Algoritmos de optimización sobre trayectorias monótonas en gráficas coloreadas por aristas.

Luis Felipe Barba Flores

30 de junio de 2009

1.Datos del alumno
Barba
Flores
Luis Felipe
56 89 77 36
Universidad Nacional Autónoma de México
Facultad de Ciencias
Matemáticas
300693401

2.Datos del tutor
Dra.
Hortensia
Galeana
Sánchez

3.Datos del sinodal 1
Dr.
Hugo
Rincón
Mejía

4.Datos del sinodal 2
Mat.
Laura
Pastrana
Ramírez

5.Datos del sinodal 3
Dra.
Rocío
Rojas
Monroy

6.Datos del sinodal 4
Dr.
Ricardo
Gómez
Aiza

7.Datos del trabajo escrito
Algoritmos de optimización sobre trayectorias monótonas en gráficas coloreadas por aristas
83 p.
2009

Índice general

Introducción

El propósito de este trabajo es mostrar el funcionamiento y la complejidad de distintos algoritmos que proporcionan herramientas de optimización, se enfoca principalmente a la búsqueda de caminos de distancia mínima entre vértices de una gráfica, distancias que se definirán tomando en cuenta diversas características de las gráficas como son el peso de sus aristas o el color de éstas.

Dado que este escrito se enfocará en distancias, demos primero una idea de que significa este concepto. La distancia en el espacio tiene muchas maneras de definirse, pero la que será usada en gráficas puede ser vista como la longitud del camino más corto entre dos puntos. Notemos que para llegar de un lugar a otro hay muchas maneras de hacerlo, pero en muchos casos aquella de menor distancia nos optimizará dinero, combustible, esfuerzo, etc., es por eso que el minimizar distancias es un problema muy útil y de aplicaciones tangibles en la vida cotidiana.

Las gráficas están formadas a partir de un conjunto de vértices y podemos decir que los segmentos sin dirección entre ellos son las aristas, si nuestro objetivo es movernos de un vértice a otro, podemos hacerlo caminando de vértice a vértice a través de aristas, hasta llegar a nuestro destino, el recorrido completo realizado al ir de un vértice a otro, tomando en cuenta que aristas y que vértices se utilizaron, será llamado un camino. Es por esto que la manera en la que se encontrarán las distancias entre vértices irá ligada a la presencia de caminos de longitudes mínimas entre ellos.

Algo que debemos notar es que si damos nuevas definiciones de caminos, restringiendo el paso de éstos sobre ciertas aristas, o pidiendoles que tengan propiedades extras, como seguir un patrón durante su recorrido, etc., al mismo tiempo estaremos definiendo nuevas distancias, siempre y cuando mantengamos el concepto de que la distancia entre dos vértices es la longitud del camino más corto que exista entre ellos.

Cada una de estas distancias nos exigirá nuevas aproximaciónes al problema de minimización, dando así como resultado una serie de algoritmos polinomiales que permitan encontrar de manera sistemática caminos de lon-

gitudes mínimas entre los vértices de una gráfica.

Una de las distancias que trataremos de minimizar como conclusión de este trabajo esta ligada a gráficas con aristas coloreadas, por lo que daremos una introducción a este tema. La idea de coloraciones es un concepto muy utilizado y conocido en la teoría de gráficas, con una gran cantidad de resultados tanto teóricos como prácticos, sin embargo la idea fundamental es muy simple, dada una paleta de colores pintar a cada una de las aristas de la gráfica con algún elemento de ésta. Cabe resaltar que también se puede pintar a los vértices de una gráfica, sin embargo durante el desarrollo de éste escrito no entraremos en ese tema.

Pensemos ahora en una gráfica coloreada por aristas y definamos en ella un tipo de camino especial, al que llamaremos monótono, en el cual se tenga que seguir un cierto patrón de colores, digamos que la pintamos de azul, rojo y verde y le pedimos a los caminos monótonos que sigan un patrón de colores específico, por ejemplo después de rojo solo puede ir azul, después de azul solo puede ir verde y después de verde puede ir cualquier color. Es claro que la cantidad de caminos que existen entre los vértices de la gráfica es mayor que la cantidad de caminos monótonos ya que estamos restringiendo las aristas por las que puede pasar. Por lo tanto la existencia de un camino entre dos vértices no implicará la existencia de uno monótono, de aquí que los algoritmos que funcionen para encontrar caminos no servirán para hallar caminos monótonos, al menos no sin una serie de modificaciones y adaptaciones.

Este patrón de colores a seguir se formaliza con el concepto de digráfica guía, gráfica dirigida que tendrá como vértices a los colores de la paleta con la que se pintó la gráfica, y en la cual las flechas entre colores representarán el patrón que se puede seguir, un camino monótono podrá pasar de un color a otro, digamos de rojo a azul, en la gráfica coloreada por aristas si y sólo si la flecha (rojo, azul) existe en la gráfica guía.

El concepto de caminos o trayectorias monótonas no es nuevo y ya ha sido abordado en algunos artículos como *Reachability Problems in Edge-Coloured Digraphs* o *Monotone reachability in arc-colored tournaments* [1], en donde se aborda la existencia de caminos que sigan patrones dados por ordenes parciales en gráficas coloreadas por aristas, a los cuales se les bautiza como monótonos, sin embargo en estos artículos se dan únicamente resultados teóricos que tratan ya sea de subconjuntos mínimos de vértices alcanzables desde todos los demás por trayectorias monótonas, llamados sumideros, y de los cuales se trata de probar su existencia dadas ciertas características de las gráficas; o de la existencia de trayectorias monótonas en gráficas especiales

[1]Referencia completa en bibliografía [1], [2]

como son lo torneos.

Sin duda estos resultados no van ligados de manera directa a los que presentaremos en esta tesis, pero al proponer la existencia de estos caminos monótonos es que se da la idea de abordarla desde el punto de vista de los algoritmos, es por esto que su mención es importante en los fundamentos del trabajo aquí presentado.

Esta tesis está dividida en cinco capítulos que muestran una evolución de coneptos, desde lo fundamental hasta llegar a los caminos monótonos y su optimización dentro de gráficas coloreadas por aristas.

Los primeros cuatro capítulos nos hablan de resultados conocidos, la mayoría expuestos en la literatura de Teoría de gráficas, como es el caso del libro de *Chartrand*[2], sin embargo el quinto capítulo se enfoca en resultados nuevos, obtenidos de la idea de generalizar los conceptos y algoritmos ya conocidos a gráficas coloreadas por aristas usando trayectorias monótonas. Una breve descripción de cada capítulo se presenta a continuación:

El primer capítulo es una breve introducción a la teoría de gráficas, donde definiremos los fundamentos, términos como vértice, arista, gráfica, adyacencia, etc., dándonos así las herramientas para poder desarrollar conceptos como el de subgráfica y conexidad. Se definen también las bases para el estudio de digráficas, básicamente serán gráficas en donde la dirección de los tramos entre vértices si importará, es decir, aristas con dirección y por lo tanto los caminos que sigamos dentro de una digráfica serán caminos dirigidos.

En el segundo capítulo daremos una introducción a algoritmos, definiéndolos y analizando sus complejidades. También daremos ejemplos sencillos que nos permitan familiarizarnos con su estructura y ver como la complejidad entre ellos puede ser muy distinta, incluso en aquellos que tengan el mismo objetivo.

El capítulo tres regresará a la teoría de gráficas presentándonos el concepto de árbol, una gráfica en la que no pueda haber caminos cíclicos, es decir, que al caminar por los vértices de un árbol nunca repetiremos ni aristas ni vértices. Mostraremos varios teoremas que describan sus propiedades y nos den herramientas para caracterizarlos. Paso siguiente veremos un caso especial de árboles llamados arraigados, extensiones de los árboles a gráficas dirigidas en donde un vértice raíz puede alcanzar a todos los demás a través de caminos dirigidos. Con esto podemos empezar con los algoritmos sobre los que se basarán los resultados de este trabajo, la Búsqueda de Amplitud Inicial y la Búsqueda de Profundidad Inicial, cada uno de los cuales presenta una manera sistemática de visitar cada vértice de una gráfica dada, en un tiempo polinomial. Se presentan también algunas aplicaciones de éstos para

[2]Referencia completa en bibliografía [3].

encontrar bloques, componentes conexas, etc.

El peso en las aristas es otro concepto que se introduce dentro del tercer capítulo, esto es, dada una gráfica, a cada arista le asignamos un valor número-rico. Presentándonos de manera natural el concepto de optimización al tratar de obtener caminos, árboles, etc., de peso mínimo, para lo cual se desarrollan varios algoritmos a lo largo de este capítulo y el siguiente.

En el cuarto capítulo entramos de lleno en el problema de las distancias y como consecuencia de caminos mínimos. La primera que se define surge al minimizar la longitud de caminos entre vértices, longitud vista como el número de aristas por las que pasa el camino. Para esto se desarrolla el algoritmo de Moore y sus variaciones, que en esencia encuentran la distancia entre cualesquiera dos vértices de la gráfica y encuentra un camino entre ellos de longitud mínima.

Como segundo tema de este capítulo se da una nueva distancia en gráficas con peso en las aristas. Primero definimos el peso de un camino como la suma de los pesos de sus aristas y si en vez de minimizar la longitud de un camino, minimizamos su peso, entonces podemos definir la distancia como el peso del camino más ligero entre dos vértices. Es claro que no existe ninguna relación entre esta nueva distancia y la previamente definida, por lo que el algoritmo para minimizarla será muy distinto. En esta sección, y con este propósito, se desarrolla el algoritmo de Dijkstra para determinar la distancia entre cualesquiera dos vértices de una gráfica con peso en las aristas.

Finalmente el capítulo cinco culmina este trabajo integrando el concepto de coloraciones, y definiendo la distancia monótona entre dos vértices en una digráfica coloreada, como la longitud del camino dirigido monótono más corto entre ellos. El resultado se generalizó para digráficas ya que el problema en gráficas es un caso particular de éste.

Terminamos por desarrollar el algoritmo central de esta tesis, algoritmo que encontrará, en una digráfica coloreada, trayectorias monótonas de longitud mínima entre cualesquiera dos vértices de ella, es decir, encontrará la distancia monótona entre ellos. Analizamos su complejidad y probamos con rigor su funcionamiento para finalizar este trabajo.

Capítulo 1

Gráficas

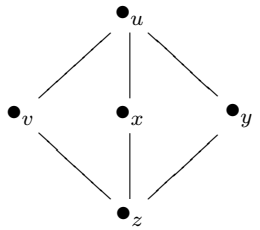
1.1. ¿Qué es una gráfica?

Una gráfica G es un conjunto finito y no vacío $V\langle G\rangle$ de objetos llamados vértices y un conjunto $E\langle G\rangle$ de pares de elementos de $V\langle G\rangle$ llamados aristas.

Por comodidad si $\{u, v\} \in E\langle G\rangle$ usaremos la siguiente notación, uv o (u, v).

Definimos también la relación binaria "ser adyacente" en $V\langle G\rangle$ de la siguiente manera: Dados $u, v \in V\langle G\rangle$ tendremos que u y v son adyacentes si y sólo si $\exists e \in E\langle G\rangle$ tal que $e = \{u, v\}$, en cuyo caso diremos que e *incide* tanto en u como en v.

Cada gráfica tiene un diagrama asociado a ella, éste consta de vértices, representados por puntos, y aristas representadas por lineas que unen vértices adyacentes. Dada G gráfica con $V\langle G\rangle = \{u, v, x, y, z\}$ y $E\langle G\rangle = \{uv, ux, uy, zv, zx, zy\}$, podemos representarla con el siguiente diagrama:



Existen problemas para los cuales, al modelarlos, requerimos de varias aristas entre dos vértices, como al modelar caminos entre dos ciudades, para lo cual definimos las multigráficas como una extensión de las gráficas en las que dos o más aristas pueden unir el mismo par de vértices. Es claro que el conjunto de las gráficas está contenido en el de las multigráficas pero no a la inversa. De forma similar si permitimos que existan aristas que unan a

un vértice consigo mismo (*bucles*) obtenemos otra extensión del conjunto de gráficas al que llamaremos pseudográficas. Una gráfica simple será aquella que no es ni multigráfica ni pseudográfica, en el transcurso del texto nos referiremos a éstas siempre que no especifiquemos una multigráfica o una pseudográfica.

Llamaremos el *orden* de una gráfica al cardinal de $V\langle G\rangle$, denotado por p, y el *tamaño* al cardinal de $E\langle G\rangle$, denotado por q.

1.2. Grado de un vértice

Dada G gráfica y $v \in V\langle G\rangle$, definimos el *grado de* v como sigue:

$$\delta(v) = |\{\ e \in E\langle G\rangle \mid e \ \ incide\ en\ \ v\ \}|$$

es decir el número de aristas incidentes en v de G.

Denotaremos a $\delta(G) = min\{\delta(v)|v \in V\langle G\rangle\}$ como el *mínimo grado* de G y a $\Delta(G) = max\{\delta(v)|v \in V\langle G\rangle\}$ como el *máximo grado* de G.

Definimos también a los *vecinos de* v como:

$$N(v) = \{\ u \in V\langle G\rangle \mid uv \in E\langle G\rangle\ \}$$

Podemos notar entonces que si una gráfica es simple $|N(v)| = \delta(v)$.

Hay varias cosas más que debemos resaltar sobre el grado de un vértice, si p es el orden de G, es claro que $0 \leq \delta(v) \leq p-1$. Un vértice de grado cero es llamado *vértice aislado*, y uno de grado uno *vértice terminal*. Llamamos *par* o *impar* a un vértice de acuerdo a la paridad de su grado.

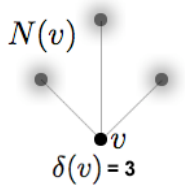


Figura 1.1:
En esta figura podemos observar al vértice v de grado tres y resaltados con un halo a sus vecinos $N(v)$.

Teorema 1.2.1 *Sea G gráfica de orden p y tamaño q con $V\langle G\rangle = \{\ v_1, v_2, \ldots, v_p\ \}$. Entonces*

$$\sum_{i=1}^{p} \delta(v_i) = 2q$$

Demostración. Al sumar el grado de cada vértice de G cada arista es contada dos veces, una vez por cada vértice en el que incide. $\square$

Del Teorema anterior es fácil concluir lo siguiente:

Corolario 1.2.2 *Cada gráfica contiene un número par de vértices impares.*

Demostración. Si definimos V_p y V_i como el conjunto de vértices de grado par e impar respectivamente, es claro que $V\langle G\rangle = V_p \cup V_i$, además de ser ajenos, por lo tanto

$$\sum_{v\in V\langle G\rangle} \delta(v) = \sum_{v\in V_i\langle G\rangle} \delta(v) + \sum_{v\in V_p\langle G\rangle} \delta(v) = 2q$$

de aquí que

$$\sum_{v\in V_i} \delta(v) = 2q \ - \ \sum_{v\in V_p} \delta(v).$$

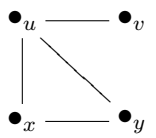
Donde los dos elementos de la derecha son claramente pares, por lo tanto la suma de la izquierda debe contener un número par de sumandos, pues cada elemento de ésta es impar. $\square$

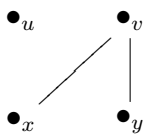
Diremos que una gráfica G es *r-regular* o *regular de grado r* si

$$\forall v \in V\langle G\rangle \quad \delta(v) = r$$

podemos observar que $0 \leq r \leq p-1$.

Dada G gráfica, definimos su complemento $\overline{G}$ como la gráfica con $V\langle\overline{G}\rangle = V\langle G\rangle$ y tal que $\forall u, v \in V\langle\overline{G}\rangle$ $uv \in E\langle\overline{G}\rangle$ *si y sólo si* $uv \notin E\langle G\rangle$. La figura siguiente muestra una gráfica y su complemento.

G : 

$\overline{G}$: 

1.3. Subgráficas

Sea G una gráfica.

Una gráfica H es subgráfica de G si $V\langle H\rangle \subseteq V\langle G\rangle$ y $E\langle H\rangle \subseteq E\langle G\rangle$, y lo denotamos como $H \subseteq G$. Notemos que la relación "ser subgráfica de $(\subseteq)$" es un orden parcial sobre el conjunto de todas las gráficas.

Algunas definiciones importantes para subgráficas son las siguientes:

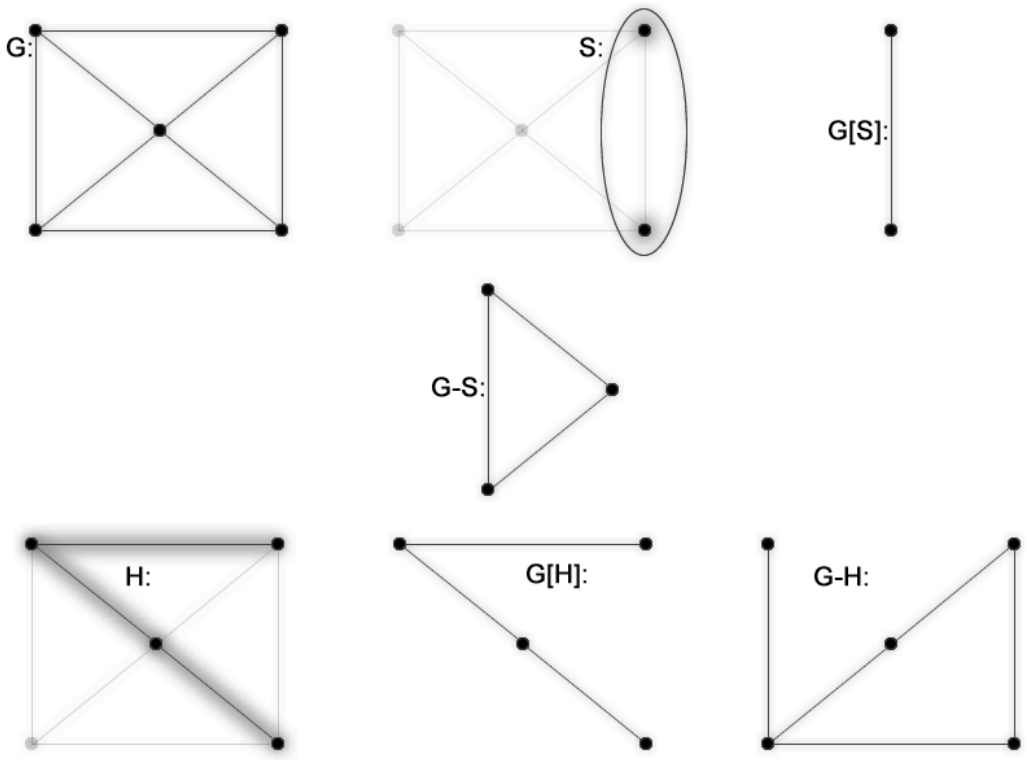


Figura 1.2:
En este ejemplo podemos observar una gráfica G y las operaciones definidas con un subconjunto de vértices S y un subconjunto de aristas H.

Dado $S \subseteq V\langle G\rangle$ *la gráfica inducida* por S es la $\subseteq$-máxima subgráfica de G con S como su conjunto de vértices, esta gráfica es denotada como $G[S]$. Denotamos a $G - S$ como la gráfica inducida obtenida a partir de $V\langle G\rangle - S$. Si $S = \{v\}$ denotaremos a $G - S$ como $G - v$ en vez de $G - \{v\}$.

Dado $X \subseteq E\langle G\rangle$ *la gráfica inducida* por X es la $\subseteq$-mínima subgráfica de G con X como su conjunto de aristas, esta gráfica la denotaremos, de igual forma, como $G[X]$.

Una subgráfica H de G es llamada *generadora* si $V\langle H\rangle = V\langle G\rangle$. De aquí podemos observar que $G - X$ es una gráfica generadora $\forall X \subseteq E\langle G\rangle$. Por comodidad si $X = \{e\}$ denotaremos a $G - X$ como $G - e$.

De igual forma si u_i, v_i son pares de vértices no adyacentes en G, $\forall i \in \{1, 2, \ldots, m\}$, entonces $G + \{\ u_1v_1, u_2v_2, \ldots, u_mv_m\ \}$ se define como la gráfica formada a partir de G añadiendo las nuevas aristas. Si u,v son vértices no adyacentes en G, denotamos $G + \{uv\}$ como $G + uv$ únicamente.

1.4. Conexidad

Dada G gráfica definimos un *camino* en G como una sucesión alternada de vértices y aristas $W : \{v_0, e_1v_1, e_2, v_2 \ldots, v_{n-1}, e_n, v_n\}$ donde $\forall i \in \{1, 2, \ldots, n\}$ $e_i = v_{i-1}v_i$. En el caso de gráficas simples podemos omitir las aristas ya que quedan implícitas y simplemente denotar el camino como sigue, $W : v_0, v_1, \ldots, v_{n-1}, v_n$ o $W : (v_0, v_1, \ldots, v_{n-1}, v_n)$, siempre y cuando la arista v_iv_{i+1} exista $\forall i \in \{0, 1, \ldots, n-1\}$. A un camino que inicia en v_0 y termina en v_n se le llama v_0-v_n-*camino*. También definimos la longitud de un camino como el número de aristas (no necesariamente distintas) que en él se encuentran. Llamaremos cerrado a un camino que inicie y termine en el mismo vértice. Habrá ocasiones en que queramos recorrer un camino $W = (v_0, v_1, \ldots, v_n)$ en el sentido opuesto, para referirnos al mismo camino recorrido inversamente usaremos la notacion $W^{-1} = (v_n, v_{n-1}, \ldots, v_0)$.

Un *paseo* en G se define como un camino en el cual ninguna arista es repetida y una trayectoria como un camino en el cual ningún vértice se repite. Podemos observar que toda trayectoria es una paseo pero no a la inversa. La idea de camino cerrado se generaliza también para paseos.

Un *ciclo* se define como una camino cerrado en el cual no hay vértices que se repitan salvo el primero y el último, llamaremos la *longitud* de un ciclo al número de vértices distintos que en éste haya. Dado un vértice o una arista de G diremos que son acíclicos si no son parte de ningún ciclo en G.

