demostración:

- (1) $\rightarrow$ (2)

  La demostración se realizará por contradicción. Supongamos que f es un flujo máximo en G pero existe una $s \rightsquigarrow t$ trayectoria aumentante p en la red residual G_f. Sabemos que la suma de los flujos f y f_p es un flujo en G con valor $|f + f_p| = |f| + |f_p|$, y como el valor de $|f_p|$ es mayor que cero, entonces $|f + f_p| > |f|$ lo cual contradice que f sea un flujo máximo. Por lo tanto G_f no contiene ninguna trayectoria aumentante.

- (2) $\rightarrow$ (3)

  Supongamos que la red residual inducida por f no contiene ninguna trayectoria aumentante. Definimos el corte $[S, T]$ de la siguiente manera: sea S el conjunto de todos los vértices $u \in V$ para los cuales existe una trayectoria desde s hasta u en la red residual G_f y T es el resto de los vértices, esto es

$$S = \{u \in V \mid \text{existe una trayectoria desde } s \text{ hasta } u \text{ en } G_f\}$$
$$T = V - S$$

[S,T] es un $s-t$ corte dado que el vértice $s \in S$; por hipótesis sabemos que de s a t no hay una trayectoria, por lo que t no puede estar en S, entonces $t \in T$.

La capacidad residual de cualquier arco (u,v) con $u \in S$ y $v \in T$ es igual a cero, porque si no fuera así existiría el arco (u,v) en la red residual G_f y existiría un camino desde s hasta v. Para los arcos (u,v), con $u,v \in S$, la capacidad residual es mayor que cero y por lo tanto $(u,v) \in A_f$.

Por el lema 1.6 sabemos que el valor de cualquier flujo siempre es menor que la capacidad de cualquier $s-t$ corte. Por lo tanto, podemos concluir:

$$|f| = f(S,T) = c(S,T)$$

- (3) $\rightarrow$ (1)

  Suponemos que dado un corte $[S,T]$ y un flujo f se cumple $|f| = c(S,T)$. Por el lema 1.6 sabemos que el valor de cualquier flujo siempre está acotado por la capacidad de cualquier $s-t$ corte, Como el valor del flujo es igual a la capacidad de un corte $[S,T]$ entonces tal valor tiene que ser máximo. Por lo tanto, f es un flujo máximo.

■

Recordemos que una de las condiciones que le pedimos a la red es que la capacidad de los arcos fueran números enteros no negativo; de ahí podemos asegurar que el flujo máximo es un número entero.

Lema 1.7 *Sea $G = (V, A, s, t, c)$, si todos los arcos $(u,v) \in A$ son enteros, entonces G tiene un flujo máximo entero.*

Demostración:

Por hipótesis sabemos que la capacidad de los arcos es un número entero mayor que cero; además, antes de aumentar el flujo en G, el valor del flujo f es cero, para cualquier flujo en G. El aumento del valor del flujo f depende de la capacidad de los arcos, pero siempre tomamos la capacidad mínima de una trayectoria aumentante, la cuál es un número entero. Por la cerradura de los enteros, podemos asegurar que el valor del flujo máximo es un número entero.

■

Con esto terminamos la presentación de los resultados de la Teoría de Redes que vamos a utilizar en el desarrollo del algoritmo genérico y en las pruebas de correctez.

Capítulo 2

Algoritmo Genérico de Flujo Máximo

Existen dos diferentes formas de encontrar el flujo máximo en una red: la primera es con una trayectoria del vértice s al vértice destino t, y aumentar el flujo por medio de ésta; y la segunda consiste en empujar el flujo a través de cada arco, comenzando con los arcos en los cuales el vértice s es el primer componente —es decir, arcos de la forma $(s, v) \in A$ con $v \in V$— y empujar la mayor cantidad de flujo posible desde los vértices v a sus vértices vecinos, hasta llegar al vértice t.

En este capítulo presentaremos un algoritmo genérico para resolver el problema de flujo máximo, así como una implementación del mismo.

2.1 El algoritmo

El *Algoritmo Genérico de Flujo Máximo* (AGFM) recibe como entrada una red $G = (V, A, c, s, t)$ que cumple con las siguientes condiciones:

- Cada vértice $u \in V$ tiene asociada su lista de incidencias, es decir, la lista de arcos de la forma $(u, v) \in A$ con $v \in V$.

- El conjunto de arcos A está dado por la unión de todas las listas de incidencia de todos los vértice $u \in V$, esto es,

$$A = \bigcup_{u \in A} Ad(u)$$

- Cada arco debe tener asociada su capacidad, la cual debe ser un número entero no negativo.

- La red debe tener dos vértices distinguidos s y t, los cuales corresponden al vértice origen y destino respectivamente.

La implementación del algoritmo se presenta a continuación:

Listado 2.1.1 Algoritmo Genérico de Flujo Máximo

```
1 public abstract class AGFM {
2
3     protected Red r;
4     protected int flujoMaximo;
5
6     public AGFM (Red r) {
7         this.r        = r;
8         flujoMaximo   = 0;
9     }
10
11    protected int algoritmoGenericoDeFlujoMaximo () {
12        inicializa ();
13        while(esPosibleIncrementarElFlujo ()){
14            incrementaFlujo();
15        }
16        return valorDelFlujoMaximo ();
17    }
18
19    protected abstract void inicializa ();
20
21    protected abstract boolean esPosibleIncrementarElFlujo ();
22
23    protected abstract void incrementaFlujo ();
24
25    protected abstract int valorDelFlujoMaximo ();
26 }
```

Pasamos a demostrar la correctez del algoritmo genérico "exigiendo" de los métodos abstractos que cumplan con ciertas precondiciones y postcondiciones.

2.1.1 Precondiciones y postcondiciones de los métodos de AGFM

Constructor

(public AGFM (Red r))
Valida el estado inicial del algoritmo, asignando un valor inicial a sus atributos.

Precondiciones: La clase debe ser invocada con una red.

Postcondiciones: Deja una representación de la red que cumple con lo siguiente:

1. Cada red tiene asociada dos gráficas, la red original que es la representación inicial de la red, o sea con flujo igual a cero, y la red residual que es una red auxiliar para ir incrementando el flujo. En un principio son equivalentes.
2. Cada vértice de la red tiene asociada su lista de adyacencias.
3. Cada arco tiene asociada una capacidad entera no negativa.
4. Cada arco tiene asociado un flujo, inicialmente cero.

Es Posible Incrementar el Flujo

(**protected abstract boolean** esPosibleIncrementarElFlujo ())
Este método revisa si es posible incrementar el flujo de la red, ya sea por medio de una trayectoria aumentante, o a través de un empuje, según sea el caso.

Precondiciones: Ninguna.

Postcondiciones: El método regresa **false** si el flujo actual es máximo; en caso contrario regresa **true**.

Incrementa el Flujo

(**protected abstract void** incrementaFlujo())
El método aumenta el flujo del vértice origen al vértice destino.

Precondiciones: Es posible incrementar el flujo; es decir, el flujo actual no es máximo.

Postcondiciones: Sea $flujoMaximo'$ igual al valor de r.flujoMaximo antes de que el método fuera invocado. Una vez que el método fue ejecutado tenemos:

$$\text{r.flujoMaximo} > flujoMaximo'.$$

Valor del Flujo Máximo

(**protected abstract int** valorDelFlujoMaximo ())
Este método actualiza el valor del flujo máximo y regresa dicho valor.

Precondiciones: Ya no es posible incrementar el flujo en la red.

Postcondiciones: El valor de r.flujoMaximo es el valor del flujo máximo que se encontró, que a su vez es el valor que regresa el método.

Inicializa

(**protected abstract void** inicializa ())
En la versión abstracta no hay necesidad de ningún preproceso.

Precondiciones: Ninguna

Postcondiciones: Ninguna

Algoritmo Genérico de Flujo Máximo

(**protected int** algoritmoGenericoDeFlujoMaximo ())
Este método revisa si es posible incrementar el flujo en la red, y mientras sea posible, lo aumenta. El método termina cuando no es posible seguir aumentando el flujo.

Precondiciones: Ninguna.

Postcondiciones: Una vez terminado el algoritmo, se cumple que

$$\text{flujoMaximo} \geq |f| \qquad \forall f \text{ un flujo de la red.}$$

2.1.2 Correctez de AGFM

Para demostrar que el Algoritmo Genérico de Flujo Máximo es correcto, debemos probar las siguientes condiciones:

1. Siempre termina.
2. Dada una red, el algoritmo siempre encuentra el flujo máximo que puede ser enviado desde s hasta t.

Suponiendo que cada uno de los métodos abstractos: inicializa, esPosibleIncrementarElFlujo, incrementaFlujo y valorDelFlujoMaximo siempre terminan y además devuelven el resultado correcto, esto es, se cumple las precondiciones correspondientes, podemos demostrar lo siguiente:

- **Siempre Termina.**

  Para demostrar que AGFM siempre termina, basta con demostrar que el ciclo **while** termina. La condición para que el **while** se ejecute es que sea posible aumentar el flujo, esto es equivalente a decir que existe una trayectoria del vértice origen al vértice destino.

  Como el valor de cualquier flujo siempre es mayor que cero y el flujo que atraviesa un arco no puede exceder su capacidad, podemos observar lo siguiente:

  $$|f| = \sum_{v\in V} f(s,v) - \sum_{v\in V} f(v,s) \leq \sum_{v\in V} c(s,v)$$

  De lo anterior, podemos concluir que el valor del flujo máximo está acotado por un número entero, y dado que en cada iteración del **while** se incrementa el valor del flujo, en algún momento ya no existirá una trayectoria aumentante del vértice s al vértice t, y por lo tanto no se cumplirá la condición del **while** y se termina el ciclo. Por lo tanto, concluimos que algoritmo siempre termin.

- **Dada una red, el algoritmo siempre encuentra el flujo máximo que puede ser enviado desde s hasta t.** Podemos ver que el algoritmo únicamente termina si no es posible aumentar el flujo actual, lo cual significa que no existe una trayectoria aumentante en la red residual y por el Teorema del Flujo Máximo y Corte Mínimo (página 16), el flujo actual es máximo.

2.1.3 Complejidad de AGFM

La complejidad de AGFM depende del tamaño de la red y de la complejidad de los métodos abstractos. Como a este nivel no se tiene más información acerca de los métodos no podemos definir la complejidad exacta del algoritmo por lo que definimos la siguiente función de complejidad para AGFM.

Sea p el número de veces que se ejecuta el ciclo **while** —el valor de p es diferente para cada una de las especializaciones concretas de AGFM, por lo que será determinada en cada una de ellas— y

$$f_{inicializa}, f_{esPosibleIncrementarElFlujo}, f_{incrementaFlujo} \text{ y } f_{valorDelFlujoMaximo}$$

son las funciones de complejidad de los métodos abstractos inicializa, esPosibleIncrementarElFlujo, incrementaFlujo y valorDelFlujoMaximo, respectivamente. Entonces, la complejidad del algoritmo está dada por la siguiente "ecuación general".

$$f_{AGFM} = f_{inicializa} + \sum_{i=1}^{q}(f_{esPosibleIncrementarElFlujo} + f_{incrementaFlujo}) + f_{valorDelFlujoMaximo}.$$

Para cada especialización de AGFM se determina el valor de cada una de las funciones de complejidad de los métodos abstractos y se obtiene la complejidad del algoritmo concreto. Para el algoritmo genérico es todo lo que podemos definir ya que lo demás se define en cada una de las especializaciones de AGFM.

Capítulo 3

Algoritmos de Trayectoria Aumentante

En este capítulo presentaremos un algoritmo genérico basado en la idea de encontrar el flujo máximo a través de trayectorias aumentantes. Este algoritmo es una extensión de AGFM, con la peculiaridad de que verificar si es posible incrementar el flujo es equivalente a revisar si existe una trayectoria aumentante del vértice origen al vértice destino, e incrementar el flujo equivale a obtener la capacidad de la trayectoria aumentante que encontró y aumentar las unidades de flujo correspondientes por dicha trayectoria.

En secciones posteriores del capítulo presentaremos las versiones concretas del *Algoritmo Genérico de Trayectoria Aumentante* (AGTA).

3.1 Algoritmo Genérico de Trayectoria Aumentante

La idea principal del algoritmo es aumentar el flujo a través de una trayectoria aumentante del vértice s al vértice t; por lo tanto, mientras le es posible busca dicha trayectoria, incrementa el flujo y actualiza la red residual correspondiente a la red original y al flujo actual. Una vez que no existe ninguna trayectoria aumentante, el algoritmo termina asegurando que el flujo encontrado hasta el momento es un flujo máximo. Este resultado será demostrado para cada versión concreta del algoritmo genérico.

El ejemplo mostrado en la figura 3.1 en la página 26 nos ilustra este algoritmo; a continuación se explica un poco el procedimiento que se sigue para encontrar el flujo máximo: sea la red inicial presentada en la figura 3.1(a). Supongamos que el algoritmo elige la trayectoria $1-2-5$ con capacidad igual al $\min\{c(1,2), c(2,5)\} = 4$. Este aumento reduce $c(1,2) = 0$ lo que provoca que el arco $(1,2)$ sea eliminado de la red residual. El resultado de este aumento se presenta en la figura 3.1(b). La siguiente trayectoria por donde incrementamos el flujo es $1-3-5$, con capacidad igual al $\min\{c(1,3), c(3,5)\} = 1$; este incremento reduce la capacidad de $(3,5)$ a cero y por lo tanto $(3,5)$ desaparece. Hasta este momento se han enviado 5 unidades de flujo, el resultado del incremento se muestra en la figura 3.1(c). El siguiente incremento es a través de la trayectoria $1-4-5$, cuya capacidad es igual al $\min\{c(1,4), c(4,5)\} = 4$; por lo tanto, las unidades de flujo enviadas desde el vértice origen al vértice destino hasta el momento son 9 y los dos arcos $(1,4)$ y $c(4,5)$ se eliminan de la red residual. El siguiente aumento se hace por medio de la trayectoria $1-3-2-5$, la capacidad de esta trayectoria es igual al $\min\{c(1,3), c(3,2), c(2,5)\} = 1$, este aumento elimina los arcos $(1,3)$ y $(2,5)$ de la red residual resultante. El total de flujo enviado hasta este momento es de 10 unidades; el resultado de

este incremento se presenta en la figura 3.1(b). Dado que ya no es posible construir ninguna trayectoria del vértice origen, en este caso el vértice 1, al vértice destino, vértice 5, concluimos que el flujo actual es máximo y su valor es de 10 unidades.

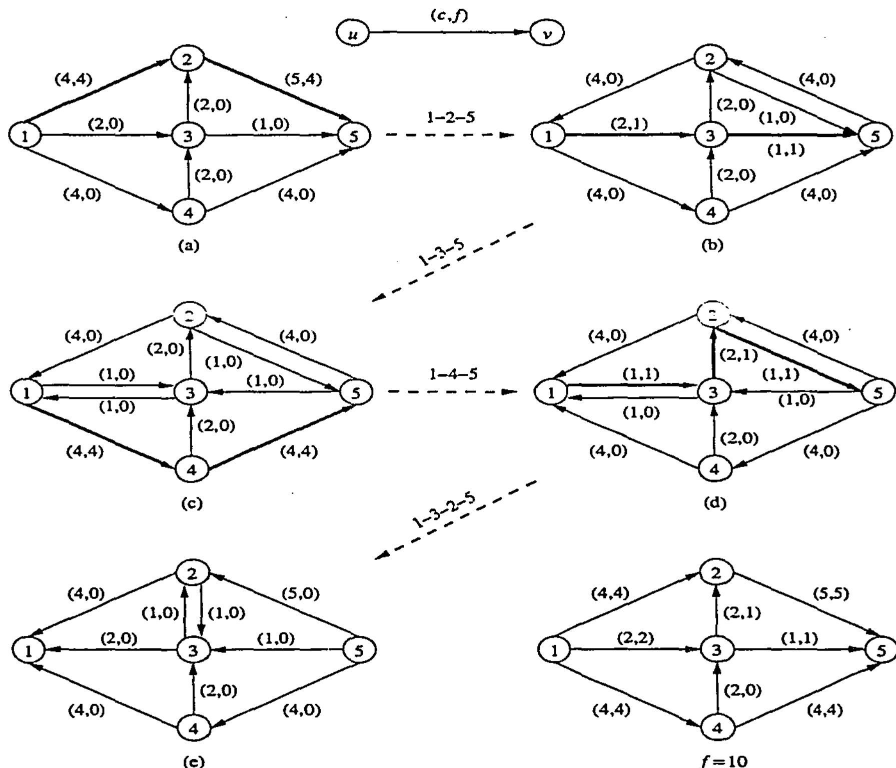


Figura 3.1: (a) red inicial. (b) resultado de incrementar el flujo a través de la trayectoria $1-2-5$ cuya capacidad es 4. (c) resultado de incrementar el flujo en una unidad a través de la trayectoria $1-3-5$. (c) red después de incrementar el flujo en una unidad a través de la trayectoria $1-4-5$, con flujo= 4. (d) resultado del último incremento de flujo realizado a través de la trayectoria $1-3-2-5$. Valor del flujo máximo: 10 unidades.

En la implementación del algoritmo la red es r, la representación inicial de la red está alma-

cenada en r.redInicial, el incremento del flujo en cada iteración del algoritmo estará almacenado en la variable flujoMaximo, y la red residual correspondiente a este flujo es r.redResidual.

La implementación del algoritmo se presenta a continuación:

Listado 3.1.1 Algoritmo Genérico de Trayectoria Aumentante

```
1  public abstract class AGTA extends AGFM {
2
3      protected LinkedList trayectoria;
4      protected int cTrayectoria;
5
6      public AGTA (Red r) {
7          super (r);
8          trayectoria  = new LinkedList();
9          cTrayectoria = 0;
10     }
11
12     protected boolean esPosibleIncrementarElFlujo () {
13         return encuentraTrayectoriaAumentante ();
14     }
15
16     protected void incrementaFlujo () {
17         encuentraCapacidadMinimaDeLaTrayectoria ();
18         actualizaRedResidual();
19         aumentaFlujo();
20     }
21
22     protected int  valorDelFlujoMaximo (){
23         return flujoMaximo;
24     }
25
26     protected  void encuentraCapacidadMinimaDeLaTrayectoria () {
27         Vertice u      = (Vertice)trayectoria.get(0);
28         Vertice v      = (Vertice)trayectoria.get(1);
29         int      i, tmp;
30         cTrayectoria =  u.capacidad(v);
31         for (i = 1; i < (trayectoria.size() - 1); i++) {
32             u = (Vertice)trayectoria.get(i);
33             v = (Vertice)trayectoria.get(i+1);
34
35             tmp = u.capacidad(v);
36             if (tmp < cTrayectoria)
37                 cTrayectoria = tmp;
38         }
39     }
40
41     protected void aumentaFlujo (){
42         flujoMaximo += cTrayectoria;
43         LinkedList verticesRedResidual = r.redResidual().obtenVertices();
44         LinkedList verticesRedInicial  = r.redInicial().obtenVertices();
```

Listado 3.1.1 Algoritmo Genérico de Trayectoria Aumentante (continuación)

```
45            int i = 0;
46            for (i = 0; i < (trayectoria.size()-1); i++) {
47                int indiceU =
48                    verticesRedResidual.indexOf((Vertice)trayectoria.get(i));
49                int indiceV =
50                    verticesRedResidual.indexOf((Vertice)trayectoria.get(i+1));
51                Vertice u = r.redInicial().obten(indiceU);
52                Vertice v = r.redInicial().obten(indiceV);
53                Arista a = u.obtenArista(v);
54                if(a != null) {
55                    a.incrementaFlujo(cTrayectoria);
56                } else {
57                    a = v.obtenArista(u);
58                    if (a != null)
59                        a.decrementaFlujo(cTrayectoria);
60                }
61            }
62        }
63
64        protected void actualizaRedResidual () {
65            int i = 1;
66            Vertice v1, v2;
67            for (i = 0; i < trayectoria.size()-1; i++) {
68                v1 = (Vertice)trayectoria.get(i);
69                v2 = (Vertice)trayectoria.get(i+1);
70                v1.decrementaCapacidad(v2,cTrayectoria);
71                if(v2.estaConectadoCon(v1))
72                    v2.incrementaCapacidad(v1,cTrayectoria);
73                else
74                    v2.conectaCon(v1,cTrayectoria);
75            }
76        }
77
78        protected void construyeTrayectoria(){
79            VerticeEtiquetado v = (VerticeEtiquetado)r.verticeDestino();
80            trayectoria.add (v);
81            while (v != r.verticeOrigen()) {
82                v = v.obtenPadre();
83                trayectoria.addFirst(v);
84            }
85        }
86
87        protected abstract boolean encuentraTrayectoriaAumentante ();
88 }
```

3.1.1 Precondiciones y postcondiciones de los métodos de AGTA

Constructor

(public AGTA (Red r)):

Valida el estado inicial del algoritmo, asignando un valor inicial a sus atributos.

Precondiciones: La clase debe ser invocada con una red.

Postcondiciones: Deja una representación de la red que cumple con las mismas condiciones expuestas en AGFM. Incluyendo las siguientes condiciones:

- Inicializa la trayectoria aumentante como vacía (es decir, trayectoria = **null**) y la capacidad de ésta la inicializa en cero, cTrayectoria = 0.
- Cada vértice tiene asociado un vértice padre que es el vértice desde el cual se exploró. Inicialmente es nulo.

Incrementa Flujo

(**protected void** incrementaFlujo ())
Dada una trayectoria aumentante desde el origen hasta el destino en la red residual, el método aumenta el flujo a través de ella y actualiza la red residual.

Precondiciones: Existe una trayectoria aumentante (trayectoria $\neq$ **null**).

Postcondiciones: Sea $flujoMaximo'$ igual al valor de flujoMaximo antes de que el método fuera invocado. Una vez que el método fue ejecutado tenemos:

$$flujoMaximo' < \text{flujoMaximo}.$$

Es Posible Incrementar el Flujo

(**protected boolean** esPosibleIncrementarElFlujo ())
Dado que este método únicamente llama a encuentraTrayectoriaAumentante, las precondiciones y postcondiciones son las mismas.

Encuentra una Trayectoria Aumentante

(**protected abstract boolean** encuentraTrayectoriaAumentante ())
El método busca una trayectoria aumentante en la red de residual; si la encuentra, actualiza la variable trayectoria con los vértices de la trayectoria que encontró. Si encuentra la trayectoria aumentante regresa **true**, y regresa **false** en caso contrario.

Precondiciones: Ninguna.

Postcondiciones: Si el método regresa **true**: trayectoria es una trayectoria aumentante. Si el método regresa **false**: no existe una trayectoria aumentante (trayectoria = **null**).

Construye Trayectoria Aumentante

(**protected void** construyeTrayectoria ())
El método construye una trayectoria aumentante siempre y cuando el padre del vértice destino esté definido.

Precondiciones: padre(t),padre(padre(t)), ..., padre(...padre(t)...) $\neq$ **null**.

Postcondiciones: trayectoria es una trayectoria aumentante del vértice s al vértice t.

Encuentra la capacidad mínima de la trayectoria aumentante

(**protected void** encuentraCapacidadMinimaDeLaTrayectoria ())
El método actualiza el valor de cTrayectoria, le asigna el valor de la capacidad mínima de los arcos (u, v) que pertenecen a la trayectoria, esto es, encuentra la capacidad de la trayectoria aumentante, siempre y cuando ésta exista.

Precondiciones: Existe una trayectoria aumentante (trayectoria $\neq$ **null**).

Postcondiciones: Si $\delta = \min\{x.capacidad | x = (u, v)$ y $x \in$ trayectoria $\}$, entonces

$$\text{cTrayectoria} = \delta.$$

Incrementar el flujo en la Red

(**protected void** aumentaFlujo ())
El método aumenta el flujo a través de la red e incrementa el valor del flujoMaximo, siempre y cuando exista una trayectoria aumentante.

Precondiciones: Existe una trayectoria aumentante (trayectoria $\neq$ **null**) con capacidad igual a cTrayectoria.

Postcondiciones: Una vez que aumentaFlujo fue ejecutado se cumplen las siguientes condiciones:

- Sea $flujoMaximo'$ igual al valor de flujoMaximo antes de que el método fuera invocado. Una vez que el método fue ejecutado tenemos:

$$\text{flujoMaximo} = flujoMaximo' + \text{cTrayectoria}$$

- Para los arcos $(u, v) \in$ trayectoria se cumple una de las siguientes opciones:
  1. Si $(u, v) \in$ r.redInicial entonces se incrementa el flujo de (u, v) cTrayectoria unidades.
  2. Si $(u, v) \notin$ r.redInicial entonces se decrementa el flujo de (u, v) cTrayectoria unidades.

Actualizar la Red Residual

(**protected void** actualizaRedResidual ())
Dada una trayectoria aumentante y un flujo mayor que cero, el método actualiza la capacidad residual de los arcos (u, v) tales que u y v están en la trayectoria.

Precondiciones: Existe una trayectoria aumentante (trayectoria $\neq$ **null**).

Postcondiciones: Sea $x = (u, v)$, con (u, v) en los arcos de la red r, definimos $x.capacidad'$ igual al valor de x.capacidad antes de que el método fuera invocado. Una vez que el método fue ejecutado se cumple la siguiente condición:

Para todo arco $x = (u, v)$ se cumple:

- x.capacidad $< x.capacidad'$ si $(u, v) \in$ trayectoria.
- x.capacidad $> x.capacidad'$ si $(u, v) \notin$ trayectoria pero u,$v \in$ trayectoria.
- x.capacidad $= x.capacidad'$ en cualquier otro caso.

3.1.2 Correctez de AGTA

Antes de demostrar la correctez del Algoritmo Genérico de Trayectoria Aumentante, demostraremos que los métodos aumentaFlujo y encuentraCapacidadMinimaDeLaTrayectoria cumplen las postcondiciones. Las demostración correspondiente al método abstracto se harán más adelante en cada una de las especializaciones concretas del algoritmo AGTA, que es donde se implementan estos métodos.

Lema 3.1 *Si existe una trayectoria aumentante (trayectoria* $\neq$ **null***) con capacidad cTrayectoria* > 0*, al terminar de ejecutarse el método aumentaFlujo se cumple lo siguiente:*

1. *El valor de flujoMaximo se incrementa cTrayectoria unidades.*

2. *Para los arcos* $(u, v) \in$ *trayectoria se cumple una de las siguientes opciones:*

   - *Si* $(u, v) \in$ *r.redInicial entonces se incrementa el flujo de* (u, v) *cTrayectoria unidades.*
   - *Si no está, y existe el arco* $(v, u) \in$ *r.redInicial, entonces se decrementa el flujo de* (v, u) *cTrayectoria unidades.*
   - *En cualquier otro caso, la capacidad del arco* (u, v) *permanece igual.*

Demostración:

Para demostrar 1 basta que observemos en qué lugar del método se ve modificado el valor de flujoMaximo, y esto solamente ocurre una vez (línea 42), cuando al valor de la variable se le suma el valor de cTrayectoria.