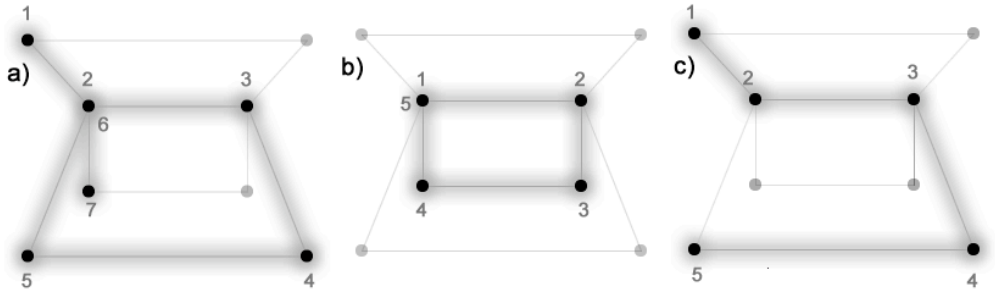


Figura 1.3:
a) Ejemplo de un paseo. b) Ejemplo de un ciclo. c) Ejemplo de una trayectoria.
En los tres cases la numeración indica el sentido en el que se recorren los vértices

Teorema 1.4.1 *Todo u-v-camino contiene una u-v-trayectoria.*

Demostración. Sea W un u-v-camino en G. Si $u = v$, entonces la trayectoria u-u resuelve el problema, por lo tanto podemos suponer que $u \neq v$. Digamos que $W : \ u = v_0, v_1, \ldots, v_n = v$. Si ningún vértice se repite en W, entonces ya es una trayectoria. De lo contrario $\exists v_j, v_k \in V\langle W\rangle$ tal que $v_j = v_k$ con $j < k$. De aquí que podemos afirmar que $W' : \ u = v_0, \ldots, v_j, v_{k+1}, v_{k+2}, \ldots, v_n = v$ es un nuevo u-v-camino contenido en W, si en W' no hay vértices que se repitan, entonces W' es una u-v-trayectoria contenida en W, de lo contrario podemos repetir el proceso anterior hasta obtener la trayectoria deseada dado que el número de vértices que aparecen más de una vez en W es finito. $\square$

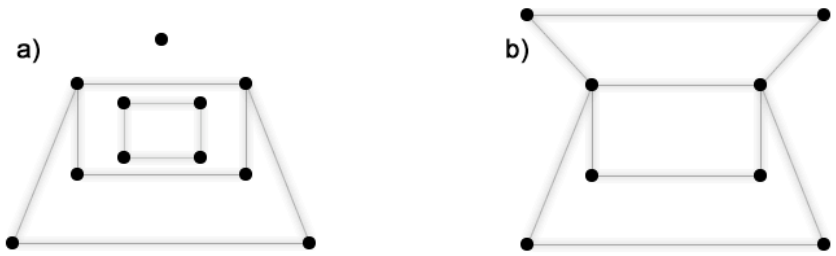


Figura 1.4:
a)Gráfica inconexa con $c(G) = 3$. b)Gráfica conexa.

Dada G gráfica definimos la siguiente relación de equivalencia sobre $V\langle G\rangle$,

para todo $u, v \in V\langle G\rangle$ diremos que u *está conectado a* v si y sólo si existe T una u-v-trayectoria.

La gráfica G es *conexa* si y sólo si para todo $u, v \in V\langle G\rangle$, u está conectado a v. Una gráfica que no cumpla con lo anterior es llamada *inconexa*.

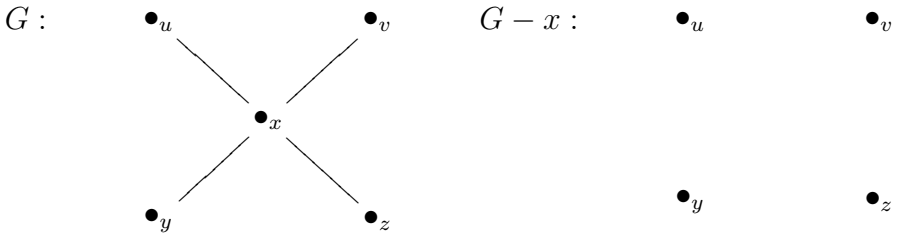
La relación "estar conectado a" es claramente una relación de equivalencia, por lo que define una partición sobre $V\langle G\rangle$, digamos $\{V_1, V_2, \ldots, V_k\}$. Definimos las *componentes conexas de una gráfica* como los elementos de la partición. Podemos observar que $G[V_i]$ es una subgráfica conexa máxima para todo $i \in \{1, 2, \ldots, k\}$, ya que de lo contrario existirían vértices $v \in V_i$ y $u \in V\langle G\rangle - V_i$ tales que u esta conectado a v y por lo tanto estarían en la misma partición bajo la relación "estar conectado a", lo cual nos lleva a una contradicción pues $u \in V\langle G\rangle - V_i$, por consiguiente V_i es una subgráfica conexa máxima.

Denotaremos con $c(G)$ al número de componentes conexas de G, por lo tanto podemos decir que una gráfica es conexa si y sólo si $c(G) = 1$.

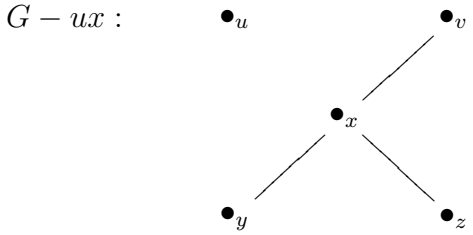
1.5. Vértices de corte y Puentes

Dada G gráfica y $v \in V\langle G\rangle$ decimos que v es un *vértice de corte* si $c(G-v) > c(G)$, con lo que podemos afirmar que si G es conexa, v es vértice de corte si y sólo si $G - v$ es inconexa.

Existe una definición análoga para aristas, a saber que dada $e \in E\langle G\rangle$ e es un *puente* si $c(G-e) > c(G)$. Podemos observar en el siguiente diagrama que un vértice de corte puede separar la gráfica en dos o más componentes conexas, sin embargo un puente separa a G en exactamente dos componentes conexas.



El vértice de corte x, al ser removido, separa a la gráfica G en cuatro componentes conexas.



El puente ux únicamente separa a G en dos componentes conexas.

Daremos a continuación una caracterización de vértices de corte y una de puentes.

Teorema 1.5.1 *Sea G conexa y sea $v \in V\langle G\rangle$, entonces son equivalentes:*

i) v es un vértice de corte.

ii) $\exists V_1, V_2$ partición de $V\langle G-v\rangle$ tales que $\forall T$ V_1-V_2-trayectoria se cumple que $v \in V\langle T\rangle$.

iii) $\exists x, y \in V\langle G - v\rangle$ *tales que* $\forall T$ *x-y-trayectoria se cumple que* $v \in V\langle T\rangle$.

Demostración. Sea G conexa y $v \in V\langle G\rangle$.

i) $\rightarrow$ *ii)* Sea v vértice de corte en G. Sabemos por definición que $c(G-v) > c(G)$, digamos entonces que $C_1, C_2, \ldots, C_k$ son las componentes conexas de $G - v$. Definimos a $V_1 = \cup_{i=1}^{k-1} C_i$ y $V_2 = C_k$.
Sea T una V_1-V_2-trayectoria en G tal que $T = (x = t_0, t_1, \ldots, t_n = y)$ con $x \in V_1$ y $y \in V_2$, sin perdida de generalidad supongamos que $x \in C_1$ y $y \in C_k$. Supongamos ahora que $v \notin V\langle T\rangle$, por lo tanto T es una V_1-V_2-trayectoria en $G - v$, más aún T sería una C_1-C_k-trayectoria, de aquí que $C_1 \cup C_k$ conexa, lo cual es una contradicción que viene de suponer que $v \notin V\langle T\rangle$, por consiguiente $v \in V\langle T\rangle$.

ii) $\rightarrow$ *iii)* Sea V_1, V_2 una partición de $V\langle G\rangle$ tal que $\forall T$ V_1-V_2-trayectoria se cumple que $v \in V\langle T\rangle$.
Sea $x \in V_1$ y $y \in V_2$, ahora como toda x-y-trayectoria es una V_1-V_2-trayectoria, entonces toda x-y-trayectoria contiene a v.

iii) $\rightarrow$ *i)* Sean $x, y \in V\langle G\rangle$ tales que $\forall T$ x-y-trayectoria se cumple que $v \in V\langle T\rangle$.
Supongamos que $G - v$ es conexa, entonces $\exists T'$ una x-y-trayectoria en $G - v$, lo cual es una contradicción pues todas las x-y-trayectorias pasaban por v y T' no lo hace. Contradicción que viene de suponer que $G - v$ conexa, de aquí que $G - v$ es inconexa y por lo tanto como G es conexa $c(G - v) > c(G)$. □

Teorema 1.5.2 *Sea* G *conexa con* $p \geq 2$ *y* $e \in E\langle G\rangle$ *entonces son equivalentes:*

i) e *es un puente.*

ii) $\exists V_1, V_2 \subseteq V\langle G\rangle$ *partición de* $V\langle G\rangle$ *tales que* $\forall T$ V_1-V_2-*trayectoria,* T *pasa por* e.

iii) $\exists u, v \in V\langle G\rangle$ *tales que toda* u-v-*trayectoria pasa por* e.

iv) e *es acíclica.*

Demostración. Sea G conexa con $p \geq 2$ y $e = (x, y) \in E\langle G\rangle$.

i) $\rightarrow$ *ii)* Sea e puente y sean C_1, C_2 las componentes conexas de $G - e$, definimos $V_1 = C_1, V_2 = C_2$ una partición de $V\langle G\rangle$. Sea $T = (u = t_0, t_1, \ldots, t_n = v)$ una V_1-V_2-trayectoria en G con $u \in V_1$ y $v \in V_2$ por lo tanto si suponemos que T no pasa por e, entonces en $G - e$ tenemos que u estaría conectado a v por T y por definición estarían en la misma componente conexa de $G - e$, lo

cual es una contradicción.
De aquí concluimos que T pasa por e.

ii) $\rightarrow$ *iii)* Sea $V_1, V_2 \subseteq V\langle G\rangle$ una partición de los vértices de G, tal que $\forall T$ V_1-V_2-trayectoria, T pasa por e.
Sea $u \in V_1$ y $v \in V_2$, como toda u-v-trayectoria es una V_1-V_2-trayectoria, entonces toda u-v-trayectoria pasa por e.

iii) $\rightarrow$ *iv)* Sean $u, v \in V\langle G\rangle$ tales que toda u-v-trayectoria pasa por $e = (x, y)$ y supongamos que $\exists\varphi$ ciclo en G tal que $e \in \varphi$. Tomamos a T una u-v-trayectoria tal que $T = (u = t_0, t_1, \ldots, t_m = x, t_{m+1} = y, \ldots, t_n = v)$. Definimos $M = \{\ i \mid t_i \in V\langle\varphi\rangle\} \neq \phi$ pues $m, m+1 \in M$. Sean $j = min(M)$ y $k = max(M)$, con esto definimos $T' = (u = t_0, t_1, \ldots, t_j, \varphi, t_k, \ldots, t_n = v)$ una u-v-trayectoria que dependiendo del sentido en el que se recorra a φ pasa o no por e, si tomamos el caso en el que no lo hace tenemos una u-v-trayectoria que no pasa por e, lo cual es una contradicción que viene de suponer que e estaba en algún ciclo.
Por lo tanto e acíclico.

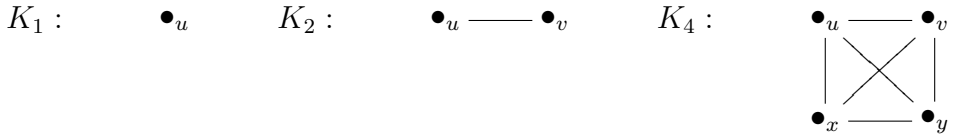
iv) $\rightarrow$ *i)* Por contrapuesta si $e = (x, y)$ no es un puente, entonces en $G - e$ hay una x-y-trayectoria digamos T.
Por lo tanto $T \cup e$ es un ciclo en G que contiene a e. $\square$

Dada G gráfica con $p \geq 3$ definimos la conexidad puntual de G, denotada por $\kappa(G)$, como el mínimo número de vértices necesarios para que, al quitarlos de G, ésta se desconecte o lleguemos a la gráfica trivial con un sólo vértice. Decimos entonces que $B \subseteq G$ es un *bloque* si $\kappa(B) \geq 2$ y B es una subgráfica máxima por contención con esta propiedad. Es claro que dos bloques B_1, B_2 pueden intersectarse a lo más en un vértice de corte, ya que si lo hicieran en dos o más, la unión $B_1 \cup B_2$, tendría también conexidad puntual mayor o igual a 2, contradiciendo la maximidad de B_1 y B_2.

Llamaremos *bloque terminal* a un bloque que tenga un sólo vértice de corte.

1.6. Gráficas especiales

En esta sección daremos notación a varios tipos de gráficas con características específicas. Por ejemplo llamaremos *gráfica completa*, denotada por K_p, a una gráfica de orden p en donde todo par de vértices distintos sean adyacentes. Podemos notar entonces que K_p es (p-1)-regular. Algunos ejemplos de gráficas completas se pueden observar en la siguiente figura.



A una gráfica que sea por si misma una trayectoria de orden n la denotaremos por P_n, de igual forma un ciclo de longitud n es llamado un *n-ciclo* y se denota por C_n. Diremos también que un ciclo es par o impar dependiendo de la paridad de su orden.

Una gráfica G es llamada *bipartita* si existe una partición $V_1, V_2 \neq \phi$ de $V\langle G\rangle$, tales que cada arista de G sea una V_1-V_2-arista. Llamaremos a V_1, V_2 una bipartición si cumple lo anterior. Una caracterización de estas gráficas es dada por el siguiente teorema.

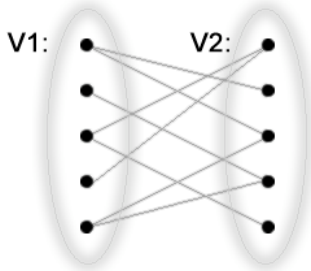


Figura 1.5:
Ejemplo de gráfica bipartita, se pueden observar la partición de $V\langle G\rangle$ V_1, V_2.

Teorema 1.6.1 *Una gráfica G no trivial es bipartita si y sólo si G no contiene ciclos impares.*

Demostración. Sea G una gráfica no trivial.

Para la primera implicación supongamos que G es bipartita, por lo tanto $\exists V_1, V_2 \subseteq V\langle G\rangle$ partición tal que $\forall e \in E\langle G\rangle$ se cumple que e es una V_1-V_2-arista. Sea $C = (v_1, v_2, \ldots, v_n)$ un n-ciclo en G, de no existir el resultado se cumple por vacuidad. Supongamos sin perdida de generalidad que $v_1 \in V_1$, en consecuencia $v_2 \in V_2$ y así para toda $1 \leq i \leq n$, $v_i \in V_1$ si y sólo si i es impar, y $v_i \in V_2$ si y sólo si i es par. Ahora como $v_n v_1 \in E\langle C\rangle$ y $v_1 \in V_1$ entonces $v_n \in V_2$, por consiguiente n es par.
De aquí concluimos que todo ciclo de G tiene longitud par.

Para el reciproco supongamos que G no contiene ciclos impares y asumamos de momento que G es conexa. Sea $u \in V\langle G\rangle$, por lo tanto $\forall v \in V\langle G\rangle$ existe una u-v-trayectoria en G, por lo que podemos definir a $T_{u,v}$ como una de longitud mínima.
Definamos ahora $V_1 \subseteq V\langle G\rangle$ como el conjunto que contiene a u y a todo vértice $v \in V\langle G\rangle$ tal que la longitud de $T_{u,v}$ sea par; y a $V_2 \subseteq V\langle G\rangle$ como el conjunto que contiene a todo vértice $v \in V\langle G\rangle$ tal que la longitud de $T_{u,v}$ sea impar. Por construcción es claro que V_1 y V_2 son una partición de $V\langle G\rangle$. Sea $e = (x, y) \in E\langle G\rangle$, veamos ahora que x y y pertenecen a distintos elementos de la partición. Podemos suponer sin perdida de generalidad que $x \in V_1$, si suponemos que $y \in V_1$ entonces existen $T_{u,x}$, u-x-trayectoria mínima y $T_{u,y}$, u-y-trayectoria mínima, ambas de longitud par, por lo tanto $\Phi =(\ u, T_{u,x}, x, y, T_{u,y}^{-1}, u)$ un ciclo impar en G, lo cual es una contradicción que viene de suponer que $y \in V_1$. De aquí que $y \in V_2$ y como consecuencia toda arista de G es una V_1-V_2-arista.

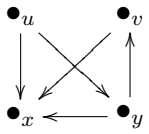
Si G no hubiera sido conexa, podemos suponer que $C_1, C_2, \ldots, C_n$ las componentes conexas de G, de tal forma que si G no contiene ciclos impares, entonces ninguna C_i contiene ciclos impares, por lo que aplicando el proceso anterior obtenemos que cada C_i es bipartita. Digamos que $X_i, Y_i \subseteq V\langle C_i\rangle$ las particiones bipartitas de cada C_i. Por lo que es claro que $\bigcup_{i=1}^{n} X_i$, $\bigcup_{i=1}^{n} Y_i \subseteq V\langle G\rangle$ son una bipartición de G. $\square$

1.7. Digráficas

Hay ocasiones en las cuales es necesario especificar si una arista puede ser recorrida en un sentido pero no en el otro, como en el caso del tráfico en las calles. Para esto surge el concepto de gráficas dirigidas mejor conocidas como *digráficas*.

Una *digráfica* D es un conjunto $V\langle D\rangle$ finito y no vacío de vértices y un conjunto $E\langle D\rangle$, posiblemente vacío, de pares *ordenados* de vértices. Los elementos de $E\langle D\rangle$ serán llamadas *flechas*. Si la flecha $(u, v) \in E\langle D\rangle$, diremos que u es adyacente hacia v o que v es adyacente desde u. En general podemos decir que u y v son adyacentes por flechas. Si existe la flecha (u, v) y la flecha (v, u) diremos que estas dos son simétricas.

Al igual que las gráficas las digráficas también pueden ser representadas por diagramas donde cada vértice se representará con un punto y cada flecha irá del primer vértice del par ordenado hacia el segundo. Un ejemplo se muestra en el siguiente diagrama:

D : 

Dada G gráfica podemos convertirla en una digráfica al darle una orientación a sus aristas, para toda arista $uv \in E\langle G\rangle$ podemos dar una de dos orientaciones (u, v) o (v, u). A la digráfica obtenida con este método se le llama una *orientación* de G. Notemos que en una orientación no hay flechas simétricas.
La gráfica *subyacente* de una digráfica D, denotada por $\underline{D}$, se define como la gráfica G obtenida a partir de remplazar cada flecha $(u, v) \in E\langle D\rangle$ por una arista uv sin dirección eliminando las aristas múltiples entre vértices. Todas las definiciones que hemos dado para gráficas se pueden trasladar a digráficas utilizando la gráfica subyacente de D. Sin embargo necesitaremos nuevas definiciones que nos ayuden a entender mejor a las digráficas.

Si $(u, v) \in E\langle D\rangle$ diremos que (u, v) *sale de* u y *llega a* v. Dado $v \in V\langle D\rangle$ definimos el *grado de entrada* de v como el número de flechas que llegan a v, denotado por $\delta^-(v)$. Y definimos el *grado de salida* de v como el número de flechas que salen de v, denotado por $\delta^+(v)$. Podemos notar aquí que $\delta(v) = \delta^+(v) + \delta^-(v)$.

Definimos también los *vecinos de entrada*

$$N^-(v) = \{\ u \in V\langle D\rangle \mid (u, v) \in V\langle D\rangle\ \}$$

y los *vecinos de salida*

$$N^+(v) = \{\ u \in V\langle D\rangle \mid (v, u) \in E\langle D\rangle\ \}$$

Notamos también que $N^+(v) \cup N^-(v) = N(v)$.

Definiremos un *camino dirigido* como una sucesión alternada de vértices y flechas, digamos $W = (v_0, e_1, v_1, e_2, \ldots, v_{n-1}, e_n, v_n)$ donde cada $e_i = (v_{i-1}, v_i) \in E\langle D\rangle$. Del mismo modo que en las gráficas un camino dirigido puede ser denotado de manera más simple como $W = (v_0, v_1, \ldots, v_n)$. Las definiciones de paseo, trayectoria y ciclo dirigido se dan de la misma forma que para gráficas pero partiendo de un camino dirigido.

Dados $u, v \in V\langle D\rangle$ diremos que v es alcanzable desde u si existe una u-v-trayectoria dirigida en D, es claro que el hecho de que exista una trayectoria dirigida de u a v no implica que haya una de v a u, así como el hecho de que haya una trayectoria entre u y v en la gráfica subyacente, no implica que

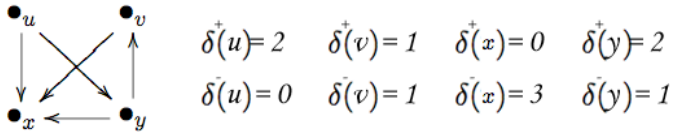


Figura 1.6:
En esta digráfica podemos observar el grado de entrada y salida de cada uno de sus vértices.

haya alguna trayectoria dirigida entre estos vértices.

Una digráfica D es regular si existe $r \in \mathbb{N}$ tal que $\forall v \in V\langle D\rangle$ se cumple que $\delta^+(v) = \delta^-(v) = r$. D es también simétrica si $\forall e = (u, v) \in E\langle D\rangle \quad \exists e' = (v, u) \in E\langle D\rangle$, en cuyo caso diremos que toda flecha tiene su conjugado.

Diremos también, dada una gráfica G que D es la digráfica simétrica de G, denotada por $D = G^*$, si y sólo si D se obtiene de G al remplazar cada arista $uv \in E\langle G\rangle$ por las flechas (u, v) y (v, u), llamaremos a estos últimos *el par conjugado de* uv y a uv *la arista subyacente de* (u, v) *o de* (v, u). Notemos también que $V\langle G\rangle = V\langle D\rangle$.

Figura 1.7:
En esta figura podemos observar a K_3 y su digráfica simétrica.

Veremos a continuación propiedades interesantes de las digráficas simétricas que después nos servirán para particularizar algunos resultados.

Dada G gráfica si $T = (v_0, v_1, \ldots, v_n)$ es una v_0-v_n-trayectoria, en G^* podemos definir a T_d *trayectoria simétrica de* T, como la v_0-v_n-ditrayectoria tal que $T_d = (v_0, (v_0, v_1), v_1, (v_1, v_2), v_2, \ldots, v_{n-1}, (v_{n-1}, v_n), v_n)$, notemos que T_d es una orientación particular de T. Llamaremos también a T la *trayectoria subyacente de* T_d. Por lo tanto se cumple que:

Proposición 1.7.1 *Para cualesquiera $u, v \in V\langle G\rangle$ T es una u-v-trayectoria en G si y sólo si T_d es una u-v-ditrayectoria en G^*.*

Capítulo 2

Introducción a algoritmos

2.1. Complejidad de algoritmos

El término *algoritmo* es usado al referirnos a un conjunto de reglas e instrucciones bien definidas para obtener un resultado a partir de un conjunto específico de argumentos en un tiempo finito. El uso de algoritmos se ha incrementado ampliamente con la evolución de los sistemas de cómputo. Pero antes de empezar con los algoritmos debemos definir un tema central en este campo que es la complejidad.

La *complejidad* de un algoritmo mide la cantidad de esfuerzo computacional que se requiere para resolver un problema utilizando un algoritmo. Hay algunos parámetros que se consideran para medir la complejidad como son el número de pasos del proceso computacional, el tiempo que tarda en correr el algoritmo y la cantidad de espacio de almacenamiento requerida. Usualmente la complejidad de un algoritmo es una función del tamaño y la presentación de los argumentos, llamada función de complejidad.

El tipo de medida de la complejidad más usado es aquel llamado *complejidad del peor caso*. Al correr un algoritmo la mayoría de las veces hay argumentos con los cuales la respuesta del algoritmos se da rápidamente. Por ejemplo si nuestro problema es encontrar la inversa de una matriz es fácil ver que invertir una matriz diagonal representa una tarea muy distinta a invertir una matriz sin ceros. Por lo que al calcular la complejidad uno resuelve el peor caso posible en cuanto a tiempo se refiere, y con base en esto se da una medida a la complejidad del algoritmo.

Dadas dos funciones de complejidad f, g diremos que el *orden de f es menor o igual que el de g* si existe una constante $C \in \mathbb{R}$ y $n_0 \in \mathbb{Z}^+$ tales que $f(n) \leq Cg(n) \quad \forall n > n_0$. Si esto sucede, lo denotaremos como $f(n) = O(g(n))$. Es decir que no crece más rápido que g. Diremos también que f y

g tienen el *mismo orden* si $f(n) = O(g(n))$ y $g(n) = O(f(n))$.

Diremos entonces que un algoritmo es *eficiente* o *polinomial* si su complejidad f, con tamaño del argumento n, es una función polinomial en n, es decir si exista una función polinomial $h(n) = n^k$ tal que $f(n) = O(h(n))$. En cuyo caso llamaremos a un problema *computable* si existe un algoritmo eficiente que lo resuelva.

Dados dos algoritmos A, B, con funciones de complejidad f, g respectivamente, que resuelven el mismo problema, diremos que A es *mejor o más eficiente* que B si $f(n) = O(g(n))$ pero $g(n) \neq O(f(n))$.

Usaremos la notacion $A \leftarrow B$ para indicar que el valor de A es remplazado por el de B. El siguiente ejemplo nos dejará más claro como funciona y como se describe un algoritmo, en éste encontraremos el mínimo dentro de una lista de elementos de un orden parcial.

Algoritmo 2.1.1 *Algoritmo para encontrar el mínimo elemento.*

[Dado un listado $P(1), P(2), \ldots, P(n)$ de elementos de un orden parcial, encuentra el mínimo y lo despliega junto con su ubicación en la lista.]

1. *[La variable FIRST representa el "primer elemento" actual.]*
   $FIRST \leftarrow P(1)$.

2. *[Declara la variable p que nos dará la ubicación del elemento deseado.]*
   $p \leftarrow 1$.

3. *[Para cada elemento $P(k)$ de la lista, distinto de $P(1)$, si el elemento $P(k)$ es menor que FIRST, entonces FIRST es remplazado por $P(k)$ y p por k.]*
   For $k = 2$ to n
   Si $P(k) < FIRST$, entonces:

   (a) $FIRST \leftarrow P(k)$.

   (b) $p \leftarrow k$.

4. *Desplegar FIRST y p.*

Como este algoritmo hace $n-1$ comparaciones es claro que su complejidad es $n - 1 = O(n)$, por lo que es polinomial y eficiente.

2.2. Algoritmos de Búsqueda

En esta sección describiremos el funcionamiento y compararemos la eficiencia de dos algoritmos utilizados para encontrar, si aparece y de ser así dónde, una palabra dentro de una lista dada ordenada alfabéticamente. Usaremos la relación $a < b$ para indicar que la palabra a, alfabéticamente, se encuentra antes que la palabra b.

El primer algoritmo llamado *algoritmo secuencial de búsqueda* realiza la búsqueda de la manera más lógica, empezando desde el principio compara cada palabra de la lista con la buscada, si la encuentra detiene el proceso y avisa que la palabra existe en la lista, si recorre toda la lista y no la encuentra despliega que no fue encontrada. La formalización de este algoritmo es la siguiente:

Algoritmo 2.2.1 *Algoritmo Secuencial de Búsqueda.*

[Determina si la palabra dada LLAVE aparece en un listado $W(1), W(2), \ldots, W(n)$ *de palabras ordenado alfabéticamente, si lo hace despliega donde.]*

1. *[Si la k-ésima palabra de la lista es LLAVE su ubicación es desplegada y el algoritmo termina.]*

   for $k = 1$ *to* n

   (a) Si $W(k) = LLAVE$*, entonces*

   desplegar k *y detener*

2. *Desplegar "no se encontró"*

Es claro que el algoritmo necesitará hacer n comparaciones antes de terminar en el peor de los casos, es decir, cuando la palabra no se encuentre en la lista. Por lo tanto la complejidad de este algoritmo es $n = O(n)$.

Veamos ahora otro algoritmo que utiliza una técnica llamada "divide y vencerás" que, como su nombre lo dice, divide el problema en subproblemas pequeños que se resuelven más fácilmente. El algoritmo que veremos lleva el nombre de *Algoritmo Binario de Búsqueda*, para esto definimos lo que es el piso de un número real x, denotado por $\lfloor x \rfloor$, como el máximo entero menor que x. Con esto podemos describir el algoritmo formalmente como sigue:

Algoritmo 2.2.2 *Algoritmo Binario de Búsqueda.*

[Determina si la palabra dada LLAVE aparece en un listado $W(1), W(2), \ldots, W(n)$ *de palabras ordenado alfabéticamente, si lo hace despliega donde.]*

1. *[*j *y* k *representan la primera y la última palabra de la sublista siendo considerada. Inicialmente* j*=1 y* k*=n]*

   1.1 $j = 1$

   1.2 $k = n$

2. *[*$W(m)$ *es la palabra a la mitad entre* $W(j)$ *y* $W(k)$*.]*

   while $j<k$ *do*

   a) $m = \lfloor \frac{j+k}{2} \rfloor$

   b) [Si la m-ésima palabra es LLAVE despliega su ubicación y el algoritmo termina]
   Si $W(m) = LLAVE$*, entonces desplegar* k *y detener.*

   c) [Si LLAVE está antes alfabéticamente que $W(m)$*, entonces la nueva sublista se tomará entre* $W(j)$ *y* $W(m-1)$*]*
   Si LLAVE < $W(m)$*, entonces* $k = m - 1$*.*

   d) [Si LLAVE está después alfabéticamente que $W(m)$*, entonces la nueva sublista se tomará entre* $W(m+1)$ *y* $W(k)$*]*
   Si LLAVE > $W(m)$*, entonces* $j = m + 1$*.*

3. *Desplegar "no se encontró"*

Veamos como funciona este algoritmo en un ejemplo concreto. Supongamos que queremos determinar si la palabra $HOMBRE$ se encuentra dentro de la siguiente lista:

W(1) = ARBOL

W(2) = BURRO

W(3) = GATO

W(4) = HOMBRE

W(5) = KILO

W(6) = MAMA

W(7) = NACION

W(8) = PESCADO

W(9) = RADIO

W(10) = TIERRA

Usando 2.2.2 como algoritmo de búsqueda primero veremos si $W(5) = KILO$ es la palabra buscada $HOMBRE$.

Como no lo es, siendo que alfabéticamente $HOMBRE < KILO$, probamos ahora la palabra a la mitad de la sublista $W(1), W(2), W(3), W(4)$, que es $W(2) = BURRO$. Siendo que tampoco es la palabra buscada y $BURRO < HOMBRE$, comparamos la palabra a la mitad de la sublista $W(3), W(4)$, digase $W(3) = GATO$. Como $GATO \neq HOMBRE$ y $GATO < HOMBRE$, sólo nos queda probar $W(4)$ en cuyo caso habremos encontrado nuestra palabra.

Para determinar la complejidad de este algoritmo observemos que los pasos uno y tres sólo se realizan una vez por lo que su orden es constante $O(1)$. Ahora en el paso dos es claro que el mayor número de iteraciones se dará si la palabra no se encuentra en la lista.

Llamemos $B(n)$ al número de iteraciones del paso 2, como después de realizar este paso por primera vez habrá una lista con a lo más $\lfloor \frac{n}{2} \rfloor$ palabras a considerar, entonces

$$B(n) \leq 1 + B\left(\left\lfloor \frac{n}{2} \right\rfloor\right). \tag{2.1}$$

Demostraremos ahora por inducción fuerte que

$$B(n) \leq 1 + \log_2 n. \tag{2.2}$$

Si $n = 1$ es claro que la desigualdad se da.

Supongamos ahora que 2.2 se cumple para todo entero n con $1 \leq n \leq k$, probaremos entonces que

$$B(k+1) \leq 1 + \log_2(k+1).$$

Por 2.1.

$$B(k+1) \leq 1 + B\left(\left\lfloor \frac{k+1}{2} \right\rfloor\right).$$

Sin embargo por Hipótesis Inductiva

$$B\left(\left\lfloor \frac{k+1}{2} \right\rfloor\right) \leq 1 + \log_2\left(\left\lfloor \frac{k+1}{2} \right\rfloor\right) \leq 1 + \log_2\left(\frac{k+1}{2}\right),$$

Por lo tanto:

$$B(k+1) \leq 2 + \log_2(k+1) - \log_2 2 = 1 + \log_2(k+1),$$

Por consiguiente $B(n) = O(\log_2(n))$. Como en cada iteración del paso 2 sólo se hacen dos comparaciones y dos asignaciones, el orden de cada una es constante. Por lo que la complejidad resultante del paso 2 es $O(\log_2(n)) = O(\log(n))$.

2.3. Gráficas dentro de la computadora

Hemos descrito ya algunos algoritmos interesantes, pero aún no hemos logrado la conexión entre éstos y las gráficas. Para ello empezaremos por representar una gráfica en la computadora.

Existen distintos modelos de almacenamiento en un ordenador, para describirlos utilizaremos bases de datos que funcionan a partir de listas de elementos en las cuales cada renglón representa un número determinado de variables llamadas campos. Estas variables usualmente representan un identificador único, vértices, aristas o apuntadores al identificador de otro renglón.

Con esta información pasaremos a representar una gráfica a partir de una base de datos, debemos notar que la eficiencia de un algoritmo puede estar afectada por el modelo de almacenamiento que usemos. Una posible representación es la matricial en donde a una gráfica G con p, q y $V\langle G\rangle = \{v_1, v_2, \ldots, v_p\}$ se le asocia una $p \times p$ *matriz de adyacencias* $A = [a_{ij}]$ definida como sigue:

$$a_{ij} = \begin{cases} 1 & \text{si } v_i v_j \in E\langle G\rangle \\ 0 & \text{de lo contrario} \end{cases}$$

Es claro que A es una matriz simétrica en donde si no hay bucles la diagonal está compuesta de ceros. Como A es una $p \times p$ matriz requiere de p^2 entradas. Si G es una gráfica con un número limitado de aristas estaríamos ingresando en la base de datos un gran número de ceros que no aportan información pero ocupan espacio, es por eso, que se usan otros métodos de almacenamiento como lo es la *lista de adyacencias* descrita a continuación.

Sea G gráfica con $V\langle G\rangle = \{v_1, v_2, \ldots, v_p\}$, la *lista de adyacencias* asocia a cada $v_i \in V\langle G\rangle$ una lista de vértices adyacentes a v_i ordenados de manera numerica o alfabética. Podemos notar que esta forma de almacenamiento utiliza a lo más $p \times q$ entradas, lo que en el caso de una gráfica con pocas aristas nos reduce la cantidad de memoria usada en la base de datos.

Una digráfica D tiene asociada una representación en la computadora muy similar, si $V\langle D\rangle = \{v_1, v_2, \ldots, v_n\}$ definimos una $p \times p$ matriz $A = [a_{ij}]$ como sigue:

$$a_{ij} = \begin{cases} 1 & \text{si } (v_i, v_j) \in E\langle D\rangle \\ 0 & \text{de lo contrario.} \end{cases}$$

Capítulo 3

Árboles.

3.1. Propiedades de los árboles.

Definimos un *árbol* como una gráfica conexa sin ciclos, esto es que cada arista de un árbol represente un puente. Diremos que una gráfica (conexa o inconexa) es un *bosque* si es una gráfica acíclica. De modo que toda componente conexa de un bosque es un árbol. Llamaremos también árbol (bosque) generador de una gráfica G a una subgráfica generadora que sea un árbol (bosque). Veremos ahora una propiedad importante que tienen estas gráficas:

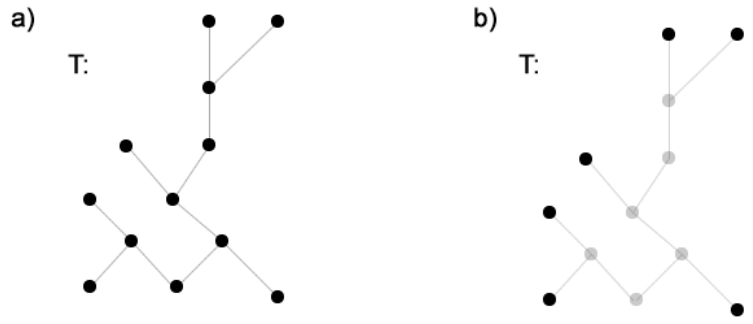


Figura 3.1:
a) Ejemplo de un árbol. b) En este esquema podemos ver resaltados los vértices terminales del mismo árbol T.

Teorema 3.1.1 *Un árbol de orden p tiene tamaño* $p-1$

Demostración. Por inducción sobre p. Como K_1 es el único árbol con $p=1$ el resultado se cumple para la base de la inducción.

Sea T árbol con $p \geq 2$ y supongamos que el resultado se cumple para toda $k < p$. Sea $e \in E\langle T\rangle$, por lo que sabemos que e es un puente, de aquí que $T-e$ tiene exactamente dos componentes conexas, digamos T_1, T_2 árboles con p_1, p_2 y q_1, q_2 como sus ordenes y tamaños.
Como $p_i < p$ por hipótesis inductiva tenemos que $q_i = p_i - 1$ con $i \in \{1,2\}$. Sabemos también que $p = p_1 + p_2$ y $q = q_1 + q_2 + 1$ por lo tanto:

$$q = (p_1 - 1) + (p_2 - 1) + 1 = p_1 + p_2 - 1 = p - 1$$

Por inducción tenemos que el tamaño de un árbol es su orden menos uno. □

Otro teorema útil que nos muestra propiedades interesantes de los árboles es el siguiente:

Teorema 3.1.2 *Todo árbol no trivial contiene al menos dos vértices terminales.*

Demostración. Sea T árbol con orden p, tamaño q y $V\langle G\rangle = \{v_1, v_2, \ldots, v_p\}$ ordenados de tal forma que $\delta(v_i) \leq \delta(v_j)$ si $i \leq j$. Ahora si suponemos que no existen 2 vértices con grado uno entonces $\forall i \geq 2$ se cumple que $\delta(v_i) \geq 2$, y $\delta(v_1) \geq 1$, por lo tanto:

$$\sum_{i=0}^{p} \delta(v_i) \geq 1 + 2(p-1) = 2p - 1 \tag{3.1}$$

Sin embargo como por 3.1.1 y 1.2.1 sabemos que

$$\sum_{v \in V\langle G\rangle} \delta(v) = 2q = 2(p-1) = 2p - 2 \tag{3.2}$$

Por lo tanto:

$$2p - 2 = \sum_{v \in V\langle G\rangle} \delta(v) = \sum_{i=0}^{p} \delta(v_i) \geq 2p - 1 \tag{3.3}$$

Lo cual es una contradicción con 3.1 que viene de suponer que T no contenía al menos 2 vértices terminales. □

Sabemos que en una gráfica conexa dados dos vértices existe una trayectoria que los une, sin embargo en árboles podemos decir un poco más:

Teorema 3.1.3 *Dado T árbol, si $u, v \in V\langle T\rangle$, entonces T contiene una única u-v-trayectoria*

Demostración. Sea T árbol y sean $u, v \in V\langle T\rangle$. Supongamos que existen P y Q dos u-v-trayectorias distintas, por lo tanto existe un vértice $x \in V\langle P\rangle \cap V\langle Q\rangle$ tal que el vértice que sigue después de x en P sea distinto del que le sigue en Q, y sea y el primer vértice en P, después de x, que también esté en Q.

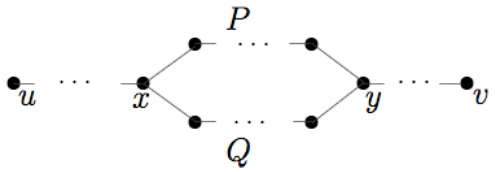


Figura 3.2:
En esta figura podemos observar las 2 u-v-trayectorias y el ciclo que se forma al suponer que son distintas.

De aquí que podamos construir el siguiente ciclo $C = (x, P, y, Q^{-1}, x)$ lo cual es una contradicción que viene de suponer que existían dos u-v-trayectorias distintas en T. $\square$

3.2. Arboles arraigados.

Extendemos el concepto de árbol a árbol dirigido definiéndolo como una digráfica cuya gráfica subyacente es un árbol. Diremos que T es un *árbol arraigado* si existe un vértice $r \in V\langle T\rangle$, llamado raíz, tal que $\forall v \in V\langle T\rangle$ existe una r-v-trayectoria dirigida. Una vez encontrada la raíz de un árbol podemos definir el *nivel de un vértice* v como la longitud de la única r-v-trayectoria dirigida. Definiremos también el *peso* de un árbol arraigado como el máximo nivel de un vértice en T.

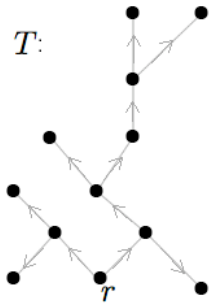


Figura 3.3:
Ejemplo de árbol arraigado en r.

El siguiente teorema nos permite caracterizar los árboles arraigados:

Teorema 3.2.1 *Un árbol dirigido es arraigado si y sólo si T contiene un vértice r con $\delta^-(r) = 0$ y $\delta^-(v) = 1 \quad \forall v \in V\langle T\rangle$ distinto de r.*

Demostración. Sea T árbol.

Para la primera implicación supongamos que T es arraigado con raíz r, por lo tanto $\delta^-(r) = 0$ ya que de lo contrario si existiera $v \in V\langle T\rangle$ tal que $(v, r) \in E\langle T\rangle$, como hay una r-v-trayectoria dirigida en T podríamos construir un ciclo dirigido $T \cup (v, r)$, lo cual sería una contradicción. Ahora dado $u \in V\langle T\rangle$ con $u \neq r$ supongamos que u está en el nivel i y por lo tanto es adyacente desde exactamente un vértice en el nivel $i - 1$, como consecuencia $\delta^-(u) = 1$.

Para el reciproco supongamos que existe $r \in V\langle T\rangle$ tal que $\delta^-(r) = 0$ y $\forall v \in V\langle T\rangle$ si $v \neq r$, entonces $\delta^-(v) = 1$. Sea $u_1 \in V\langle T\rangle$ con $u_1 \neq r$, como $\delta^-(u_1) = 1$ entonces existe un vértice adyacente a u_1 digamos u_2. Si $u_2 = r$ tenemos una u_1-r-trayectoria, de lo contrario podemos repetir este proceso cuantas veces sea posible, notemos que como no hay ciclos en T este proceso

debe ser finito, así que llegaremos a u_n con $\delta^-(u_n) = 0$, por consiguiente $u_n = r$, con lo que encontramos una u_1-r-trayectoria y por lo tanto T es arraigado. □

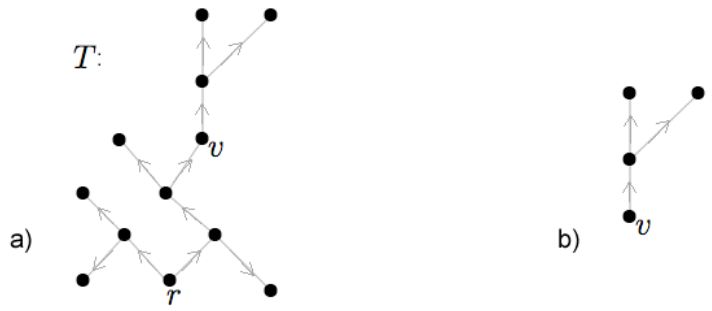


Figura 3.4:
El ejemplo (a) representa a T un árbol arraigado en r y (b) es un subárbol máximo de T arraigado a v.

Dado T árbol arraigado y $u, v \in V\langle T\rangle$ vértices adyacentes tales que u está en el nivel anterior a v diremos que u es el *nodo anterior* de v (o "parent node") o que v es un *nodo posterior* de u (o "child node"). Diremos también que un vértice $w \in V\langle T\rangle$ es descendiente de otro vértice $x \in V\langle T\rangle$ (o que x es antecesor de w) si existe una x-w-trayectoria en T. Al subárbol inducido por v y todos sus descendientes se le conoce como *subárbol máximo de T arraigado a v*. Un vértice $v \in V\langle T\rangle$ sin nodos posteriores será llamado una *hoja* del árbol T mientras que todos los demás serán llamados *vértices internos*.

Un árbol arraigado T es llamado *m-ario* si todo vértice de T tiene a lo más m nodos posteriores. En un árbol 2-ario o binario cada vértice tiene a lo más 2 nodos posteriores, un posterior derecho y un posterior izquierdo. Llamamos a un árbol arraigado T *m-ario completo* si todo vértice de T tiene cero o m nodos posteriores, de igual forma se define un árbol *binario completo*.