Por demostrar 2. Sea trayectoria [1] $= v_0, v_1, v_2, \ldots, v_{n-1}, v_n$. Veamos que el **for** se repite a lo más el tamaño de la trayectoria aumentante trayectoria. Observemos que el contador empieza desde cero; entonces, en la primera ejecución del ciclo obtenemos los vértices u y v que corresponden al primer vértice(línea 51) y segundo vértices (línea 52) de la trayectoria, respectivamente. Después obtenemos el arco correspondiente a los vértices u y v (línea 53). Si $(u, v) \in$ r.redInicial (línea 54), aumentamos el flujo de el arco cTrayectoria unidades (línea 55). Si no, verificamos que el arco $(v, u) \in$ r.redInicial() (línea 57 y 58); si está, decrementamos el flujo del arco cTrayectoria unidades (línea 59). Posteriormente, se incrementa el valor del contador i y se verifica si el arco (v_i, v_{i+1}) pertenece a r.redInicial; si es así, se incrementa el flujo; en caso contrario se decrementa. Este proceso se repite para cada una de los arcos formado por los vértices (v_i, v_{i+1}) con $0 \leq i < n$. En ningún momento se modifica la capacidad de cualquier otro arco. Por lo tanto, se cumple la condición 2.

■

Lema 3.2 *Si trayectoria* $\neq$ **null***, al terminar de ejecutarse el método encuentraCapacidadMinimaDeLaTrayectoria() el valor de cTrayectoria es igual a la capacidad de trayectoria, es decir*

$$cTrayectoria = \min\{x.capacidad | x = (u, v) \text{ y } x \in trayectoria\}.$$

[1] A partir de este momento nos referiremos a una trayectoria como una secuencia de vértices, es decir, si la trayectoria es: $v_0, (v_0, v_1), v_1, \ldots, v_{n-1}, (v_{n-1}, v_n), v_n$ entonces sólamente listamos los vértices que la forman en el orden correspondiente, esto es: $v_0, v_1, \ldots, v_{n-1}, v_n$

Demostración:

Sea

$$\text{trayectoria} = v_0, v_1, v_2, \ldots, v_n,$$

y $(v_k, v_{k+1}) \in$ trayectoria, con $0 \leq k < n$ la arista con capacidad mínima, esto es,

$$c(v_k, v_{k+1}) = \min\{c(v_i, v_{i+1}) | (v_i, v_{i+1}) \in \text{trayectoria con } 0 \leq i < n\}.$$

Si existen más de una arista con capacidad mínima tomamos a (v_k, v_{k+1}) igual a la primera arista que aparece en trayectoria, sin pérdida de generalidad.
P.D. cTrayectoria $= c(v_k, v_{k+1})$.

Las únicas veces que el valor de cTrayectoria es modificado son las siguientes:

- Cuando es inicializado, que se le asigna la capacidad de la primera arista en la trayectoria, es decir, cTrayectoria $= c(v_0, v_1)$ (línea 30).
- Y cuando alguna arista (v_l, v_{l+1}) con $1 \leq l < n$ tiene capacidad menor que el valor actual de cTrayectoria (línea 36 y 37).

Dado que el algoritmo comprueba la capacidad de todos los arcos que pertenecen a trayectoria y, además, lo hace en orden (línea 31), en alguna de las iteraciones del **for** comprobará la arista (v_k, v_{k+1}), y como ésta tiene capacidad mínima se cumple la condición del **if** (línea 36); por lo tanto, el valor de cTrayectoria es modificado y es igual a $c(v_k, v_{k+1})$. Por hipótesis tenemos que la capacidad de la arista (v_k, v_{k+1}) es mínima, lo cual implica que para los arcos (v_j, v_{j+1}), $k < j < n$, la capacidad debe ser mayor o igual que la capacidad de (v_k, v_{k+1}), es decir, $c(v_j, v_{j+1}) \geq c(v_k, v_{k+1})$. Por lo tanto no se cumple la condición del **if** (línea 36) en ninguna de ellas lo cual implica que el valor de cTrayectoria no vuelve a ser modificado. De donde podemos concluir que cTrayectoria $= c(v_k, v_{k+1})$.

■

Lema 3.3 *Sean $s = u_0, u_1, u_2, \ldots, u_k = t$ los vértices de una trayectoria aumentante del vértice s al vértice t. Para todos los arcos (u_i, u_{i+1}) con $0 \leq i < k$, al terminar de ejecutarse el método actualizaRedResidual su capacidad se habrá decrementado, y para los arcos de la forma (u_{i+1}, u_i) se habrá incrementado, y la capacidad de cualquier otro arco cuyos vértices no estén en la trayectoria aumentante permanecerá igual.*

Demostración:

Para cada arco $(u_i, u_{i+1}) \in$ trayectoria con $0 \leq i < k$ se decrementa su capacidad (línea 70). Para los arcos de la forma (u_{i+1}, u_i) la capacidad se incrementa (línea 72) si es que éste existe en la red residual; si no, se crea el arco con capacidad igual a la de la trayectoria (línea 74).

■

Lema 3.4 *Si se cumple que padre(t),padre(padre(t)), ..., padre(...padre(t)...) $\neq$* **null** *entonces podemos garantizar que al terminar de ejecutarse el método construyeTrayectoria la variable trayectoria contiene una trayectoria aumentante dada por:*

$$s = padre(padre(\ldots(padre(t))\ldots)), \ldots, padre(padre(t)), padre(t), t.$$

Demostración:

El método inicializa el valor del vértice v=t (línea 79), introduciéndolo en trayectoria (línea 80). Después se ejecuta el ciclo while, donde se asigna a v el vértice padre(v) (línea 82) y después lo agrega al inicio de trayectoria; observemos que en la primera iteración se incluye el vértice padre(t), en la segunda se agrega el vértice padre(padre(t)) y así sucesivamente hasta llegar al vértice s.

∎

Teorema 3.1 *El algoritmo AGTA encuentra un flujo máximo.*

Demostración:

Si en cada una de las especializaciones concretas de AGTA el método encuentraTrayectoriaAumentante cumple con las invariantes expuestas entonces aseguramos que no existe una trayectoria aumentante del vértice s al vértice t en la red residual. Por lo tanto, no es posible incrementar el valor del flujo. De lo anterior concluimos que el flujo encontrado es un flujo máximo.

∎

3.1.3 Complejidad de AGTA

En la siguiente tabla se muestran la complejidad[2] de aquellos métodos definidos en AGTA, la complejidad de los métodos abstractos es definida en cada algoritmo concreto de AGTA.

Método	Complejidad	Justificación
aumentaFlujo	$O(n)$	Actualiza el flujo de todos los arcos de la trayectoria y a lo más son n
encuentraCapacidadMinimaDeLaTrayectoria	$O(n)$	Revisa todos los arcos de la trayectoria y a lo más son n
incrementaFlujo	$O(n)$	Es la suma de la complejidad de los métodos encuentraCapacidadMinimaDeLaTrayectoria, aumentaFlujo y actualizaRedResidual
valorDelFlujo	$O(1)$	Únicamente se regresa el valor de flujoMáximo
aumentaFlujo	$O(n)$	Actualiza el flujo de todos los arcos de la trayectoria en la red original y a lo más son n.
actualizaRedResidual	$O(n)$	Actualiza la capacidad de los arcos de la trayectoria.
construyeTrayectoriaAumentante	$O(n)$	Construye la trayectoria revisando los vértices padre, iniciando con el padre del vértice t; a lo más revisa n vértices.

[2]La complejidad de cualquier método o algoritmo presentado en este trabajo es evaluada en el peor de los casos

Sustituyendo las complejidades de los métodos definidos en AGTA en la ecuación general dada en el Algoritmo Genérico de Flujo Máximo obtenemos lo siguiente:

$$\begin{aligned}
f_{AGFM} &= f_{inicializa} + \sum_{i=1}^{q} (f_{esPosibleIncrementarElFlujo} + f_{incrementaFlujo}) \\
&\quad + f_{valorDelFlujoMaximo} \\
&= f_{inicializa} + \sum_{i=1}^{q} (f_{esPosibleIncrementarElFlujo} + O(n)) + O(1) \\
&= f_{inicializa} + \sum_{i=1}^{q} (f_{encuentraTrayectoriaAumentante} + O(n))
\end{aligned}$$

Nuevamente, la complejidad exacta del algoritmo depende de los métodos abstractos. Entonces, para cada especialización concreta de AGTA la complejidad varía y podemos definir una función de complejidad para aquellos algoritmos que sean especialización de AGTA.

$$f_{AGTA} = f_{inicializa} + \sum_{i=1}^{q} (f_{encuentraTrayectoriaAumentante} + O(n)).$$

Dada la función anterior, en cada uno de los algoritmos concretos basados en AGTA únicamente hay que revisar la complejidad de los métodos abstractos inicializa y encuentraTrayectoriaAumentante para definir la complejidad de la especialización concreta.

3.2 Algoritmo de Flujo Máximo Etiquetado

El *Algoritmo de Flujo Máximo Etiquetado* (AFME), presentado en [15], es una implementación concreta del Algoritmo Genérico de Trayectoria Aumentante (AGTA). En cada iteración del algoritmo, el método encuentraTrayectoriaAumentante etiqueta todos los vértices $u \in V$ para los cuales es posible construir una trayectoria del vértice origen s al vértice u. Al finalizar la ejecución de este método el conjunto de vértices habrá quedado partido en dos subconjuntos de V; en uno de ellos quedarán los *vértices etiquetados*, es decir, aquellos vértices para los cuales fue posible construir una trayectoria del vértice origen a ellos; y el otro conjunto estará formado por los *vértices no etiquetados*, aquellos para los cuales no existe una trayectoria desde el vértice origen.

Si en algún punto de la ejecución del método encuentraTrayectoriaAumentante el vértice t se etiqueta entonces podremos construir una trayectoria aumentante del vértice s al vértice t por medio de la cual incrementamos el flujo, y actualizamos la red residual. Este procedimiento se repite mientras sea posible construir dicha trayectoria, es decir, mientras el vértice t sea

etiquetado. Una vez que el vértice t no puede ser etiquetado, el algoritmo termina regresando el valor del flujo máximo.

La implementación del algoritmo AFME se presenta a continuación.

Listado 3.2.1 Algoritmo de Flujo Máximo Etiquetado

```
1 public class AFME  extends AGTA   {
2
3     public AFME (Red r) {
4         super(r);
5     }
6
7     protected void inicializa () {
8         Grafica  residual    = r.redResidual ();
9         int      numVertices = residual.numeroDeVertices ();
10        trayectoria = new LinkedList();
11        cTrayectoria = 0;
12
13        int       i;
14        for (i = 0; i < numVertices; i++) {
15            VerticeEtiquetado ve = (VerticeEtiquetado)residual.obten(i);
16            ve.desetiqueta();
17            ve.definePadre(null);
18        }
19    }
20
21    public boolean encuentraTrayectoriaAumentante (){
22        VerticeEtiquetado s      = (VerticeEtiquetado)r.verticeOrigen ();
23        VerticeEtiquetado t      = (VerticeEtiquetado)r.verticeDestino ();
24        LinkedList         lista = new LinkedList ();
25
26        inicializa();
27        s.etiqueta();
28        lista.add(s);
29        while (lista.size() > 0 && !(t.estaEtiquetado())) {
30            VerticeEtiquetado i=(VerticeEtiquetado)lista.get(lista.size()-1);
31            lista.remove(i);
32            LinkedList ady = i.obtenAdyacencias();
33            int k;
34            for (k = 0; k < ady.size(); k++) {
35                VerticeEtiquetado v =
36                    (VerticeEtiquetado)((Arista)ady.get(k)).obtenVertice();
37                if (!v.estaEtiquetado()) {
38                    v.definePadre(k);
39                    v.etiqueta();
40                    lista.add(v);
41                }
42            }
43        }
44        if(t.estaEtiquetado()){
45            construyeTrayectoria();
46            return true;
47        } else
```

Listado 3.2.1 Algoritmo de Flujo Máximo Etiquetado (continuación).

```
48              return false;
49      }
50 }
```

3.2.1 Precondiciones y postcondiciones de los métodos de AFME

Constructor

(**public** AFME (Red r))
Valida el estado inicial del algoritmo, asignando un valor inicial a sus atributos.

Precondiciones: La clase debe ser invocada con una red.

Postcondiciones: Deja una representación de la red que cumple con las mismas condiciones dadas en AGTA, incluyendo la siguiente: cada vértice tiene una variable booleana que indica si el vértice está etiquetado. Inicialmente es falsa.

Preproceso

(**protected void** inicializa ())
Inicializa el estado de los vértices y de la trayectoria aumentante.

Precondiciones: Ninguna.

Postcondiciones: Al terminar de ejecutarse el método inicializa() se cumplen las siguientes condiciones:

1. Todos los vértices de r.redResidual estarán no etiquetados.
2. Ningún vértice de r.redResidual tiene asociado un vértice padre.
3. La variable trayectoria no tiene asociada ninguna trayectoria aumentante.
4. La variable cTrayectoria es igual a cero.

Encuentra una Trayectoria Aumentante

(**protected boolean** encuentraTrayectoriaAumentante ())
El método etiqueta todos los vértices para los cuales es posible construir una trayectoria del vértice origen a ellos. Si el vértice t es etiquetado actualiza la variable **trayectoria** con los vértices de la trayectoria que encontró y regresa **true**. Si no es posible etiquetar al vértice t durante la ejecución del método entonces no es posible construir ninguna trayectoria aumentante, por lo que el método regresa **false**.

Precondiciones: Ninguna.

Postcondiciones: Si el método regresa **true**: trayectoria es una trayectoria aumentante. Si el método regresa **false**: no existe una trayectoria aumentante (trayectoria = **null**).

3.2.2 Correctez de AFME

Para demostrar que si se cumplen las precondiciones, de cada uno de los métodos, entonces se cumplen las postcondiciones, demostraremos los siguientes resultados.

Lema 3.5 *Al terminar de ejecutarse el método inicializa() se cumplen las siguientes condiciones:*

1. *Todos los vértices de r.redResidual estarán no etiquetados.*
2. *Ningún vértice de r.redResidual tendrá asociado un vértice padre.*
3. *La variable trayectoria no tendrá asociada ninguna trayectoria aumentante.*
4. *La variable cTrayectoria* = 0.

Demostración:
El **for** se ejecuta n veces, donde n es el número de vértices que tiene la red. Por lo tanto, para cada vértice u en la red residual (línea 14) el vértice es desetiquetado (línea 16) y el padre de cada vértice es **null** (línea 17), es decir, no tiene asignado ningún vértice padre. Por último, la variable trayectoria es inicializada con una trayectoria vacía y con capacidad igual a cero(línea 10 y 11). De donde, al terminar de ejecutarse el método inicializa las postcondiciones se cumplen.

■

Observemos que siempre que el método encuentraTrayectoriaAumentante se ejecuta, lista contiene a los vértices que ya fueron etiquetados pero aún no han sido examinados. Esta lista es inicializada con una lista vacía e inmediatamente se introduce a s; el método no termina hasta que el vértice t está etiquetado o la lista se haya vaciado. El ciclo **while** se ejecuta por lo menos una vez dado que en un principio la lista contiene al vértice origen. En cada iteración del ciclo se examina un vértice u de la lista, esto quiere decir que verificamos cada arco del cual u es el primer componente (esto es (u, v)), e incluimos a la lista de vértices etiquetados aquel vértice v que aún no ha sido etiquetado y para los cuales existe un arco de u a v. Además de incluirlos a la lista los etiquetamos y les asignamos como vértice padre a u, dado que éste es el vértice desde el cual fueron descubiertos.

Lema 3.6 *Para todo vértice $v \in V$ para el cual existe una $s \rightsquigarrow v$ trayectoria en la red residual, al terminar de ejecutarse el método encuentraTrayectoriaAumentante estos vértices estarán etiquetados.*

Demostración:
La demostración de este lema será realizada por contradicción. Supongamos que existe un vértice v para el cual existe una trayectoria desde el vértice origen y al terminar de ejecutarse el método el vértice no está etiquetado. Sea $s = u_0, u_1, u_2, \ldots, u_{k-1}, u_k = v$ la $s \rightsquigarrow v$ trayectoria.

Antes de entrar al ciclo (líneas 27 y 28) el vértice s es etiquetado e introducido a la lista, además de ser el único vértice que contiene lista en este punto; por lo tanto, en la primera iteración del **while** se revisan los arcos donde s es el primer componente, lo cual implica que se examina el arco (s, u_1) (línea 32); así que el vértice u_1 es etiquetado (línea 39) y se le asigna

como vértice padre a s (línea 38), es decir, $padre(u_1) = s$. Además se introduce a la lista (línea 40), por lo que en algún momento se revisan los arcos adyacentes a u_1, dado que el vértice u_2 forma parte del arco (u_1, u_2); entonces en el momento de examinar a u_1 se etiqueta u_2 y se le asigna como padre el vértice u_1. Este procedimiento se repite para cada arco y vértice de la trayectoria. Por lo tanto, cuando estemos revisando los arcos adyacentes al vértice u_{k-1} etiquetaremos al vértice t. Y dado que dentro del método un vértice que ha sido etiquetado no se desetiqueta, t estará etiquetado.

■

Lema 3.7 *Si el vértice t es etiquetado en el método encuentraTrayectoriaAumentante entonces es posible construir una trayectoria aumentante del vértice s al vértice t de la siguiente manera:*

$$s = padre(padre(\ldots(padre(t))\ldots)), \ldots, padre(padre(t)), padre(t), t.$$

Demostración:

Por el lema anterior podemos asegurar que si el vértice t es etiquetado entonces existe una trayectoria del vértice s a t; sólo nos falta demostrar que esta trayectoria se puede construir a través de los vértices padres, yéndonos en reversa a partir de t.

Cada vez que un vértice es descubierto se le asigna su vértice padre, el cual es el vértice desde el cual fue descubierto (línea 38). Por lo tanto, existe el arco $(padre(u), u)$ para cualquier vértice $u \in V$ que ha sido etiquetado. Dado que t es etiquetado entonces existe el arco $(padre(t), t)$ y como t fue descubierto desde el vértice $padre(t)$ entonces éste también es un vértice etiquetado (porque fue introducido a la lista); por lo tanto también existe el arco $(padre(padre(t)), padre(t))$, y así sucesivamente hasta llegar al vértice origen, s. De esto, podemos construir una trayectoria de la siguiente manera:

$$s = padre(padre(\ldots(padre(t))\ldots)), \ldots, padre(padre(t)), padre(t), t$$

■

Lema 3.8 *El método encuentraTrayectoriaAumentante() regresa verdadero únicamente si existe una trayectoria aumentante del vértice s al vértice t, y los vértices que la forman están en la lista trayectoria.*

Demostración:

El método sólo regresa **true** si el vértice t está etiquetado (líneas 44-46); por el lema 3.7 podemos construir una trayectoria aumentante del vértice s al vértice t a partir de los vértices padres, comenzando con t, y dicha trayectoria es construida y almacenada en la variable trayectoria (línea 45) –por el lema 3.4.

■

Corolario 3.1 *El método encuentraTrayectoriaAumentante() regresa* **false** *únicamente si no existe una trayectoria aumentante del vértice s al vértice t.*

Demostración:

La demostración de este corolario es inmediata del lema anterior: el método sólo regresa **false** si t no está etiquetado (línea 48), en cuyo caso no existe una trayectoria del vértice s al vértice t.

■

Lema 3.9 *Si el método encuentraTrayectoriaAumentante() regresa falso entonces el conjunto de vértices etiquetados y el conjunto de vértices no etiquetados forman un $s-t$ corte.*

Demostración:

Sea $S = \{v \in V | v \text{ está etiquetado}\}$ y $T = \{v \in V | v \text{ no está etiquetado}\}$. El método solamente regresa **false** si el vértice t no ha sido etiquetado; esto significa que $t \in T$, y dado que s es etiquetado al inicio entonces $s \in S$. Por lo tanto, $[S,T]$ es un $s-t$ corte.

■

Teorema 3.2 *Al terminar de ejecutarse el Algoritmo de Flujo Máximo Etiquetado, flujoMaximo es el valor del flujo máximo.*

Dado que el algoritmo algoritmoGenericoDeTrayectoriaAumentante() de AGTA sólo termina si el método encuentraTrayectoriaAumentante regresa **falso**, esto sucederá si no existe una trayectoria aumentante del vértice s al vértice t en la red residual. Y por el teorema 1.1 (página 16) podemos asegurar que el flujo obtenido es máximo.

■

3.2.3 Complejidad de AFME

A continuación se presenta una tabla con la complejidad de cada método del algoritmo AFME.

Método	Complejidad	Justificación
inicializa	$O(n)$	Dado que desetiqueta y asigna el padre como nulo a cada vértice $u \in V$
encuentraTrayectoriaAumentante	$O(m)$	Para cada vértice $u \in V$ revisa todos sus arcos

Sustituyendo las complejidades de los métodos en la ecuación general del Algoritmo Genérico de Trayectoria Aumentante, obtenemos lo siguiente:

$$\begin{aligned} f_{AGTA} &= f_{inicializa} + \sum_{i=1}^{q}(f_{encuentraTrayectoriaAumentante} + O(n)) \\ &= O(n) + \sum_{i=1}^{q}(O(m) + O(n))) \end{aligned}$$

Como las capacidades son números enteros entonces la capacidad máxima U es también un número entero, lo cual implica que la capacidad de cualquier $s-t$ corte es a lo más nU; por lo tanto el flujo máximo es a lo más nU. Suponiendo que el algoritmo aumente en una unidad el valor del flujo por cada incremento entonces $q \leq nU$. Sustituyendo en la ecuación anterior obtenemos el siguiente teorema:

Teorema 3.3 *El Algoritmo de Flujo Máximo Etiquetado resuelve el problema de flujo máximo etiquetado en $O(nmU)$, donde U es la capacidad máxima de los arcos.*

Demostración:
La complejidad de AFME este definida por la siguiente ecuación:

$$\begin{aligned} f_{AFME} &= f_{inicializa} + \sum_{i=1}^{q}(f_{encuentraTrayectoriaAumentante} + O(n)) \\ &= O(n) + \sum_{i=1}^{nU}(O(m) + O(n))) \\ &= O(nmU) + O(n^2U) \end{aligned}$$

Como para cualquier redes que proviene de una gráfica $2 - conexa$ se cumple que $n < m$ (para los árboles el problema es trivial), la complejidad de AFME se reduce a $O(nmU)$.

■

3.3 Implementaciones concretas polinomiales

En la sección anterior presentamos la implementación genérica del algoritmo que resuelve el problema de flujo máximo a través de trayectorias aumentantes. Además revisamos una implementación concreta de éste, el algoritmo AFME. La desventaja de este algoritmo es que su complejidad depende de la capacidad de los arcos, lo cual implica que si tenemos una red cuyas capacidades son muy grandes el algoritmo AFME se llevará en principio mucho tiempo en resolver el problema.

En esta sección presentaremos varias implementaciones concretas del algoritmo AGTA, las cuales, a diferencia del algoritmo AFME, resuelven el problema de flujo máximo en tiempo polinomial independientemente de la capacidad máxima de los arcos. La idea básica para reducir la complejidad de los algoritmos está en decrementar el número de aumentos necesarios para encontrar el flujo máximo. Las formas como alcanzaremos esta meta son las siguientes:

- La primera de ellas consiste en enviar la mayor cantidad de flujo a través de una trayectoria aumentante, es decir, en cada aumento se enviará una fuerte cantidad de flujo desde el vértice origen hasta el vértice destino.

- La segunda es condicionar el tipo de trayectorias aumentantes; en este caso pediremos que la trayectoria aumentante sea de longitud mínima, esto es, una trayectoria del vértice s al vértice t con el mínimo número de arcos posibles.

- Y la última es permitiendo que no se cumpla la condición de conservación de flujo en cada paso intermedio del algoritmo, lo cual nos permitirá aumentar el flujo a través de cada arco, es decir, no será necesario aumentar el flujo a través de una trayectoria (no todo

el flujo que sale desde s tiene que llegar a t, en cada aumento). Solamente necesitamos que al final del algoritmo dicha condición se cumpla: como es una condición de flujo y el objetivo es encontrar el flujo máximo, al terminar el algoritmo la conservación de flujo en cada vértice $v \in V - \{s, t\}$ debe cumplirse.

El último método lo estudiaremos en el siguiente capítulo; por ahora sólo nos dedicaremos a ver algoritmos que utilizan las primeras dos ideas con el fin de reducir su complejidad.

3.4 Algoritmo de Escalamiento de Capacidad

La idea de querer aumentar el flujo en grandes cantidades nos sugiere que debemos incrementar el flujo a través de una trayectoria con capacidad máxima, aunque para ello tendríamos que revisar cuál de las trayectorias aumentantes del vértice origen al vértice destino tiene la mayor capacidad; para evitar esto se sugiere aumentar el flujo a través de cualquier trayectoria con una capacidad "suficientemente" grande. Esta es la idea principal del *algoritmo de escalamiento de capacidad* (AEC) [14] , el cual incrementa el flujo a través de una trayectoria aumentante del vértice s al vértice t con capacidad muy grande (no necesariamente capacidad máxima). Para poder explicar cómo trabaja el algoritmo AEC introducimos un parámetro Δ, y definimos $\Delta-$*red residual* con respecto a un flujo f como la red que contiene los arcos con capacidad residual mayor o igual que Δ, denotada por $G_f(\Delta)$. Llamamos *fase de escalamiento* al tiempo en que el valor de Δ permanece constante durante la ejecución del algoritmo, y $\Delta-$*fase de escalamiento* a la fase en que Δ tiene un valor específico.

Al inicio del algoritmo asignamos $\Delta = 2^{\log U}$, donde U es la capacidad máxima de los arcos de la red; una vez definido este valor se actualiza la $\Delta-$*red residual* correspondiente a Δ y entonces procedemos a buscar trayectoria aumentante del vértice origen al vértice destino, si la encontramos incrementamos el flujo a través de ella y actualizamos la $\Delta-$*red residual*. Mientras que sea posible encontrar una trayectoria aumentante, se repite este procedimiento. Cuando ya no es posible encontrar una trayectoria, se actualiza el valor de Δ y $\Delta-$*red residual*, repitiendo nuevamente el procedimiento. El algoritmo termina cuando $\Delta = 1$ y no es posible construir una trayectoria aumentante. Observemos que la red $G_f(1)$ es igual a la red residual correspondiente al flujo f y a la red G dado que en la red residual únicamente están los arcos cuya capacidad es mayor a cero.