Teorema 3.2.2 *Un árbol m-ario completo con i vértices internos tiene orden $mi + 1$.*

Demostración. Sea T un árbol m-ario completo de orden p con i vértices internos. Si contamos los vértices de T, sabemos que por cada vértice interno de T tenemos m nodos posteriores y como cada vértice tiene un nodo anterior único hasta aquí hemos contados mi vértices, sin embargo como la raíz no es

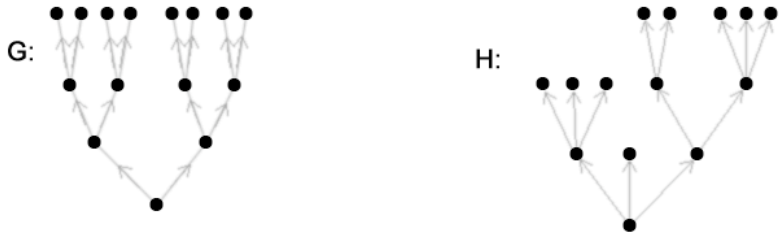


Figura 3.5:
En esta figura G representa un árbol binario completo y H un árbol 3-ario.

un nodo posterior de nadie nos falta contarla por lo tanto $p = mi + 1$. □

Corolario 3.2.3 *Cada árbol binario completo con i vértices internos tiene $i + 1$ hojas.*

Demostración. Sea T un árbol binario completo con i vértices internos. Por el teorema anterior sabemos que $p = 2i + 1$, por lo tanto como T tiene exactamente i vértices internos entonces el resto, $i + 1$ vértices, tienen que ser hojas. □

Los siguientes resultados nos dan información entre la relación orden y peso de un árbol binario.

Lema 3.2.4 *Se cumple que para toda $n \in \mathbb{N}$:*

$$\sum_{k=0}^{n} 2^k = 1 + 2 + 2^2 + \ldots + 2^n = 2^{n+1} - 1$$

Demostración. Por inducción sobre n. Para $n = 1$ es claro que

$$\sum_{k=0}^{1} 2^k = 1 + 2 = 2^2 - 1 = 3 = 2^{1+1} - 1$$

Supongamos el resultado válido para n, y probemos para $n + 1$. Desarrollemos ahora la siguiente expresión:

$$\sum_{k=0}^{n+1} 2^k = 1 + 2 + \ldots + 2^n + 2^{n+1} =^{HI} 2^{n+1} - 1 + 2^{n+1} = 2(2^{n+1}) - 1 = 2^{n+2} - 1$$

Con lo que queda demostrada la proposición. □

Teorema 3.2.5 *Si T es un árbol binario de peso h y orden p, entonces*

$$h+1 \leq p \leq 2^{h+1}-1$$

Demostración. Definamos p_k como el número de vértices de T en el nivel k, con $0 \leq k \leq h$. Por lo tanto $\sum_{k=0}^{h} p_k = p$. Veamos por inducción sobre el peso del árbol que $p_k \leq 2^k$ si $0 \leq k \leq h$. Para $h = 1$ es claro que la raíz tiene a lo más dos nodos posteriores, por lo tanto $p_1 \leq 2$. Supongamos el resultado válido para h, como $p_{h+1} \leq 2p_h$ pues cada vértice tiene a lo más 2 vértices posteriores y $p_h = 2^h$ entonces $p_{h+1} \leq 2^{h+1}$.

De aquí que usando 3.2.4:

$$h+1 = \sum_{k=0}^{h} 1 \leq \sum_{k=0}^{h} p_k = p \leq \sum_{k=0}^{h} 2^k = 2^{h+1}-1.$$

Con lo que queda demostrado el teorema. □

Diremos que un árbol arraigado T está *balanceado* si cada hoja de T esta en el nivel h o $h-1$.

Corolario 3.2.6 *Si T es árbol binario de peso h y orden p, entonces*

$$h \geq \lceil \log_2((p+1)/2) \rceil$$

La igualdad se da si T está balanceado.

Demostración. Por el Teorema anterior $p \leq 2^{h+1}-1$ ó $2^h \geq (p+1)/2$ por lo tanto

$$h \geq \log_2((p+1)/2)$$

De aquí que, como h un entero, $h \geq \lceil \log_2((p+1)/2) \rceil$.

Ahora si T está balanceado entonces

$$p > 1 + 2 + 2^2 + \ldots + 2^{h-1} = 2^h - 1$$

por lo tanto $2^h - 1 < p \leq 2^{h+1} - 1$ o lo que es lo mismo

$$2^{h-1} < \frac{p+1}{2} \leq 2^h$$

Sacando logaritmos tenemos que

$$h - 1 < \log_2\left(\frac{p+1}{2}\right) \leq h$$

Por lo tanto $h = \lceil \log_2((p+1)/2) \rceil$ □

3.3. Búsqueda de Profundidad Inicial

La *Búsqueda de Profundidad Inicial* es un método sistemático para visitar cada vértice de una gráfica mediante un algoritmo eficiente. La primera aplicación que veremos de este método será la de encontrar sub-árboles o sub-bosques generadores dentro de una gráfica arbitraria.

Una descripción inicial de este método es la siguiente. Supongamos que tenemos una gráfica conexa, lo primero que haremos será tomar como nuestro vértice actual a un véritce arbitrario dentro de ésta, de aquí nos fijaremos en cada vértice adyacente que tenga, no visitado por el algoritmo previamente, elegiremos uno que será nuestro nuevo vértice actual y repetiremos el proceso hasta que todos los vértices adyacentes a nuestro vértice actual hayan sido ya visitados por el algoritmo, en cuyo caso regresaremos por el camino seguido vértice a vértice y repetiremos el proceso en cada ocasión.

Cuando todos los vértices de la gráfica hayan sido visitados, el camino que seguimos durante este proceso nos definirá un sub-árbol generador, en caso de que la gráfica no sea conexa repetiremos el proceso para cada componente conexa obteniendo así un sub-bosque generador.

Definiremos para cada $v \in V\langle G\rangle$ $ipi(v) = n$ si y sólo si v es el n-ésimo vértice visitado por primera vez en el algoritmo. Este último puede ser expresado formalmente de la siguiente manera:

Algoritmo 3.3.1 *Algoritmo de Búsqueda de Profundidad Inicial.*

[Para conducir una búsqueda de profundidad inicial dentro de una gráfica arbitraria G]

1. *[Inicialmente a todos los vértices se les da un indice de búsqueda de 0]* $ipi(v) \leftarrow 0$ *para cada* $v \in V\langle G\rangle$.

2. *[La variable i es definida, y será asignada al i-esimo vértice visitado en el proceso]*
   $i \leftarrow 1$.

3. *[El conjunto S es definido y será el que guarde todos las aristas visitadas dentro del proceso]*
   $S \leftarrow \phi$.

4. *[Si no han sido visitados todos los vértices de G se define una nueva raíz desde la cual se iniciará el proceso de búsqueda]*
   Si $ipi(v) \neq 0 \quad \forall v \in V\langle G\rangle$, *entonces detener y desplegar S;*
   de lo contrario definir r como el primer vértice tal que $ipi(v) = 0$ *y* $w \leftarrow r$.

5. $ipi(w) \leftarrow i$.

6. $i \leftarrow i + 1$.

7. *[La búsqueda es llevada a cabo para un vértice no visitado anteriormente]*

   7.1 *Si* $ipi(v) = 0$ *para algun v adyacente a w, entonces continuar al siguiente paso, de lo contrario, ir al paso 7.4.*

   7.2 $S \leftarrow S \cup \{(w, v)\}$ *y* $PARENT(v) \leftarrow w$

   7.3 $w \leftarrow v$ *y regresar al paso 5.*

   7.4 *Si* $w \neq r$, *entonces* $w \leftarrow PARENT(w)$ *y regresar al paso 7.1; de lo contrario, regresar al paso 4.*

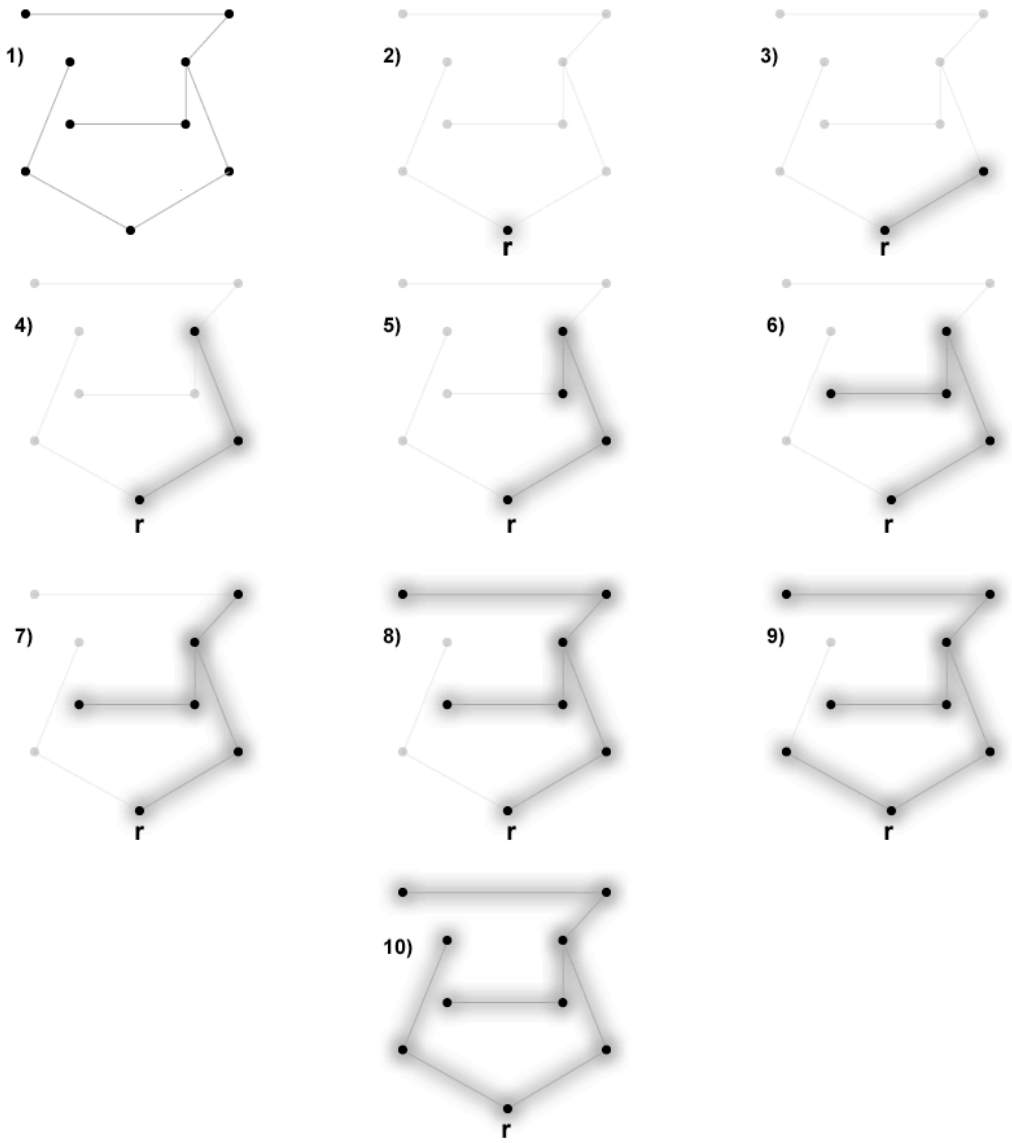


Figura 3.6:
En esta figura podemos observar el actuar de un algoritmo de Búsqueda de Profundidad Inicial, comenzamos en el paso 2) por definir un vértice r como la raíz y el vértice activo para, paso siguiente, empezar a recorrer la gráfica buscando vértices adyacentes al activo con $ipi(v) = 0$. Podemos observar que en el cambio del paso 6) al 7), como el algoritmo no encuentra como seguir avanzando regresa por su camino hasta encontrar un nuevo vértice con $ipi(v) = 0$, lo mismo ocurre en el cambio del paso 8) al 9). Todo para que terminemos con un sub-árbol generador.

Si G es una (p, q) gráfica, entonces la complejidad del algoritmo de búsqueda de profundidad inicial es $O(max\{p, q\})$, ya que, notemos primero que para cada vértice w la igualdad $ipi(v) = 0$ es verificada exactamente una vez para cada vértice v en la lista de adyacencias de w. Además los pasos 7.2 y 7.3 son ejecutados a lo más $\delta(w)$ veces para cada vértice w. Como $\sum_{w \in V\langle G\rangle} \delta(w) = 2q$ tenemos que la complejidad de los pasos 7.1-7.3 es $O(q)$. Como también regresamos pasos a lo más una vez por cada $w \in V\langle G\rangle$, el paso 7.4 tiene complejidad $O(p)$. Por lo que la complejidad del paso 7 es $O(max\{p, q\})$. Pasos 1, 2, 3, 5 y 6 tiene complejidad constante. El 4 requiere a lo más de p iteraciones. Por todo lo anterior tenemos que la complejidad del algoritmo completo es $O(max\{p, q\})$.

Dado T un árbol de búsqueda de profundidad inicial en una gráfica G, definimos una *arista trasera* como una arista de G que no pertenece a T. El siguiente resultado muestra algunas propiedades de este término.

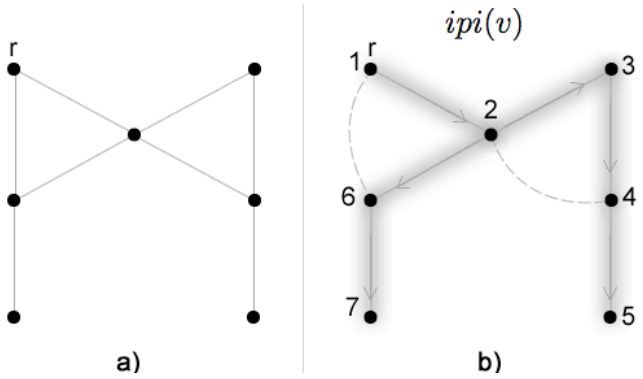


Figura 3.7:
En (b) podemos observar el árbol arraigado en r obtenido a partir del algoritmo de BPI en la gráfica de la figura (a). En punteado tenemos ejemplos de aristas traseras y al lado de cada vértice se encuentra el valor de su *ipi*

Teorema 3.3.2 *Cada arista trasera e de G une a un antecesor y a un descendiente. En particular, si $e = uv$ y u es visitado antes de v en la búsqueda de profundidad inicial de G, entonces v es descendiente de u.*

Demostración. Por hipótesis, v es visitado por primera vez después de la primera visita a u. La subgráfica del árbol de búsqueda de profundidad inicial inducida por u y todos los vértices de G visitados entre la primera y la última visita a u es un árbol T, arraigado en u. Por lo tanto, cada vértice

de T distinto de u es un descendiente de u. Como cada vértice adyacente a u y visitado después de u es vértice de T, entonces el vértice v está en T y por lo tanto es descendiente de u. $\square$

3.4. Búsqueda de Profundidad Inicial: Una herramienta para encontrar bloques.

Con la ayuda de la búsqueda de profundidad inicial se puede determinar el número de vértices de corte de una gráfica dada. Para esto introduciremos algunos conceptos nuevos. Dada G gráfica y $v \in V\langle G\rangle$, una *rama* de G en v es una sugráfica conexa máxima H de G que contenga a v pero no como un vértice de corte. Sin importar si v es o no vértice de corte en G podemos observar que el número de ramas de G en v es igual al número de bloques que contengan a v.

Veamos ahora un par de resultados que nos permitirán caracterizar los vértices de corte de una gráfica conexa, sin embargo como un vértice es de corte en una gráfica G si y sólo si lo es en alguna componente conexa de G, basta demostrar el resultado para gráficas conexas.

Teorema 3.4.1 *Sea T un árbol de búsqueda de profundidad inicial en una gráfica G, y r la raíz de T. Entonces r es vértice de corte de G si y sólo si r tiene al menos dos nodos posteriores en T.*

Demostración. Sea T un árbol de búsqueda de profundidad inicial en una gráfica G.

Para el primer condicional sea r un vértice de corte de G. Por lo tanto r pertenece a al menos 2 bloques y por ende a 2 ramas de G. Sea x el primer vértice visitado después de r durante la búsqueda de profundidad inicial de G. De aquí que x sea un nodo posterior de r en T y pertenece a una rama, digamos B_1 de G en r. Sea $B_2 \neq B_1$ otra rama de G en r. Por lo tanto $B_1 \cap B_2 = \{r\}$ ya que dos bloques se intersectan a lo más en un vértice. Como $x \in B_1$ antes de visitar algún vértice de B_2 la búsqueda de profundidad inicial debe, necesariamente, volver a pasar por r. De aquí podemos deducir que r tendrá al menos un segundo nodo posterior en T al visitar el primer vértice de B_2.

Para la segunda implicación supongamos que r tiene al menos dos nodos posteriores en T. Sea x el primer vértice visitado después de r en la búsqueda de profundidad inicial de G. Denotemos a T_1 como el máximo subárbol de

T tal que T_1 sea arraigado en x. Es fácil observar que $V\langle T_1\rangle$ consiste de x y todos los vértices descendientes de x.

Supongamos que existe $uw \in E\langle G\rangle$ tal que $u \in V\langle T_1\rangle$ y $w \notin (V\langle T_1\rangle \cup \{r\})$. De aquí podemos deducir que w es visitado después de u en la búsqueda de profundidad inicial pues no es descendiente de x y todos los descendientes de x son visitados antes de regresar a r. Como $uw \notin E\langle T_1\rangle$ entonces uw es una arista trasera y gracias a 3.3.2 sabemos que w es descendiente de u lo cual es una contradicción pues todo descendiente de u pertenece a T_1, contradicción que viene de suponer la existencia de uw. Al no existir tal arista toda trayectoria en G que conecte a un vértice de $V\langle T_1\rangle$ con vértices de $V\langle T\rangle - (V\langle T_1\rangle \cup \{r\})$ tiene, necesariamente, que pasar por r. De aquí que r sea vértice de corte de G. $\square$

Para completar la caracterización de vértices de corte necesitamos introducir un nuevo concepto. Sea F un bosque de búsqueda de profundidad inicial de una gráfica G y supongamos que v es un vértice que pertenece a alguna componente conexa de F digamos T. Definimos *el punto bajo de* v $\lambda(v)$ como el mínimo $ipi(u)$ de un vértice $u \in V\langle T\rangle$ tal que u es alcanzable desde v por una v-u-trayectoria (dirigida) compuesta de aristas (flechas) de T seguida por a lo más una arista trasera de G. Como la v-u-trayectoria puede ser trivial, tenemos que $\lambda(v) \leq ipi(v) \ \ \forall v \in V\langle G\rangle$.

De hecho de la definición es fácil hacer la siguiente observación:

Proposición 3.4.2 *$\lambda(v) < ipi(v)$ si y sólo si existe una arista trasera de un descendiente a un antecesor de v.*

Teorema 3.4.3 *Sea T un árbol de búsqueda de profundidad inicial en una gráfica conexa G, y sea u un vértice de G que no sea raíz de T. Entonces u es vértice de corte de G si y sólo si u tiene un nodo posterior $v \in V\langle T\rangle$ tal que $\lambda(v) \geq ipi(u)$.*

Demostración. Sea G gráfica y T árbol de búsqueda de profundidad inicial en G.

Para la primera implicación sea $u \in V\langle G\rangle$ vértice de corte de G que no sea raíz de T. Sea $r \neq u$ la raíz de T. Es claro que r pertenece a alguna rama, digamos B_1, de G en u. Ya que u es vértice de corte, podemos observar que el algoritmo de búsqueda de profundidad inicial debe visitar u cuando todavía quedan vértices no visitados dentro de G. Por lo tanto podemos asumir que existe $v \in V\langle G\rangle$ adyacente a u que no pertenezca a B_1, ya que si todo vértice adyacente a u estuviera en B_1, todos pertenecerían al mismo bloque y u no sería vértice de corte, por lo tanto podemos asumir entonces que $v \in B_2$, otra rama de G.

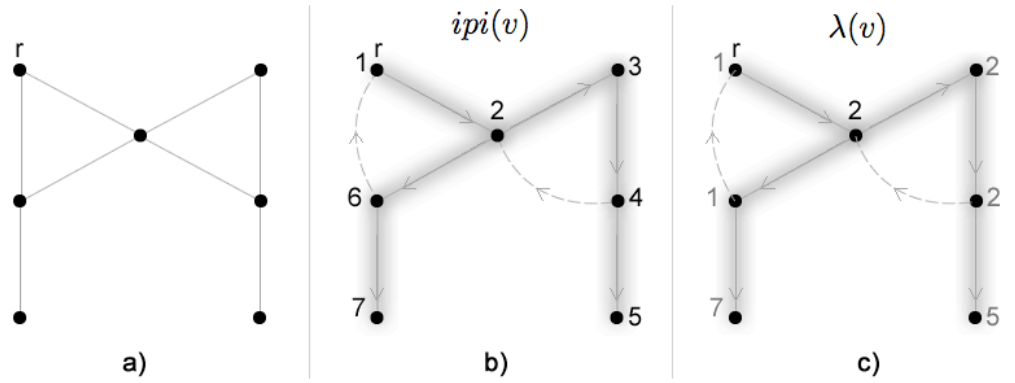


Figura 3.8:
(a) nos muestra una gráfica cualquiera. (b) es un árbol arraigado generado por el algoritmo de profundidad inicial en la misma gráfica, en punteado tenemos las aristas traseras y podemos observar también para cada vértices v, el valor de $ipi(v)$ a su lado. (c) nos muestra el mismo árbol, pero en cada vértice tenemos el valor de $\lambda(v)$, notemos que utilizando las aristas traseras hay varios vértices desde los cuales se pueden alcanzar vértices con menor ipi y por lo tanto $\lambda(v) < ipi(v)$.

Por definición de rama sabemos que ningún vértice de B_2 distinto de u es adyacente a alguno de $V\langle G - B_2\rangle$, entonces toda arista trasera de un descendiente de v une a dos vértices de B_2 y todo vértice de B_2 o es u o es visitado después de u, por lo tanto $\lambda(v) \geq ipi(u)$. Por construcción del algoritmo de búsqueda de profundidad inicial uv es una arista de T, por consiguiente v es un nodo posterior de u.