El método que este algoritmo utiliza para encontrar la trayectoria aumentante es el mismo usado en AFME. La implementación del algoritmo se presenta a continuación:

Listado 3.4.1 Algoritmo de Escalamiento de Capacidad

```
1 public class AEC  extends AFME {
2     private int        Delta;
3
4     public AEC  (Red r) {
5         super(r);
6         int cMaxima = defineCapacidadMaxima();
7         Delta  = (int)(Math.pow(2,(int)(Math.log(cMaxima)/Math.log(2))));
8     }
```

Listado 3.4.1 Algoritmo de Escalamiento de Capacidad (continuación)

```
9       public int algoritmoDeEscalamientoDeCapacidad () {
10          int     flujo = 0;
11          while (Delta >= 1) {
12              flujoMaximo = 0;
13              actualizaRedResidualMayorQueDelta ();
14              flujo += algoritmoGenericoDeFlujoMaximo ();
15              Delta = Delta /2;
16          }
17          flujoMaximo = flujo;
18          return flujoMaximo;
19      }
20      private void actualizaRedResidualMayorQueDelta () {
21          r.copiaRedInicialEnRedResidual ();
22          LinkedList vertices = r.redResidual().obtenVertices();
23
24          int i;
25          for(i = 0; i < vertices.size(); i++ ){
26              Vertice    u    = (Vertice)vertices.get(i);
27              LinkedList ady = u.obtenAdyacencias();
28              int j;
29              for(j = 0; j < ady.size(); j++ ){
30                  Arista aristaUV  = (Arista)ady.get(j);
31                  int    capacidad = aristaUV.obtenCapacidad();
32                  int    flujo     = aristaUV.obtenFlujo();
33                  Vertice v = (Vertice)(aristaUV.obtenVertice());
34                  if(  (capacidad - flujo) < Delta ){
35                      u.desconectaDe(v);
36                      j--;
37                  } else
38                      aristaUV.defineCapacidad(capacidad - flujo);
39                  if(flujo >= Delta)
40                      v.conectaCon(u,flujo);
41
42              }
43          }
44      }
45
46      private void actualizaRedResidual () {
47          Vertice v1, v2;
48          int     i;
49          for (i = 0; i < trayectoria.size()-1; i++) {
50              v1 = (Vertice)trayectoria.get(i);
51              v2 = (Vertice)trayectoria.get(i+1);
52              v1.decrementaCapacidad(v2,cTrayectoria);
53              Arista a = v1.obtenArista(v2);
54              if(a != null)
55                  if(a.obtenCapacidad() < Delta)
56                      v1.desconectaDe(v2);
57              if(v2.estaConectadoCon(v1))
58                  v2.incrementaCapacidad(v1,cTrayectoria);
```

Listado 3.4.1 Algoritmo de Escalamiento de Capacidad (continuación)

```
59             else if(cTrayectoria >= Delta)
60                 v2.conectaCon(v1,cTrayectoria);
61         }
62     }
63
64     private int defineCapacidadMaxima () {
65         int i, tmp = 0;
66         for(i = 0; i < r.redResidual().numeroDeVertices() ; i++){
67             Vertice    u  = (Vertice)r.redResidual().obten(i);
68             LinkedList ady = u.obtenAdyacencias();
69             int j;
70             for(j = 0; j < ady.size(); j++ ){
71                 if( ((Arista)ady.get(j)).obtenCapacidad() > tmp)
72                     tmp = ((Arista)ady.get(j)).obtenCapacidad();
73             }
74         }
75         return tmp;
76     }
77 }
```

3.4.1 Precondiciones y postcondiciones de los métodos de AEC

Constructor

(public AEC (Red r))

Valida el estado inicial del algoritmo, asignando un valor inicial a sus atributos.

Precondiciones: La clase debe ser invocada con una red.

Postcondiciones: Deja una representación de la red que cumple con los mismas condiciones dadas en el algoritmo AFME. Además inicializa el valor de las variables cMaxima y Delta con los siguientes valores:

$$\text{cMaxima} = \max\{c(u,v)|(u,v) \in E\}\text{Delta} = 2^{\log U}.$$

Algoritmo de Escalamiento de Capacidad

(public int algoritmoDeEscalamientoDeCapacidad ())

Este método encuentra el flujo máximo en la red residual correspondiente al valor de Delta, utilizando el algoritmoGenericoDeTrayectoriaAumentante() de la clase AGTA. Una vez que se ha encontrado el flujo máximo en la red residual correspondiente a Delta, es decir, que no existe una trayectoria aumentante en ésta, se disminuye el valor de Delta y se repite este procedimiento mientras que Delta ≥ 1.

Precondiciones: Delta≥ 1.

Postcondiciones: Al terminar de ejecutarse el método, éste encuentra el flujo máximo que puede ser enviado desde el vértice origen al vértice destino, y se cumple Delta< 1.

Actualiza la Red Residual Correspondiente a Delta

(private void actualizaRedResidualMayorQueDelta ())
Obtiene la red residual correspondiente al flujo actual f y al valor de Delta. Es decir, únicamente están los arcos cuya capacidad residual es mayor o igual al valor de la variable Delta.

Precondiciones: El valor de la variable Delta debe ser al menos uno.

Postcondiciones: En r.redResidual sólo están los arcos $a = (u, v)$ tales que a.capacidad≥Delta.

Actualiza Red Residual

(private void actualizaRedResidual ())
Dada una trayectoria aumentante del vértice origen al vértice destino, el método actualiza la capacidad residual de los arcos (u, v) tales que u, v están en la trayectoria; además elimina aquellos arcos cuya capacidad residual sea menor que Delta.

Precondiciones: Las mismas que en el algoritmo genérico AGTA.

Postcondiciones: Además que las de AGTA, se incluye la siguiente condición: cualquier arco $a = (u, v) \in$ r.redResidual cumple a.capacidad≥ Delta, donde Delta corresponde a la fase actual.

Define Capacidad Máxima

(private int defineCapacidadMaxima ())
Devuelve la capacidad máxima de los arcos de la red original.

Precondiciones: Ninguna.

Postcondiciones: Sea max el valor que regresa el método defineCapacidadMaxima entonces se cumple:

$$\text{max} = \max\{c(u,v) | (u,v) \in A\}$$

3.4.2 Correctez de AEC

Lema 3.10 *El método defineCapacidadMaxima() regresa el valor de la capacidad máxima de los arcos de la red original. En otras palabras, si max es el valor que regresa defineCapacidadMaxima entonces*

$$max = \max\{c(u,v) | (u,v) \in A\}$$

Demostración:
En un principio se inicializa el valor de tmp=0 (línea 64); después, para cada vértice (línea 66) revisa los arcos correspondientes a éste (línea 68) para encontrar el que tiene mayor capacidad. Esto último lo realiza de la siguiente manera: si en el momento de revisar la capacidad de

una arista el valor de ésta es mayor que el de tmp entonces se asigna el valor de la capacidad a la variable tmp. Una vez que tmp es igual a la capacidad máxima, este valor no vuelve a cambiar porque la única forma en que el valor de la variable cambia es si existe un arco con capacidad mayor que tmp (línea 71). De lo anterior, tenemos que el método regresa la capacidad máxima de los arcos de la red (línea 75).

■

Lema 3.11 *Al terminar de ejecutarse el método constructor AEC, Delta* $= 2^{\log U}$, *donde* $U = \max\{c(u,v) : c(u,v) \geq 0 \text{ y } (u,v) \in A\}$.

Demostración:
El valor de Delta es inicializado en la línea 7; por el lema 3.10 tenemos que cMaxima tiene asignado el valor de la capacidad máxima (línea 6). Por lo tanto, Delta $= 2^{\log U}$.

■

Lema 3.12 *La* Δ*−red residual,* $G_f(\Delta)$ *correspondiente a una red* G *y un flujo* f, *cuando* $\Delta = 1$ *es equivalente la red residual* G_f *correspondiente a* G *y* f.

Demostración:
Por definición de red residual tenemos, que en G_f están todos los vértices de G, o sea V, y solamente están los arcos cuya capacidad residual es mayor que cero ($c_f(u,v) \geq 1$). Ahora bien, $G_f(1)$ es igual a la red formada por los vértices de G, y los arcos cuya capacidad residual es mayor o igual a uno. Por lo tanto, $G_f(1) = G_f$.

■

Lema 3.13 *Sean* $s = u_0, u_1, u_2, \ldots, u_k$ *los vértices de una trayectoria aumentante del vértice* s *al vértice* t *al terminar de ejecutarse el método actualizaRedResidual en la* Δ*−fase de escalamiento. Entonces se cumplen las siguientes propiedades:*

1. *Para todos los arcos* (u_i, u_{i+1}), $0 \leq i < k$ *su capacidad se habrá decrementado y para los arcos* (u_{i+1}, u_i) *se habrá incrementado. Para cualquier otro arco cuyos vértices no estén en la trayectoria aumentante su capacidad permanecerá igual.*

2. *En r.redResidual sólo están los arcos* (u,v) *tal que* $c(u,v) \geq \Delta$.

Demostración:
Para cada arco $(u_i, u_{i+1}) \in$ trayectoria con $0 \leq i < k$, se decrementa su capacidad (línea 52); además, si $c_f(u_i, u_{i+1}) < \Delta$ lo borramos (línea 56). Para los arcos de la forma (u_{i+1}, u_i) se tienen dos casos: si el arco ya estaba en la red residual entonces sólo incrementamos la capacidad residual de éste (línea 58); si no estaba, lo incluimos solamente si $c_f(u_{i+1}, u_i) \geq \Delta$ (línea 60). Por lo tanto, actualizamos la capacidad residual únicamente de los arcos de la trayectoria y quitamos aquellos cuya capacidad residual sea menor a Δ.

■

Lema 3.14 *Una vez ejecutado el método actualizaRedResidualMayorQueDelta, en r.redResidual sólo están los arcos cuya capacidad es mayor o igual que Delta.*

Demostración:
Observemos que en la primera línea en r.redResidual copiamos la red original (línea 26); después, para cada vértice u de la red (línea 25) se obtiene su lista de adyacencias (línea 27). Para cada arco (u, v) se tiene uno de los siguientes casos:

- Si $c_f(u, v) \geq$ Delta entonces incluimos el arco (u, v) en la red residual.
- Si $c_f(u, v) <$ Delta, borramos el arco de r.redResidual.
- Si $f(u, v) \geq$ Delta agregamos el arco (v, u) a r.redResidual con capacidad residual $c_f(v, u) = f(u, v)$.

Con estas condiciones aseguramos que r.redResidual únicamente contiene arcos (u, v) tal que $c_f(u, v) \geq$ Delta.

■

Teorema 3.4 *Al terminar de ejecutarse el método algoritmoDeEscalamientoDeCapacidad, el valor de la variable flujoMaximo es el valor del flujo máximo.*

Demostración:
Demostrar que el método termina se reduce a probar que el ciclo **while** termina. Por el lema 3.14 aseguramos que el método actualizaRedResidualMayorQueDelta termina. Además, por 3.13 sabemos que el proceso de actualizar la red una vez que ésto se ha incrementado también finaliza. Y dado que AEC utiliza el mismo método que el algoritmo AFME para encontrar la trayectoria aumentante, podemos asegurar que el método algoritmoGenericoDeTrayectoriaAumentante también finaliza. Asimismo, una vez que se ha encontrado el flujo máximo en la $\Delta-$red residual correspondiente se decrementa el valor de la variable Delta (línea 15). Por lo anterior, Delta<1 en un número finito de pasos, dado que Delta es un número entero. Por lo tanto, el método termina.

Ahora solamente nos falta demostrar que regresa el valor del flujo máximo. Por el lema 3.12 tenemos que si Delta $= 1$ la $\Delta-$red residual asociada al flujo actual f es igual a la red residual correspondiente a f $(G_f(1) = G_f)$. Por lo tanto, en la última ejecución del ciclo, es decir en la $1-$fase de escalamiento, la red residual en la que aumentamos el flujo f es G_f. Por lo anterior y por el lema 3.2 concluimos que el valor que regresa el método algoritmoDeEscalamientoDeCapacidad es el del flujo máximo, dado que AEC hereda los métodos de AFME.

■

3.4.3 Complejidad de AEC

Antes de demostrar la complejidad del algoritmo probaremos que en una red existen a lo más m trayectorias simples del vértice origen al vértice destino con flujo distinto de cero. A los flujos que son trayectorias, en otras palabras, únicamente circulan por una trayectoria aumentante del vértice origen al vértice destino, les llamamos *flujo de trayectoria*. La definición formal es la siguiente:

Definición 3.1 *Un* flujo de trayectoria *en una red G, es un flujo f que toma valores distintos de cero únicamente en algunas trayectorias simples (trayectorias no dirigidas), del vértice s al*

vértice t. En otras palabras, existe un número δ y una trayectoria simple $P = u_0, u_1, u_2, \ldots, u_k$ con $u_0 = s, u_k = t$ tal que

$$f(u,v) = \begin{cases} \delta & \text{si } (u,v) \in P \text{ y es un arco de avance} \\ -\delta & \text{si } (u,v) \in P \text{ y es un arco de retroceso} \\ 0 & \text{en cualquier otro caso} \end{cases}$$

Lema 3.15 *Cualquier flujo f en una red $G = (V, A, c, s, t)$ puede ser expresado como la suma de a lo más m flujos de trayectorias en G y un flujo en la red residual cuyo valor es cero, donde m es el número de arcos en G.*

Demostración:

Sea f un flujo en G con $|f| > 0$ (en el caso en que $|f| = 0$, el lema se cumple trivialmente). Definimos c', una nueva función de capacidad de la siguiente manera:

$$c'(u,v) = \max\{f(u,v), 0\}$$

y sea $G' = (V, A', c', s, t)$. Entonces f' es un flujo en G' y dado que $c'(u,v) \leq c(u,v)$ cualquier flujo en G' es un flujo en G. Por el teorema 1.1 G' debería tener una trayectoria aumentante del vértice s al vértice t; por construcción de G', cualquier arco en la trayectoria está saturado por f. Sea p un flujo de trayectoria cuyo valor es la capacidad del arco que juega el papel de cuello de botella, es decir, el arco (u,v) en la trayectoria aumentante tal que tiene la menor capacidad. Entonces los flujos p y $f - p$ son flujos en G', por los lemas 1.3 y 1.4, y al menos un arco de la trayectoria se satura; por lo tanto, en la red residual correspondiente a este nuevo flujo ese arco desaparece. Lo anterior nos garantiza que hay un arco menos en la red residual. Repetimos este proceso con el flujo $f - p$ para obtener $c''(u,v) \leq c'(u,v)$, donde $c''(u,v)$ se define de manera similar a $c'(u,v)$, y $G'' = (V, A'', c'', s, t)$. Observemos que G'' tiene menos arcos que G', ya que al menos un arco fue saturado y por lo tanto eliminado de la gráfica. Este procedimiento puede repetirse a lo más m veces. Por lo tanto, el flujo original f es la suma de un flujo con valor cero y la suma de los flujos de trayectorias encontradas en cada paso.

■

Lema 3.16 *El Algoritmo de Escalamiento de Capacidad realiza $O(m \log U)$ aumentos de flujo.*

Demostración:

Sea f un flujo en la red G y f^* el flujo máximo de la misma. Si aplicamos el lema anterior a la red residual, aseguramos que podemos encontrar m o menos trayectorias dirigidas del vértice origen al vértice destino cuya suma de las capacidades residuales sea $|f^*| - |f|$. Entonces existe una trayectoria con capacidad al menos $\frac{|f^*|-|f|}{m}$. Consideremos una secuencia de $2m$ aumentos consecutivos con capacidad máxima cuyo flujo inicial es el flujo f. Cada uno de estos aumentos incrementa el valor del flujo en al menos $\frac{|f^*|-|f|}{2m}$ unidades, lo cual implica que en $2m$ aumentos o menos alcanzaremos el flujo máximo. Observemos que si alguno de estos aumentos incrementa el flujo en una cantidad menor que $\frac{|f^*|-|f|}{2m}$ unidades de flujo, entonces del flujo inicial f reducimos la capacidad residual de la trayectoria aumentantes a la mitad, por lo menos. Por lo tanto, en $2m$ iteraciones consecutivas el algoritmo establece un flujo máximo o reduce la capacidad de la trayectoria aumentante por lo menos a la mitad. Ya que la capacidad residual de una

trayectoria aumentante es a lo más $2U$ y al menos 1, después de $m \log U$ aumentos, el flujo es máximo. La eficiencia de este algoritmo se debe a que realiza $2m$ aumentos por cada Δ -fase y el número total de fases que lleva a cabo es $O(\log U)$. Por lo tanto, en total realiza $O(m \log U)$ aumentos. ■

A continuación se presenta una tabla con la complejidad de cada método del algoritmo *AEC*.

Método	Complejidad	Justificación
actualizaRedResidualMayorQueDelta	$O(m)$	Revisa todos los arcos para verificar que su capacidad residual sea mayor o igual que Delta. Si alguno de ellos no cumple esta condición, es eliminado en la red residual.
actualizaRedResidual	$O(n)$	Actualiza la capacidad de los arcos pertenecientes a la trayectoria aumentante encontrada, y borra aquellos cuya capacidad residual sea menor que Delta.
defineCapacidadMaxima	$O(m)$	Revisa todas los arcos para ver cual de ellos tiene la capacidad máxima.

Recordemos que el algoritmo AEC está basado en el algoritmo AFME expuesto en la sección anterior, por lo que, la complejidad del método encuentraTrayectoriaAumentante es la misma y la complejidad de actualizaRedResidual también es igual a la del método definido en AGTA. Y ya se demostró en el teorema anterior que el número de aumentos es a lo más $m \log U$, entonces $q = m \log U$. Sustituyendo los valores en la ecuación general de AGTA obtenemos el siguiente teorema.

Teorema 3.5 *El Algoritmo de Escalamiento de Capacidad resuelve el problema de flujo máximo etiquetado en $O(m^2 \log U)$.*

Demostración:

La complejidad del Algoritmo de Escalamiento de Capacidad está dada por la siguiente ecuación:

$$\begin{aligned} f_{AEC} &= f_{inicializa} + \sum_{i=1}^{q} (f_{encuentraTrayectoriaAumentante} + O(n)) \\ &= O(n) + \sum_{i=1}^{m \log U} (O(m) + O(n))) \\ &= O(n) + m \log U (O(m) + O(n))) \end{aligned}$$

Dado las redes cumplen con $n < m^3$, la complejidad de AEC es $O(m^2 \log U)$.

∎

3.5 Distancias Etiquetadas

Una función de distancia $d : V \rightarrow \mathbb{Z}^+ \cup \{0\}$ con respecto a la capacidad residual, es una función del conjunto de vértices a los enteros mayores o iguales que cero. Decimos que una ***función de distancia*** es ***válida*** con respecto a un flujo f si cumple con las siguientes condiciones:

- La distancia del vértice destino es cero, es decir, $d(t) = 0$
- Para cada arco (u, v) en la red residual inducida por el flujo f se cumple que $d(u) \leq d(v) + 1$.

Nos referimos a $d(u)$ como la ***distancia etiquetada del vértice*** u; en este caso particular, es la distancia del vértice u al vértice destino t.

El siguiente resultado nos muestra que dicha distancia corresponde a una cota mínima de la longitud de la trayectoria más corta del vértice u al vértice destino. Por ejemplo, para el vértice t la trayectoria más corta a él mismo es de longitud cero, y para todos los vértice u para los cuales existe una arco (u, t), $d(u) = 1$.

Lema 3.17 *Si las* distancias etiquetadas *son válidas, la distancia etiquetada del vértice u es una cota mínima de la longitud de la trayectoria más corta del vértice u al vértice t, en la red residual G_f.*

Demostración:

Sea

$$p : u = u_0, u_1, u_2, \ldots, u_{k-1}, u_k = t$$

una trayectoria del vértice u al vértice destino t. Y sea d la función de distancia válida. Como existen los arcos (u_{i-1}, u_i), con $0 < i \leq k$ tenemos:

$$\begin{array}{lcll} d(u_{k-1}) & \leq & d(u_k) + 1 & = 1 \\ d(u_{k-2}) & \leq & d(u_{k-1}) + 1 & = 2 \\ & & \vdots & \\ d(u_0) & \leq & d(u_1) + 1 & = k \end{array}$$

Dado que no dimos ninguna restricción de la trayectoria p, se cumple que para cualquier $u \rightsquigarrow t$ trayectoria $d(u)$ es menor o igual que la longitud de esta trayectoria.

∎

Lema 3.18 *Si la distancia etiquetada del vértice s es mayor que el número de vértices en G, entonces la red residual G_f no contiene una trayectoria del vértice s al vértice t.*

Demostración:

Por el lema anterior tenemos que $d(s)$ es una cota mínima de la longitud de la trayectoria más corta del vértice s al vértice t. Además, ninguna trayectoria puede tener más de $(n - 1)$ arcos. Por lo tanto, si $d(s) \geq n$ no existe ninguna trayectoria de s a t en la red residual.

∎

[3] Dado que son gráficas 2−conexas, y para los árboles el problema es trivial

3.6 Arcos y Trayectorias Admisibles

Una vez definido el concepto de función de distancia válida podemos dar la definición de *arco admisible.* Un arco (u, v) es admisible si $d(v) = d(u) + 1$. Ya hemos demostrado que la distancia de un vértice u es una cota mínima para cualquier $u \rightsquigarrow t$ trayectoria. Podemos observar que si un arco es admisible entonces dicho arco pertenece a una trayectoria del vértice u al vértice t. Además, una trayectoria que está formada únicamente por arcos admisibles es llamada *trayectoria admisible.* Más adelante demostraremos que dicha trayectoria es de longitud mínima. A continuación se presenta la definición formal de *arco* y *trayectoria admisible.*

Definición 3.2 *Un arco* (u, v) *es* admisible *si se cumple:*

$$d(u) = d(v) + 1;$$

en caso contrario, decimos que el arco (u, v) no es admisible.

Definición 3.3 *Una trayectoria del vértice s al vértice t es* admisible *si todos los arcos que la forman son admisibles.*

Lema 3.19 *Si p es una trayectoria admisible entonces p es una trayectoria de aumento de longitud mínima del vértice s al vértice t.*

Demostración:

Sea

$$p : s = u_0, u_1, u_2, \ldots, u_{k-1}, u_k = t.$$

Para cada arco (u_{i-1}, u_i), $0 < i \leq k$, se cumple que $c_f(u_{i-1}, u_i) > 0$ (por definición de red residual, únicamente están los arcos que cumplen con esta condición). De esto podemos concluir que la trayectoria p es una trayectoria aumentante. Ahora bien, como p es una trayectoria admisible entonces $d(u_{i-1}) = d(u_i) + 1$. Por lo tanto, $d(s) = k$ y $d(s)$ es una cota mínima de las trayectorias aumentantes del vértice origen al vértice destino. Por lo anterior, concluimos que p es una trayectoria con longitud mínima.

■

3.7 Algoritmo de Trayectoria Aumentante de Longitud Mínima

El *Algoritmo de Trayectoria Aumentante de Longitud Mínima* (ATALM), presentado en [14], es una especialización concreta del Algoritmo Genérico de Trayectoria Aumentante, (AGTA). Por lo tanto, también incrementa el flujo a través de una trayectoria aumentante, con la peculiaridad de que dicha trayectoria es de longitud mínima (como su nombre lo indica). Además, la trayectoria aumentante es una trayectoria admisible.

El algoritmo ATALM inicialmente asigna las distancias etiquetadas a cada vértice de la red, con el fin de construir trayectorias admisibles a los vértices u para los cuales exista dicha trayectoria. Si en algún punto se ha construido una $s \rightsquigarrow t$ trayectoria admisible se aumenta el

flujo a través de ella. Y se repite el procedimiento. La manera en la cual el algoritmo construye las trayectorias es la siguiente: en cada iteración un vértice u es examinado; a este vértice le llamamos *vértice actual*. Cuando un vértice está siendo examinado se realiza una de las siguientes operaciones:

- Se hace una *operación de avance* si para el vértice actual existe una arco admisible, esta operación consiste en avanzar un arco, es otras palabras, actualizar el vértice actual con el vértice correspondiente a la segunda componente del arco por el cual estamos avanzando.

- En caso contrario, es decir, que el vértice actual no contenga ningún arco admisible, se realiza una *operación de retroceso*, la cual consiste en retroceder un arco, reetiquetar el vértice actual y sustituir el vértice actual con el vértice al cual retrocedemos.

Este último procedimiento se ejecuta mientras que t no sea el vértice actual. Una vez que llegamos al vértice destino habremos encontrado una trayectoria aumentante del vértice s al vértice t. Como en el algoritmo AGTA, todo el procedimiento se repite mientras sea posible encontrar una trayectoria de s a t.

La implementación del algoritmo se presenta a continuación:

Listado 3.7.1 Algoritmo de Trayectoria Aumentante de Longitud Mínima

```
 1 public class ATALM extends AGTA{
 2     public ATALM(Red r){
 3         super(r);
 4
 5     }
 6     protected boolean encuentraTrayectoriaAumentante (){
 7         VerticeEtiquetado s  = (VerticeEtiquetado)r.verticeOrigen();
 8         VerticeEtiquetado t  = (VerticeEtiquetado)r.verticeDestino();
 9         VerticeEtiquetado i  = s;
10         int  numeroDeVertices = r.redResidual().numeroDeVertices();
11         do{
12             Arista a = i.aristaAdmisible();
13             if( a != null)
14                 i = (VerticeEtiquetado)avanza(i,a);
15             else
16                 i = (VerticeEtiquetado)retrocede(i);
17         }while( (i != t) && (s.obtenDistancia() < numeroDeVertices) );
18         if(i == t){
19             trayectoria = new LinkedList();
20             cTrayectoria = 0;
21             construyeTrayectoria();
22             return true;
23         }
24         else
25             return false;
26     }
```

Listado 3.7.1 Algoritmo de Trayectoria Aumentante de Longitud Mínima (continuación)

```
27     protected void inicializa (){
28         Grafica      tmp         = r.redInicial().inversa();
29         Vertice      tR          = r.verticeDestino();
30         LinkedList   vertices    = tmp.obtenVertices();
31         LinkedList   verticesR   = r.redResidual().obtenVertices();
32         Vertice      t           = (Vertice)vertices.get(verticesR.indexOf(tR));
33         BFS          bfs         = new BFS(tmp,t);
34         bfs.busquedaPorAmplitud ();
35         int i;
36         for( i = 0; i < vertices.size(); i++){
37             VerticeEtiquetado v  =  (VerticeEtiquetado)vertices.get(i);
38             VerticeEtiquetado vR =
39                     (VerticeEtiquetado)verticesR.get(vertices.indexOf(v));
40             vR.defineDistancia(v.obtenDistancia());
41         }
42     }
43
44     protected Vertice retrocede(VerticeEtiquetado v){
45         int d = v.distanciaMinimaDeLosVerticesAdyacentes();
46         if(d == -1)
47             d = v.obtenDistancia();
48         v.defineDistancia(d+1);
49         if(v != r.verticeOrigen())
50             return ((Vertice)v.obtenPadre());
51         else
52             return v;
53     }
54
55     protected Vertice avanza(VerticeEtiquetado u, Arista a){
56         VerticeEtiquetado  v  =  (VerticeEtiquetado)a.obtenVertice();
57         v.definePadre(u);
58         return v;
59     }
60 }
```

3.7.1 Precondiciones y postcondiciones de los métodos de ATALM

Constructor

(public ATALM (Red r))

Valida el estado inicial del algoritmo, asignando un valor inicial a sus atributos.

Precondiciones: La clase debe ser invocada con una red.

Postcondiciones: Deja una representación de la red que cumple con las mismas condiciones dadas en AGTA, además de la siguiente: cada vértice tiene una variable para almacenar la distancia etiquetada que le corresponde; inicialmente dicha distancia es cero.

Preproceso

(protected void inicializa())

El método asigna las distancias etiquetadas válidas a cada vértice de la red de flujo.

Precondiciones: Ninguna.

Postcondiciones: Al terminar de ejecutarse el método las distancias de los vértices son válidas.

Encuentra una Trayectoria Aumentante

(**protected boolean** encuentraTrayectoriaAumentante ())
El método construye trayectorias admisibles del vértice s a los vértices u para los cuales sea posible construir dicha trayectoria; si en algún punto el vértice actual es t el método termina regresando **true**. Si en algún punto no es posible realizar ninguna operación de avance el método termina y regresa **false**.

Precondiciones: Ninguna.

Postcondiciones: Si el método regresa **true**, trayectoria es una trayectoria aumentante de longitud mínima. Si el método regresa **false**, no existe una trayectoria aumentante (trayectoria = **null**).

Avanza un Arco Admisible

(**protected void** avanza ())
Si el vértice u tiene un arco admisible (u, v), entonces al terminar de ejecutarse el método el vértice actual será v.

Precondiciones: El vértice u tiene un arco admisible (u, v).

Postcondiciones: Al terminar de ejecutarse el método se cumplen las siguientes condiciones,

- $padre(v) = u$
- El vértice actual es v.

Retrocede un Arco

(**protected** Vertice retrocede ())
Si el vértice actual u no tiene ningún arco admisible entonces retrocedemos un arco, es decir, el vértice actual será $padre(u)$.

Precondiciones: El vértice actual u no contiene un arco admisible.

Postcondiciones: Al terminar de ejecutarse el método se cumplen las siguientes condiciones:

- La distancia de u se habrá incrementado, es decir: sea $d(u)$ la distancias antes de ejecutar el método y $d'(u)$ la distancia una vez que fue ejecutado el método; entonces se cumple $d(u) < d'(u)$. Además las distancias siguen siendo válidas.
- El vértice actual es $padre(u)$.

3.7.2 Correctez de ATALM

Lema 3.20 *Una vez que el método inicializa fue ejecutado todos los vértices de la red tienen distancias etiquetadas válidas.*

Demostración:
Dado que el método ejecuta BFS sobre la red inversa, es decir voltea los arcos y el vértice del que parte es t, el algoritmo siempre encuentra la distancia de la trayectoria más corta de cualquier vértice u en la red al vértice destino t. Para cualquier arco (u, v) se cumple que $d(u) \leq d(v) + 1$. Si el arco está en la trayectoria más corta del vértice u a t, la demostración es inmediata. El caso interesante es cuando el arco no está en la trayectoria, en cuyo caso, existe el arco (u, v') el cual está en la trayectoria más corta del vértice u al vértice t y $d(v') \leq d(v)$; en este caso también se cumple la desigualdad porque $d(u) = d(v') + 1 \leq d(v) + 1$.

■

Lema 3.21 *Una vez que el método avanza fue ejecutado las distancias etiquetadas siguen siendo válidas.*

Demostración:
La demostración es inmediata. El método nunca redefine la distancia etiquetada de ningún vértice.

■

Lema 3.22 *El método retrocede mantiene las distancias etiquetadas válidas después de ser ejecutado. Más aún, incrementa la distancia etiquetada del vértice.*

Demostración:
Este método modifica la distancia del vértice v (línea 48) y por lo tanto debemos asegurarnos que esta nueva distancia es válida. Tenemos que verificar que para todos los arcos en los cuales v es el primer componente, es decir, para los arcos (v, u) tenemos $d(v) \leq d(u) + 1$. Veamos porque es así. El método sólo se ejecuta cuando el vértice v no tiene ningún arco admisible (línea 16); entonces para cualquier arco (v, u) tenemos $d(v) < d(u) + 1$. Y definimos la nueva distancia de v, digamos $d'(v)$, como $d'(v) = \min\{d(u) + 1 | (v, u) \in A(v) \text{ y } c_f(v, u) > 0\}$ (línea 45). De lo cual se sigue que $d'(v) \leq d(u) + 1$ para cualquier arco (u, v). Ahora solamente nos falta probar $d'(v)$ es válida para los arcos donde v es el segundo componente (esto es (u, v)). Por hipótesis tenemos que las distancias eran válidas al entrar al método, y por lo tanto $d(u) \leq d(v) + 1 \leq d'(v)$. De lo anterior concluimos que una vez que el método retrocede es ejecutado las distancias siguen siendo válidas.

■

Lema 3.23 *El método encuentraTrayectoriaAumentante preserva las distancias etiquetadas válidas.*

Demostración:
Dado que los métodos avanza y retrocede conservan las distancias etiquetadas válidas, y el método construyeTrayectoria no modifica las distancias de los vértices, el método encuentraTrayectoriaAumentante preserva las distancias etiquetadas válidas.

■

Lema 3.24 *El método actualizaRedResidual mantiene las distancias etiquetadas válidas después de ser ejecutado.*

Demostración:

Este método no modifica las distancias de los vértices pero crea nuevos arcos: si aumentamos δ unidades de flujo a través de un arco (u, v), entonces se crea el arco (v, u) en caso que no exista, que es justamente el caso que nos interesa, dado que si el arco ya existía por hipótesis tenemos que las distancias son válidas. Entonces, un aumento de flujo en el arco (u, v) crea un nuevo arco (v, u) tal que $c_f(v, u) > 0$ y se cumple $d(v) \leq d(u) + 1$. Como hubo un aumento de flujo a través del arco (u, v) entonces éste era un arco admisible, lo que significa que $d(u) = d(v) + 1$. Por lo tanto, se cumple la desigualdad $d(v) \leq d(u) + 1$.

■

Lema 3.25 *Al terminar de ejecutar el método encuentraTrayectoriaAumentante éste regresa* **false** *solamente si no existe una trayectoria admisible del vértice origen al vértice destino en la red residual inducida por G y el flujo actual f.*

Demostración:

El método regresa **false** solamente cuando $d(s) \geq n$ (línea 25), dado que el ciclo do ... **while** únicamente termina cuando hemos llegado hasta el vértice t o no se cumple que $d(s) < n$ (línea 17). Y en el primero de los casos el algoritmo regresa **true** (líneas 18-22). Ahora bien, por el lema 3.18 garantizamos que no existe una trayectoria aumentante en la red residual G_f.

■

Lema 3.26 *Al terminar de ejecutar el método encuentraTrayectoriaAumentante éste regresa Verdadero solamente si existe una trayectoria admisible del vértice origen al vértice destino. Además, trayectoria almacena dicha trayectoria.*

Demostración:

Observemos que el método únicamente regresa **true** si el vértice actual es igual a t, en cuyo caso t tiene asignado un padre que es el vértice desde el cual se descubrió. Lo mismo se sigue para el vértice $padre(t)$: si se revisaron sus arcos adyacentes antes debió de habérsele asignado un vértice padre. Y así sucesivamente hasta llegar al vértice s que es el vértice del cual partimos. Dado que si se cumple la condición i = t, es decir, el vértice actual es el vértice destino se cumple: padre(t), padre(padre(t)), ..., padre(...padre(t)...) $\neq$ null, entonces, una vez ejecutado el método construyeTrayectoria (línea 21) el lema 3.4 nos garantiza que trayectoria es una trayectoria aumentante.

■

Teorema 3.6 *El Algoritmo de Trayectoria Aumentante de Longitud Mínima ATALM encuentra un flujo máximo.*

Demostración:

El algoritmo sólo termina si el método encuentraTrayectoriaAumentante regresa falso. Por el lema 3.25 garantizamos que no existe una trayectoria aumentante del vértice s al vértice t. Y el teorema 1.1 nos garantiza que el flujo encontrado hasta el momento es un flujo máximo.

■

3.7.3 Complejidad de ATALM

Los siguientes resultados serán de gran utilidad en la demostración de la complejida de algoritmo ATALM.

Lema 3.27 *Suponiendo que el algoritmo ATALM reetiqueta un vértice v a lo más k veces, entonces el tiempo total que tarda en encontrar los arcos admisibles y reetiquetar todos los vértices es $O(km)$.*

Demostración:

Recordemos que cada vértice u tiene asignada una lista de adyacencias, denotada por $Ad(u)$, entonces el algoritmo busca en la lista de adyacencias un arco admisible revisando uno por uno en el orden en que están en $Ad(u)$; en el peor de los casos encuentra el arco al final de la lista, lo cual nos toma $|Ad(u)|$ pasos. En el caso que la lista de adyacencias del vértice u no contenga ningún vértice, se hará una operación de retroceso y dicho vértice será reetiquetado; esta operación también se lleva $|Ad(u)|$ pasos, dado que tiene que revisar todos los arcos para obtener la distancia mínima de los arcos adyacentes. Por lo tanto, si un vértice es reetiquetado a lo más k veces el algoritmo tarda $O(k \sum_{v \in V} |Ad(u)|)$ en reetiquetar todos los vértice y encontrar los arcos admisibles. De lo anterior concluimos, el tiempo total que nos lleva encontrar los arcos admisibles y reetiquetar los vértices es $O(km)$.

■

Lema 3.28 *Entre dos saturaciones consecutivas del arco (u, v) las distancias de los vértices pertenecientes a dicho arco se incrementan por lo menos en dos unidades, es decir, si d es la función de distancia antes de realizar las saturaciones y d' es la función de distancia después de dichas saturaciones, podemos asegurar que:*

$$d'(u) \geq d(u) + 2 \quad y \quad d'(v) \geq d(v) + 2$$

Demostración:

Supongamos que un incremento del flujo satura el arco (u, v), esto significa que el arco era admisible y por lo tanto se cumplía:

$$d(u) = d(v) + 1$$

Dado que $c_f(u, v) = 0$, necesitamos enviar flujo del vértice v al vértice u para que (u, v) tenga capacidad residual mayor a cero y pueda ser saturado nuevamente. Sea $d'(u)$ y $d'(v)$ las distancias de los vértices una vez que fue saturado el arco (u, v) la primera vez, debe cumplirse:

$$d'(v) = d'(u) + 1$$

Ya que utilizamos el arco (v, u) para enviar flujo dicho arco debe ser admisible. Si saturamos nuevamente el arco (u, v) también debe cumplirse:

$$d''(u) = d''(v) + 1$$

donde $d''(u)$ y $d''(v)$, son las distancias de u y v respectivamente una vez que se incrementó el flujo utilizando la arista (v, u).

Por el lema 3.22 y lo anterior podemos concluir:

$$\begin{aligned} d''(u) &= d''(v) + 1 \\ &\geq d'(v) + 1 \\ &= (d'(u) + 1) + 1 \\ &\geq d(u) + 2 \end{aligned}$$

Para el vértice v se demuestra de manera análoga.

■

Lema 3.29 *Si el algoritmo ATALM reetiqueta cualquier vértice a lo más k veces entonces el algoritmo realiza a lo más $\frac{km}{2}$ saturaciones de arcos.*

Demostración:
Por hipótesis tenemos que un vértice es reetiquetado a lo más k veces; además por el lema anterior tenemos que después de dos saturaciones consecutivas de un arco las distancias de los vértices correspondientes se incrementan en dos unidades; entonces un arco puede ser saturado a lo más $\frac{k}{2}$ veces. Por lo tanto, el número total de saturaciones es a lo más $\frac{km}{2}$.

■

Lema 3.30 *El algoritmo ATALM incrementa distancia de un vértice a lo más n veces. Consecuentemente, el número total de reetiquetamientos de los vértices es a lo más n^2.*

Demostración:
Por el lema 3.22 sabemos que después de que el método retrocede fue ejecutado, la distancia del vértice u se ve incrementada por lo menos en una unidad. Supongamos que dicho vértice es reetiquetado n veces; después de esto el vértice no se vuelve a reetiquetar porque $d(u) < d(s) < n$. Por lo tanto, el algoritmo reetiqueta un vértice a lo más n veces y el total de operaciones de reetiquetamiento (retroceso) es a lo más n^2.

■

Corolario 3.2 *El número total de operaciones de aumento es menor o igual a $\frac{nm}{2}$.*

Demostración:
Por el lema 3.29 y el lema anterior tenemos que el número total de saturaciones es $\frac{nm}{2}$. Y dado que cada aumento de flujo reduce la capacidad residual de por lo menos un arco a cero, es decir, lo satura, el número total de aumentos es a lo más $\frac{nm}{2}$.

■

Corolario 3.3 *El tiempo que lleva encontrar todos los arcos admisibles es $O(nm)$.*

Demostración:
La demostración es inmediata de los lemas 3.27 y 3.30.

■

La siguiente tabla nos muestra la complejidad de cada uno de los métodos definidos en ATALM.

Método	Complejidad	Justificación
avanza	$O(1)$	Solamente tiene que actualizar el vértice actual.
retrocede	$O(1)$	Únicamente reetiqueta la distancia del vértice u.
inicializa	$O(m+n)$	Es la complejidad de BFS más el número de vértices.
encuentraTrayectoriaAumentante	$O(n)$	Revisa los vértices para realizar una operación de avance.

Teorema 3.7 *El orden del algoritmo $ATALM$ es $O(n^2m)$.*

Demostración:
El tiempo total que lleva encontrar los arcos admisibles y reetiquetar los vértices es $O(nm)$. El número de aumentos es a lo más $O(nm)$. Y dado que cada aumento requiere $O(n)$, el tiempo total que se llevan todos los aumentos es $O(n^2m)$. Por otro lado, el número total de operaciones de retroceso es a lo más $O(n^2)$. Ahora bien, cada operación de retroceso elimina un arco de la trayectoria parcial que se construye, mientras que cada operación de avance agrega un arco a ésta. Como cualquier trayectoria aumentante del vértice s al vértice t no puede ser mayor que $n-1$, entonces el número total de avances es $O(n^2+n^2m)$, donde el primer término corresponde al número total de retrocesos y el segundo al número de aumentos. Por lo tanto, la complejidad del algoritmo es $O(n^2m)$.

Observemos que sucede con nuestra ecuación general. Dado que el número de aumentos es a lo más nm, el valor de $q = nm$.

$$
\begin{aligned}
f_{ATALM} &= f_{inicializa} + \sum_{i=1}^{q}(f_{encuentraTrayectoriaAumentante} + O(n)) \\
&= O(m+n) + \sum_{i=1}^{mn}(O(n) + O(n))) \\
&= O(m+n) + mn(O(n) + O(n)))
\end{aligned}
$$

Como las redes 2−conexas [4] $n < m$, $f_{ATALM} = O(n^2m)$.

■

3.8 Algoritmo Dinic

El *Algoritmo Dinic* (ADinic) [11], es un algoritmo que extiende a AGTA, por lo que, también resuelve el problema de flujo máximo incrementando el flujo a través de trayectorias aumentantes. Este algoritmo construye una *red de capas* a partir de las distancias etiquetadas. Intuitivamente, una red de capas es una gráfica dirigida formada por las trayectorias más cortas del vértice origen a cada uno de los vértices $v \in V$. Formalmente

[4] Nuevamente, para el caso en que la gráfica correspondiente a la red sea un árbol el problema es trivial.

Definición 3.4 *Definimos una* red de capas V *con respecto a un flujo dado* f *de la siguiente manera: una vez determinadas las distancias etiquetadas* d *en* G_f*, la red de capas consiste de aquellos arcos* (u, v) *en* G_f *que satisfacen la condición* $d(u) = d(v) + 1$. *Además el conjunto de vértices es particionado en clases* $V_0, V_1, \ldots, V_2$. *El conjunto* V_k *contiene a todos los vértices cuya distancia etiquetada es igual a* k*; además, para cada arco* (u, v) *en la red de capas,* $u \in V_k$ *y* $v \in V_{k-1}$ *para alguna* k.

Como ya mencionamos, el algoritmo ADinic construye una red de capas y aumenta el flujo en la misma mientras exista una trayectoria aumentante en ésta. Cuando la red de capas no contiene ninguna trayectoria aumentante se establece un *flujo bloqueado*. Después de establecerse dicho flujo se recalculan las distancias etiquetadas para volver a construir una nueva red de capas y repetir el mismo procedimiento. El algoritmo termina cuando en una red de capas recién construida no es posible encontrar una trayectoria aumentante. Cada construcción de una red de capas se le conoce como una *fase*.

El algoritmo ADinic es semejante a ATALM ya que también construye trayectorias admisibles del vértice s a los vértices u para los cuales exista dicha trayectoria. Pero tiene las siguientes diferencias:

1. En la operación de retroceso no cambiamos la distancia etiquetada del vértice u, sólo marcamos al vértice como bloqueado. Una vez que un vértice ha sido bloqueado garantizamos que éste no tiene una trayectoria admisible al vértice t, esto es, no existe una $u \rightsquigarrow t$ trayectoria en la red de capas actual.

2. Definimos un arco (u, v) admisible si $d(u) = d(v) + 1$, $c_f(u, v) > 0$ y el vértice v está bloqueado.

3. Cuando el vértice s ha sido bloqueado, volvemos a calcular las distancias etiquetadas de todos los vértices.

Observemos que el tamaño de la trayectoria más corta es $d(s)$ y esta distancia se incrementa cada vez que establecemos un flujo bloqueado y formamos una nueva red de capas correspondiente al flujo actual.

El siguiente lema formaliza este resultado.

Lema 3.31

a). Sea p *una trayectoria aumentante de longitud mínima en* G*, sea* G' *la gráfica residual obtenida al aumentar el flujo a través de* p*, y sea* q *una trayectoria aumentante de longitud mínima en* G'. *Entonces* $|q| \geq |p|$.

b). Podemos aumentar el flujo a través de trayectorias aumentantes de la misma longitud a lo más m *veces.*

Demostración:

Sea p cualquier trayectoria de s a t en la red de capas a través de la cual incrementamos el flujo. Dado que c_p es la mínima capacidad de los arcos que la forman, al menos un arco es saturado y desaparecerá de la red residual. Ahora bien, a lo más son añadidos a la red residual

$|p|$ arcos, los cuales son arcos de retroceso y no pueden contribuir para que exista una trayectoria más corta de s a t mientras t sea alcanzable desde s en la red de capas. Este procedimiento se repite mientras sea posible encontrar una trayectoria aumentante en la red de capas, lo cual es posible hacer a lo más m veces dado que en cada aumento desaparecen los arcos de p con capacidad mínima, y hay al menos uno de éstos. Una vez que t ya no es alcanzable desde s en la red de capas, cualquier otra trayectoria debería usar los arcos de retroceso o los arcos que no estaban en la red de capas, en cuyo caso la longitud de esta trayectoria se incrementa.

La implementación del algoritmo se presenta a continuación:

Listado 3.8.1 Algoritmo Dinic

```
1  public class ADINIC extends AGTA{
2      public ADINIC(Red r){
3          super(r);
4      }
5
6      protected boolean encuentraTrayectoriaAumentante(){
7          VerticeEtiquetado s = (VerticeEtiquetado)r.verticeOrigen();
8          VerticeEtiquetado t = (VerticeEtiquetado)r.verticeDestino();
9          VerticeEtiquetado i = s;
10         int numeroDeVertices = r.redResidual().numeroDeVertices();
11         do{
12             Arista a = i.aristaAdmisible();
13             if(a != null)
14                 i = (VerticeEtiquetado)avanza(i,a);
15             else
16                 i = (VerticeEtiquetado)retrocede(i);
17             if(i == s && s.estaEtiquetado()){
18                 inicializa();
19                 s = (VerticeEtiquetado)r.verticeOrigen();
20                 t = (VerticeEtiquetado)r.verticeDestino();
21                 i = s;
22             }
23
24         }while((i != t) && (s.obtenDistancia() < numeroDeVertices));
25         if(i == t){
26             trayectoria = new LinkedList();
27             cTrayectoria = 0;
28             construyeTrayectoria();
29             return true;
30         }
31         else
32             return false;
33     }
34
35     public void inicializa (){
36         creaRedResidual();
37         Grafica     tmp        = r.redInicial().inversa();
38         Vertice     tR         = r.verticeDestino();
39         LinkedList  vertices   = tmp.obtenVertices();
40         LinkedList  verticesR  = r.redResidual().obtenVertices();
```

Listado 3.8.1 Algoritmo Dinic (continuación)

```
41         Vertice      t           = (Vertice) vertices.get(verticesR.indexOf(tR));
42         BFS          bfs         = new BFS(tmp, t);
43         bfs.busquedaPorAmplitud ();
44         int i;
45         for( i = 0; i < vertices.size(); i++){
46             VerticeEtiquetado v =
47                     (VerticeEtiquetado) vertices.get(i);
48             VerticeEtiquetado vR =
49                     (VerticeEtiquetado) verticesR.get(vertices.indexOf(v));
50             vR.defineDistancia(v.obtenDistancia());
51             vR.desetiqueta();
52             vR.definePadre(null);
53         }
54         VerticeEtiquetado s = (VerticeEtiquetado)r.verticeOrigen();
55         if(s.obtenDistancia() == -1)
56             s.defineDistancia (vertices.size());
57         creaRedDeCapas();
58     }
59
60     public void creaRedResidual () {
61         r.copiaRedInicialEnRedResidual ();
62         LinkedList vertices = r.redResidual().obtenVertices();
63         int i;
64
65         for(i = 0; i < vertices.size(); i++){
66             VerticeEtiquetado u = (VerticeEtiquetado) vertices.get(i);
67             LinkedList ady = u.obtenAdyacencias();
68             int j;
69             for(j = 0; j < ady.size(); j++){
70                 Arista aristaUV   = (Arista)ady.get(j);
71                 int    capacidad = aristaUV.obtenCapacidad();
72                 int    flujo     = aristaUV.obtenFlujo();
73                 VerticeEtiquetado v =
74                         (VerticeEtiquetado) aristaUV.obtenVertice();
75                 if(  (capacidad - flujo) <= 0 ){
76                     u.desconectaDe(v);
77                     j--;
78
79                 }else
80                     aristaUV.defineCapacidad(capacidad - flujo);
81                 if(flujo > 0)
82                     v.conectaCon(u, flujo);
83             }
84         }
85     }
86
87     public void creaRedDeCapas () {
88         LinkedList vertices = r.redResidual().obtenVertices();
89
90         int i;
91         for(i = 0; i < vertices.size(); i++ ){
92             VerticeEtiquetado u = (VerticeEtiquetado) vertices.get(i);
93             LinkedList ady = u.obtenAdyacencias();
```

Listado 3.8.1 Algoritmo Dinic (continuación)

```
 94                 int j;
 95                 for(j = 0; j < ady.size(); j++ ){
 96                     Arista aristaUV = (Arista)ady.get(j);
 97                     VerticeEtiquetado v =
 98                            (VerticeEtiquetado)aristaUV.obtenVertice();
 99                     if(u.obtenDistancia() != (v.obtenDistancia()+1) )
100                         u.desconectaDe(v);
101                 }
102             }
103         }
104
105         public Vertice retrocede(VerticeEtiquetado v){
106             v.etiqueta();
107             if(v != r.verticeOrigen())
108                 return ((Vertice)v.obtenPadre());
109             else
110                 return v;
111         }
112
113         public Vertice avanza(VerticeEtiquetado u, Arista a){
114             VerticeEtiquetado v = (VerticeEtiquetado)a.obtenVertice();
115             v.definePadre(u);
116             return v;
117         }
118 }
```

3.8.1 Precondiciones y postcondiciones de los métodos de ADinic

Constructor

(**public** ADinic (Red r))
Valida el estado inicial del algoritmo, asignando un valor inicial a sus atributos.

Precondiciones: La clase debe ser invocada con una red.

Postcondiciones: Deja una representación de la red que cumple con las mismas condiciones dadas en AGTA, además de las siguientes:

- Cada vértice tiene una variable para almacenar la distancia etiquetada que le corresponde; inicialmente dicha distancia es cero.
- Cada vértice tiene asociada una variable booleana que indica si el vértice está bloqueado o no.

Crea Red Residual

(**private void** creaRedResidual ()) El método construye la red residual r.redResidual a partir de la red original y el flujo actual.

Precondiciones: El vértice s ha sido bloqueado.

Postcondiciones: Si el arco $(u, v) \in$ r.redResidual entonces se cumple una y sólo una de las siguientes condiciones:

- $(u, v) \in$ redOriginal y $c(u, v) - f(u, v) > 0$
- $(v, u) \in$ redOriginal y $f(v, u) > 0$

Crea Red de Capas

(**private void** creaRedDeCapas ()) El método actualiza la red residual r.redResidual dejando únicamente los arcos que son admisibles.

Precondiciones: Están definidas las distancias etiquetadas de los vértices.

Postcondiciones: Si el arco $(u, v) \in$ r.redResidual entonces se cumple:

$$d(u) = d(v) + 1$$

Encuentra una Trayectoria Aumentante

(protected boolean encuentraTrayectoriaAumentante ())
El método construye trayectorias admisibles del vértice s a los vértices u para las cuales sea posible construir dicha trayectoria; si en algún momento no es posible avanzar desde el vértice actual, éste se bloquea, indicando con ello que no existe una trayectoria admisible de este vértice al vértice destino. Cuando el vértice s ha sido bloqueado se vuelve a asignar las distancias etiquetas a los vértices. El método regresa **true** en el momento en que el vértice actual sea t, y regresa **false** una vez $d(s) \geq n$ lo cual implica que no existe una trayectoria de s a t.

Precondiciones: Ninguna.

Postcondiciones: Si el método regresa **true**, trayectoria es una trayectoria aumentante de longitud mínima. Si el método regresa **false**, no existe una trayectoria aumentante (trayectoria = **null**).

Avanza un Arco Admisible

(**protected void** avanza ())
Si el vértice u tiene un arco admisible (u, v), entonces, al terminar de ejecutarse el método, el vértice actual será v.

Precondiciones: El vértice u tiene un arco admisible (u, v).

Postcondiciones: Al terminar de ejecutarse el método se cumplen las siguientes condiciones.

- $padre(v) = u$.
- El vértice actual es v.

Retrocede un Arco

(protected Vertice retrocede ())
Si el vértice actual u no tiene ningún arco admisible entonces retrocedemos un arco, es decir, el vértice actual será $padre(u)$.

Precondiciones: El vértice actual u no contiene un arco admisible.

Postcondiciones: Al terminar de ejecutarse el método se cumplen las siguientes condiciones:

- El vértice u es etiquetado, es decir, bloqueado.
- El vértice actual es $padre(u)$.

Preproceso

(protected void inicializa())
El método asigna las distancias etiquetadas válidas a cada vértice de la red y la red residual r.redResidual es una red de capas.

Precondiciones: Ninguna.