Para el reciproco supongamos que u no es la raíz de T y que u tiene un nodo posterior v en T tal que $\lambda(v) \geq ipi(u)$. Sea r la raíz de T, y sea P la única r-u-trayectoria en T. Sea $S_1 = V\langle P\rangle - \{u\}$. Sea S_2 el conjunto de vértices en el subárbol máximo de T arraigado en v. Como todos los antecesores en T de vértices de S_2 están en $H = S_1 \cup S_2 \cup \{u\}$, por teorema 3.3.2, como todas las aristas traseras unen antecesores con descendientes, entonces sabemos que todas las aristas de vértices de S_2 inciden en vértices de H.

Supongamos que G contiene una arista x_1x_2, tal que $x_i \in S_i$ $(i = 1, 2)$. Por lo tanto x_1x_2 es una arista trasera tal que $ipi(x_1) < ipi(u)$, de esta forma al tomar la v-x_2-trayectoria en T seguida de x_2x_1 tenemos que $\lambda(v) \leq ipi(x_1) < ipi(u)$. Lo cual es una contradicción que viene de suponer que existe alguna arista entre vértices de S_1 y S_2, por lo tanto toda S_1-S_2-trayectoria en G pasa por u, lo cual implica por 1.5.1 que u sea vértice de

corte de G. $\square$

Dentro del manejo de gráficas hay ocasiones donde nos es necesario conocer los puentes que ésta contenga, para lo cual el algoritmo de búsqueda de profundidad inicial también nos es útil, como veremos en el siguiente teorema de caracterización.

Teorema 3.4.4 *Sea T un árbol de búsqueda de profundidad inicial en una gráfica G. Sea $uv \in E\langle G\rangle$ tal que $ipi(u) < ipi(v)$, entonces uv es puente de G si y sólo si uv es una arista en T y $\lambda(v) > ipi(u)$.*

Demostración. Sea T un árbol de búsqueda de profundidad inicial en una gráfica G dada.

Para el primer condicional sea uv un puente de G tal que $ipi(u) < ipi(v)$, como la única u-v-trayectoria que existe en G es la arista uv, entonces necesariamente uv pertenece a T, y v es un nodo posterior de u. Ahora si $\lambda(v) \leq ipi(u) < ipi(v)$, entonces por 3.4.2 existiría una arista trasera uniendo a v, o un descendiente de v con un antecesor de u o u misma. En cuyo caso uv pertenecería a un ciclo, lo cual es una contradicción que viene de suponer que $\lambda(v) \leq ipi(u)$. Por lo tanto $\lambda(v) > ipi(u)$.

El segundo condicional se probará por contrapuesta. Supongamos que uv es una arista de G que, o no pertenece a T, o $\lambda(v) \leq ipi(u)$. Si $uv \notin E\langle T\rangle$, entonces es una arista trasera y por lo tanto pertenece a un ciclo de G. Ahora si uv es una arista de T tal que $\lambda(v) \leq ipi(u) < ipi(v)$, entonces por 3.4.2 existiría una arista trasera uniendo a v, o un descendiente de v con un antecesor de u o u misma. Por lo tanto uv pertenecería también a un ciclo de G. De lo cual podemos concluir que en ambos casos uv es cíclica y en consecuencia por 1.5.2 no es un puente de G. $\square$

Utilizando estos teoremas tenemos las herramientas suficientes para generar un algoritmo que encuentre bloques dentro de una gráfica G. Este empezará en algún vértice arbitrario r de la gráfica. Notemos que no importando como sea la gráfica ésta tendrá por lo menos 2 bloques terminales, de aquí que una vez corriendo el algoritmo de BPI, iniciado en r, en algún momento entrará a uno de estos bloques terminales y lo recorrerá por completo.

Cuando el algoritmo empiece a recular sobre el mismo camino que recorrió, volverá a analizar al único vértice de corte del bloque terminal en cuestión, digamos u, y si el algoritmo hubiese determinado $\lambda(v)$ para los vértices vistados, entonces podríamos comparar el $ipi(u)$ con el valor de $\lambda(v)$ de sus nodos posteriores de manera que como u es de corte, existirá un vértice tal que $\lambda(v) \geq ipi(u)$, con lo que u quedará identificado como vértice de corte de la gráfica.

Si usamos una pila en el cual vayamos colocando los vértices en el orden en que se van analizando, tendríamos que después de u estarán en la pila todos los vértices del bloque terminal en cuestión, si los removemos en el momento que u sea identificado, tendremos caracterizado al bloque.

El algoritmo utilizará entonces dos arreglos, $STACK$ y $SCAN$, de longitudes p y q respectivamente. Inicialmente $STACK(v)$ será falso para todos los vértices de G, es decir, indicamos que ningún vértice ha sido puesto en la pila aún, y $SCAN(e)$ será también falso para todas las aristas e de G, es decir, que ninguna arista ha sido escaneada aún.

Conforme los vértices sean puestos en la pila se actualizará el valor de $STACK(v)$, y siempre que una arista sea analizada, entonces $SCAN(e)$ será asignado como verdadero.

Vayamos ahora con la definición formal del algoritmo.

Algoritmo 3.4.5 *Algoritmo para determinar los bloques de una gráfica G.*

[Para determinar los bloques de una gráfica G con $V\langle G\rangle = \{v_1, v_2, \ldots, v_p\}$ y $E\langle G\rangle = \{e_1, e_2, \ldots, e_q\}$, con G representada por su lista de adyacencias.]

1. *[Definir los arreglos STACK, SCAN y PARENT , definir la "pila" S, la variable i que es asignada al i-ésimo vértice visitado, y el indice de profundidad inicial de cada vértice]*

   1.1 Para $j = 1, 2, \ldots, p$, definimos $STACK(v_j) \leftarrow falso$, $PARENT(v_j) \leftarrow \phi$, y $ipi(v_j) = 0$.

   1.2 Para $j = 1, 2, \ldots, q$, sea $SCAN(e_j) \leftarrow falso$.

   1.3 $S \leftarrow \phi$.

   1.4 $i \leftarrow 0$.

2. *[Definir el vértice activo v y la raíz r, al iniciar es claro que j será igual a 1]*
   Sea j el mínimo entero positivo tal que $ipi(v_j) = 0$, y sea $v \leftarrow v_j$ y $r \leftarrow v_j$. Si no existe tal j, entonces detener ya que en ese caso todos los bloques de G habrán quedado ya determinados.

3. *[En este paso la variable i es actualizada y se asigna al i-ésimo vértice visitado. Además el punto bajo de este vértice es iniciado con su máximo valor posible y es añadido a la pila]*

   3.1 $i \leftarrow i + 1$.

   3.2 $ipi(v) \leftarrow i$.

3.3 $\lambda(v) \leftarrow i$.

3.4 Añadir v *al tope de la pila* S *y asignar* $STACK(v) \leftarrow verdadero$.

4. *[Este paso determina si hay aristas no escaneadas incidentes con el vértice actualmente activo.]*
Si no hay aristas no escaneadas incidentes desde v*, entonces ir al paso 7, de otra manera sea* u *el primer vértice adyacente a* v *en la lista de adyacencias de* G *tal que* $SCAN(vu) = falso$*, es decir que* vu *no haya sido escaneada.*

5. *[Este paso actualiza el arreglo* $SCAN$ *al indicar que la arista hallada en el paso anterior esta ya escaneada. Además, si* u *no ha sido visitado previamente, entonces* v *se vuelve ahora el nodo anterior de* u*, y* u *es designado como el nuevo vértice activo]*

   5.1 $SCAN(vu) \leftarrow verdadero$.

   5.2 Si $ipi(u) = 0$*, entonces continuar, de otra forma, ir al paso 6.*

   5.3 $PARENT(u) \leftarrow v$.

   5.4 $v \leftarrow u$ *y regresar al paso 3.*

6. *[Si el paso 5 determina que el vértice* u *encontrado en el paso 4 ya ha sido etiquetado, entonces este paso actualiza el punto bajo de* v *ya que* vu *es una arista trasera.]*
Si $ipi(u) \neq 0$*, entonces* $\lambda(v) = min\{\lambda(v), ipi(u)\}$ *y regresar al paso 4.*

7. *[Este paso determina si el nodo anterior de* v *es la raíz de la componente en consideración.]*
Si $PARENT(v) = r$*, entonces ir al paso 11, de otra manera, continuar.*

8. *[Si el nodo anterior de* v *no es la raíz, entonces este paso determina si el punto bajo del nodo anterior de* v *debe ser revaluado.]*
Si $\lambda(v) < ipi(PARENT(v))$*, entonces*
$\lambda(PARENT(v)) \leftarrow min\{\lambda(v), \lambda(PARENT(v))\}$ *e ir al paso 10, de otra manera continuar.*

9. *[Un punto de corte es descubierto y un bloque queda determinado.]*
$PARENT(v)$ *es un vértice de corte de* G*. Todos los vértices en* S *hasta* v *son removidos de* S*, ellos, junto con* $PARENT(v)$ *inducen un bloque en* G*.*

10. *[La búsqueda retrocede hasta el nodo anterior de* v*.]*
$v \leftarrow PARENT(v)$ *y regresar al paso 4.*

11. *[El bloque que contiene a la raíz es determinado]*
Todos los vértices en S incluyendo a v son removidos de S, juntos con r, inducen un bloque en G.

12. *[Este paso determina si hay aristas no escaneadas incidentes con la raíz]*
Si no hay aristas no escaneadas incidentes con r, entonces regresar al paso 2; de otra forma sea $v \leftarrow r$ y regresar al paso 4.

Analicemos ahora la complejidad de este algoritmo. Sea G gráfica en la cual se llevo a cabo el algoritmo 3.4.5. Notemos primero que la complejidad del paso 1 es $O(max(p, q))$ ya que son asignaciones a todas las aristas y vértices, el paso 2 tiene complejidad constante, y se lleva a cabo una vez por cada componente conexa de G. El paso 3 tiene cuatro acciones, cada una de complejidad constante, y en conjunto se lleva a cabo 2 veces por cada vértice de corte y una más por cada otro que no lo sea, por lo tanto su complejidad es $O(p)$.

El paso 4 escanéa cada arista, incidente en cada vértice de G, exactamente una vez, y de no haber aristas salta al paso 7 a lo más una vez por cada vértice, por lo que su complejidad es $O(max(p, q))$. Como cada acción del paso 5 es ejecutada a lo más una vez para cada arista de G, entonces tiene complejidad $O(q)$. El paso 6 actualiza el punto bajo de cada vértice a lo más una vez por cada arista incidente en ese vértice, por lo tanto es proporcional a la suma de los grados de todos los vértices, es decir, por 1.2.1 $2q$, de donde la complejidad queda determinada como $O(q)$.

El paso 7 tiene complejidad $O(p)$, y como del paso 8 al 11 se llevan a cabo cuando el algoritmo regresa sobre su camino, y éste regresa a lo más una vez por cada vértice, entonces la complejidad de estos pasos es $O(p)$. Finalmente el paso 12 revisa las aristas no escaneadas incidentes con r o hace una asignación a v, por lo tanto su complejidad es $O(max(p, q))$.

En consecuencia la complejidad total del algoritmo 3.4.5 es $O(max(p, q))$.

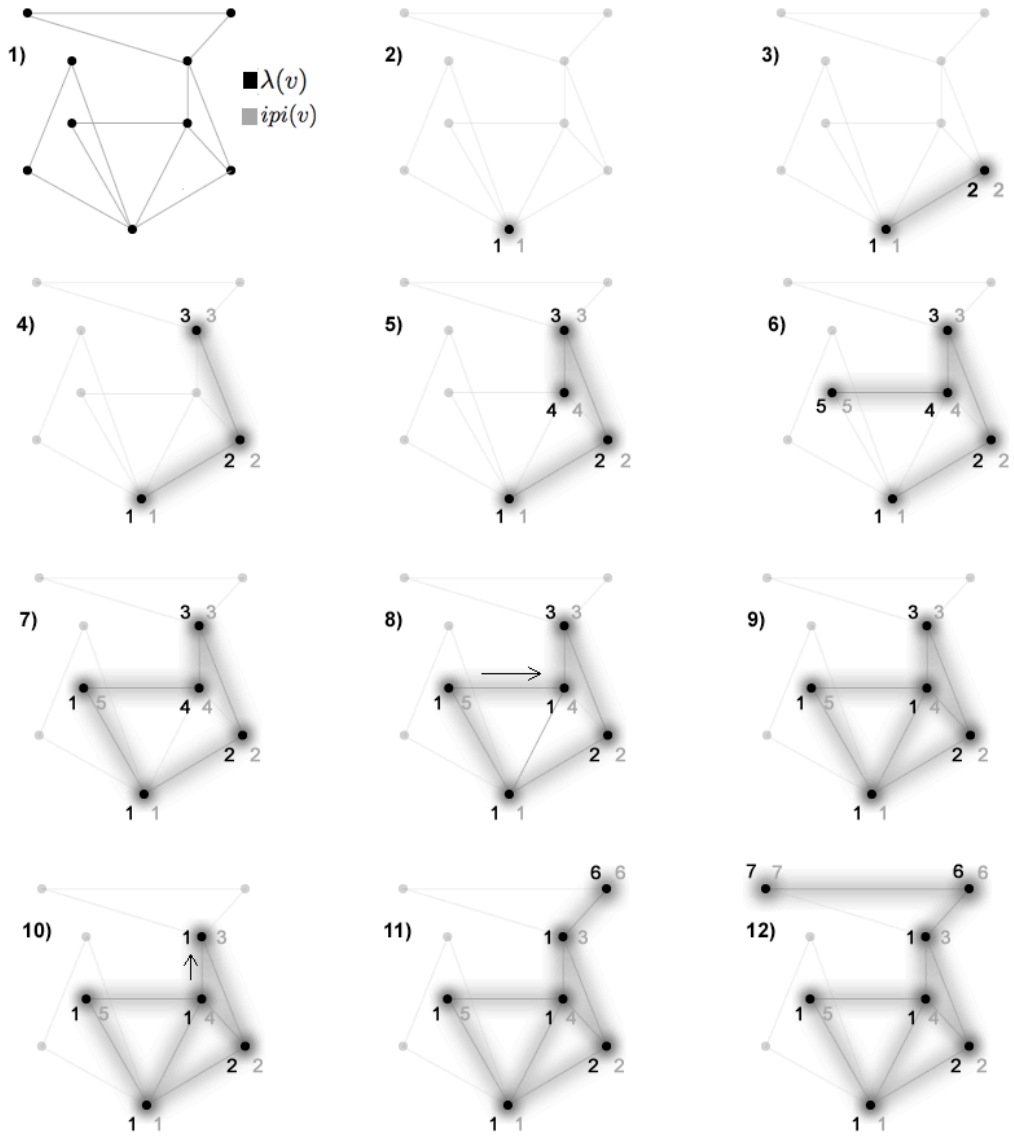


Figura 3.9:
En esta figura podemos observar los primeros 12 pasos del recorrido del algoritmo 3.4.5 sobre la gráfica mostrada en (1). Cada vértice por el que se pasa se coloca en la pila y se le da en la figura un halo gris oscuro, al igual que a cada arista escaneada se le da un halo gris un poco más claro. Al costado de cada vértice analizado se coloca un par de números, en color negro a la izquierda el valor de $\lambda(v)$ y en gris a la derecha el valor de $ipi(v)$, de manera que podamos analizar las interacciones y cambios de estos valores conforme avanza el algoritmo. Las flechas indicarán los pasos en los que el algoritmo regresa por el camino previamente recorrido, actualizando y comparando los valores de λ e ipi de los vértices y sus nodos anteriores.

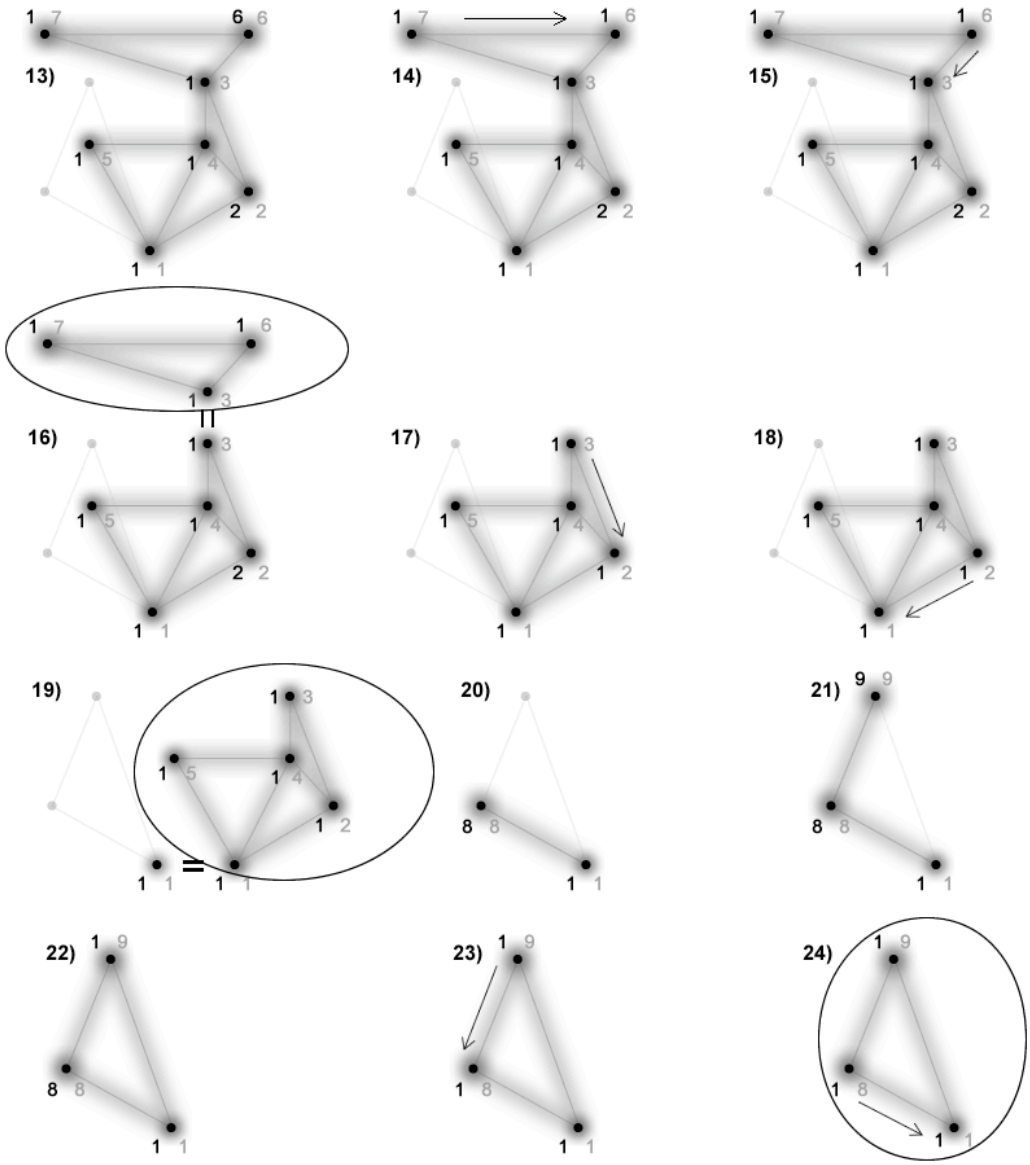


Figura 3.10:
En esta figura tenemos la conclusión del recorrido del algoritmo 3.4.5. Podemos observar en pasos como el (15) y (16) que al regresar sobre el camino recorrido nos encontramos con un vértice cuyo nodo anterior tiene el valor de su λ igual que el *ipi* del vértice en cuestión. Esto nos indica que encontramos un vértice de corte y por lo tanto retiramos de la pila a todos los vértices que ingresaron a éste después del vértice de corte, que junto con él nos determinan un bloque. De manera similar en (19) y el (24) por el paso 11 del algoritmo como regresamos a la raíz quedan determinados los dos bloques restantes de la gráfica.

3.5. Búsqueda de Amplitud Inicial

La Búsqueda de Amplitud Inicial, abreviada como *BAI*, es otra poderosa herramienta utilizada en muchos algoritmos de gráficas como veremos más adelante.

Al igual que la búsqueda de profundidad inicial, la BAI es una forma de visitar todos los vértices de una gráfica de forma sistemática y óptima. Este algoritmo parte de una gráfica expresada a partir de su lista de adyacencias. Utilizaremos para este algoritmo el concepto de *cola* Q en vez de una pila o un arreglo, una cola será un conjunto linealmente ordenado, en el cual un elemento irá primero que otro si fue agregado a Q antes, en la práctica a la cola Q se le irán añadiendo elementos al final y removiéndolos del inicio conforme avance el proceso.

Supongamos que inicialmente todos los vértices de la gráfica están rotulados con 0. Definimos de manera arbitraria a r, vértice de la gráfica, como la ráiz de la búsqueda y como vértice activo, lo rotulamos con 1 y lo añadimos al final de la cola Q (notemos que en este momento Q sólo contendrá a r). Como paso siguiente tomamos todos los vértices adyacentes desde r (suponiendo que existan) en orden según la lista de adyacencias, los rotulamos con el siguiente índice disponible si están rotulados con cero, de otra forma dejamos su rótulo intacto y los añadimos al final de Q, es decir, que si u es el primer vértice de la lista de adyacencias ligado a r, u será rotulado con 2 y añadido después de r en la cola.

Al terminar de hacer esto con todos los vértices adyacentes desde r, retiramos a r de Q, y definimos como nuevo vértice activo al primer elemento de la cola, en este caso u.

Continuamos con el proceso anterior hasta que Q este vacía, si aún quedan vértices de la gráfica sin rotular, es decir, si la gráfica es inconexa, tomamos como nuevo vértice activo a un vértice arbitrario con rótulo 0 y reiniciamos el proceso.

Este algoritmo define un bosque generador F dentro de la gráfica G como sigue, la raíz de cada componente de F es el vértice con rótulo más pequeño dentro de ésta. Y una arista uv de G estará en F si y sólo si u es agregado a Q mientras v es el vértice activo o viceversa.

En una búsqueda de amplitud inicial aplicada a una (p, q)-gráfica G, al visitar todos los vértices de manera sistemática, cada arista es visitada a lo más dos veces, una por cada extremo de ésta. Más aún cada vértice terminal es visitado a lo más una vez. Por lo tanto la complejidad de este algoritmo es $O(max\{p, q\})$.

Podremos encontrar una formalización de este algoritmo en el siguiente capítulo, 4.1.1 Algoritmo de Moore.

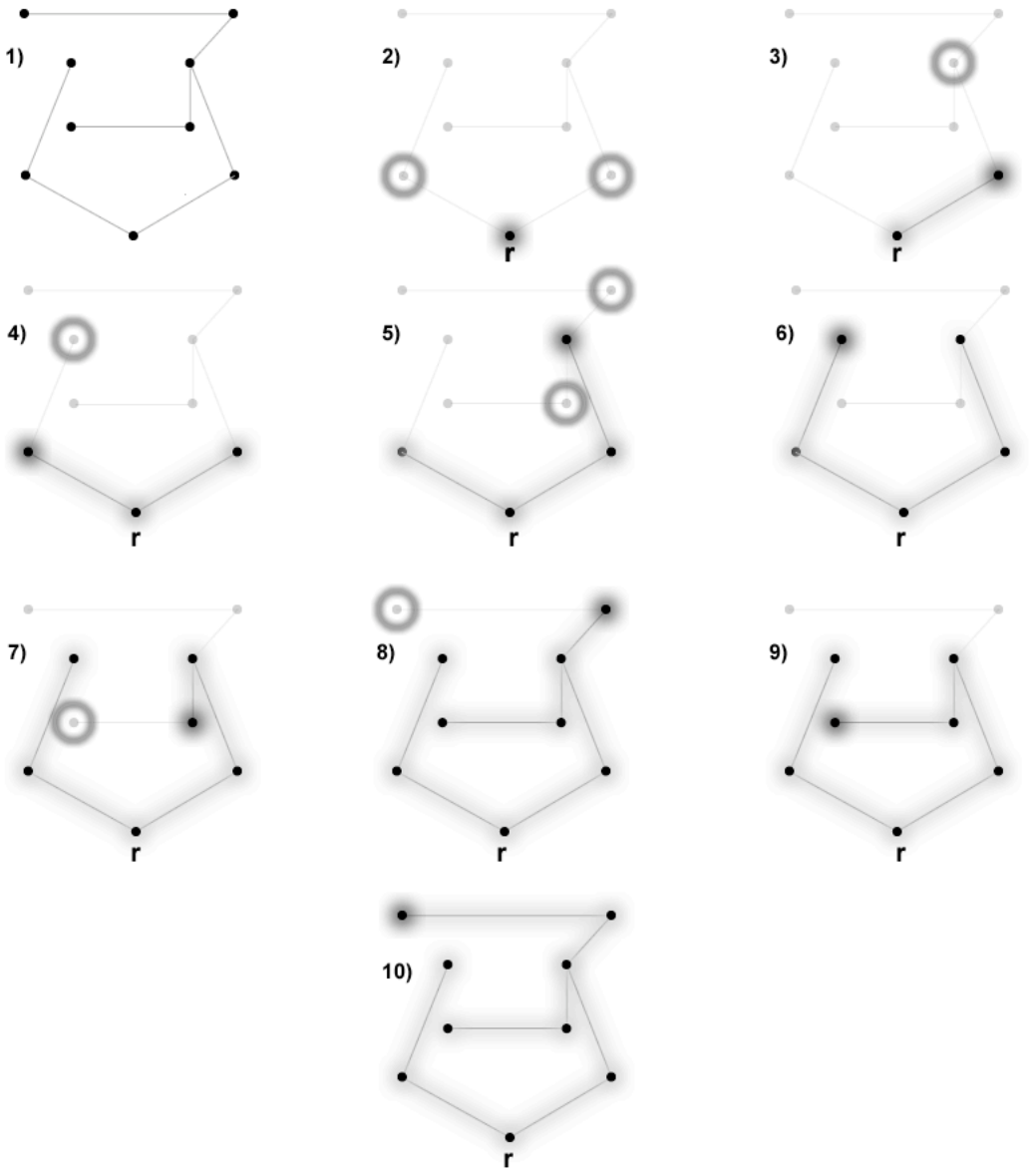


Figura 3.11:
En esta figura podemos observar el accionar de un algoritmo de búsqueda de amplitud inicial, el vértice más obscuro en cada diagrama representa el vértice activo y los vértices circulados representan los agregados a Q al analizar a este último.

3.6. El problema del mínimo árbol generador

Para hablar de árboles de peso mínimo debemos de introducir el concepto de *peso en las aristas*, el cual es asignar un valor numérico a cada arista de la gráfica. Con lo cual podemos definir también el *peso de una gráfica* $w(G)$ como la suma del peso de cada arista de la gráfica.

Por ejemplo, supongamos que se tiene una red de carreteras entre diferentes ciudades, pero un huracán destruye las vías, y se quiere obtener, remodelando el menor número posible de kilómetros de carretera, una red funcional en la cual se pueda circular entre cualesquiera dos ciudades.

En este caso el problema se puede representar con una gráfica, en la cual cada ciudad se vea como un vértice y cada camino destruido entre dos ciudades como una arista. En esta gráfica le podemos asignar a cada arista la distancia de la carretera que representa, por lo que nuestro problema se reduce a encontrar un sub-árbol generador con el menor número de kilometros por remodelar, es decir, un sub-árbol generador T donde $w(T)$ sea mínimo entre todos los sub-árboles generadores. A este árbol se le llamara árbol de peso mínimo o mínimo árbol generador.

Para resolver un problema como éste, se le atribuye a Kruskal la elaboración del siguiente algoritmo que elige aristas de peso mínimo de una gráfica de manera sistemática, sin formar ciclos.

Algoritmo 3.6.1 *(Algoritmo de Kruskal).*

[Para determinar un árbol generador de peso mínimo en una gráfica conexa no trivial G con peso en las aristas.]

1. *[Definir el conjunto S, que consistirá de las aristas del árbol de peso mínimo.]*
   $S \leftarrow \phi$.

2. *[El conjunto S es incrementado.]*
   Sea e una arista de peso mínimo, tal que $e \notin S$ y $(S \cup \{e\})$ acíclico, y sea $S \leftarrow S \cup \{e\}$.

3. *[Este paso determina si se ha construido ya un árbol generador de peso mínimo.]*
   Si $|S| = p - 1$, entonces desplegar S, de otra forma regresar al paso 2.

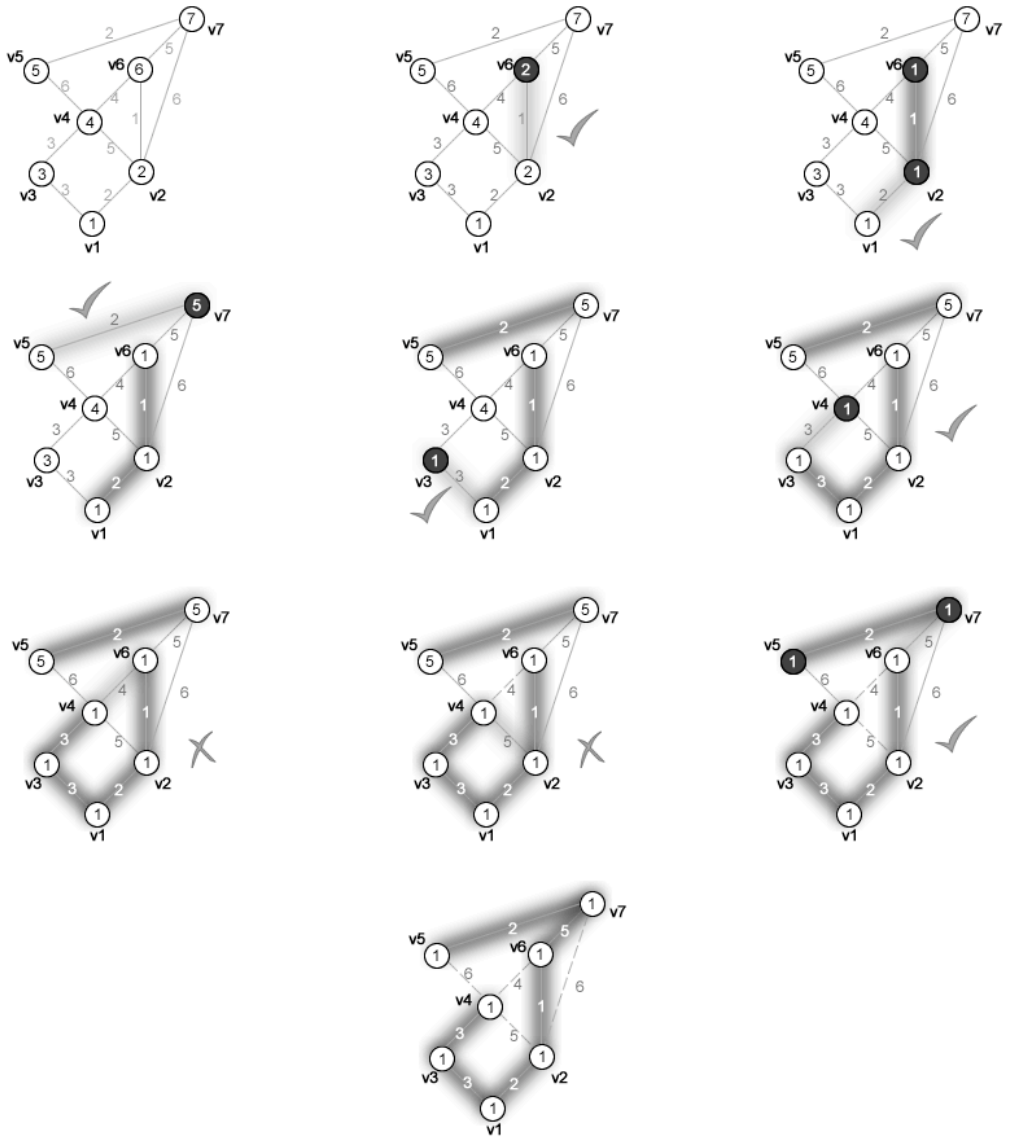


Figura 3.12:
En esta figura podemos observar el funcionamiento del algoritmo 3.6.1. Con un halo gris representaremos a las aristas que se están revisando por el paso 2 del algoritmo. Una paloma o la cruz nos indicarán cuando la arista analizada es aceptada o no por el paso 2 del algoritmo. Dentro de cada vértice tenemos un número que es el valor del arreglo $COMP(v_i)$, usado para verificar si la nueva arista es o no acícilica. En color negro representaremos a los vértices que actualizaron el valor de $COMP(v_i)$ al validar una nueva arista.

Analizando el algoritmo podemos notar que para poder realizar el paso 2, se necesitan un par de cosas. Primero necesitamos ordenar las aristas de G por su peso. Una vez realizado el ordenamiento, es necesario tener una forma óptima de decidir si al añadir una nueva arista el resultado será acíclico o no. Para esto definimos el arreglo $COMP$ de longitud p, donde inicialmente, $COMP(v_i) = i$ para toda i $(1 \leq i \leq p)$.

Ahora, cuando el paso dos se realice por primera vez, la arista $e_1 = v_{j_1}v_{k_1}$ es añadida a S, después asignamos $COMP(v_{j_1}) = COMP(v_{k_1}) \leftarrow min\{j_1, k_1\}$. En general supongamos que las aristas $e_1, e_2, \ldots, e_{i-1}$ han sido comparadas como posibles miembros de S por este proceso. Cuando la arista $e_i = v_jv_k$ sea probada, si $COMP(v_j) = COMP(v_k)$, entonces procedemos con la siguiente arista, ya que e_i no es acíclica con las aristas de S; de lo contrario sea $S \leftarrow S \cup \{e\}$ y definimos $m = min\{COMP(v_j), COMP(v_k)\}$ y $M = max\{COMP(v_j), COMP(v_k)\}$, ahora asignamos $COMP(u) \leftarrow m$ para todo vértice u tal que $COMP(u) = M$. Este proceso es un método eficiente para determinar si al añadir una nueva arista ésta formara un ciclo o no.

Para calcular su eficiencia veamos que $COMP(v_j)$ y $COMP(v_k)$ son comparados a lo más una vez por cada arista v_jv_k, por lo tanto a lo más q comparaciones son requeridas. Además actualizamos el valor de $COMP(u)$, para cada vértice u, a lo más $p-1$ veces, una vez después de cada adición de un vértice a S. Esta actualización tiene una complejidad de $O(p^2)$, de aquí que el algoritmo tenga complejidad $O(max\{p^2, q^2\})$.

Ahora demostraremos que el algoritmo realmente produce un árbol generador de peso mínimo.

Teorema 3.6.2 *El algoritmo de Kruskal produce un árbol generador de peso mínimo en una gráfica G conexa y no trivial.*

Demostración. Sea G un gráfica conexa no trivial con peso en las aristas de orden p, y sea T la subgráfica producida por el algoritmo de Kruskal. Ciertamente T es un árbol generador por construcción. Como todo árbol T de orden p tiene tamaño $p-1$, podemos decir que:

$$E\langle T\rangle = \{e_1, e_2, \ldots, e_{p-1}\}.$$

donde $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_{p-1})$. por lo tanto el peso de T está dado como sigue:

$$w(T) = \sum_{i=1}^{p-1} w(e_i).$$

Supongamos ahora que T no es un árbol generador de peso mínimo, por lo tanto podemos elegir de entre los árboles generadores mínimos de G aquel

árbol H que tenga el máximo número de aristas en común con T. Sabemos que T y H son distintos, por lo que al menos una arista pertenece a T y no a H, digamos que e_i es la de menor índice en cumplir esta propiedad.

Definamos ahora $G_0 = H + e_i$, es claro que G_0 contiene exactamente un ciclo C en donde se encuentra e_i. Como T no contiene ciclos hay una arista en C que no pertenece a T, digamos e_0.
Construimos ahora $T_0 = G_0 - e_0$, árbol generador de G tal que:

$$w(T_0) = w(H) + w(e_i) - w(e_0).$$

Como $w(H) \leq w(T_0)$, pues H es árbol generador mínimo, entonces $w(e_0) \leq w(e_i)$. Sabemos por el algoritmo de Kruskal que e_i es una arista de peso mínimo tal que $\{e_1, e_2, \ldots, e_{i-1}\} \cup \{e_i\}$ es acíclico. Sin embargo $\{e_1, e_2, \ldots, e_{i-1}\} \cup \{e_0\}$ es una subgráfica de H también acíclica, lo cual implica, por el paso 2 del algoritmo de Kruskal, que $w(e_i) \leq w(e_0)$, pues e_i es de peso mínimo con esa característica. De aquí que $w(T_0) = w(H)$, de donde tenemos que T_0 es un árbol generador de peso mínimo con más aristas en común con T que H, lo cual es una contradicción que viene de suponer que T no era árbol generador de peso mínimo. □

Analizaremos ahora otro algoritmo que permite encontrar un árbol generador de peso mínimo en una gráfica conexa con peso en las aristas, este algoritmo atribuido a Prim utiliza un método distinto que Kruskal para llegar al resultado. El algoritmo funciona como sigue:

Algoritmo 3.6.3 *(Algoritmo de Prim).*

[Para determinar un árbol generador de peso mínimo en una (p, q) gráfica conexa no trivial G con peso en las aristas.]

1. *[Iniciar la variable T representando al árbol.]*
   Sea u un vértice arbitrario dentro de G, y sea $T \leftarrow u$.

2. *[Actualizar la variable T.]*
   Sea e una arista de peso mínimo uniendo un vértice de T con algún vértice que no esté en T, y sea $T \leftarrow T + e$.

3. *[Este paso determina si se ha construido ya un árbol generador de peso mínimo.]*
   Si $|S| = p - 1$, entonces desplegar $E\langle T\rangle$, de otra forma regresar al paso 2.

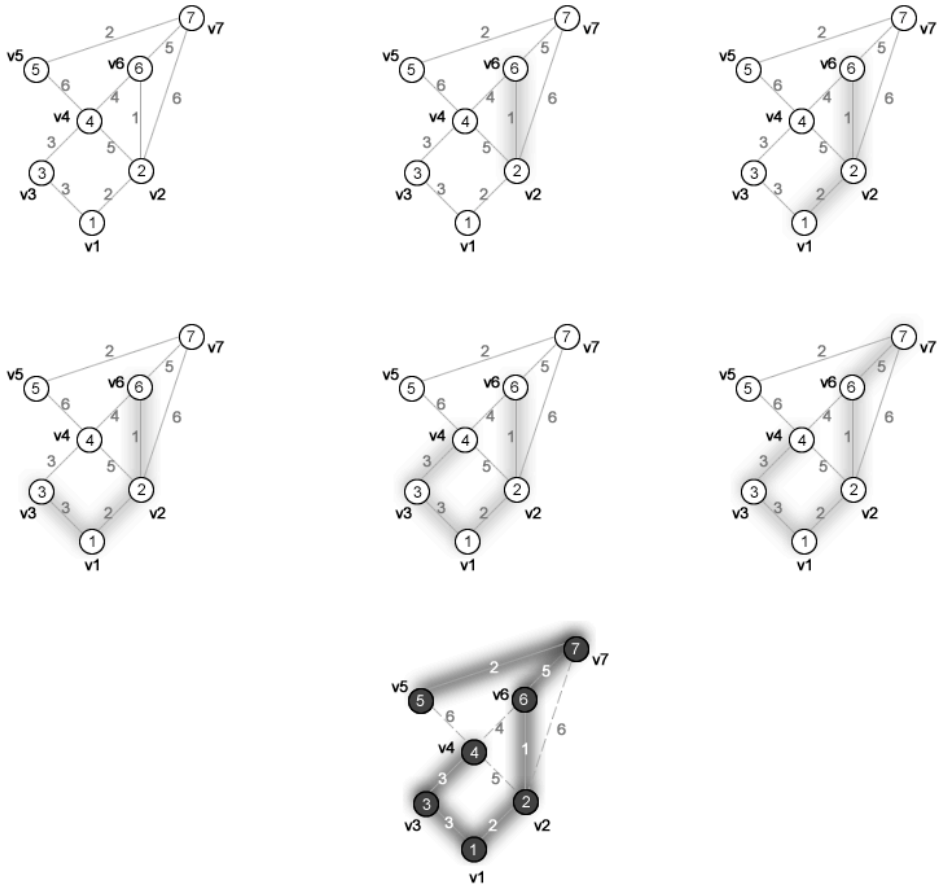


Figura 3.13:
En esta figura podemos observar el accionar del algoritmo 3.6.3, Algoritmo de Prim, en la misma gráfica que observamos el de Kruskal. Las aristas con un halo gris son las que pertenecen al árbol de peso mínimo siendo generado por el algoritmo.

En el paso dos de este algoritmo, si T tiene k vértices, entonces cada elemento de $V\langle T\rangle$ es adyacente a lo más a $p-k$ vértices, de aquí se necesita encontrar el de menor peso de entre a lo más $k(p-k)$ vértices y $k(p-k) < (p-1)^2$. Además necesitamos ordenar por peso, por lo que la complejidad total del paso 2 es $O(p^2)$. Como el paso 2 es realizado $p-1$ veces, entonces la complejidad del algoritmo de Prim es $O(p^3)$.

Al igual que con el de Kruskal, ahora veremos que el algoritmo de Prim produce el resultado deseado.

Teorema 3.6.4 *El Algoritmo de Prim produce un árbol generador de peso mínimo en una gráfica G conexa y no trivial con peso en las aristas.*

Demostración. Sea G un gráfica conexa no trivial con peso en las aristas de orden p, y sea T la subgráfica producida por el algoritmo de Prim. Ciertamente T es un árbol generador por construcción. Como todo árbol T de orden p tiene tamaño $p-1$ podemos decir que:

$$E\langle T\rangle = \{e_1, e_2, \ldots, e_{p-1}\}.$$

donde e_i es la i-ésima arista producida por el algoritmo de Prim. Sabemos también que

$$w(T) = \sum_{i=1}^{p-1} w(e_i).$$

Supongamos ahora que T no es un árbol generador de peso mínimo, por lo tanto podemos elegir de entre los árboles generadores mínimos de G aquel árbol H que tenga el máximo número de aristas en común con T. Sabemos que T y H son distintos, por lo que al menos una arista pertenece a T y no a H, digamos que e_i es la de menor índice en cumplir esta propiedad.

Para $i = 1$, definamos $U = \{u\}$ el conjunto que contiene al vértice arbitrario elegido por paso 1 del algoritmo. Si $i \geq 2$, entonces definimos a $U = V\langle\{e_1, e_2, \ldots, e_{i-1}\}\rangle$. Por otro lado sabemos que la gráfica $H + e_i$ contiene exactamente un ciclo, digamos C. Por construcción es claro que, e_i une un vértice de U con uno de $V\langle T\rangle - U$. Como T no contiene ciclos, entonces necesariamente hay una arista en C que no pertenece a T uniendo un vértice de U con uno de $V\langle T\rangle - U$, digamos e_0. Por lo tanto $T' = H + e_i - e_0$ es también un árbol generador.

Ahora como T fue construido a partir del algoritmo de Prim y ambos e_i y e_0 unen un vértice de U con otro de $V\langle T\rangle - U$, sabemos que $w(e_i) \leq w(e_0)$, de aquí que $w(T') \leq w(H)$; pero como H es árbol de peso mínimo entonces $w(H) \leq w(T')$. Por lo tanto T' es un árbol de peso mínimo en G que tiene más vértices en común con T que H, lo cual es una contradicción que viene

de suponer que T no era árbol de peso mínimo. □

En el caso que las aristas tengan distintos pesos entre si, existe otro algoritmo para encontrar un árbol generador de peso mínimo. En primera instancia nos podría parecer que pedir que todas las aristas tengan distinto peso es demasiado, pero en el mundo práctico eso ocurre a menudo, ya que cada trayectoria es única, por lo que el siguiente algoritmo, atribuido a Boruvka, es otro método socorrido al buscar árboles generadores mínimos.

La idea del algoritmo es empezar con una gráfica que contenga los mismo vértices que G pero sin aristas, nos fijaremos en las componentes conexas que tenga y añadiremos aristas de peso mínimo de G entre componentes conexas distintas, al ser únicamente aristas entre componentes conexas distintas garantizamos que nunca se formarán ciclos. Este proceso continúa hasta formar un árbol generador. El algoritmo es descrito formalmente como sigue:

Algoritmo 3.6.5 *(Algoritmo de Boruvka)*

[Para determinar un árbol generador de peso mínimo en una (p, q) gráfica conexa no trivial G cuyas aristas tengan pesos distintos entre si.]

1. *[Iniciar la variable F representando un bosque generador.]*
   $F \leftarrow \overline{K_p}$

2. *[Actualizar la variable F.]*
   Por cada componente F' de F, unir un vértice de F' con un vértice de otra componente de F por una arista de peso mínimo. Denotamos como S al conjunto de estas aristas y sea $F \leftarrow F + S$.