Postcondiciones: Al terminar de ejecutarse el método las distancias de los vértices son válidas. Además, r.redResidual es una red de capas.

3.8.2 Correctez de ADinic

Lema 3.32 *El método creaRedResidual() construye la red residual r.redResidual a partir de la red original y el flujo actual, donde el conjunto de arcos está formado por todos los arcos (u, v) tales que:*

- $(u, v) \in$ *redOriginal y* $c(u, v) - f(u, v) > 0$ *y* $c_f(u, v) = c(u, v) - f(u, v)$*, o bien*
- $(v, u) \in$ *redOriginal y* $f(v, u) > 0$ *y* $c_f(u, v) = 0$.
- *Únicamente se debe cumplir una de las condiciones anteriores.*

Demostración:
Lo primero que hace el método es copiar la red original en la red residual; después revisa la lista de adyacencias de todos los vértices (líneas 66-70) para definir la capacidad residual de ellos. Si algún arco (u, v) cumple con $c(u, v) - f(u, v) \leq 0$ lo elimina (líneas 76-77); en caso contrario actualiza su capacidad en la red residual $c_f(u, v) = c(u, v) - f(u, v)$ (línea 80). Además, crea los arcos (u, v) si el arco $(v, u) \in$ r.redOriginal y $f(v, u) > 0$ con $c_f(u, v) = f(v, u)$ (líneas 81 y 82). El método solamente realiza lo anterior.

∎

Lema 3.33 *El método creaRedDeCapas() actualiza la red residual r.redResidual dejando únicamente los arcos que son admisibles.*

Demostración:
El método revisa la lista de adyacencias de todos los vértices de r.redResidual (líneas 91-95), y elimina los arcos (u, v) para los cuales no se cumple $d(u) = d(v) + 1$ (líneas 99 y 100), es decir, elimina todos los arcos que no son admisibles.

■

Lema 3.34 *Si el vértice actual u no tiene ningún arco admisible entonces retrocedemos un arco, es decir, el vértice actual será $padre(u)$.*

Demostración:
El único momento en el que el método se llama es cuando el vértice actual u no tiene un arco admisible (línea 16), y el método únicamente regresar el vértice ($padre(u)$) (línea 108) siempre y cuando el vértice actual no sea s (línea 107); este vértice es el asignado al vértice actual (línea 16).

■

Lema 3.35 *Si el vértice u tiene un arco admisible (u, v), entonces, al terminar de ejecutarse el método, el vértice actual será v.*

Demostración:
El método sólo se ejecuta si el vértice actual tiene un arco admisible $a = (u, v)$ (líneas 13 y 14) y lo único que hace es regresar el valor del vértice v (línea 116) una vez que estableció $padre(v) = u$ (línea 115).

■

Lema 3.36 *En cada fase del algoritmo, es decir, cada vez que se forma una red de capas, $d(s)$ es estrictamente incrementada.*

Demostración:
Por el lema 3.31 tenemos que después de aumentar el flujo a través de una trayectoria p cualquier trayectoria aumentante en la red residual, inducida por p, tiene longitud al menos $|p|$. Ahora bien, el único momento en el cual las distancias son asignadas es en el método inicializa (línea 56) y éste únicamente se ejecuta cuando no existe una trayectoria aumentante del vértice origen al vértice destino en la red de capas actual, que es lo mismo que decir que s esté bloqueado (líneas 17-18). Entonces, se actualiza la red residual y se reetiquetan las distancias de los vértices de la misma para obtener la nueva red de capas. Por lo tanto, cualquier trayectoria en esta nueva red debe tener longitud mayor a las anteriores. Dado que $d(s)$ es la longitud de la trayectoria más corta en la red, podemos concluir que $d(s)$ se incrementa después de cada fase.

■

Corolario 3.4 *El algoritmo ADinic ejecuta a lo más n fases.*

Demostración:
La demostración es directa del lema anterior, dado que en cada fase la distancia de s es incrementada por lo menos una unidad y la condición del algoritmo para continuar iterando

es que exista una trayectoria aumentante del vértice origen al vértice destino; en este caso, es equivalente a decir que $d(s) < n$. Entonces, podemos concluir que el algoritmo realiza a lo más n fases.

■

Teorema 3.8 *El algoritmo ADinic encuentra un flujo máximo.*

Demostración:
Por el lema 3.36 y el corolario anterior podemos ver que que en la última fase la distancia de $d(s) \geq n$. Y por lema 3.18 sabemos que no existe una trayectoria aumentante del vértice s al vértice t en la red residual. Por lo tanto, el flujo encontrado es un flujo máximo.

■

3.8.3 Complejidad de ADinic

Los siguientes resultados serán de gran utilidad en la demostración de la complejida del algoritmo [illegible].

■

Lema 3.37 *El número total de llamadas al método avanza en cada fase es $O(mn)$.*

Demostración:
Podemos hacer a lo más $2mn$ avances en cada fase, porque podemos realizar a lo más n avances por cada aumento o retroceso, y existen a lo más m aumentos y m retrocesos. Dado que el método se ejecuta en tiempo constante, el tiempo total que toma realizar todas las operaciones de avance es $O(mn)$.

■

Lema 3.38 *El tiempo total que toma realizar todas las operaciones de retroceso durante una fase es $O(m + n)$.*

Demostración:
Existen a lo más n retrocesos por cada fase, porque al menos un vértice es eliminado en cada retroceso. Dado que el tiempo que me lleva retroceder es constante, sólo nos fijamos en los arcos que se pueden borrar, que a lo más son m. Por lo tanto, el tiempo total que lleva hacer todos los retrocesos es $O(m + n)$.

■

Lema 3.39 *El tiempo total que me lleva realizar los aumentos de flujo en una fase es $O(mn)$.*

Demostración:
Hay a lo más m aumentos en cada fase, porque un arco es borrado en cada aumento. Cada aumento toma $O(n)$, dado que se tienen que actualizar las capacidades de los arcos de la trayectoria aumentante. De lo anterior, podemos concluir que el tiempo total que se requiere para realizar todos los incrementos de flujo en una fase es $O(mn)$.

■

La siguiente tabla nos muestra la complejidad de cada uno de los métodos de ADinic.

Método	Complejidad	Justificación
avanza	$O(1)$	Solamente tiene que redefinir el vértice actual.
retrocede	$O(1)$	Únicamente reetiqueta la distancia del vértice u.
inicializa	$O(m+n)$	Es la complejidad de BFS más el número de vértices.
creaRedResidual	$O(m)$	Revisa todos los arcos, para ver cuales tienen capacidad residual o flujo mayor a cero.
creaRedDeCapas	$O(m)$	Revisa todos los arcos, para eliminar las que no son admisibles.
encuentraTrayectoriaAumentante	$O(n)$	Revisa los vértices para realizar una operación de avance, y a lo más son n.

Teorema 3.9 *La complejidad de ADinic es* $O(n^2m)$.

Demostración:
Dado que cada fase requiere $O(mn)$ para incrementar el flujo durante esa fase y el número de fases es a lo más n, la complejidad de ADinic es $O(n^2m)$.
Observemos que sucede con nuestra ecuación general. Como que el número de aumentos es a lo más nm, el valor de $q = nm$.

$$\begin{aligned} f_{ADinic} &= f_{inicializa} + \sum_{i=1}^{q}(f_{encuentraTrayectoriaAumentante} + O(n)) \\ &= O(n+m) + \sum_{i=1}^{mn}(O(n) + O(n)) \\ &= O(n+m) + mn(O(n) + O(n)) \end{aligned}$$

Como las gráficas asociadas a una red son conexas $n < m$, $f_{ADinic} = O(n^2m)$.

■

Con esto damos por terminado el capítulo correspondiente a los algoritmos concretos del Algoritmo Genérico de Flujo Máximo basados en la idea de aumentar el flujo a través de trayectorias aumentantes.

Capítulo 4

Algoritmos de Preflujo

En el capítulo anterior se mencionan tres diferentes formas por medio de las cuales se puede reducir la complejidad de los algoritmos de flujo máximo. Las dos primeras se estudiaron en el mismo; recordemos que éstas consistían en encontrar trayectorias aumentantes del vértice s al vértice t con capacidad lo "suficientemente grande" y la idea de la segunda es enviar flujo a través de trayectorias de longitud mínima. Por lo tanto, únicamente nos falta explicar en qué consiste el tercer método; la idea básica de este método es enviar la mayor cantidad de flujo posible del vértice origen al vértice destino empujándolo por cada una de las aristas de forma individual. Observemos que cuando incrementamos el flujo a través de una trayectoria aumentante lo que realmente estamos haciendo es empujar flujo a través de cada uno de los arcos que pertenecen a dicha trayectoria, y esto lo repetimos con cada una de las trayectorias aumentantes. Los algoritmos que presentaremos en este capítulo resuelven el problema de flujo máximo empujando flujo a través de los arcos de manera individual, de tal forma, desde el principio enviamos todo el flujo que sea posible por los arcos de la forma (s, v) intentando que la mayor cantidad de flujo que enviamos no sea regresada al vértice origen y llegue al vértice destino. Una vez enviadas la mayor cantidad de flujo a los vértices adyacentes al vértice s, éstos envían todo el flujo posible a través de los arcos que son adyacentes a ellos (esto es, (v, v')), y así sucesivamente, hasta que la mayor cantidad de flujo llega el vértice destino. La cantidad de flujo que no pueda ser enviada desde un vértice v hasta el destino es regresada al origen.

Observemos que en este método se permite que un vértice u reciba más flujo del que envía; a esta cantidad la llamaremos el *exceso de u*. Por lo tanto, no se satisface la conservación del flujo en cada paso intermedio, pero como todo el flujo que no puede ser enviado al vértice destino se regresa al origen, al final del algoritmo dicha condición siempre se cumple. Al hecho de enviar flujo a través de arcos de manera individual le llamamos *preflujo*, y la definición formal se presenta a continuación:

Definición 4.1 *Un* preflujo *es una función $f : A \rightarrow \mathbb{R}$ que satisface las siguientes condiciones:*

1. Restricción de la capacidad: *El flujo de una arco $(u, v) \in A$ nunca debe exceder su capacidad:*

$$\forall (u, v) \in A \qquad f(u, v) \leq c(u, v)$$

2. *Para cualquier vértice $u \in V - \{s, t\}$ se cumple:*

$$\sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v) \geq 0$$

Dado un preflujo f, definimos el exceso de cada vértice u como el flujo que es enviado al vértice u y que éste no envía. Formalmente tenemos:

Definición 4.2 *El* exceso de flujo *de un vértice u, denotado por $e(u)$, se define como:*

$$e(u) = \sum_{v \in V} f(v,u) - \sum_{v \in V} f(u,v)$$

En un preflujo el exceso de todos los vértices $u \in V - \{s,t\}$ es un número entero no negativo ($e(u) \geq 0$). Dado que ningún arco tiene como segundo componente al vértice t, el exceso del flujo de t es mayor que cero. Por lo tanto, el vértice s es el único vértice que tiene exceso negativo.

Decimos que un vértice u está *activo* si $e(u) > 0$

4.1 Algoritmo Genérico de Preflujo

El *Algoritmo Genérico de Preflujo* (AGP) [19], se relaja la ley de conservación de flujo y se permite que en iteraciones intermedias se "viole", pero al final de la ejecución todos los vértices $v \in V - \{s,t\}$ deben satisfacerla. Mientras un vértice no cumple con la ley de la conservación de flujo se etiqueta como *activo.* La idea principal del algoritmo es encontrar un vértice activo y deshacerse del exceso empujando el flujo a través de los arcos adyacentes a éste; en otras palabras, trata de eliminar el exceso enviándolo a sus vértices vecinos. La idea que sigue para encontrar el flujo máximo es la siguiente: sea u un vértice activo; como el objetivo es enviar el flujo al vértice t, entonces u empuja el exceso de flujo a través de un arco (u,v) tal que v sea el vértice más cercano a t, lo cual es equivalente a enviar el flujo a través de un arco admisible. Si el vértice activo que estamos considerando no tiene ningún arco admisible, incrementamos su distancia etiquetada hasta que tengamos al menos uno. El algoritmo termina cuando la red residual no contiene ningún vértice activo.

Observemos que un empuje de δ unidades de flujo de un vértice activo u a un vértice v decrementa el exceso del vértice u y la capacidad residual del arco (u,v) en δ unidades, e incrementa el exceso del vértice v y la capacidad residual del arco (v,u) en δ unidades. De lo anterior tenemos la siguiente definición:

Definición 4.3 *Decimos que un empuje de flujo es* saturado *si las unidades de flujo que se envían son igual a la capacidad residual del arco; en caso contrario decimos que es* no saturado.

Observemos que un empuje no saturado al vértice u reduce el exceso de flujo a cero, mientras que un empuje saturado no logra deshacerse de todo el exceso del vértice activo.

El algoritmo AGP realiza las siguientes tareas: lo primero que hace es enviar flujo a los vértices adyacentes a s y definir $d(s) = n$. Por lo tanto, para cualquier vértice u adyacente a s se cumple $e(u) > 0$ garantizando que existe al menos un vértice con exceso positivo, de lo contrario no habría ningún arco adyacente a s. Después, empuja flujo desde un vértice activo u a un vértice v para el cual existe un arco admisible (u,v) en la red residual, y repite este procedimiento mientras exista un vértice activo u distinto de s y t. Cuando ya no hay ningún vértice activo podemos asegurar que el preflujo actual es un flujo, dado que para cualquier

vértice excepto s y t se cumple la conservación de flujo, y también garantizamos que no existe ninguna trayectoria aumentante desde el vértice s al vértice t, ya que $d(s) \geq n$ —el algoritmo nunca decrementa las distancias y en un principio se definió $d(s) = n$. Por lo tanto, el algoritmo encuentra un flujo máximo.

Antes de presentar la implementación del Algoritmo Genérico de Preflujo, ilustraremos su funcionamiento con el siguiente ejemplo.

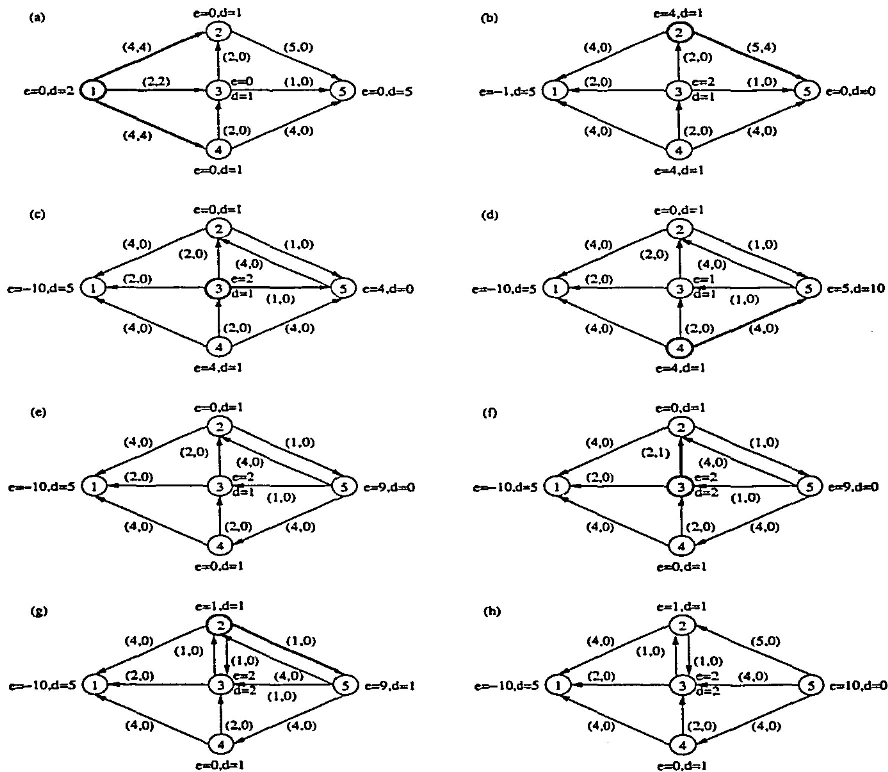


Figura 4.1: Ejemplo del funcionamiento de AGP

Revisemos la figura 4.1. La red inicial se muestra en la figura (a); en la figura (b) se muestra el preflujo determinado por el preproceso, es decir, después de enviar flujo a través de los arcos admisibles adyacentes a 1 . Supongamos que en la primera iteración del algoritmo el vértice activo seleccionado es 2, entonces, realizamos un empuje saturado a través del arco admisible $(2,5)$ de δ unidades de flujo, donde $\delta = \min\{e(2), c(2,5)\} = 4$. El resultado de este empuje saturado se muestra en la figura (c). En la siguiente iteración el vértice seleccionado es 1, empujando δ unidades de flujo a través del arco $(3,5)$ con $\delta = \min\{e(3), c(3,5)\} = 1$. El estado de la red después de realizar este empuje saturado se muestra en la figura (d). Supongamos que el siguiente vértice que elegimos es 4; entonces realizamos un empuje saturado a través de $(4,5)$ de δ unidades donde $\delta = \min\{e(4), (4,5)\} = 4$; el resultado de esta operación se presenta en la figura (e). Ahora, el único vértice activo es 3 pero como no tiene ningún arco admisible realizamos un reetiquetamiento; el resultado de éste se muestra en la figura (f). En la figura (g) se muestra el resultado después de hacer un empuje no saturado a través del arco $(3,2)$. Y por último en la figura (h) se muestra el estado final de la red, después de hacer un empuje saturado del vértice 2 al vértice destino 5. Dado que la red no contiene ningún vértice activo, excepto 5, el algoritmo termina. Observemos que el preflujo final es un flujo dado que se cumple la propiedad de conservación de flujo. Además, como $d(s) = n$ no existe ninguna trayectoria aumentante de 1 a 5, por lo tanto, el flujo actual es máximo.

La implementación del algoritmo se presenta a continuación:

Listado 4.1.1 Algoritmo Genérico de Preflujo

```
1  public abstract class AGP extends AGFM{
2
3      public AGP (Red r){
4          super (r);
5          asignaDistanciasEtiquetadas();
6      }
7
8      protected boolean esPosibleIncrementarElFlujo (){
9          return contieneVerticeActivo ();
10     }
11
12     protected void incrementaFlujo (){
13         VerticeEtiquetado u = seleccionaVerticeActivo();
14         Arista a = u.aristaAdmisible();
15         if(a != null){
16             empujaFlujo (u,a);
17         } else
18             reetiqueta(u);
19
20     }
21
22     protected int valorDelFlujoMaximo (){
23         VerticeEtiquetado t = (VerticeEtiquetado)r.verticeDestino();
24         flujoMaximo = t.obtenExceso();
25         return flujoMaximo;
26     }
```

Listado 4.1.1 Algoritmo Genérico de Preflujo (continuación)

```
27      protected void asignaDistanciasEtiquetadas (){
28          Grafica tmp = r.redInicial().inversa();
29          Vertice tR = r.verticeDestino();
30          Vertice t  = (Vertice)vertices.get(verticesR.indexOf(tR));
31          LinkedList vertices = tmp.obtenVertices();
32          LinkedList verticesR = r.redResidual().obtenVertices();
33          Vertice t = (Vertice)vertices.get( verticesR.indexOf(tR) );
34          BFS   bfs = new BFS(tmp,t);
35          bfs.busquedaPorAmplitud ();
36          int i;
37          for( i = 0; i < vertices.size(); i++){
38              VerticeEtiquetado v =
39                     (VerticeEtiquetado)vertices.get(i);
40              VerticeEtiquetado vR =
41                     (VerticeEtiquetado)verticesR.get(vertices.indexOf(v));
42              vR.defineDistancia(v.obtenDistancia());
43          }
44      }
45
46      protected void inicializa (){
47          VerticeEtiquetado s = (VerticeEtiquetado)r.verticeOrigen();
48          LinkedList       ady = s.obtenAdyacencias();
49          int               na = ady.size();
50
51          int i;
52          for(i = 0; i < na; i++) {
53              Arista              sv   = (Arista) ady.get(0);
54              VerticeEtiquetado   v    = (VerticeEtiquetado)sv.obtenVertice();
55              int                 c    = sv.obtenCapacidad();
56              s.decrementaCapacidad(v,c);
57              v.aumentaExceso(c);
58              s.disminuyeExceso(c);
59              agregaVerticeActivo(v);
60              if(v.estaConectadoCon(s))
61                  v.incrementaCapacidad(s,c);
62              else
63                  v.conectaCon(s,c);
64              actualizaFlujoEnRedOriginal(s,v,c);
65          }
66          s.defineDistancia(r.redResidual().numeroDeVertices());
67      }
68
69      protected  void empujaFlujo (VerticeEtiquetado u, Arista a){
70          VerticeEtiquetado v = (VerticeEtiquetado)a.obtenVertice();
71
72          int delta  = u.obtenExceso();
73          if (delta > a.obtenCapacidad())
74              delta = a.obtenCapacidad();
75          else
76              eliminaVerticeActivo(u);
77
78          if(v != (VerticeEtiquetado)r.verticeDestino()
79             && v != (VerticeEtiquetado)r.verticeOrigen() )
80              agregaVerticeActivo(v);
```

Listado 4.1.1 Algoritmo Genérico de Preflujo (continuación)

```
 81             u.decrementaCapacidad(v, delta);
 82             v.aumentaExceso(delta);
 83             u.disminuyeExceso(delta);
 84
 85             if(v.estaConectadoCon(u))
 86                 v.incrementaCapacidad(u, delta);
 87             else
 88                 v.conectaCon(u, delta);
 89
 90             actualizaFlujoEnRedOriginal(u,v, delta);
 91         }
 92
 93         protected void reetiqueta (VerticeEtiquetado u){
 94             LinkedList          ady   =   u.obtenAdyacencias();
 95             if(ady.size() > 0){
 96                 eliminaVerticeActivo(u);
 97
 98                 Arista                  uv   = (Arista) ady.get(0);
 99                 VerticeEtiquetado        v   = (VerticeEtiquetado)uv.obtenVertice();
100                 int                      d   = v.obtenDistancia();
101                 int i;
102                 for(i = 1; i < ady.size(); i++){
103                     uv   = (Arista) ady.get(i);
104                     v    = (VerticeEtiquetado)uv.obtenVertice();
105
106                     if(v.obtenDistancia() < d)
107                         d =  v.obtenDistancia();
108                 }
109                 u.defineDistancia(d+1);
110                 agregaVerticeActivo(u);
111             }
112
113         }
114
115         protected void actualizaFlujoEnRedOriginal(Vertice u,Vertice v,int delta)
116         {
117             LinkedList verticesRedResidual = r.redResidual().obtenVertices();
118             LinkedList verticesRedInicial  = r.redInicial().obtenVertices();
119             int indiceU = verticesRedResidual.indexOf(u);
120             int indiceV = verticesRedResidual.indexOf(v);
121             Vertice u1 = r.redInicial().obten(indiceU);
122             Vertice v1 = r.redInicial().obten(indiceV);
123             Arista a = u1.obtenArista(v1);
124             if(a != null)
125                 a.incrementaFlujo(delta);
126             else{
127                 a = v1.obtenArista(u1);
128                 if (a != null  )
129                     a.decrementaFlujo(delta);
130             }
131         }
```

Listado 4.1.1 Algoritmo Genérico de Preflujo (continuación)

```
132     protected abstract VerticeEtiquetado seleccionaVerticeActivo ();
133
134     protected abstract void eliminaVerticeActivo (VerticeEtiquetado u);
135
136     protected abstract void agregaVerticeActivo (VerticeEtiquetado u);
137
138     protected abstract boolean contieneVerticeActivo ();
139 }
```

4.1.1 Precondiciones y postcondiciones de los métodos de AGP

Constructor

(**public** AGP (Red r))
Valida el estado inicial del algoritmo, asignando un valor inicial a sus atributos.

Precondiciones: La clase debe ser invocada con una red.

Postcondiciones: Deja una representación de la red que cumple con lo siguiente:

1. Cada red tiene asociadas dos gráficas, la red original que es la representación inicial de la red, o sea, con flujo igual a cero, y la red residual que es una red auxiliar para ir incrementando el flujo. En un principio son equivalentes.
2. Cada vértice de la red tiene asociada su lista de adyacencias.
3. Cada vértice tiene asociada una distancia etiquetada válida.
4. Cada vértice tiene asociado un exceso de flujo, inicialmente cero.
5. Cada arco tiene asociada una capacidad entera no negativa.
6. Cada arco tiene asociado un flujo cero.

Incrementa Flujo

(**protected void** incrementaFlujo ())
Este método empuja flujo desde un vértice activo u a través de un arco admisible (u, v), siempre y cuando exista dicho arco, en caso contrario incrementa la distancia de u.

Precondiciones: Existe un vértice activo.

Postcondiciones: Si el vértice activo u contiene un arco admisible a se cumplen las postcondiciones del método empujaFlujo(u,a). Si no es así entonces $d(u)$ se habrá incrementado.

Define el Valor del Flujo Máximo

(protected int valorDelFlujoMaximo ())

Este método asigna el exceso del vértice t a la variable flujoMaximo, dado que éste es el valor del flujo máximo de la red r, y regresa dicho valor.

Precondiciones: No existe ningún vértice activo en la red residual r.redResidual.

Postcondiciones: El valor flujoMaximo es el valor del flujo máximo encontrado.

Asigna Distancias Etiquetadas

(protected void asignaDistanciasEtiquetadas ())

Este método ejecuta BFS sobre la red inversa a r; por lo tanto, asigna la distancia de la trayectoria más corta de cada vértice al vértice destino.

Precondiciones: Ninguna

Postcondiciones: Al finalizar la ejecución del método las distancias etiquetadas de cada vértice son válidas.

Preproceso

(protected abstract void inicializa ())

Este método realiza un empuje saturado de flujo desde el vértice s a los vértices v tales que existe un arco de s a v.

Precondiciones: Ninguna

Postcondiciones: Al salir del método, se cumple que:

- Para todos los vértices v para los cuales existe un arco de s a ellos, se cumple $e(v) > 0$.
- La distancia del vértice origen cumple: $d(s) = n$.

Empuja Flujo A Través de Arcos Admisibles

(protected void empujaFlujo (VerticeEtiquetado u))

Sea u un vértice activo para el cual existe un arco admisible (u, v); entonces el método empujaFlujo envía flujo a través de (u, v).

Precondiciones: Existe un arco admisible (u, v) y u está activo.

Postcondiciones: Supongamos que que el empuje es de δ unidades de flujo. Al finalizar la ejecución se cumple: el exceso del vértice u fue decrementado en δ unidades al igual que la capacidad residual del arco (u, v), mientras que el exceso del vértice v y la capacidad residual del arco (v, u) fueron incrementados en δ unidades de flujo. Además, si $\delta = e(u)$ el vértice u es quitado de la lista de vértice activos.

Reetiqueta un Vértice

(protected void reetiqueta (VerticeEtiquetado u))
Sea v un vértice activo, el cual no contiene un arco admisible; el método reetiqueta incrementa el valor de la distancia actual en al menos una unidad.

Precondiciones: El vértice activo u no contiene ningún arco admisible.

Postcondiciones: $d(u)$ se incrementa por lo menos en una unidad.

Contiene Vértice Activo

(protected boolean contieneVerticeActivo ())
El método verifica si la red contiene algún vértice con exceso mayor a cero, en caso afirmativo regresa **true** y devuelve **false** en caso contrario.

Precondiciones: Ninguna.

Postcondiciones: Si el método regresa true entonces existe un vértice u tal que $e(u) > 0$.

En caso contrario, el método regresa **false**.

Selecciona un Vértice Activo

(protected VerticeEtiquetado seleccionaVerticeActivo ())
El método regresa un vértice cuyo exceso sea mayor que cero.

Precondiciones: Existe un vértice activo en la red.

Postcondiciones: El método regresa un vértice v tal que $e(v) > 0$.

Agrega un Vértice Activo

(protected void agregaVerticeActivo (VerticeEtiquetado u))
El método agrega el vértice activo u al conjunto de vértices activos, siempre y cuando u no sea el vértice destino o el vértice origen.

Precondiciones: $e(u) > 0$, $u \neq t$ y $u \neq s$.

Postcondiciones: Si L es el conjunto de vértices activo, al terminar de ejecutarse la función $u \in L$.

Elimina un Vértice Activo

(protected void eliminaVerticeActivo (VerticeEtiquetado u))
El método elimina un vértice u del conjunto de vértices activo.

Precondiciones: El vértice u tiene exceso igual a cero.

Postcondiciones: Si L es el conjunto de vértices activo, al terminar de ejecutarse la función, $u \notin L$.

4.1.2 Correctez de AGP

Lema 4.1 *Una vez que el método constructor AGP fue ejecutado todos los vértices de la red tienen distancias etiquetadas válidas.*

Demostración:
La demostración de este lema es análoga a la del lema 3.20.

∎

Lema 4.2 *Una vez que el método inicializa() fue ejecutado se cumple:*

$$\forall u \in V \text{ tal que } (s,u) \in A, e(u) > 0.$$

Demostración:
El método obtiene la lista de adyacencias del vértice origen (línea 48) y para cada uno de los arcos (s,u) se obtiene la capacidad del arco (línea 55); recordemos que la capacidad de cualquier arco que está en la red siempre es mayor a cero. El método satura todos los arcos (s,u) (línea 56), lo cual provoca que el exceso del vértice u se incremente (línea 57). Por lo tanto, después de la ejecución del método inicializa() se cumple:

$$\forall u \in V \text{ tales que } (s,u) \in A, e(u) > 0.$$

∎

Lema 4.3 *Si para un vértice activo u existe un arco admisible (u,v), al terminar de ejecutarse el método empujaFlujo() con $\delta = \min\{e(u), c(u,v)\}$ se cumplen las siguientes condiciones:*

- *$e(u)$ es decrementado en δ unidades.*
- *$c(u,v)$ también se decrementa en δ unidades.*
- *$e(v)$ se incrementa en δ unidades.*
- *$c(v,u)$ también se incrementa en δ unidades*
- *Si el empuje es no saturado se elimina de la lista de vértices activos a u.*
- *El vértice v es agregado a la lista de vértices activos.*

Demostración:
Si u contiene un arco admisible el método incrementaFlujo llama al método empujaFlujo el cual verifica cuántas son las unidades de flujo que va a enviar, es decir, el valor de la variable delta; delta es inicializa en la línea 72 con el valor del exceso del vértice u; después se verifica si $c(u,v)$ <delta (línea 73); en caso de que se cumpla dicha condición se actualiza el valor de delta = c(u,v). Por lo tanto, $\delta = \min\{e(u), c_f(u,v)\}$. Una vez definidas las unidades de flujo que se van a empujar, se procede a realizar lo siguiente: el exceso del vértice u es decrementado en δ unidades (línea 83) al igual que la capacidad residual del arco (u,v) (línea 81), mientras que el exceso del vértice v y la capacidad residual del arco (v,u) son incrementados en δ unidades de flujo (líneas 82 y 86); si el arco (v,u) no está en la red entonces se crea con capacidad igual a δ (línea 87). Además se agrega el vértice v a la lista de vértices activos (línea 80), siempre y cuando $v \notin \{s,t\}$.

∎

ESTA TESIS NO SALE
DE LA BIBLIOTECA

Lema 4.4 *Si para un vértice activo u existe un arco admisible (u, v), al terminar de ejecutarse el método incrementaFlujo(u) ocurre lo siguiente: el exceso del vértice u es decrementado en δ unidades al igual que la capacidad residual del arco (u, v), mientras que el exceso del vértice v y la capacidad residual del arco (v, u) son incrementados en δ unidades de flujo. Donde* $\delta = \min\{e(u), c_f(u, v)\}$.

Demostración:

La demostración se sigue del lema anterior.

■

Lema 4.5 *Si para un vértice activo u no existe un arco admisible (u, v), al terminar de ejecutarse el método reetiqueta() la distancia de u se habrá incrementado en al menos una unidad.*

Demostración:

Recordemos que las distancias etiquetadas son válidas, por lo que para cualquier arco (u, v) se cumple $d(u) \leq d(v) + 1$. Además, por hipótesis tenemos que no existe ningún arco admisible, lo cual significa que para cualquier vértice v adyacente a u tenemos $d(u) \leq d(v)$. Sea d el valor de la mínima distancia de los vértices adyacentes a u, inicialmente el método asigna el valor de d igual al valor del primer vértice que aparece en la lista de adyacencias de u (línea 100). Después, revisa todas las distancias de los vértice adyacentes a u y si alguna es menor actualiza el valor de d (líneas 106-107). Por lo tanto, al terminar el ciclo **for** (línea 102) el valor de $d = \min\{d(v) | (u, v) \in E\}$. Por lo anterior, podemos concluir $d(u) < d + 1$, que es el valor con el que se actualiza $d(u)$ (línea 109). Por lo tanto, $d(u)$ se incrementa después de ejecutar el método reetiqueta().

■

Lema 4.6 *Al finalizar el algoritmo AGP el preflujo existente es un flujo.*

Demostración:

Para demostrar que el preflujo es un flujo únicamente es necesario revisar que se cumple la condición de conservación del flujo, lo cual es inmediato del hecho que el algoritmo termina cuando el método *contieneVerticeActivo()* regresa **false**, y esto ocurre únicamente si no hay ningún vértice en lista. De lo anterior y por el lema 4.12 podemos concluir que el preflujo obtenido, una vez ejecutado el algoritmo, es un flujo.

■

Lema 4.7 *En cualquier estado del AGP, para cualquier vértice activo v existe una $s \rightsquigarrow v$ trayectoria dirigida en la red residual.*

Demostración:

Si v está activo es porque empujamos flujo desde otro vértice que estaba activo, llamémosle v_1. Si $v_1 = s$, ya terminamos porque en red residual existe el arco (v, s), que sería una trayectoria dirigida de longitud igual a uno; si $v_1 \neq s$ entonces, dado que empujamos flujo desde v_1 hasta v, en algún momento v_1 estuvo activo y por lo tanto existe un vértice v_2 desde el cual enviamos flujo a v_1; si $v_2 = s$ ya terminamos, en caso contrario repetimos el procedimiento hasta llegar a s. Podemos asegurar que en algún momento llegamos a s porque es el vértice desde el

cual empezamos a empujar flujo, por lo tanto ningún vértice puede ser activo a menos que reciba flujo desde s. Entonces, para cualquier vértice activo existe una trayectoria dirigida $v, v_1, v_2, \ldots, v_k = s$, en la cual para cada arco (v_k, v_{k+1}) en algún momento del algoritmo el vértice v_{k+1} empujó flujo al vértice v_k.

■

Lema 4.8 *Para cualquier vértice $v \in V$ se cumple que $d(v) < 2n$.*

Demostración:

La última vez que el algoritmo define la distancia de un vértice v, éste tiene $e(v) > 0$. Por lo tanto, en la red residual tenemos una trayectoria p del vértice v al vértice origen s. Sea $p = v, v_1, v_2, \ldots, v_k = s$ dicha trayectoria, cuya longitud es estrictamente menor que $n-1$ $(k < n-1)$; además, para cada arco (v_k, v_{k+1}) se tiene que $d(v_k) \leq d(v_{k+1})$, tomando en cuenta esto último y que $d(s) = n$ podemos deducir lo siguiente:

$$\begin{aligned} d(v) &\leq d(v_1) + 1 \\ &\leq (d(v_2) + 1) + 1 \\ &\vdots \\ &\leq d(s) + |p| < 2n \end{aligned}$$

■

Cada vez que reetiquetamos un vértice la distancia de éste siempre se incrementa en al menos una unidad. El siguiente resultado nos muestra una cota máxima para el número de aumentos que podemos realizar a la distancia de un vértice, en otras palabras, el número máximo de operaciones de reetiquetamiento que se hacen en total.

Corolario 4.1 *El número máximo de operaciones de reetiquetamiento es $2n^2$.*

Demostración:

Por el lema anterior tenemos que cualquier distancia etiquetada se incrementa a lo más $2n$ unidades. En cada operación de reetiquetamiento la distancia del vértice se incrementa, en el peor de los casos únicamente en una unidad. De lo anterior y del hecho que el número de vértices es n podemos concluir que a lo más podemos realizar $2n^2$ reetiquetamientos.

■

Para demostrar que el Algoritmo Genérico de preflujo es correcto deberemos probar que:

1. Siempre termina.

2. Dada una red, el algoritmo siempre encuentra el flujo máximo que puede ser enviado desde s hasta t.

Suponiendo que cada uno de los métodos abstractos son correctos, es decir, que siempre terminan y cumplen con las postcondiciones dadas en la sección anterior, podemos demostrar lo siguiente:

- **Siempre Termina.** La condición para que el Algoritmo Genérico de Flujo Máximo termine es que no sea posible incrementar el flujo; en AGP esto es equivalente a decir que no exista un vértice activo. Podemos garantizar que en algún momento no hay ningún vértice activo en la red por lo siguiente: supongamos que en el preproceso enviamos desde s un total de δ unidades de flujo, dado que δ es un número finito y como ningún vértice es etiquetado más de $2n$ veces, entonces en algún momento toda la cantidad de flujo enviada desde s llega a t, o bien, las cantidades que no sea posible enviar hasta t son regresadas a s.

- **Encuentra un flujo máximo.** Para demostrar que el algoritmo encuentra un flujo con valor máximo enunciaremos el siguiente teorema:

  Teorema 4.1 *El algoritmo AGP encuentra el flujo máximo.*

  Demostración:

  El algoritmo termina cuando el preflujo actual es un flujo, lema 4.6. Además, como $d(s) = n$, la red residual no contiene ninguna trayectoria aumentante del vértice s al vértice t. Como esta condición es la que se utiliza para el AGFM, el exceso que está en el vértice t es el valor del flujo.

  ■

De lo anterior podemos concluir que AGP es correcto.

4.1.3 Complejidad de AGP

Para analizar la complejidad del algoritmo, analizaremos los siguientes resultados:

Lema 4.9 *El algoritmo realiza a lo más nm empujes saturados.*

Demostración:
Por el lema 3.29 podemos asegurar que si un vértice es reetiquetado a lo más $2n$ veces entonces el número total de empujes no saturados es $O(nm)$.

■

Ahora únicamente nos falta demostrar el número de empujes no saturados que realiza el algoritmo, siendo esto lo que define la complejidad del algoritmo; suponiendo que mantener la lista de vértices activos nos tome $O(1)$:

Lema 4.10 *El AGP realiza $O(n^2m)$ empujes no saturados.*

Demostración:
Para esta demostración vamos a introducir las siguientes definiciones. Sea

$$I = \{v | v \text{ es un vértice activo}\}$$

y denotamos por Δ la suma de las distancias de los vértices activos, es decir,

$$\Delta = \sum_{v \in I} d(v).$$

Observemos que el valor de Δ después de la ejecución del método inicializa siempre es menor a $2n^2$ unidades. Ahora bien, cuando realizamos una operación de empuje o reetiquetamiento ocurren cualquiera de los siguientes casos.

El primer caso ocurre cuando el vértice u que estamos examinando no tiene ningún arco admisible; entonces se procede a reetiquetar el vértice, es decir, se incrementa la distancia de éste. Como consecuencia inmediata el valor de Δ también se incrementa.

En el segundo caso el vértice u sí contiene un arco admisible (u, v) por medio del cual empujamos flujo. El valor de Δ se incrementa o decrementa dependiendo del tipo de empuje que se haga. Si el empuje es saturado el valor de Δ se incrementa, dado que el vértice v ahora es activo y u también sigue estando activo, pero como la distancia de cualquier vértice no es mayor a $2n$ entonces el valor de Δ se incrementa a lo más $2n$ unidades. Suponiendo que en cada arco se realiza un empuje saturado, el incremento total de Δ por empujes saturados es $2n^2m$. Analicemos qué pasa con el valor de Δ si el empuje de flujo a través del arco (u, v) es no saturado: en este caso v también se vuelve activo, pero como el exceso de u se disminuye a cero, u deja de estar activo y ya no forma parte del conjunto I. Entonces tenemos dos casos: Si v estaba en I antes de enviar flujo a través de (u, v) el valor de Δ se decrementa $d(u)$ unidades, pero si v no estaba activo el incremento es de una unidad porque $d(u) = d(v) + 1$. Podemos concluir que después de realizar un empuje no saturado el valor de Δ se decrementa por lo menos una unidad.

De lo anterior tenemos que Δ a lo más vale $4n^2 + 2n^2m$ unidades, resultado que obtenemos al sumar el número total de incrementos de Δ durante la ejecución del algoritmo, más el valor de Δ después de ejecutar el método de preproceso. Como al terminar la ejecución del algoritmo $\Delta = 0$, dado que el único vértice activo es el vértice t y $d(t) = 0$, y cada empuje no saturado decrementa el valor de Δ en por lo menos una unidad, concluimos que el número de empujes no saturados es $O(n^2m)$.

∎

Teorema 4.2 *La complejidad de AGP es de $O(n^2m)$.*

Demostración:

La complejidad del algoritmo se reduce al número de empujes no saturados y al costo total de mantener la lista de vértices activos. La primera parte ya demostramos que es $O(n^2m)$ y sabemos que podemos implementar una lista en donde sacar y meter un elemento sea $O(1)$. Por lo tanto, AGP tiene $O(n^2m)$.

∎

La siguiente tabla muestra la complejidad de cada uno de los métodos definidos en AGP.

Método	Complejidad	Justificación
asignaDistanciasEtiquetadas	$O(m)$	Es la misma complejidad de BFS
inicializa	Se definirá en cada versión concreta del algoritmo	Depende de los métodos abstractos agregaVerticeActivo
valorDelFlujo	$O(1)$	Únicamente revisa el exceso de t y se lo asigna a flujoMáximo
empujaFlujo	Se definirá en cada versión concreta del algoritmo	Depende de los métodos abstractos agregaVerticeActivo y eliminaVerticeActivo
reetiqueta	$O(n)$	Tiene que verificar todos los vértices adyacentes al vértice actual, y a lo más son n.
actualizaFlujoEnRedOriginal	$O(1)$	Únicamente actualiza el flujo en un arco de la red original

Sustituyendo las complejidades de los métodos definidos en AGTA en la ecuación general dada en el Algoritmo Genérico de Flujo Máximo obtenemos lo siguiente:

$$
\begin{aligned}
f_{AGP} &= f_{inicializa} + \sum_{i=1}^{q} (f_{esPosibleIncrementarElFlujo} + f_{incrementaFlujo}) \\
&\quad + f_{valorDelFlujoMaximo} \\
&= \sum_{i=1}^{m} f_{agregaVerticeActivo} + \sum_{i=1}^{q} \Big(f_{contieneVerticeActivo} \\
&\quad + (f_{selecionaVerticeActivo} + f_{empujaFlujo} + f_{reetiqueta}) \Big) + O(1) \\
&= \sum_{i=1}^{m} f_{agregaVerticeActivo} + \sum_{i=1}^{q} \Big(f_{contieneVerticeActivo} + (f_{selecionaVerticeActivo} \\
&\quad + (f_{agregaVerticeActivo} + f_{eliminaVerticeActivo}) + O(n)) \Big) + O(1)
\end{aligned}
$$

Nuevamente, la complejidad exacta del algoritmo depende de los métodos abstractos. Entonces, para cada especialización concreta de AGP la complejidad varía, podemos definir una función de complejidad para aquellos algoritmos que sean especializaciones de AGP.

$$f_{AGP} = \sum_{i=1}^{m} f_{agregaVerticeActivo} + \sum_{i=1}^{q} \Big(f_{contieneVerticeActivo} + (f_{selecionaVerticeActivo}$$
$$+ (f_{agregaVerticeActivo} + f_{eliminaVerticeActivo}) + O(n)) \Big)$$

Dada la función anterior, en cada uno de los algoritmos concretos basados en AGP únicamente hay que revisar la complejidad de los métodos abstractos agregaVerticeActivo, contieneVerticeActivo, selecionaVerticeActivo, empujaFlujo y reetiqueta para definir la complejidad de la especialización concreta.

4.2 Implementaciones concretas del Algoritmo Genérico de Preflujo

En esta sección presentamos tres implementaciones concretas del algoritmo AGP: el *Algoritmo de Preflujo FIFO (APFIFO)* [32], el *Algoritmo de Preflujo de Distancia Máxima (APDM)* [6] y el *Algoritmo de Preflujo de Escalabilidad en el Exceso (APEE)* [1]. La diferencia entre cada uno de ellos está concretamente en la manera cómo seleccionan el vértice activo que van a examinar. La idea básica de cada uno de ellos es la siguiente.

- El *Algoritmo de Preflujo FIFO* examina todos los vértices activos en el orden en que los va activando, es decir, el primer vértice que revisa es el primero que examina.
- El *Algoritmo de Preflujo de Distancia Máxima* siempre realiza un empuje de flujo a un vértice activo cuya distancia etiquetada sea la más grande.
- Y por último, el *Algoritmo de Preflujo de Escalamiento de Exceso* empuja flujo de un vértice con exceso muy alto a un vértice con exceso muy pequeño.

En las siguientes secciones revisaremos tanto la implementación como la complejidad de los tres algoritmos.

4.3 Algoritmo de Preflujo FIFO

El Algoritmo de Preflujo FIFO es un algoritmo concreto de AGP. Recordemos que este tipo de algoritmos eligen un vértice que esté activo para deshacerse de su exceso, enviándolo a los vértices vecinos a través de arcos admisibles. El método que sigue para elegir cuál vértice activo escoger es el siguiente: en una lista L va introduciendo los vértices cuyo exceso sea mayor a cero en el orden en que dichos vértices van estando activos. La lista L se mantiene como una cola, el primero que entra es el primero que sale (First-In First-Out), de ahí el nombre del algoritmo. Entonces, el algoritmo APFIFO examina los vértices activos en orden de una cola. Selecciona un vértice u de la lista L y realiza un empuje de este vértice, y agrega los nuevos vértices activos a L; el algoritmo examina al vértice u mientras que éste esté activo, o bien, mientras que no se

reetiquete. El algoritmo termina cuando la lista L no contiene ningún vértice, lo cual implica que no hay más vértices activos, salvo el vértice destino t.

En cada iteración del algoritmo AGP se elige a un vértice activo para hacer un empuje de flujo, el cual puede ser saturado o no saturado. En el primer caso, si el empuje es saturado implica que agotamos la capacidad del arco por el cual enviamos flujo; pero el exceso del vértice activo no necesariamente se redujo a cero, entonces este vértice permanece activo. En el algoritmo APFIFO una vez que un vértice activo es elegido éste se mantiene realizando empujes hasta que su exceso se reduce a cero, es decir, dejó de estar activo; o el vértice es reetiquetado, lo cual ocurre cuando éste no tiene ningún arco admisible. Consecuentemente, el algoritmo desarrolla varios empujes saturados seguido de un empuje no saturado o de un reetiquetamiento.

Definición 4.4 *Un* vértice es examinado *si después de una secuencia de empujes dejó de estar activo, o bien fue reetiquetado.*

La implementación del algoritmo APFIFO se presenta a continuación.

Listado 4.3.1 Algoritmo de Preflujo FIFO

```
 1 public class APFIFO extends AGP {
 2     protected LinkedList lista;
 3
 4     public APFIFO (Red r) {
 5         super(r);
 6         lista = new LinkedList();
 7     }
 8
 9     protected void agregaVerticeActivo (VerticeEtiquetado u){
10         if(lista.indexOf(u) == -1)
11             lista.addFirst(u);
12     }
13
14     protected void eliminaVerticeActivo (VerticeEtiquetado u){
15         lista.removeLast();
16     }
17
18     protected VerticeEtiquetado seleccionaVerticeActivo (){
19         VerticeEtiquetado v = (VerticeEtiquetado)lista.getLast();
20         return v;
21     }
22
23     protected boolean contieneVerticeActivo (){
24         int i = lista.size();
25         return (i > 0);
26     }
27 }
```

4.3.1 Precondiciones y postcondiciones de los métodos de APFIFO

Constructor

(public APFIFO (Red r))

Valida el estado inicial del algoritmo, asignando un valor inicial a sus atributos.

Precondiciones: La clase debe ser invocada con una red.

Postcondiciones: Son exactamente las misma que en AGP, sólo que además, se inicializa la lista que contiene a los vértices activo (lista = **null**)

Contiene Vértice Activo

(**protected boolean** contieneVerticeActivo ())
El método verifica si la red contiene algún vértice con exceso mayor a cero, en caso afirmativo regresará **true** y devuelve **false** en caso contrario.

Precondiciones: Ninguna.

Postcondiciones: Si el método regresa **true** entonces existe un vértice u tal que $e(u) > 0$. En caso contrario, el método regresa false.

Selecciona un Vértice Activo

(**protected** VerticeEtiquetado seleccionaVerticeActivo ())
El método regresa un vértice cuyo exceso sea mayor que cero.

Precondiciones: Existe un vértice activo en la red.

Postcondiciones: El método regresa un vértice v tal que $e(v) > 0$

Agrega un Vértice Activo

(**protected void** agregaVerticeActivo (VerticeEtiquetado u))
El método agrega el vértice activo u al conjunto de vértices activo, siempre y cuando u no sea el vértice destino.

Precondiciones: El vértice u tiene exceso mayor a cero. Y además, u es distinto de t.

Postcondiciones: Si *lista* es el conjunto de vértices activos, al terminar de ejecutarse la función, $u \in lista$.

Elimina un Vértice Activo

(**protected void** eliminaVerticeActivo (VerticeEtiquetado u))
El método elimina un vértice u del conjunto de vértices activo.

Precondiciones: $e(u) = 0$.

Postcondiciones: Si *lista* es el conjunto de vértices activo, al terminar de ejecutarse la función $u \notin lista$.

4.3.2 Correctez de APFIFO

Lema 4.11 *Una vez que el método constructor APFIFO fue ejecutado todos los vértices de la red tienen distancias etiquetadas válidas.*

Demostración:
La demostración de este lema es similar a la del lema 3.20.

■

Lema 4.12 *En lista están todos los vértices que están activos en cada iteración, respectivamente; asimismo no contiene a ningún otro vértice.*

Demostración:
Observemos que los únicos momentos en los que lista es modifica es cuando ejecutamos el método empujaFlujo(v,a) del algoritmo AGP, en el cual, agregamos un nuevo vértice activo (línea 80 de AGP). en este caso el vértice v que se añade a lista cumple $e(v) > 0$ ya que delta> 0. Un vértice es eliminado de lista cuando $e(v) = 0$ (línea 76 de AGP). Además de este método, se elimina y agrega el mismo vértice en el método reetiqueta(u) del algoritmo AGP en ese orden, por lo tanto, una operación anula a la otra. De lo anterior concluimos que en la lista de vértices activos están todos y únicamente los vértices que hasta ese momento están activos.

■

Lema 4.13 *El método contieneVerticeActivo únicamente regresa* **true** *si la red contiene un vértice activo.*

Demostración:
Este método únicamente verifica que lista tenga al menos un elemento. Por lo anterior y por el lema 4.12 podemos asegurar que el método únicamente regresa **true** si existe un vértice activo.

■

Lema 4.14 *El método seleccionaVerticeActivo regresa un vértice activo, respetando el orden en que los vértices fueron activados. El primer vértice que fue activado es el primero que regresa.*

Demostración:
El método regresa el primer vértice que se agregó a lista (línea 19). Esto lo podemos asegurar porque el método agregaVérticeActivo(u) agrega un vértice al inicio de lista (línea 11) y el método seleccionaVerticeActivo regresa el último elemento de la lista (línea 19).

■

Teorema 4.3 *El algoritmo APFIFO encuentra el flujo máximo.*

Demostración:
Dado que se cumplen las precondiciones de los métodos abstractos, podemos asegurar por el teorema 4.1 que el flujo actual es máximo.

■

4.3.3 Complejidad de APFIFO

Para poder demostrar la complejidad del algoritmo dividiremos la ejecución del algoritmo en *fases*. En la primera fase examinamos todos los vértices adyacentes al vértice origen. En la segunda fase examinamos los adyacentes a los vértices de la primera fase. De esta manera, en la $k-$ésima fase examinamos lo vértices adyacentes a los vértices de la fase $k-1$.

Como la complejidad del algoritmo depende del número de empujes no saturados que se realizan, podemos ver que el número total de empujes no saturados es la suma de los empujes no saturados que se hacen en cada fase. Por lo tanto, basta con acotar el número de empujes no saturados que se realizan en cada fase y saber cuál es el número máximo de fases que se realizan.

Lema 4.15 *En cada fase el número total de empujes no saturados es $O(n)$.*

Demostración:

En cada fase cualquier vértice activo que esté en la fase es examinado una única vez. Y una vez que se realiza un empuje no saturado el vértice examinado deja de estar activo, y es sacado de la lista. Por lo tanto, en esta fase no se realiza ningún otro empuje de flujo desde el vértice en cuestión. De lo anterior, podemos concluir que en una fase únicamente se realiza un empuje no saturado por cada vértice. Por lo tanto, el número de empujes no saturados en cada fase es a lo más n.

■

Lema 4.16 *APFIFO realiza a lo más n^2 fases.*

Demostración:

Consideremos la siguiente función $\Phi = \max\{d(u)|u$ es un vértice activo $\}$, la cual se actualiza en cada fase del algoritmo. Definimos el número total de cambios de la función Φ como la diferencia entre el valor inicial y el valor final de la misma. Consideremos los siguientes casos: el primer caso es cuando el algoritmo realiza al menos una operación de reetiquetamiento durante una fase; en tal caso el valor de Φ se incrementa tanto como el incremento máximo que pueda sufrir cualquier distancia, el cual es a lo más $2n$. Tomando en cuenta que en cada fase el incremento es máximo, el incremento total de Φ en todas las fases es $O(n^2)$. Ahora supongamos que el algoritmo no realiza ninguna operación de reetiquetamiento; en este caso el exceso de cualquier vértice activo es enviado a otro vértice con menor distancia, por lo que el valor de Φ se decrementa en al menos una unidad, dado que para todos los vértices de la fase actual el exceso se reduce a cero.