3. *[Este paso determina si se ha construido ya un árbol generador de peso mínimo.]*
   Si $|S| = p - 1$, entonces desplegar $E\langle F\rangle$, de otra forma regresar al paso 2.

La demostración del funcionamiento así como el cálculo de la complejidad del algoritmo de Boruvka se dejan al lector.

Capítulo 4

Distancia en Gráficas.

4.1. Distancia en gráficas.

Dada una gráfica G y dos vértices $u, v \in V\langle G\rangle$, definimos la *distancia* $d_G(u, v)$ (o $d(u, v)$ si es claro que nos referimos a la gráfica G) entre los vértices u y v como la longitud de una u-v-trayectoria mínima en G, si es que existe, de lo contrario definimos $d_G(u, v) = \infty$.

Para una gráfica dirigida D definiremos la *distancia* $d_D(u, v)$ del vértice u al vértice v como la longitud de una u-v-trayectoria dirigida de longitud mínima, si es que tal trayectoria existe, de lo contrario $d_D(u, v) = \infty$.

Aunque métodos de prueba y error pueden llevarnos a encontrar trayectorias de longitud mínima, ciertamente no son eficientes ni útiles cuando hablamos de gráficas de orden mayor. Por lo tanto empezaremos describiendo un algoritmo, atribuido a Moore, que nos permite encontrar trayectorias de longitud mínima en una gráfica de forma sistemática, utilizando el método de búsqueda de amplitud inicial.

Supongamos que queremos determinar la distancia entre un par de vértices u, v en una gráfica o digráfica. Inicialmente a u le damos el rótulo 0, que indica la distancia de u a si mismo. Y damos el rótulo de ∞ a todos los demás vértices. Después rotulamos con 1 a todos los vértices adyacentes a u que tengan rótulo ∞. A continuación a todo vértice, con rótulo ∞, adyacente a un vértice rotulado con 1 le asignamos el rótulo 2 y así sucesivamente hasta que v tenga un rótulo finito o todos los vértices con rótulo finito sean adyacentes a vértices con rótulo finito. De esta manera, al terminar, cada vértice w con rótulo finito tendrá asignado $d(u, w)$, donde $d(u, w) \leq d(u, v)$.

A continuación presentamos el algoritmo de Moore de manera formal:

Algoritmo 4.1.1 *Algoritmo de búsqueda de amplitud inicial de Moore.*

[Para encontrar $d(u, v)$ para un par de vértices distintos u, v en una gráfica G, así como la u-v-trayectoria mínima, si alguna existe.]

1. *[Inicialmente, cada vértice $w \neq u$ es rotulado con el rótulo $\lambda(w) = \infty$ y a u se le asigna el rótulo $\lambda(u) = 0$. La cola Q contiene inicialmente solo a u.]*
   Para cada vértice $w \neq u$, sea $\lambda(w) \leftarrow \infty$, sea $\lambda(u) = 0$ y $Q \leftarrow \{u\}$.

2. *Si $Q \neq \phi$, entonces borrar un vértice x de Q, de otra forma detener ya que no existe una u-v-trayectoria en G.*

3. *[Dejemos que $PARENT(w)$ denote el nodo anterior del vértice w en el árbol de búsqueda de amplitud inicial. Para cada vértice w que sea adyacente a x con rótulo ∞, asignamos x al $PARENT(w)$ y remplazamos $\lambda(w)$ con la cantidad finita $\lambda(x) + 1$. Después w es añadida a la cola Q.]*
   Para cada vértice w adyacente a x tal que $\lambda(w) = \infty$, asignamos $PARENT(w) \leftarrow x$, sea también, $\lambda(w) \leftarrow \lambda(x) + 1$ y añadimos w a la cola Q.

4. *Si $\lambda(v) = \infty$, entonces regresar al paso 2, de lo contrario ir al paso 5.*

5. *[Este paso encuentra una mínima u-v-trayectoria.]*

   5.1 *Sea $k \leftarrow \lambda(v)$ y $u_k \leftarrow v$.*

   5.2 *Si $k \neq 0$, entonces $u_{k-1} \leftarrow PARENT(u_k)$, de lo contrario ir a 5.4.*

   5.3 *Sea $k \leftarrow k - 1$ e ir a 5.2.*

   5.4 *Desplegar $u_0, u_1, \ldots, u_k$, quienes representan una u-v-trayectoria mínima.*

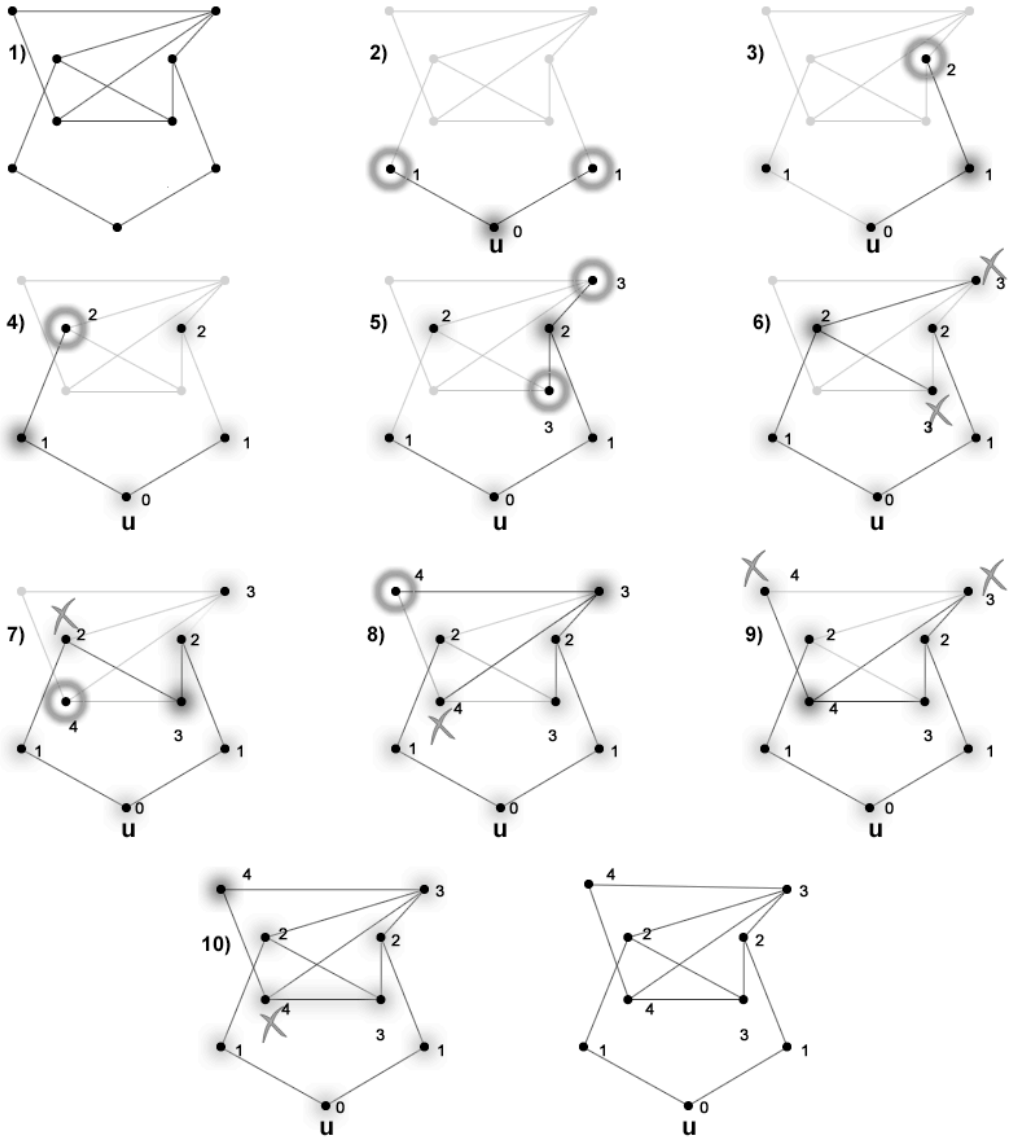


Figura 4.1:
En esta figura podemos observar el accionar del algoritmo 4.1.2, en (1) la gráfica a recorrer. El vértice con el halo en gris oscuro será el vértice activo en cada paso del algoritmo, los vértices circulados representarán a los vértices adyacentes al activo con $\lambda(v) = \infty$, los que tengan cruz a su lado serán los adyacentes con $\lambda(v) \neq \infty$. Los vértices con un halo gris claro serán los ya recorridos.

Este último algoritmo puede ser modificado ligeramente para encontrar la distancia de un vértice u con todos los demás de la gráfica como se muestra a contiunación:

Algoritmo 4.1.2

[Para encontrar $d(u, v)$ para un vértice u fijo en una gráfica G y cualesquiera otro vértice v.]

1. *Para cada vértice $v \neq u$, sea $\lambda(v) \leftarrow \infty$, sea $\lambda(u) = 0$ y definimos a Q de manera que inicialmente contenga únicamente a u.*

2. *Si $Q \neq \phi$, entonces borrar un vértice x de Q e ir al paso 4.*

3. *Si $Q = \phi$, entonces desplegar los pares $(v, \lambda(v))$ para todos los vértices v de G, y detener.*

4. *Para cada vértice w adyacente a x tal que $\lambda(w) = \infty$, sea $\lambda(w) \leftarrow \lambda(x) + 1$ y añadir w a la cola Q. Regresar al paso 2.*

Cuando este algoritmo termine desplegará cada vértice v de G acompañado por $d(u, v)$. Podemos también observar que este algoritmo determina la componente conexa en la que se encuentra u ya que todo vértice con rótulo finito esta conectado con u mediante alguna trayectoria.

Es claro que podemos modificar los algoritmos 4.1.1 y 4.1.2 para que funcionen en digráficas, simplemente remplazando "adyacente a" por "adyacente desde", obteniendo así los mismo resultados que en gráficas.

4.2. Distancia en gráficas con peso en las aristas

Una gráfica G con peso en las aristas se define como aquella a la cual, mediante una función de asignación $w : E\langle G\rangle \to \mathbb{N}$ se le dan valores numéricos a las aristas, por consiguiente, dada $e \in E\langle G\rangle$ nos referiremos a $w(e)$ como el *peso de* e. Extendiendo este concepto, si $T = (v_0, v_1, \ldots, v_n)$ es una trayectoria en G, definimos a $w(T) = \sum_{i=0}^{n-1} w(v_i v_{i+1})$ como el *peso de* T.

Dada una gráfica G con peso en las aristas definimos la *distancia* $d(u, v)$ entre un par de vértices u, v de G como $min\{w(T) \mid T$ es una u-v-trayectoria en $G\}$, si dichas u-v-trayectorias existen, de lo contrario $d(u, v) = \infty$. Podemos observar que el algoritmo de Moore no nos ayuda a encontrar trayectorias de peso mínimo, ya que la longitud de la trayectoria no está relacionada de ninguna manera con el peso de ésta. Por lo tanto es necesario recurrir a un nuevo algoritmo eficiente que nos ayude a resolver este problema.

A continuación presentaremos el algoritmo de Dijkstra desarrollado para encontrar trayectorias de peso mínimo desde un vértice fijo a todo otro vértice de la gráfica, pero para ello serán necesarias las siguientes herramientas.

Sea u_0 un vértice de un gráfica G con peso en las aristas, y supongamos que S es un subconjunto propio de $V\langle G\rangle$ tal que $u_0 \in S$. Sea $\overline{S} = V\langle G\rangle - S$ y definamos la distancia $d(u_0, \overline{S})$ de u_0 a $\overline{S}$ como

$$d(u_0, \overline{S}) = min\{d(u_0, x) \mid x \in \overline{S}\}.$$

Por lo tanto, si no existe una u_0-$\overline{S}$-trayectoria $d(u_0, \overline{S}) = \infty$. De lo contrario existe al menos un vértice $v \in \overline{S}$ tal que $d(u_0, v) = d(u_0, \overline{S}) < \infty$. Además si

$$P : u_0, u_1, \ldots, u_n, v$$

es una u_0-v-trayectoria de peso mínimo, entonces $u_i \in S$ para $i = 0, 1, \ldots, n$, y $u_0, u_1, \ldots, u_i$ es una u_0-u_i-trayectoria de peso mínimo. Más aún.

$$d(u_0, \overline{S}) = min\{d(u_0, u) + w(uv) \mid u \in S, v \in \overline{S} \text{ y } uv \in E\langle G\rangle\}.$$

Ahora si suponemos que el mínimo es alcanzado cuando $u = x$ y $v = y$ entonces

$$d(u_0, \overline{S}) = d(u_0, x) + w(xy) = d(u_0, y). \tag{4.1}$$

Lo cual nos da una expresión para la distancia entre u_0 y y.

Algoritmo 4.2.1 *Algoritmo de Dijkstra.*

[Para determinar la distancia en una gráfica G con peso en las aristas de orden p, desde un vértice u_0 a todo otro vértice de G.]

1. *[Este paso define el rótulo de todos los vértices $v \neq u_0$ de G como ∞ y el de u_0 como 0. Además inicializa la variable índice $i = 0$. El conjunto S, que consistirá de los vértices de G cuya distancia a u_0 haya sido determinada, se define inicialmente como el conjunto que contiene únicamente a u_0. El conjunto $\overline{S}$ se define como $V\langle G\rangle - \{u_0\}$.]*
   Sea $i \leftarrow 0$. $S \leftarrow \{u_0\}$. $\overline{S} \leftarrow V\langle G\rangle - \{u_0\}$. $\lambda(u_0) \leftarrow 0$ y $\lambda(v) \leftarrow \infty$ para todo vértice $v \in V\langle G\rangle - \{u_0\}$.

2. *[Este paso actualiza, de ser necesario, los rótulos de los vértices v en $\overline{S}$ adyacentes a u_i. Además si el rótulo $\lambda(v)$ es cambiado para un vértice v adyacente a u_i, entonces $PARENT(v)$ es cambiado por u_i.]*
   Para cada $v \in \overline{S}$ tal que $u_iv \in E\langle G\rangle$ hacer lo siguiente: Si $\lambda(v) < \lambda(u_i) + w(u_iv)$, entonces continuar, de lo contrario $\lambda(v) \leftarrow \lambda(u_i) + w(u_iv)$ y $PARENT(v) \leftarrow u_i$.

3. *[Este paso determina el vértice u_{i+1} de $\overline{S}$ para el cual $d(u_0, v)$ ha sido encontrada.]*
   Sea $m = min\{\lambda(v) \mid v \in \overline{S}\}$. Si $m = \infty$ detener, de lo contrario sea $w \in \overline{S}$ es un vértice tal que $\lambda(w) = m$, entonces desplegar m como la distancia entre u_0 y w y $u_{i+1} \leftarrow w$.

4. *[Este paso actualiza el conjunto S y $\overline{S}$]*
   $S \leftarrow S \cup \{u_{i+1}\}$ y $\overline{S} \leftarrow \overline{S} - \{u_{i+1}\}$.

5. *[Actualizar el índice i y checar si el algoritmo ha terminado]*
   $i \leftarrow i + 1$. Si $i = p - 1$, entonces detener, de lo contrario regresar al paso 2.

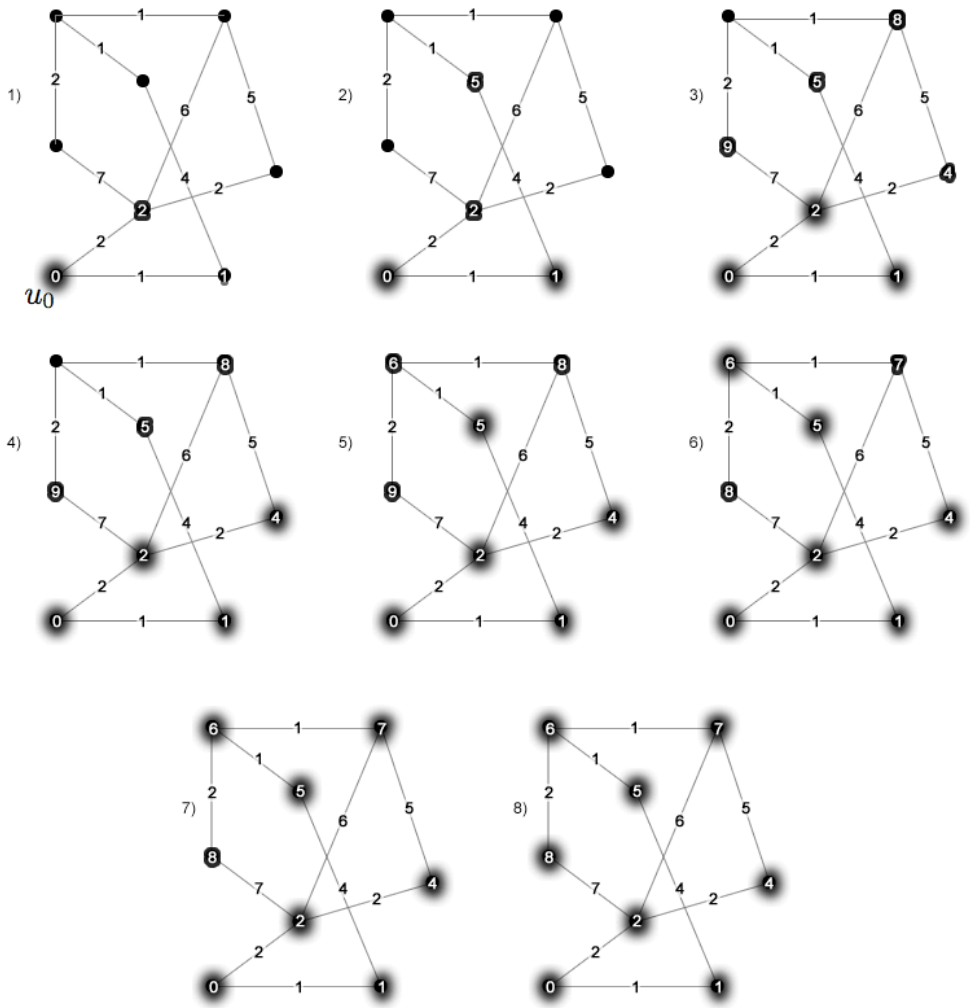


Figura 4.2:
En esta figura podemos analizar el funcionamiento del algoritmo 4.2.1 sobre una gráfica con pesos en las aristas. Iniciamos definiendo el conjunto S, conteniendo únicamente al vértice u_0. Para identificar a S, sus vértices tendrán un halo gris obscuro a su alrededor. Todo vértice v con $\lambda(v) \neq \infty$ tendrá un número blanco sobre él, el resto serán aquellos cuyo $\lambda(v) = \infty$. En (6) podemos observar como se actualiza y mejora el valor de λ para dos vértices previamente visitados por el paso 2 del algoritmo.

Como cada gráfica puede ser vista como una gráfica con peso en las aristas, donde cada arista tiene peso 1, entonces el algoritmo de Dijkstra funciona para encontrar distancias y trayectorias mínimas en gráficas.

A continuación verificaremos que el algoritmo de Dijkstra encuentre lo que necesitamos.

Teorema 4.2.2 *Sea G una gráfica con peso en las arista de orden p. Entonces el algoritmo de Dijkstra encuentra la distancias desde un punto fijo $u_0 \in V\langle G\rangle$ a cualquier otro vértice de G. Esto es que:*

$$\lambda(v) = d(u_0, v) \quad \forall v \in V\langle G\rangle. \tag{4.2}$$

Además si $\lambda(v) \neq \infty$ y $v \neq u_0$, entonces

$$(u_0 = w_0, w_1, w_2, \ldots, w_k = v) \tag{4.3}$$

Es una u_0-v-trayectoria de peso mínimo, donde $w_{i-1} = PARENT(w_i)$ para todo $i = 1, 2, 3, \ldots, k$.

Demostración. Comenzaremos por verificar 4.2, es claro que basta probarlo para una gráfica G conexa, ya que de ser inconexa un vértice w de G para el cual no exista una u_0-w-trayectoria tendrá rótulo ∞ en el momento que el algoritmo termine.

Para lo cual procederemos por inducción sobre i, probaremos que después de que u_i quede determinada para algún $0 \leq i < p - 1$

$$\lambda(v) = d(u_0, v) \text{ para todo } v \in S_i = \{u_0, u_1, \ldots, u_i\}. \tag{4.4}$$

Esto se cumple claramente cuando $i = 0$. Supongamos ahora que 4.4 se cumple para una $i \in \{0, 1, 2, \ldots, p-2\}$ dada, probaremos ahora que cumple para $i + 1$. Es suficiente demostrar que $\lambda(u_{i+1}) = d(u_0, u_{i+1})$. Ahora por el paso 3 del algoritmo de Dijkstra sabemos que u_{i+1} es un vértice tal que $\lambda(u_{i+1}) = min\{\lambda(v) \mid v \in \overline{S_i}\}$, notemos que $u_{i+1} \in \overline{S_i}$, por lo que el mínimo se da cuando $v = u_{i+1}$, de aquí que

$$\begin{aligned} \lambda(u_{i+1}) &= min\{\lambda(v) \mid v \in \overline{S_i}\} \\ &= min\{\lambda(u) + w(uv) \mid u \in S_i, v \in \overline{S_i}, uv \in E\langle G\rangle\} \\ &= min\{d(u_0, u) + w(uv) \mid u \in S_i, v \in \overline{S_i}, uv \in E\langle G\rangle\}. \end{aligned} \tag{4.5}$$

Igualdad que viene de la hipótesis inductiva. Como sabemos que el mínimo en 4.5 ocurre cuando $v = u_{i+1}$, usando el resultado 4.1 tenemos que

$$\lambda(u_{i+1}) = d(u_0, u_{i+1}).$$

Ahora verificaremos 4.3, sea $v \in V\langle G\rangle$ tal que $\lambda(v) \neq \infty$, y $v \neq u_0$. En el momento que algoritmo termine tendremos que

$$\lambda(v) = \lambda(v_1) + w(v_1v).$$

Donde $v_1 = PARENT(v)$ y

$$d(u_0, v) = d(u_0, v_1) + w(v_1, v).$$

Esto garantiza que v_1 es el penúltimo vértice en alguna u_0-v-trayectoria de peso mínimo. Con lo cual, continuando de esta forma, podemos construir una u_0-v-trayectoria mínima

$$T : u_0 = v_n, v_{n-1}, \ldots, v_2, v_1, v.$$

En donde $PARENT(v_i) = v_{i+1}$ para $i = 1, 2, \ldots, n-1$. Si $w_i = v_{n-i}$ para $i = 0, 1, 2, \ldots, n-1$, entonces hemos verificado 4.3. $\square$

Ahora analizaremos la complejidad de este algoritmo, podemos observar que los pasos que más tiempo consumen son el 2 y el 3. Al terminar de correr el algoritmo cada arista de G ha sido utilizada a lo más una vez por el paso 2, si G tiene q aristas entonces la complejidad del paso 2 es $O(q)$. Cada vez que el paso 3 es realizado, el mínimo rótulo de $\overline{S}$ debe ser encontrado, lo cual necesita $|\overline{S}| - 1$ comparaciones. Si G tiene orden p, entonces $|\overline{S}| < p$. Además el paso 3 es realizado $p - 1$ veces, así que la complejidad total del paso 3 esta dada por $O(p^2)$. De aquí que la complejidad total del algoritmo sea $O(max\{p^2, q\})$. Como cada vértice es adyacente a lo más a $p-1$ vértices entonces al sumar las áristas de cada vértice tenemos que:

$$\sum_{v \in V\langle G\rangle} d(v) = 2q \leq \sum_{v \in V\langle G\rangle} (p-1) \leq p(p-1)$$

Por lo tanto como $q \leq p(p-1)/2$ de donde podemos deducir que la complejidad del algoritmo de Dijkstra es $O(p^2)$.