De lo anterior podemos concluir que el número total de fases es a lo más el valor inicial de Φ, el cual es a lo más n, más el incremento máximo que puede sufrir Φ durante todas las fases del algoritmo. Por lo tanto, el número de fases es a lo más $2n^2 + n$.

■

La siguiente tabla nos muestra la complejidad de los métodos definidos en APFIFO.

Método	Complejidad	Justificación
agregaVerticeActivo	$O(1)$	Únicamente debemos agregar un vértice a la cola.
eliminaVerticeActivo	$O(1)$	Únicamente sacamos un vértice de la cola.
selecionaVerticeActivo	$O(1)$	Elegimos el último vértice de la cola.
contieneVerticeActivo	$O(1)$	Verificamos si la cola contiene un elemento.

Sustituyendo las complejidades de los métodos de APFIFO en la ecuación de complejidad de AGP y asignando q al número de fases obtenemos la siguiente ecuación:

$$\begin{aligned} f_{APFIFO} &= \sum_{i=1}^{m} f_{agregaVerticeActivo} + \sum_{i=1}^{q} \Big(f_{contieneVerticeActivo} + (f_{selecionaVerticeActivo} \\ &\quad + (f_{agregaVerticeActivo} + f_{eliminaVerticeActivo})) + O(n) \Big) \\ &= \sum_{i=1}^{m} O(1) + \sum_{i=1}^{n^2} \Big(O(1) + (O(1) + (O(1) + O(1))) + O(n) \Big) \\ &= O(m) + O(n^3) \end{aligned}$$

El siguiente teorema nos asegura que la complejidad del algoritmo únicamente es $O(n^3)$.

Teorema 4.4 *El tiempo de ejecución de APFIFO es $O(n^3)$.*

Demostración:
Dado que la operación de la cual depende la complejidad del algoritmo es el número total de empujes no saturados, por los lemas 4.15 y 4.16 podemos garantizar que la complejidad de APFIFO es $O(n^3)$.

∎

4.4 Algoritmo de Preflujo por Distancia Máxima

El Algoritmo de Preflujo por Distancia Máxima, como su nombre lo indica, empuja el flujo desde un vértice activo cuya distancia es máxima a través de un arco admisible. Sea d^* la distancia máxima de los vértices activos. El algoritmo primero envía flujo desde los vértices con distancia igual a d^* a los vértices con distancia igual $d^* - 1$, y a su vez envía flujo desde estos vértices a los vértices con distancia $d^* - 2$. Este procedimiento se sigue mientras ningún vértice sea reetiquetado. Una vez que se realiza una operación de reetiquetamiento se calcula nuevamente el valor de la distancia máxima y se actualiza d^*. Repetimos el procedimiento hasta que no exista ningún vértice activo, excepto el vértice destino. Para seleccionar el vértice

activo con mayor distancia desde el cual empujaremos flujo, utilizamos la siguiente estructura: para cada $k = 1, 2, \ldots, 2n-1$ mantenemos una lista de vértices activos cuya distancia sea igual a k, es decir:

$$L(k) = \{u \mid u \text{ es un vértice activo y } d(u) = k\}$$

Cada una de estas listas será definida como una pila o una cola[1]. Además, mantenemos una referencia, llamada nivel, a la lista con el máximo valor de k para el cual la lista $L(k)$ no sea vacía. Entonces, la operación de elegir un vértice con distancia máxima se reduce obtener un vértice de la lista a la cual se refiere la variable nivel. Si en algún momento se reetiqueta el valor del vértice con distancia máxima, entonces se actualiza el valor de nivel con la nueva distancia. Observemos que el número total de incrementos de la variable nivel es a lo más $2n^2$ (por el lema 4.1), entonces el número total de decrementos es a lo más $2n^2 + n$.

La implementación de APDM se presenta a continuación.

Listado 4.4.1 Algoritmo de Preflujo de Distancia Máxima

```
 1 public class APDM extends AGP {
 2     private LinkedList   listasL;
 3     private int          nivel;
 4
 5     public APDM (Red r) {
 6         super(r);
 7         listasL = new LinkedList();
 8         nivel=0;
 9         creaListasDeDistancias();
10     }
11
12     protected VerticeEtiquetado seleccionaVerticeActivo (){
13         LinkedList        l = (LinkedList)listasL.get(nivel);
14         VerticeEtiquetado v = (VerticeEtiquetado)l.getLast();
15         return v;
16     }
17
18     protected boolean contieneVerticeActivo (){
19         if(nivel>0)
20             return true;
21         return false;
22     }
23
24     private void creaListasDeDistancias (){
25         LinkedList  vertices = r.redResidual().obtenVertices();
```

[1]No importa cuál de las dos, siempre y cuando se garantice que tanto meter como sacar a un elemento de la lista sea O(1).

Listado 4.4.1 Algoritmo de Preflujo de Distancia Máxima (continuación)

```
 1         listasL = new LinkedList();
 2         int i;
 3         for(i = 0; i <= (2*r.redInicial().numeroDeVertices()); i++){
 4             LinkedList  l = new LinkedList();
 5             listasL.addLast(l);
 6         }
 7     }
 8
 9     protected void agregaVerticeActivo (VerticeEtiquetado u){
10         int        d = u.obtenDistancia();
11         LinkedList  l = (LinkedList)listasL.get(d);
12         if(l.indexOf(u) == -1){
13             l.addLast(u);
14             if(d > nivel)
15                 nivel = d;
16         }
17     }
18
19     protected void eliminaVerticeActivo (VerticeEtiquetado u){
20         int        d = u.obtenDistancia();
21         LinkedList  l = (LinkedList)listasL.get(d);
22         l.remove(u);
23
24         if(l.size() == 0){
25             int i;
26             boolean band = true;
27             for(i = nivel; i > 0 && band ; i--){
28                 if(((LinkedList)listasL.get(i)).size() > 0){
29                     nivel = i;
30                     band = false;
31                 }
32             }
33             if(band == true)
34                 nivel = 0;
35         }
36     }
37 }
```

4.4.1 Precondiciones y postcondiciones de los métodos de APDM

Constructor

(public APDM (Red r))

Valida el estado inicial del algoritmo, asignando un valor inicial a sus atributos.

Precondiciones: Las mismas que para la superclase.

Postcondiciones: Son las mismas del constructor de AGP, incluyendo que la estructura listasL también estará inicializada, es decir, no contiene ningún vértice activo.

Contiene Vértice Activo

(protected boolean contieneVerticeActivo ())
El método verifica si la red contiene algún vértice con exceso mayor a cero, en caso afirmativo regresará **true** y devuelve **false** en caso contrario.

Precondiciones: Ninguna.

Postcondiciones: Si el método regresa **true** entonces existe un vértice u tal que $e(u) > 0$.
En caso contrario, el método regresa **false**.

Selecciona un Vértice Activo

(protected VerticeEtiquetado seleccionaVerticeActivo ())
El método regresa un vértice cuyo exceso sea mayor que cero.

Precondiciones: Existe un vértice activo en la red.

Postcondiciones: Si el método regresa un vértice v, entonces $e(v) > 0$ y

$$d(v) = \max\{d(u)|u \text{ es activo}\}.$$

Agrega un Vértice Activo

(protected void agregaVerticeActivo (VerticeEtiquetado u))
El método agrega el vértice activo u al conjunto de vértices activos, siempre y cuando u no sea el vértice destino.

Precondiciones: $e(u) > 0$ y $u \neq s$.

Postcondiciones: Si L es el conjunto de vértices activos, al terminar de ejecutarse el método $u \in L$.

Elimina un Vértice Activo

(protected void eliminaVerticeActivo (VerticeEtiquetado u))
El método elimina un vértice u del conjunto de vértices activos.

Precondiciones: $e(u) > 0$.

Postcondiciones: Si L es el conjunto de vértices activo, al terminar de ejecutarse el método, $u \notin L$.

Crea Estructura de Vértices Activos basándose en su Distancia

(protected void creaListaDeDistancias ())

Crea una estructura donde va a almacenar los vértices activos. Dependiendo de la distancia del vértice, es la lista que le corresponde.

Precondiciones: Ninguna.

Postcondiciones: Al terminar de ejecutarse el método se habrá creado la estructura listasL, que es una lista de listas $L(i)$, como se definieron anteriormente.

4.4.2 Correctez de APDM

Lema 4.17 *Una vez que el método constructor APDM fue ejecutado todos los vértices de la red tienen distancias etiquetadas válidas.*

Demostración:

La demostración de este lema es análoga a la del lema 3.20.

∎

Lema 4.18 *En listaL están únicamente los vértices que están activos en cada fase.*

Demostración:

La demostración de este lema es similar a la demostración del lema 4.12, con la sutil diferencia de que la estructura en donde almacenamos los vértices activos no es lista sino listaL. Por lo tanto, cuando agregamos un vértice cuya distancia es d, éste es introducido a la lista $L(d)$ (línea 28).

∎

Lema 4.19 *La variable nivel guarda la posición de la primera lista listaL que contiene un vértice activo u tal que*

$$d(u) = \max\{d(v) | v \textit{ es activo}\}.$$

Demostración:

En los únicos método donde se modifica la variable nivel son los siguientes:

1. En el constructor, en donde se inicializa con valor cero.

2. En el método agregaVerticeActivo(u) en el cual si agregamos un vértice u tal que,

$$d(u) > nivel$$

   entonces actualizamos el valor de la variable nivel= $d(u)$. Por lo tanto, en este caso nivel almacena la posición de la lista de listaL que contiene un vértice activo con distáncia máxima.

3. Por último también se modifica la variable nivel en eliminaVerticeActivo(u), en donde se verifica si u tiene distancias máxima y en tal caso se revisa si era el único vértice en la lista a la cual se refería nivel; si la respuesta a esta pregunta es afirmativa entonces se busca cuál es la lista en listaL que contiene un vértice activo y con distancia máxima y se actualiza el valor de nivel con la nueva posición.

■

Lema 4.20 *El método contieneVerticeActivo únicamente regresa* **true** *si la red contiene un vértice activo.*

Demostración:
Recordemos que tenemos una variable llamada nivel la cual guarda la posición de la primera lista en la estructura listaL que no es vacía y que contiene a los vértices activos con distancia máxima. El método contieneVerticeActivo verifica si nivel > 0, dado que el único vértice u cuya $d(u) = 0$ es t, y las distancias nunca se decrementan, si se cumple la condición anterior, podemos asegurar que existe un vértice activo.

■

Lema 4.21 *El método seleccionaVerticeActivo siempre regresa un vértice u con $e(u) > 0$ y*

$$d(u) = \max\{d(v)|v \textit{ es activo}\}.$$

Demostración:
Los argumentos son los mismos que en la demostración anterior: nivel se refiere a la primera lista no vacía que contiene los vértices con distancia máxima. Y por el lema 4.19 el método seleccionaVerticeActivo únicamente regresa un vértice que pertenezca a la lista que se refiere nivel.

■

Teorema 4.5 *El algoritmo APDM encuentra un flujo máximo.*

Demostración:
Dado que se cumplen las precondiciones de los métodos abstractos, podemos asegurar por el teorema 4.1 que el flujo actual es máximo.

■

4.4.3 Complejidad de APDM

Antes de demostrar la complejidad del algoritmo definimos un *árbol* como una gráfica conexa y acíclica con un vértice distinguido llamado *raíz.* En un árbol cualquier vértice, excepto la raíz, tiene asignado un único vértice padre, es decir, es un *árbol enraizado.*

Sea F el conjunto de todos los arcos admisibles actuales, es decir, lo arcos por medio de los cuales empujamos flujo en un estado del algoritmo. Observemos que en cualquier estado un vértice tiene a lo más un arco admisible actual por donde empujar flujo. De lo anterior podemos asegurar que $|F| < n - 1$ porque cualquier vértice es el primer componente de a lo

más un arco y además podemos asegurar que no hay ciclos. Estas dos últimas condiciones nos llevan a definir a F como un *bosque* con los arcos invertidos, es decir, un conjunto de árboles en el cual la raíz corresponde al vértice que no es la primera componente de ningún arco en F; dicho de otro modo, del vértice raíz no sale ningún arco.

Para cada vértice u mantenemos una lista $D(u)$ con los vértices descendientes de u en el conjunto. Y decimos que un *vértice activo es máximal* si el único descendiente es él mismo.

Denotamos por M al conjunto de vértices activos maximales. Introducimos un nuevo parámetro K cuyo valor óptimo será definido más adelante. Además definimos la función Φ de la siguiente forma:

$$\Phi = \sum_{v \in M} \Phi(u)$$

donde $\Phi(u) = \max\{0, K + 1 - |D(u)|\}$. Notemos $\Phi(u) < K$ dado que para cualquier vértice se cumple que él mismo es un descendiente.

La razón por la cual introducimos la función Φ es porque si analizamos la forma en la cual se afecta el valor de la función cuando realizamos una operación de reetiquetamiento o empuje de flujo, podemos obtener una cota máxima del número de empujes no saturados. Los resultados presentados a continuación nos muestran las consecuencias de estas observaciones.

Lema 4.22 *Un empuje no saturado de un vértice activo maximal no incrementa el valor de Φ. Al contrario, si $|D(u)| \leq K$ el valor de Φ se decrementa por lo menos una unidad.*

Demostración:

Sea u un vértice activo maximal y (u, v) el arco por el cual empujamos flujo; una vez realizado un empuje no saturado a través de este arco el exceso de flujo del vértice u es enviado al vértice v, de tal forma que ahora el vértice v se convierte en un vértice activo máximo mientras que u deja de serlo. Como $|D(u)| < |D(v)|$ se cumple lo siguiente: $\Phi(u) + \Phi(v)$ se decrementa si $|D(u)| \leq K$; en cualquier otro caso permanece igual.

■

Lema 4.23 *Un empuje saturado de un vértice activo maximal incrementa el valor de Φ a lo más K unidades.*

Demostración:

Si realizamos un empuje de flujo desde el vértice maximal u hasta un vértice v, a través del arco (u, v), ocurre lo siguiente: el arco deja de pertenecer a F dado que su capacidad se reduce a cero, mientras que el vértice v se convierte en un vértice activo maximal al mismo tiempo que u también lo sigue siendo. Por lo tanto, v se agrega a M lo cual provoca que Φ se incremente en $\Phi(v)$ unidades; recordemos que este incremento no puede ser mayor que K.

■

Lema 4.24 *Cuando un vértice activo maximal es reetiquetado el valor de Φ se incrementa a lo más K unidades.*

Demostración:
Un vértice u es reetiquetado únicamente cuando no contiene ningún arco admisible por el cual pueda empujar flujo; lo anterior implica que u es la raíz de un árbol en el bosque actual. Además ninguno de sus descendientes es un vèrtice activo. Después de que el vértice u es reetiquetado cualquier arco donde u sea el segundo componente deja de pertenecer al árbol actual. Por lo tanto, $|D(u)| = 1$ dado que el único descendiente de u es él mismo. Esto último implica que el valor de $\Phi(u)$ es a lo más K. Podemos concluir que el valor de Φ se incrementa a lo más K unidades después de realizar un empuje saturado.

■

Lema 4.25 *Introducir un arco al árbol actual no incrementa el valor de* Φ.

Demostración:
Si introducimos el arco (u, v) al bosque actual no añadimos ningún vértice activo maximal. Más bien, incrementamos el número de descendientes de algunos vértices, lo cual puede provocar que algunos vértices activos que eran maximales, dejen de serlo. Por lo tanto, el valor de Φ no se incrementa.

■

La siguiente tabla muestra la complejidad de los algoritmos definidos en APDM.

Método	Complejidad	Justificación
seleccionaVerticeActivo	$O(1)$	Únicamente regresa un vértice de la lista $L(nivel)$.
contieneVerticeActivo	$O(1)$	Solamente verifica que $nivel > 0$
creaListasDeDistancias	$O(n)$	Crea a lo más $2n + 1$ listas.
agregaVerticeActivo	$O(1)$	Sólamente agrega un vértice u a la lista $L(d(u))$.
eliminaVerticeActivo	$O(n)$	Tiene que actualizar el valor de la variable $nivel$ para lo cual a lo más hace $2n + 1$ comparaciones.

Sustituyendo las complejidades de los métodos de APDM en la ecuación de complejidad de AGP obtenemos la siguiente ecuación:

$$
\begin{aligned}
f_{APDM} &= \sum_{i=1}^{m} f_{agregaVerticeActivo} + \sum_{i=1}^{q} \Big(f_{contieneVerticeActivo} + (f_{selecionaVerticeActivo} \\
&\quad + (f_{agregaVerticeActivo} + f_{eliminaVerticeActivo})) + O(n) \Big) \\
&= \sum_{i=1}^{m} O(1) + \sum_{i=1}^{q} \Big(O(1) + (O(1) + (O(1) + O(n))) + O(n) \Big) \\
&= \sum_{i=1}^{m} O(1) + \sum_{i=1}^{q} O(n)
\end{aligned}
$$

En la ecuación anterior aún falta determinar el valor de q, por lo que, el siguiente resultado nos demuestra la complejidad del algoritmo.

Teorema 4.6 *El tiempo de ejecución de APDM es $O(n^2m^{\frac{1}{2}})$.*

Demostración:
Para analizar la complejidad del algoritmo definamos lo que es una *fase*. Sea

$$d_{max} = \max\{d(u) | \text{ u es un vértice activo}\}.$$

Definimos un *fase* como una secuencia de empujes en la cual d_{max} no cambia. El número de fases es $O(n^2)$, y la demostración de esto es similar a la del lema 4.16.

Llamamos a una fase *económica* si el número de empujes no saturados en la fase es menor a $\frac{2n}{K}$, en cualquier otro caso decimos que la fase es *costosa*.

El número de empujes no saturados realizados en todas las fases económicas es a lo más $O(n^2\frac{2n}{K})$. Para encontrar el número de empujes no saturados realizados en las fases costosas, observamos lo siguiente: una fase costosa realiza más de $\frac{2n}{K}$ empujes no saturados. Como la red contiene a lo más $\frac{n}{K}$ vértices con K descendientes o más, entonces al menos $\frac{n}{K}$ empujes no saturados deberían ser de un vértice con menos de K descendientes.

El algoritmo APDM realiza una operación de empuje saturado, no saturado o un reetiquetamiento en un vértice activo maximal. De los lemas anteriores podemos concluir lo siguiente: un empuje no saturado reduce el valor de Φ, mientras que las operaciones de reetiquetamiento y empuje saturado incrementan su valor, y este incremento es a lo más $O(nmK)$. Por lo tanto, el número de empujes no saturado es a lo más $O(nmK)$. Ahora sí llegó la hora de encontrar un valor óptimo para K: balanceemos los empujes saturados con los no saturados. Entonces tenemos que

$$\frac{n^3}{K} = nmK$$

de lo cual podemos concluir que $K = \frac{n}{m^{\frac{1}{2}}}$.

Por lo tanto, la complejidad del algoritmo es $O(n^2m^{\frac{1}{2}})$.

■

4.5 Algoritmo de Preflujo por Escalabilidad de Exceso

El Algoritmo de Preflujo por Escalabilidad de Exceso APEE, como su nombre lo indica, está basado en la idea de empujar todo el flujo posible desde el vértice origen s a los vértices adyacentes, y después empujar el flujo desde dichos vértice a los vértices adyacentes a ellos, y así sucesivamente. El algoritmo selecciona el vértice que contenga el mayor exceso y envía flujo a través de los arcos adyacentes a éste.

Como ya es costumbre, introducimos la función $e_{\max}$, la cual se define de la siguiente manera:

$$e_{\max} = max\{e(u)|u \text{ es un vértice activo}\}$$

Definida la función ahora sólo nos resta verificar de qué manera será de utilidad. Observemos que durante la ejecución del Algoritmo Genérico de Preflujo no es posible ver ningún patrón

particular en el valor de e_{max}, excepto que su valor se reduce a cero al finalizar el algoritmo. En el algoritmo APEE intentamos reducir el valor de la función de una forma más sistemática.

Luego entonces, el algoritmo APEE asegura que cada empuje no saturado envía una cantidad de flujo lo suficientemente grande, de tal forma que el número de empujes no saturados se reduzca a la menor cantidad posible.

Denotamos con Δ a la cota máxima que puede alcanzar la función e_{max}. Decimos que un vértice tiene *exceso muy alto* si $e(u) \geq \frac{\Delta}{2}$, y es un *vértice con exceso pequeño* en cualquier otro caso.

De lo anterior podemos decir que el algoritmo APEE empuja flujo desde un vértice con exceso muy alto intentando que el flujo llegue al vértice destino. El algoritmo nunca incrementa el valor de Δ, evitando realizar empujes innecesarios, los cuales suceden cuando varios vértices envían flujo a un mismo vértice; si éste último no puede enviar este flujo al vértice destino tendrá que regresarlo a los vértices que se lo enviaron.

El algoritmo selecciona de entre todos los vértices con exceso muy alto a aquel cuya distancia sea menor que la de los demás.

El algoritmo APEE procede de la misma manera que el algoritmo AGP con la siguiente diferencia: $\delta = \min\{e(u), c_f(u,v), \Delta - e(v)\}$. Recordemos que δ son las unidades de flujo que vamos a empujar del vértice u al vértice v, en el algoritmo genérico el valor de δ se define como sigue: $\delta = \min\{e(u), c_f(u,v)\}$. Con esto podemos asegurar que el exceso de un vértice nunca es mayor que Δ.

Definimos una $\Delta-$*fase de escalamiento* como una secuencia de empujes realizados mientras que el valor de Δ permanece constante. El valor inicial de $\Delta = 2^{\log U}$ donde U es el valor de la capacidad máxima de todos los arcos en la red. Notemos que $U \leq \Delta \leq 2U$. Durante una $\Delta-$fase de escalamiento se cumple que $\frac{\Delta}{2} \leq e_{max} \leq \Delta$; el valor de e_{max} puede incrementarse y/o decrementarse durante la fase, pero una vez que dicho valor es menor que $\frac{\Delta}{2}$ empezamos una nueva fase donde el valor de Δ se reduce a la mitad. Una vez que el algoritmo realiza $\log U + 1$ fases, el valor de e_{max} se habrá decrementado a cero y habremos obtenido un flujo máximo.

Para seleccionar el vértice activo con mayor exceso y distancia mínima desde el cual empujaremos flujo, utilizamos la siguiente estructura: para cada $k = 1, 2, \ldots, 2n - 1$ mantenemos una lista de vértices activos cuya distancia sea igual a k, es decir:

$$L(k) = \{u \mid e(u) \geq \frac{\Delta}{2} \text{ y } d(u) = k\}$$

Cada una de estas listas estará definida como una pila o una cola. A diferencia de la referencia nivel de la estructura correspondiente al algoritmo APDM, la cual se refería a la lista con el máximo valor de k para el cual la lista $L(k)$ no sea vacía; en este algoritmo nivel se refiere a la lista con el mínimo valor de k, tal que $L(k)$ no sea vacía.

A continuación presentamos la implementación del algoritmo APEE.

Listado 4.5.1 Algoritmo de Preflujo por Escalamiento de Exceso

```
1 public class APEE extends AGP {
2     private int          Delta;
3     private boolean      bandera;
4     private LinkedList   listasL;
5     private int          nivel;
6
7     public APEE (Red r) {
8         super(r);
9         listasL = new LinkedList();
10        nivel=2*r.redResidual().numeroDeVertices() + 1;
11        creaListasDeDistancias();
12        bandera = true;
13        int cMaxima = defineCapacidadMaxima();
14        Delta=(int)(Math.pow(2,(Math.ceil(Math.log(cMaxima)/Math.log(2)))));
15    }
16
17    public int algoritmoDeEscalamientoDeExceso () {
18         inicializa();
19         while (Delta >= 1) {
20             flujoMaximo = 0;
21             while(esPosibleIncrementarElFlujo ()){
22                 incrementaFlujo();
23             }
24             Delta = Delta/2;
25             actualizaListaDeVerticesActivosConExcesoGrande();
26         }
27         return valorDelFlujoMaximo ();
28    }
29
30    protected VerticeEtiquetado seleccionaVerticeActivo  (){
31        LinkedList         l = (LinkedList)listasL.get(nivel);
32        VerticeEtiquetado v = (VerticeEtiquetado)l.getLast();
33        return v;
34    }
35
36    protected boolean contieneVerticeActivo (){
37        if( nivel <= 2*r.redResidual().numeroDeVertices() && nivel>0 )
38            return true;
39        return false;
40    }
41
42    private void creaListasDeDistancias (){
43        LinkedList  vertices = r.redResidual().obtenVertices();
44
45        listasL = new LinkedList();
46        int i;
47        for(i = 1; i <= (2*r.redInicial().numeroDeVertices()); i++){
48            LinkedList  l = new LinkedList();
49            listasL.addLast(l);
50        }
51    }
```

Listado 4.5.1 Algoritmo de Preflujo por Escalamiento de Exceso (continuación)

```
52      protected void agregaVerticeActivo (VerticeEtiquetado u){
53         if(u.obtenExceso()>=(Delta/2) && u!=r.verticeDestino() && Delta/2>0){
54               int        d = u.obtenDistancia();
55               LinkedList l = (LinkedList)listasL.get(d);
56               if(l.indexOf(u) == -1){
57                   l.addLast(u);
58                   if(d < nivel)
59                       nivel = d;
60               }
61         }
62      }
63
64      protected void eliminaVerticeActivo (VerticeEtiquetado u){
65           int         d = u.obtenDistancia();
66           LinkedList l = (LinkedList)listasL.get(d);
67           l.remove(u);
68           if(l.size() == 0){
69               int i;
70               boolean band = true;
71               for(i = nivel; i < listasL.size() && band ; i++){
72                   if(((LinkedList)listasL.get(i)).size() > 0){
73                       nivel = i;
74                       band = false;
75                   }
76               }
77               if(band == true)
78                   nivel = 2*r.redResidual().numeroDeVertices() + 1;;
79           }
80      }
81
82      protected  void empujaFlujo (VerticeEtiquetado u, Arista a){
83           VerticeEtiquetado v = (VerticeEtiquetado)a.obtenVertice();
84           int     delta   =  u.obtenExceso();
85           if(delta > a.obtenCapacidad())
86               delta = a.obtenCapacidad();
87           if(v != (VerticeEtiquetado)r.verticeDestino() &&
88              delta > (Delta - v.obtenExceso()) )
89               delta = Delta - v.obtenExceso();
90
91           if((u.obtenExceso() - delta) < (Delta/2))
92               eliminaVerticeActivo(u);
93
94           if(v != (VerticeEtiquetado)r.verticeDestino()  &&
95              v != (VerticeEtiquetado)r.verticeOrigen() )
96               agregaVerticeActivo(v);
97           u.decrementaCapacidad(v,delta);
98           v.aumentaExceso(delta);
99           u.disminuyeExceso(delta);
100
101          if(v.estaConectadoCon(u))
102              v.incrementaCapacidad(u,delta);
103          else
104              v.conectaCon(u,delta);
```

Listado 4.5.1 Algoritmo de Preflujo por Escalamiento de Exceso (continuación)

```
105             actualizaFlujoEnRedOriginal(u,v,delta);
106         }
107
108         private void actualizaListaDeVerticesActivosConExcesoGrande(){
109             LinkedList  vertices = r.redResidual().obtenVertices();
110
111             int i;
112             for(i = 0; i < vertices.size(); i++)
113                 agregaVerticeActivo((VerticeEtiquetado) vertices.get(i));
114
115         }
116
117         private int  defineCapacidadMaxima () {
118             int i, tmp = 0;
119
120             for(i = 0; i < r.redResidual().numeroDeVertices() ; i++){
121                 Vertice     u   = (Vertice)r.redResidual().obten(i);
122                 LinkedList ady = u.obtenAdyacencias();
123
124                 int j;
125                 for(j = 0; j < ady.size(); j++ ){
126                     if( ((Arista)ady.get(j)).obtenCapacidad() > tmp)
127                         tmp = ((Arista)ady.get(j)).obtenCapacidad();
128                 }
129             }
130             return tmp;
131         }
132 }
```

4.5.1 Precondiciones y postcondiciones de los métodos de APEE

Constructor

(public APEE (Red r))

Valida el estado inicial del algoritmo, asignando un valor inicial a sus atributos.

Precondiciones: La clase debe ser invocada con una red. Mismas que para la superclase.

Postcondiciones: La red estará inicializada cumpliendo con las mismas condiciones de AGP. Se inicializa el valor de $\Delta = 2^{\log U}$.

Contiene Vértice Activo

(protected boolean contieneVerticeActivo ())

El método verifica si la red contiene algún vértice con exceso mayor a cero, en caso afirmativo regresará **true** y devuelve **false** en caso contrario.

Precondiciones: Ninguna.

Postcondiciones: Si el método regresa **true** entonces existe un vértice u tal que $e(u) \geq \frac{\Delta}{2}$. En caso contrario, el método regresa false.

Selecciona un Vértice Activo

(**protected** VerticeEtiquetado seleccionaVerticeActivo ())
El método regresa un vértice cuyo exceso sea mayor que cero.

Precondiciones: Existe $u \in V$ tal que $e(u) > 0$.

Postcondiciones: El método regresa un vértice v tal que $e(v) \geq \frac{\Delta}{2}$ además

$$d(u) = \min\{d(v)|e(v) \geq \frac{\Delta}{2}.$$

Agrega un Vértice Activo

(**protected void** agregaVerticeActivo (VerticeEtiquetado u))
El método agrega el vértice activo u al conjunto de vértices activo, siempre y cuando u no sea el vértice destino y $e(u) \geq \frac{\Delta}{2}$.

Precondiciones: $e(u) \geq \frac{\Delta}{2}$, y $u \notin \{s, t\}$.

Postcondiciones: Si L es el conjunto de vértices activo, al terminar de ejecutarse la función, $u \in L$.

Elimina un Vértice Activo

(**protected void** eliminaVerticeActivo (VerticeEtiquetado u))
El método elimina un vértice u del conjunto de vértices activo si $e(u) \leq \frac{\Delta}{2}$.

Precondiciones: $e(u) \leq \frac{\Delta}{2}$.

Postcondiciones: Si L es el conjunto de vértices activo, al terminar de ejecutarse la función, $u \notin L$.

Crea Estructura de Vértices Activos basándose en su Distancia

(**protected void** creaListaDeDistancias ())
Crea una estructura donde va a almacenar los vértices activos. Dependiendo de la distancia del vértice, es la lista que le corresponde.

Precondiciones: Ninguna.

Postcondiciones: Al terminar de ejecutarse el método se habrá creado la estructura listasL, que es una lista de listas $L(i)$, definidas anteriormente en esta sección.

Actualiza Listas de Vértices Activos con gran Exceso

(**protected void** actualizaListaDeVerticesConExcesoGrande ())

Actualiza la lista de vértices activo con exceso al menos $\frac{\Delta}{2}$, después de que se decrementó el valor de Δ

Precondiciones: El valor de Δ se redujo a la mitad.

Postcondiciones: Al terminar de ejecutarse el método, listasL contiene únicamente a los vértices activos con exceso mayor o igual que $\frac{\Delta}{2}$.

4.5.2 Correctez de APEE

Lema 4.26 *Una vez que el método constructor APEE fue ejecutado, todos los vértices de la red tienen distancias etiquetadas válidas.*

Demostración:

La demostración de este lema es análoga a la del lema 3.20.

■

Lema 4.27 *En listaL están todos y únicamente cada uno de los vértices u tal que $e(u) \geq \frac{\Delta}{2}$.*

Demostración:

Nuevamente, la demostración se realiza de manera similar que la demostración del lema 4.12, sólo que en este método únicamente agregamos vértices activos u cuyo $e(u) \geq \frac{\Delta}{2}$. Y una vez que $e(u) < \frac{\Delta}{2}$ el vértice u es eliminado.

■

Lema 4.28 *La variable nivel guarda la posición de la primera lista listaL que contiene un vértice u tal que $e(u) \geq \frac{\Delta}{2}$ y $d(u)$ sea mínima.*

Demostración:

Por el lema anterior aseguramos que cualquier vértice que esté en listaL cumple con $e(u) \geq \frac{\Delta}{2}$. Entonces sólo nos falta verificar que nivel guarde la posición de la lista que contiene por lo menos un vértice activo con distancia mínima. Los únicos método donde se modifica la variable nivel son los siguientes:

1. En el constructor, en donde es inicializa con valor $2n + 2$, que es la cota máxima del incremento de la distancia de cualquier vértice.

2. En el método agregaVerticeActivo(u) en el cual si agregamos un vértice u tal que

$$d(u) < nivel$$

actualizamos el valor de la variable nivel= $d(u)$. Por lo tanto, en este caso nivel almacena la posición de la lista de listaL que contiene un vértice activo con distancia etiquetada mínima.

3. El último método donde se modifica la variable es en eliminaVerticeActivo(u) en el cual se verifica si u tiene distancia mínima y se revisa si hay otro vértice en la lista a la cual se refería nivel; si la respuesta a esta pregunta es negativa entonces se revisa cuál es la lista en listaL que contiene un vértice activo con distancia menor se actualiza el valor de nivel con la nueva posición.

■

Lema 4.29 *El método contieneVerticeActivo únicamente regresa* **true** *si la red contiene un vértice activo con exceso al menos $\frac{\Delta}{2}$.*

Demostración:

Recordemos que tenemos una variable llamada nivel, la cual almacena la posición de la primera lista en la estructura listaL que no es vacía y que contiene a los vértices activos cuya distancia es mínima. El método contieneVerticeActivo verifica si nivel > 0 y nivel $\leq 2n+1$, en tal caso podemos asegurar que existe al menos un vértice activo.

■

Lema 4.30 *El método seleccionaVerticeActivo regresa un vértice activo u con $e(u) \geq \frac{\Delta}{2}$ y*

$$d(u) = \min\{d(v)|e(v) \geq \frac{\Delta}{2}\}.$$

Demostración:

El lema 4.29 nos asegura que en listaL únicamente están los vértices u tales que $e(u) \geq \frac{\Delta}{2}$. Y por el lema anterior tenemos que existe un vértice activo u tal que $e(u) \geq \frac{\Delta}{2}$. Además nivel almacena la primera lista con al menos un vértice activo de distancia mínima. Por lo anterior y verificando que el algoritmo regresa un elemento de la lista a la cual se refiere nivel. Podemos asegurar que el método regresa un vértice activo cuyo $e(u) \geq \frac{\Delta}{2}$.

■

Teorema 4.7 *El algoritmo APEE encuentra el flujo máximo.*

Demostración:

Dado que se cumplen las precondiciones de los métodos abstractos, podemos asegurar por el teorema 4.1 que el flujo actual es máximo.

■

4.5.3 Complejidad de APEE

Lema 4.31 *El algoritmo satisface las siguientes condiciones:*

1. *Cualquier empuje no saturado envía al menos $\frac{\Delta}{2}$ unidades de flujo.*

2. *Ningún exceso es mayor que Δ.*

Demostración:

Primero demostraremos la primera condición. Supongamos que realizamos un empuje no saturado del vértice u al vértice v a través del arco (u,v). Como el arco es admisible entonces se cumple que $d(u) > d(v)$, y como el algoritmo selecciona un vértice con distancia mínima podemos asegurar que $e(u) \geq \frac{\Delta}{2}$ mientras que $e(v) < \frac{\Delta}{2}$; dado que el empuje es no saturado $\delta = \min\{e(u), \Delta - e(v)\}$. Por lo tanto, $\delta \geq \frac{\Delta}{2}$ unidades, con lo cual se cumple la primera parte del lema. Para la segunda parte observemos lo siguiente: el exceso $e'(v)$ del vértice v después del empuje está definido de la siguiente manera:

$$e'(v) = e(v) + \min\{e(u), \Delta - e(v)\} \leq e(v) + (\Delta - e(v)\} \leq \Delta$$

.

■

Lema 4.32 *El algoritmo APEE realiza $O(n^2)$ empujes no saturados por cada fase y $O(n^2 \log U)$ empujes no saturados en total.*

Demostración:

Una vez demostrada la primera parte del lema, la segunda es consecuencia inmediata de la primera y del hecho de que el algoritmo realiza a lo más $O(\log U)$ fases, entendiendo por fase el periodo de tiempo que Δ permanece constante.

Para la primera parte del lema consideremos la siguiente función: sea $\Phi = \sum_{u \in V} e(u)\frac{d(u)}{\Delta}$; el valor inicial de Φ en una $\Delta-$fase de escalamiento está acotado por $2n^2$ dado que $e(u)$ es una cota para Δ y la $d(u) < 2n$. Ahora bien, mientras empujamos flujo ocurre una de las siguientes situaciones: en el primer caso consideramos que el algoritmo no encuentra ningún arco admisible para el vértice activo u por el cual podamos enviar flujo; entonces el vértice u es reetiquetado, es decir, su distancia se incrementa, lo cual implica que el valor de Φ también se incrementa; pero como la distancia de cualquier vértice está acotada por $2n$ entonces el incremento es a lo más de $2n$ unidades en cada $\Delta-$fase de escalamiento. En el otro caso, cuando sí existe un arco admisible para el vértice activo u, realizamos un empuje de flujo del vértice u al vértice v, donde v es la segunda componente del arco admisible. Si realizamos un empuje no saturado enviamos al menos $\frac{\Delta}{2}$ unidades de flujo del vértice u al vértice v, y dado que $d(u) > d(v)$ entonces el valor de Φ se decrementa en al menos $\frac{1}{2}$ unidad. Dado que el valor inicial de Φ es $2n^2$ cuando comenzamos una fase, y el incremento de Φ en la fase es a lo más $2n^2$, entonces el número de empujes no saturados es a lo más n^2.

■

La siguiente tabla muestra la complejidad de los algoritmos definidos en APDM.

Método	Complejidad	Justificación
seleccionaVerticeActivo	$O(1)$	Únicamente elegimos un vértice de la lista $L(nivel)$.
contieneVerticeActivo	$O(1)$	Verificamos si $nivel > 0$ y $nivel \leq 2n+1$.
creaListasDeDistancias	$O(n)$	Crea $2n+1$ lista, dado que el crear una lista es $O(1)$.

Método	Complejidad	Justificación
agregaVerticeActivo	$O(1)$	Únicamente agregamos un vértice a la lista correspondiente.
eliminaVerticeActivo	$O(n)$	Debemos actualizar el valor de *nivel* para lo cual realizamos a lo más $2n+1$ comparaciones.
empujaFlujo	$O(n)$	Depende de la complejidad de eliminaVerticeActivo y agregaVerticeActivo.
actualizaListaDeVertices ActivosConExceso-Grande	$O(n)$	A lo más revisa todos los vértice para saber cual de ellos está activo.
defineCapacidadMaxima	$O(m)$	Revisa la capacidad de todas las aristas para saber cual de ellas tiene capacidad mínima.

Sustituyendo las complejidades de los métodos de APEE en la ecuación de complejidad de AGP obtenemos la siguiente ecuación:

$$\begin{aligned} f_{AGP} &= f_{inicializa} + \sum_{i=1}^{q}(f_{esPosibleIncrementarElFlujo} + f_{incrementaFlujo}) \\ &\quad + f_{valorDelFlujoMaximo} \\ &= \sum_{i=1}^{m} f_{agregaVerticeActivo} + \sum_{i=1}^{q}\Big(f_{contieneVerticeActivo} \\ &\quad + (f_{selecionaVerticeActivo} + f_{empujaFlujo} + f_{reetiqueta})\Big) + O(1) \\ &= \sum_{i=1}^{m} O(1) + \sum_{i=1}^{q}\Big(O(1) + (O(1) + (O(1) + O(n)) + O(n))\Big) + O(1) \\ &= O(m) + \sum_{i=1}^{q} O(n) \end{aligned}$$

Ya que aún no hemos determinado el valor de q, por lo que, el siguiente resultado nos demuestra la complejidad del algoritmo.

Teorema 4.8 *El tiempo de ejecución de APEE es* $O(nm + n^2 \log U)$.

Demostración:

Por el lema anterior tenemos que el número de empujes no saturados es $O(n^2 \log U)$ y como el número de empujes saturados y operaciones de reetiquetamiento está acotado por

$O(nm)$, el tiempo total que el algoritmo APEE necesita para encontrar el flujo máximo es $O(nm + n^2 \log U)$.

■

Con este último algoritmo damos por terminado el capítulo correspondiente a las implementaciones concretas de AGFM basados en preflujo, las desventajas y ventajas de cada uno de los algoritmos expuestos en este trabajo se analizarán en las conclusiones.

Conclusiones

Los algoritmos genéricos cuentan con varias ventajas, aunque como en todo lo concerniente a las ciencias de la computación, también ofrecen ciertas desventajas.[2]

Algunas de las ventajas que obtenemos al abstraer las características comunes de una misma clase de problemas y desarrollar un algoritmo genérico que resuelva, de forma abstracta cada uno de estos problemas son las siguientes: una vez que detectamos cuáles son las características que tienen en común todos los algoritmos pertenecientes a una misma clase, podemos entender el funcionamiento de cada uno de ellos con una visión más general, podemos olvidarnos del cómo cada algoritmo resuelve el problema y enfocarnos a probar que si se cumplen ciertas precondiciones, también se cumplen las postcondiciones dadas para cada una de las funciones que abstraímos. Suponiendo lo anterior, podemos demostrar que el algoritmo genérico es correcto, para lo cual se definen invariantes que se extenderán a cada especialización concreta del algoritmo genérico, ahorrándonos la latosa tarea de demostrar que estas invariantes también se cumplen en cada uno de los algoritmos concretos. De esta manera, en las implementaciones concretas del algoritmo genérico únicamente nos resta verificar que se cumplan las postcondiciones dadas para los métodos que abstraímos. Por ejemplo, en el Algoritmo Genérico de Trayectoria Aumentante demostramos que si no es posible encontrar una trayectoria aumentante del vértice origen al vértice destino, el flujo encontrado es un flujo máximo, para lo cual asumimos que el método abstracto encuentraTrayectoriaAumentante regresa **true** si es posible construir una trayectoria aumentante y regresa **false** en caso contrario. Por lo tanto, en los algoritmo concretos de AGTA tuvimos que demostrar que dicha condición se cumple, pero una vez demostrado esto probar que el algoritmo concreto resuelve el problema de flujo máximo era algo muy trivial. Otra de las ventajas que nos proporcionan los algoritmos genéricos está relacionada con la implementación, donde nos evitan tener código repetido, esto es, una vez definidos los métodos que son iguales en cada algoritmo concreto únicamente tenemos que definir en cada algoritmo concreto los métodos abstractos.

Si a estas ventajas le sumamos el hecho de que una vez entendido el funcionamiento del algoritmo genérico, sólo nos resta entender cómo resuelve las operaciones abstractas cada uno de los algoritmos concretos, podemos ver que la idea de utilizar algoritmos genéricos para resolver algún problema no solamente cumplen su objetivo sino que también nos ahorran trabajo.

También se mencionó que tenían ciertas desventajas. El algoritmo genérico por sí mismo no resuelve el problema, dado que asume que se cumplen las invariantes para los métodos abstractos, lo cual no garantiza que esto ocurra: no da métodos concretos para llevar a cabo las operaciones básicas del algoritmo.

En el caso de redes, las ventajas obtenidas al utilizar algoritmos genéricos para resolver

---

[2] Por un lado ganamos y por otro perdemos.

el problema de flujo máximo fueron varias. Una vez demostrada la correctez de cada uno de los algoritmos genéricos, demostrar cada especialización concreta de estos se redujo a demostrar que las invariantes expuestas en la versión abstracta se cumplieran en cada una de las implementaciones concretas. Además se evitó la duplicación de código.

Las siguientes tablas nos muestra las implementaciones concretas de los algoritmo genéricos AGTA y AGP, su complejidad y la idea que siguen para resolver los métodos abstractos.

Algoritmos Concretos de AGTA

En el caso de los algoritmos concretos del Algoritmo Genérico de Trayectoria Aumentante, el método abstracto principal es encuentraTrayectoriaAumentante, la tabla siguiente nos da un breve resumen de como cada algoritmo encuentra la trayectoria aumentante.

Algoritmo	Complejidad	Idea principal del algoritmo
AFME	$O(nmU)$	Encuentra la trayectoria aumentante etiquetando todos los vértices u para los cuales existe una trayectoria dirigida de s a u. Si t es etiquetado entonces existe una trayectoria aumentante.
AEC	$O(m^2 \log U)$	Utiliza la misma idea de AFME con la peculiaridad de que en cada fase el algoritmo intenta encontrar una trayectoria con capacidad "suficientemente grande".
ATALM	$O(n^2 m)$	Encuentra una trayectoria aumentante del vértice s al vértice t con longitud mínima. Para encontrar dicha trayectoria se utilizan las distancias etiquetadas de los vértices de la red.
ADinic	$O(n^2 m)$	Este algoritmo también busca trayectorias aumentantes de longitud mínima, pero crea una red de capas para lograr su propósito.

De los algoritmos presentados en la tabla anterior podemos observar que si la capacidad de los arcos de la red en cuestión es pequeña el algoritmo AFME podría ser una buena elección, dado que la implementación de este algoritmo es muy sencilla. Pero si los arcos de la red tienen capacidades muy grandes entonces la mejor opción son los algoritmos ATALM ó ADinic. Las desventajas de estos últimos recaen principalmente en la implementación, dado que se vuelve más compleja; sobretodo ADinic, el cual mantiene una red de capas en cada iteración. Pero en lo concerniente al tiempo de ejecución son más eficientes. La complejidad de AEC podría reducirse a $O(nm \log U)$ si utilizamos el método encuentraTrayectoriaAumentantes() usado en ATALM, pero aún así, si la capacidad de las aristas es muy alta es más conveniente utilizar ATALM o ADinic.

Algoritmo Concretos de AGP

Ahora bien, para los algoritmo basados en el Algoritmo Genérico de Preflujo la operación abstracta principal es elegir el vértice activo. La siguiente tabla nos muestra una breve idea

de cómo cada algoritmo elige el siguiente vértice activo para ser explorado y la complejidad de cada uno de ellos.

Algoritmo	Complejidad	Idea principal del algoritmo
APFIFO	$O(n^3)$	Este algoritmo mantiene una lista de vértices activos en forma de cola, es decir, el primer vértice que entra es el primero que sale (First-In First-Out).
APDM	$O(n^2 m^{\frac{1}{2}})$	Elige el vértice activo con distancia etiquetada máxima, para lo cual, mantiene una estructura de listas $L(k)$ donde cada lista contiene los vértices activos u con $d(u) = k$, para cada $k = 1, \ldots, 2n+1$
APEE	$O(nm + n^2 \log U)$	Elige un vértice activo con exceso muy grande, para lo cual mantiene una estructura de listas $L(k)$ donde cada lista contiene los vértices activos u con $e(u) > \frac{\Delta}{2}$ y $d(u) = k$, para cada $k = 1, \ldots, 2n+1$, donde Δ es un parámetro definido al inicio del algoritmo con $2^{\log U}$ y actualizado en cada fase del algoritmo.

Las ventajas de los algoritmos presentados en la tabla anterior, obviamente, es que la complejidad de cualquiera de ellos es menor que la de los algoritmos basados en AGTA cuando el número de aristas no es lineal con respecto al número de vértices. Para cualquier problema donde la red es muy extensa utilizar un algoritmo basado en AGP es la mejor opción. De los algoritmos presentados en la tabla anterior el mejor es APDM. La única desventaja de este algoritmo es que mantiene una estructura de listas $L(k)$ con $k = 1, 2, \ldots, 2n$. Cada $L(k)$ contiene a los vértices activos con distancia igual a k y además mantiene una referencia a la lista no vacía con distancias máximas, así que mantener y almacenar esta estructura es más complicado y costoso que mantener una cola de vértices activos, que es una de las ventajas del APFIFO.

Comparando todos los algoritmos presentados, podemos concluir que en la práctica el mejor algoritmo en términos de complejidad es APDM, ya que la complejidad es menor que la de los otros algoritmo, excepto APEE, y no es difícil de implementar.

Bibliografía

[1] **Ahuja, R. K. and J. B. Orlin,**
A Fast and Simple ALgorithm for the Maximum Flow Problem,
Operation Research 37, 748-759, 1989.

[2] **Ahuja, R. K., J. B. Orlin and R. E. Tarjan,**
Improved Time Bounds for the Maximum Flow Problem,
SIAM Journal on Computing 18, 939-954, 1989.

[3] **Ahuja, R. K. and J. B. Orlin,**
Distance-directed Augmenting Path Algorithms for Maximum Flow Problems,
Naval Research Logistics Quarterly 38, 413-430, 1991.

[4] **Ahuja, R. K., T. L. Magnanti, J. B. Orlin,**
Network FLows: Theory, ALgorithms, and Applications.,
Prentice-hall, 1993.

[5] **Chartrand, Gary, and Ortrud R.Oellermann,**
Applied and Algorithmic Graph Theory,
McGrawHill, Inc, 1993.

[6] **Cheriyan, J. and S. N. Maheshwari,**
Analysis of Preflow Push Algorithms for Maximum Network Flow,
SIAM Journal on Computing 18, 1057-1086, 1989.

[7] **Copper, James,**
Java Design Patterns
Adisson Wesley, 2002.

[8] **Cormen, T.H.,C.L. Leiserson, y R.L. Riverst,**
Introduction to Algorithms,
MIT Press ans McGrawHill, New York. 1990.

[9] **Dantzig. G. B.,**
Application of Simplex Method to Transportation Problem.
In T.C. Koopmans, editor, Activity Analisys and Production and Allocation, 359-373.
Wiley, New York, 1951.

[10] **Dantzig. G. B.,**
Linear programming and extensions,
Princenton University Press, Princeton, NJ, 1992.

[11] **Dinic, E. A.,**
Algorithm for Solutions of a Problem of Maximum Flow in Networks with Power Estimation,
Sovieth Math. Dokl., 11:1277-1280, 1970.

[12] **Dinic, E. A.,**
The Method of Scaling and Transportation Problems,
Issled. Diskret. Mat. Science, Moscow, 1973.

[13] **, Dung Nguyen, Stephen B. Wong,**
"Design Patterns for Sorting",
"SIGCSE Bulletin", "ACM/SIGCSE", feb, 2001.

[14] **Edmonds, J. and R. M. Karp,**
Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems,
J. Assoc. Comput. Mach., 19:248-264, 1972.

[15] **Ford, L. R., Jr. and D. R. Fulkerson,**
Maximal Flow Through a Network,
Canadian Journal of Math., 8:399-404, 1956.

[16] **Gabow H. N.,**
Scalling Algorithms for Network Problems,
J. of Comp. and Sys. Sci., 31:148-168, 1985.

[17] **Galil, Z and A. Naamad,**
An $O(EV \log^2 V)$ Algorithm for the Maximal Flow Problem,
J. Comput. System Sci., 21:203-217, 1980.

[18] **Goldberg, A. V.,**
A New Max-Flow Algorithm,
Technical Report MIT/LCS/TM-219, Laboratory for Computer Science, M.I.T, 1985.

[19] **Goldberg, A. V. and R. E. Tarjan,**
A New Approach to the Maximum Flow Problem,
J. Assoc. Comput. MAch., 35:921-940, 1988.

[20] **Goldberg, A. V. and S. Rao,**
Flows in Undirected Unit Capacity Networks,
In Proc. 38th IEEE Annual Symposium on Fundations of Computer Science, pages 32-35, 1997

[21] **Goldberg, A. V.,**
Recent Developments in Maximum Flow Algorithms,
Technical Report 98-045, NEC Research Institute, Inc. 1998.

[22] **Harary, Frank,**
Graph Theory,
Addison-Wesley, 1972.

[23] **Heaney, Matthew,**
Charles: A Data Structure Library for Ada95,
2002.

[24] **Jewell, W. S.,**
Optimal flow through networks with gains.
Operations Researchs, 10: 476 -499, 1992.

[25] **Karzanov, A. V.,**
Determining the the Maximal Flow in a Network by the Method of Preflows,
Soviet Math. Dok., 15:434-437, 1974.

[26] **Kozen, Dexter C.,**
The Design and Analysis of Algorithms,
Springer-Verlag New York, 1992.

[27] **Merrit, M. Susan,**
An Inverted Taxonomy of Sorting Algorithms,
Comunications of the ACM,
Vol 28, No 1, pp 96-99, 1985.

[28] **Merrit, M. Susan,**
An Inverted Taxonomy of Sorting Algorithms,
Comunications of the ACM,
Vol 28, No 1, pp 96-99, 1985.

[29] **Mohri, Mehryar,**
Generic ϵ-Removal Algorithm for Weighted Automata,
AT&T Labs-Research.
USA.

[30] **Musser, R. David., Nishanov, Gor V.** (1998),
A Fast Generic Sequence Matching Algorithm,
Computer Science Department
Rensselaer Politechnic Institute, Troy, NY. USA.

[31] **Musser, R. David., Saini, Atul.,**
C++ Programing whit the Standard Template Library

[32] **Shiloach, Y. and U. Vishkin,**
An $O(n^2 \log n)$ Parallel Max-flow Algorithm,
Journal of Algorithms 3, 128-146, 1982.

[33] **Even, Shimon,**
Graph Algorithms,
Computer Science Press, 1979.

[34] **Sleator, D. D. and R. E. Tarjan,**
A Data Structure for Dynamic Trees,
Journal of Computer and System Sciences 24, 362-391, 1983.

[35] **Tarjan, R. E.,**
Data Structures and Network Algorithms,
CBMS-CNFS. Regional Conference
Series in Applied Math. Society for Industrial and Applied Math., USA, Fifth edition, 1988.

[36] **Wayne, K. D.,**
Generalized Maximum Flow Algorithms,
Cornell University,1999