4.3. El centro y la mediana de una gráfica

Para poder definir los conceptos de centro y mediana de una gráfica necesitaremos analizar nuevas propiedades de una gráfica y sus elementos.

Comenzaremos por definir, en una gráfica G con o sin peso en las aristas, la *excentricidad* $e(v)$ de un vértice de G como la distancia desde v al vértice "más lejano" de v en G, esto es,

$$e(v) = max\{d(u,v) \mid u \in V\langle G\rangle\}.$$

Definiremos también el *radio* $rad(G)$ de una gráfica conexa G como

$$rad(G) = min\{e(v) \mid v \in V\langle G\rangle\}.$$

Y el diámetro $diam(G)$ de G como

$$diam(G) = max\{e(v) \mid v \in V\langle G\rangle\}.$$

Teorema 4.3.1 *Dada una gráfica G. Entonces*

$$rad(G) \leq diam(G) \leq 2rad(G).$$

Demostración. La desigualdad de la izquierda viene de la definición. Para verificar la desigualdad derecha, sea $u, v \in V\langle G\rangle$ tales que $d(u,v) = diam(G)$. Sea $w \in V\langle G\rangle$ un vértice tal que $e(w) = rad(G)$. Ahora por la desigualdad del triángulo tenemos que $diam(G) = d(u,v) \leq d(u,w)+d(w,v) \leq 2rad(G)$. □

Definimos también la *distancia* $d(v)$ de un vértice v en una gráfica G con peso en las aristas como la suma de las distancias desde v a cada vértice de G.

Con estas herramientas podemos definir el *centro* $C(G)$ de una gráfica G como la subgráfica inducida por el conjunto de vértices de G que tienen excentricidad igual al radio de G y la *mediana* $M(G)$ de una gráfica G como la subgráfica inducida por el conjunto de vértices de G que tengan distancia mínima. Con esto podemos notar que el centro de una gráfica representa el conjunto de vértices "menos lejanos" a todos los demás, mientras que la mediana representa el conjunto de vértices desde los cuales se optimiza la distancia a los demás. Estos dos conjuntos pueden llegar a ser ajenos.

Capítulo 5

Trayectorias Monótonas

5.1. Coloraciones

En esta sección analizaremos trayectorias dirigidas dentro de una digráfica que sigan un patrón determinado a partir de un código de colores, para lo cual necesitamos definir el uso de coloraciones dentro de una gráfica dirigida.

Dada D una digráfica decimos que $C(D)$ es una *k-coloración por flechas* o simplemente k-coloración de D si $C(D) = (c_1, c_2, \ldots, c_k)$ es una partición de $E\langle D\rangle$. Esto es que para cualesquiera dos $c_i, c_j \in C(D)$

$$c_i \cap c_j = \phi \;\; y \;\; \cup_{i=0}^{k} c_i = E\langle D\rangle.$$

Podemos ver entonces que cada flecha de D pertenecerá a un único elemento de $C(D)$, es decir, que podemos definir una función $c : E\langle D\rangle \to C(D)$ tal que $c(e) = c_i$ si y sólo si $e \in c_i$, en este caso diremos que $c(e)$ es el "color de e". Dado un vértice $v \in V\langle D\rangle$ diremos que el color c_i incide en v si y sólo si $\exists e \in E\langle D\rangle$ tal que e incide en v y $c(e) = c_i$.

Dada una digráfica D con una k-coloración por flechas llamaremos a una digráfica H, *digráfica guía de* D si y sólo si $V\langle H\rangle = C(D)$.

Diremos ahora que una trayectoria dirigida $T = (v_0, v_1, \ldots, v_n)$ en D es *monótona* si y sólo si para todo $i \in \{0, 1, \ldots, n-2\}$ se cumple que:

$$(c((v_i, v_{i+1})), c((v_{i+1}, v_{i+2}))) \in E\langle H\rangle \tag{5.1}$$

En cuyo caso diremos que las flechas (v_i, v_{i+1}) y (v_{i+1}, v_{i+2}) son compatibles. Es decir, que los colores contiguos en las flechas de T sean adyacentes por flechas en la digráfica guía.

Con esto podemos definir, dados dos vértices $u, v \in V\langle D\rangle$, la *distancia monótona* $\delta_M(u, v)$ como la longitud de una u-v-ditrayectoria monótona mínima en D si ésta existe, de lo contrario $\delta_M(u, v) = \infty$.

El siguiente algoritmo nos permitirá decidir si existen o no ditrayectorias monótonas en una digráfica D coloreada por flechas con H como digráfica guía, entre un vértice u fijo y cualquier otro vértice v de la digráfica, y de existir las presentará explícitamente.

Dada una u-w-trayectoria dirigida T en D usaremos la notación de $ext(T)$ para referirnos a w el extremo de T. Llamaremos *segmento terminal o último segmento* a la última flecha de T, es decir, la que contenga a w. Usaremos también la función $PARENT : V\langle D\rangle \times \{T \mid T \textit{ trayectoria dirigida de } D\} \rightarrow V\langle D\rangle \cup \{\phi\}$, tal que $PARENT(x, T)$ representa el vértice anterior a x en la trayectoria dirigida T en caso de existir, de lo contrario la función regresa ϕ.

La idea principal de este algoritmo es extender la búsqueda de amplitud inicial de manera que ésta, siga un patrón de color definido por la gráfica dirigida H.

En un búsqueda de amplitud inicial normal se pasa una vez por cada vértice, en este algoritmo la diferencia radica en que analizaremos cada vértice a lo más una vez por cada color que incida en él. Otra diferencia que debemos notar es que el conjunto Q, en vez de contener vértices, en este algoritmo contendrá trayectorias dirigidas completas, de forma que nos permita, al momento de analizar su extremo, saber por qué color se ha llegado, además de proporcionarnos ditrayectorias monótonas explícitas al termino del algoritmo.

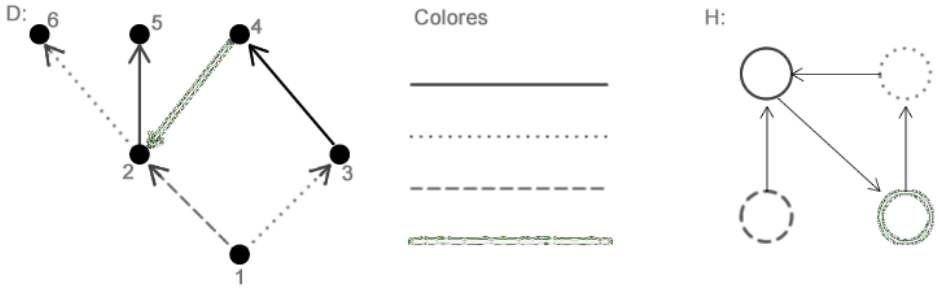


Figura 5.1:
En esta figura podemos notar que si recorremos los vértices de D, tomando como raíz a 1 y usando BAI, llegaremos al vértice 2 por la arista $(1, 2)$, sin embargo la trayectoria $T : (1, 2, 6)$ no es monótona, por lo que si el algoritmo nos restringiera a pasar una sola vez por cada vértice, núnca alcanzaríamos al vértice 6, de aquí que el algoritmo a diseñar pasará más de una vez por vértice, de hecho, a lo más una vez por cada color que incida en él. Siendo así, en algún momento dentro del algoritmo podremos construir la trayectoria monótona $P : (1, 3, 4, 2, 6)$.

Algoritmo 5.1.1

[Dada una digráfica coloreada por flechas D con una digráfica guía H. Algoritmo para encontrar $\delta_M(u,v)$ para un vértice u fijo y cualquier otro vértice v.]

0. *[Este paso se realiza una única vez durante el algoritmo. En este definimos el arreglo tray, que representará para cada vértice v de D la mínima u-v-trayectoria dirigida monótona encontrada por el algoritmo. El arreglo colores que representará el conjunto de los colores que inciden en v y el arreglo λ que representará la distancia monótona entre u y v.*
   Finalmente se definirá al conjunto Q como el que inicialmente consta de la u-u-trayectoria dirigida trivial.]
   $\forall v \in V\langle D\rangle$ *si* $v \neq u$, *entonces* $\lambda(v) \leftarrow \infty$, $tray(v) \leftarrow \phi$ *y*
   $colores(v) \leftarrow \{c(e) \mid e \text{ incide en } v\}$.
   $\lambda(u) \leftarrow 0, tray(u) \leftarrow T : u$.
   $Q \leftarrow \{T : u\}$

1. *[En este paso si Q no es vacío le removemos una trayectoria dirigida de longitud mínima y la enviamos al paso 2.*
   Si Q está vacía, quiere decir que el algoritmo ha terminado y desplegamos los resultados.]
   Si $Q \neq \phi$, *entonces removemos de Q una trayectoria dirigida T tal que* $long(T) = min\{long(P) \mid P \in Q\}$ *y continuamos con el paso 2.*
   De lo contrario terminar y desplegar resultados.

2. *[En este paso se analiza la trayectoria T obtenida del paso 1.*
   Sea x el extremo de T, para cada flecha $e = (x, v)$, si la ditrayectoria monótona T se puede extender con e a una nueva trayectoria dirigida monótona T', es decir que el color del segmento terminal de T y e sean compatibles, y al extremo de T' no se ha llegado antes en el algoritmo con ninguna otra trayectoria de color $c(e)$ en su último segmento, entonces añadimos T', la trayectoria construida, a Q y actualizamos el arreglo colores para el extremo de T', esto último removiendo del conjunto $colores(ext(T'))$ a $c(e)$, de forma que, la próxima vez que una trayectoria con el color de su último segmento igual a $c(e)$ sea analizada en este paso, ésta no entrará a Q.
   Si la trayectoria construída mejora (es de menor longitud) a la anteriormente encontrada para su extremo o no existía ninguna anterior,

entonces actualizamos los arreglos λ y tray para el extremo de la nueva trayectoria dirigida.]
Sea $x = ext(T)$
$\forall e = (x, w) \in E\langle D\rangle$ *tal que* $w \notin V\langle T\rangle$,
si $x = u$ *ó* $(c((PARENT(x, T), x)), c(e)) \in E\langle H\rangle$, *entonces:*
Si $c(e) \in colores(w)$, *entonces* $colores(w) \leftarrow colores(w) - \{c(e)\}$,
$T' \leftarrow T \cup e$, $Q \leftarrow Q \cup \{T'\}$, *y si además* $long(T') < \lambda(w)$, *entonces*
$\lambda(w) \leftarrow long(T')$ *y* $tray(w) \leftarrow T'$.
Regresar al Paso 1.

Denotaremos para el siguiente lema a T_k como la k-ésima trayectoria dirigida en ser removida de Q por el paso 1 del algoritmo. Notemos que T_1 será la u-u-trayectoria trivial que inicialmente contiene Q.

Lema 5.1.2 *Sea D una digráfica coloreada por flechas con una digráfica guía H. Después de realizar el algoritmo 5.1.1. Dada T_i se cumple que para toda T_j, si $i < j$, entonces $long(T_i) \leq long(T_j)$.*

Demostración.

Sean T_i la i-ésima trayectoria en ser removida de Q. Haremos la demostración por inducción sobre j con i arbitraria pero fija.
Si $j = i + 1$ (base de la inducción), entonces tenemos dos casos:

Caso 1: T_j ya estaba en Q al momento que T_i fue removida, por lo tanto, por la construcción del paso 1 del algoritmo, podemos garantizar que $long(T_i) \leq long(T_j)$.

Caso 2: T_j no estaba en Q al momento que T_i fue removida, sin embargo T_j es la siguiente trayectoria en ser removida de Q, por lo tanto tuvo que ser construida a partir de T_i, ya que entre la remoción de T_i y la de $T_{i+1} = T_j$ sólo fueron creadas trayectorias agregando flechas al extremo de T_i, por lo tanto $long(T_j) = long(T_i) + 1 > long(T_i)$.

Supongamos el resultado para toda k, tal que $i + 1 \leq k < j$, y demostremos el resultado para j.

Al igual que en la base tenemos dos casos:

Caso 1: T_j ya estaba en Q al momento que T_i fue removida, por lo tanto, por la construcción del paso 1 del algoritmo, podemos garantizar que $long(T_i) \leq long(T_j)$.

Caso 2: T_j no estaba en Q al momento que T_i fue removida, de aquí que T_j fue agregada a Q después de la remoción de T_i. Por lo tanto T_j fue construida al agregar una flecha al extremo de algún T_m con $i \leq m < j$. Si $m = i$, entonces es claro que $long(T_i) < long(T_i) + 1 = long(T_j)$. De lo contrario, si $i < m$, entonces por hipótesis inductiva sabemos que $long(T_i) \leq$

$long(T_m)$, además $long(T_j) = long(T_m) + 1$. De aquí que podamos concluir que $long(T_i) \leq long(T_m) < long(T_j)$. Con lo que queda demostrada nuestra inducción. □

Lema 5.1.3 *Dada una digráfica coloreada por flechas D con una digráfica guía H. Después de realizar el algoritmo 5.1.1, si $T : u = v_0, v_1, v_2, \ldots, v_k = v$ una u-v-ditrayectoria monótona en D, entonces podemos afirmar que en el algoritmo 5.1.1 entró a Q una u-v-trayectoria dirigida monótona P de D, tal que $long(P) \leq long(T)$ y el color del segmento terminal de ambas trayectorias coincide. Más aún $\lambda(v) \leq long(P)$.*

Demostración. Por inducción sobre $long(T)$.
Sea T una ditrayectoria monótona en D.

Si $long(T) = 1$, entonces $T = (u, v)$ y por lo tanto la primera vez que se realizó el paso 2 del algoritmo, como v adyacente desde u y $c((u, v)) \in colores(v)$, entonces T entró a Q y $tray(v) = T$ de aquí que $\lambda(v) = long(T) = 1$.

Supondremos ahora el resultado para toda $k < n$ y veremos que si $T = (u = v_0, v_1, \ldots, v_{n-1}, v_n)$ tal que $long(T) = n$, entonces el resultado se cumple.

Sea $T' = (u = v_0, v_1, \ldots, v_{n-1})$, con lo que por hipótesis inductiva entró a Q una u-v_{n-1}-ditrayectoria monótona P' con el color de su segmento terminal igual al de T' y $long(P') \leq long(T')$. De aquí que en el momento que P' sea analizada por el paso 2 del algoritmo, como v_n adyacente desde v_{n-1}, entonces $P = P' \cup (v_{n-1}v_n)$ es una ditrayectoria monótona, pues T lo era y coinciden en el color de sus últimas dos flechas; ahora tenemos dos casos a revisar para ese mismo momento en el algoritmo:

Caso 1 : Si $c((v_{n-1}v_n)) \in colores(v_n)$, entonces P fue agregada a Q, una u-v_n-ditrayectoria monótona con el color de su segmento terminal igual al de T. Además $long(P) = long(P') + 1 \leq long(T') + 1 = long(T)$.

Caso 2 : Si $c((v_{n-1}v_n)) \notin colores(v_n)$, quiere decir que ya entró a Q una u-v_n-ditrayectoria monótona R con el color de su segmento terminal igual al de P y T. Como R entró a Q, sabemos que R fue construida a partir de una trayectoria R', agregando una flecha a su extremo en el paso 2 del algoritmo. Como R' fue removida de Q antes que P', por 5.1.2 tenemos que, $long(R') \leq long(P')$, lo cual implica que:

$$long(R) = long(R') + 1 \leq long(P') + 1 \leq long(T') + 1 = long(T). \quad (5.2)$$

De aquí que en ambos casos tengamos una u-v_n-ditrayectoria monótona de longitud menor o igual que T, que coincide en el color de su segmento terminal con T y que entró a Q. Notemos que cuando esta trayectoria

sea agregada a Q, por construcción del algoritmo, podremos asegurar que $\lambda(v_n) \leq long(P) \leq long(T)$. Con lo que por inducción queda demostrado el Lema. $\square$

Teorema 5.1.4 *Dada una digráfica coloreada por flechas D con una digráfica guía H. Al terminar el algoritmo 5.1.1 para todo vértice v en D, se cumple que $\lambda(v) = \delta_M(u, v)$. Más aún $tray(v)$ es una u-v-ditrayectoria monótona mínima.*

Demostración.

Sea $v \in V\langle D\rangle$.

Notemos que al terminar el algoritmo como se modifican simultáneamente los arreglos $tray$ y λ, entonces $\lambda(v) = \infty$ si y sólo si $tray(v) = \phi$, también podemos observar que por definición $\lambda(v) = long(tray(v))$.

Veamos que si existe alguna u-v-ditrayectoria monótona T en D, por el lema anterior tenemos que $tray(v) \neq \phi$ y esto sucede si y sólo si $\lambda(v) \neq \infty$. Por lo tanto acabamos de probar por contrapuesta que si $\lambda(v) = \infty$, entonces no existe ninguna u-v-ditrayectoria monótona en D.

Ahora si T una u-v-ditrayectoria monótona mínima, por el lema 5.1.3 entró a Q una trayectoria dirigida monótona P, tal que $long(P) \leq long(T)$ y $\lambda(v) \leq long(P)$, por lo tanto como T mínima:

$$long(T) \leq long(tray(v)) \leq long(P) \leq long(T) = \delta_M(u, v) \tag{5.3}$$

De aquí que:

$$\lambda(v) = long(tray(v)) = long(P) = long(T) = \delta_M(u, v) \tag{5.4}$$

Por lo tanto $tray(v)$ es también mínima y $\lambda(v) = \delta_M(u, v)$. $\square$

Gracias al teorema anterior sabemos que el algoritmo funciona, ahora sólo nos falta revisar la complejidad de éste, para la cual analizaremos la complejidad de cada paso del algoritmo y cuántas veces se lleva a cabo cada uno de ellos.

El paso 0 se lleva a cabo una única vez con una complejidad $O(p)$ ya que para cada vértice de la gráfica D se llevan acabo varias asignaciones de variables. La complejidad del paso 1 es, usando 2.1.1 para ordenar las trayectorias por longitud, del orden del número de elementos que contenga Q, pero podemos observar que a lo más a cada vértice de la gráfica llegará una trayectoria de Q por cada color, de aquí que el máximo número de elementos que puede llegar a contener Q durante todo el algoritmo es $|V\langle H\rangle|p$, por lo

que la complejidad del paso 1 es $O(p)$. De modo que lo único que debemos revisar es cuántas veces se lleva a cabo, para esto notemos que como el paso 2 es el que regresa el algoritmo al paso 1, entonces este último se llevará a cabo tantas veces como el paso 2 más una. Por lo que sólo nos queda revisar el paso 2.

En el último paso del algoritmo cada vez que entra una trayectoria T se revisan todos los vértices adyacentes a su extremo y las ditrayectorias monótonas que con éstos se puedan generar, revisando que ninguno de éstos pertenezca a T, por lo tanto su complejidad es $\Delta(D)|V\langle T\rangle| = O(p^2)$. Notemos ahora que, por construcción del algoritmo, para cada vértice v de D distinto de u, a lo más vamos a tener $|\{c(e) \mid e \textit{ incide en } v\}|$ distintas u-v-ditrayectorias monótonas entrando al paso 2, lo cual nos dice que éste, a lo más, se llevará a cabo $|V\langle H\rangle|$ veces por cada vértice, dándonos así una complejidad total del paso 2 y del algoritmo de $O(p^3)$.

5.2. Trayectorias monótonas en gráficas

Ahora intentaremos particularizar los resultados de este capítulo a gráficas sin dirección, para lo cual primero daremos algunas definiciones basadas en las existentes para digráficas.

Dada G gráfica, diremos que $C(G)$ es una *k-coloración por aristas* o simplemente una k-coloración de G si $C(G) = (c_1, c_2, \ldots, c_k)$ partición de $E\langle G\rangle$.

Como cada arista de G pertenece a un único elemento de $C(G)$, podemos definir una función $c : E\langle G\rangle \to C(G)$ que asigne a cada $e \in E\langle G\rangle$ el elemento de la partición al que pertenece, diremos entonces que $c(e) = c_i$ es el "color de e" si y sólo si $e \in c_i$.

Una vez definida una coloración para G, podemos asignarle una digráfica guía H de igual forma que en digráficas, es decir, $V\langle H\rangle = C(G)$.

Diremos también que una trayectoria $T = (v_0, v_1, \ldots, v_n)$ es monótona si y sólo si para todo $i \in \{0, 1, \ldots, n-2\}$ se cumple que $(c(v_iv_{i+1}), c(v_{i+1}v_{i+2})) \in E\langle H\rangle$.

Con esto y dados $u, v \in V\langle G\rangle$, podemos definir la *distancia monótona simple* entre u y v, denotada por $\delta_m(u, v)$ como la longitud de una u-v-trayectoria monótona mínima, si ésta existe, de lo contrario $\delta_m(u, v) = \infty$. Notemos que a diferencia de la distancia monótona, en gráficas se cumple la reflexividad, es decir, que $\delta_m(u, v) = \delta_m(v, u)$, cosa que no ocurre en digráficas.

Veremos ahora que el problema de encontrar trayectorias monótonas mínimas en una gráfica G coloreada por aristas con una digráfica guía H ya lo tenemos resuelto.

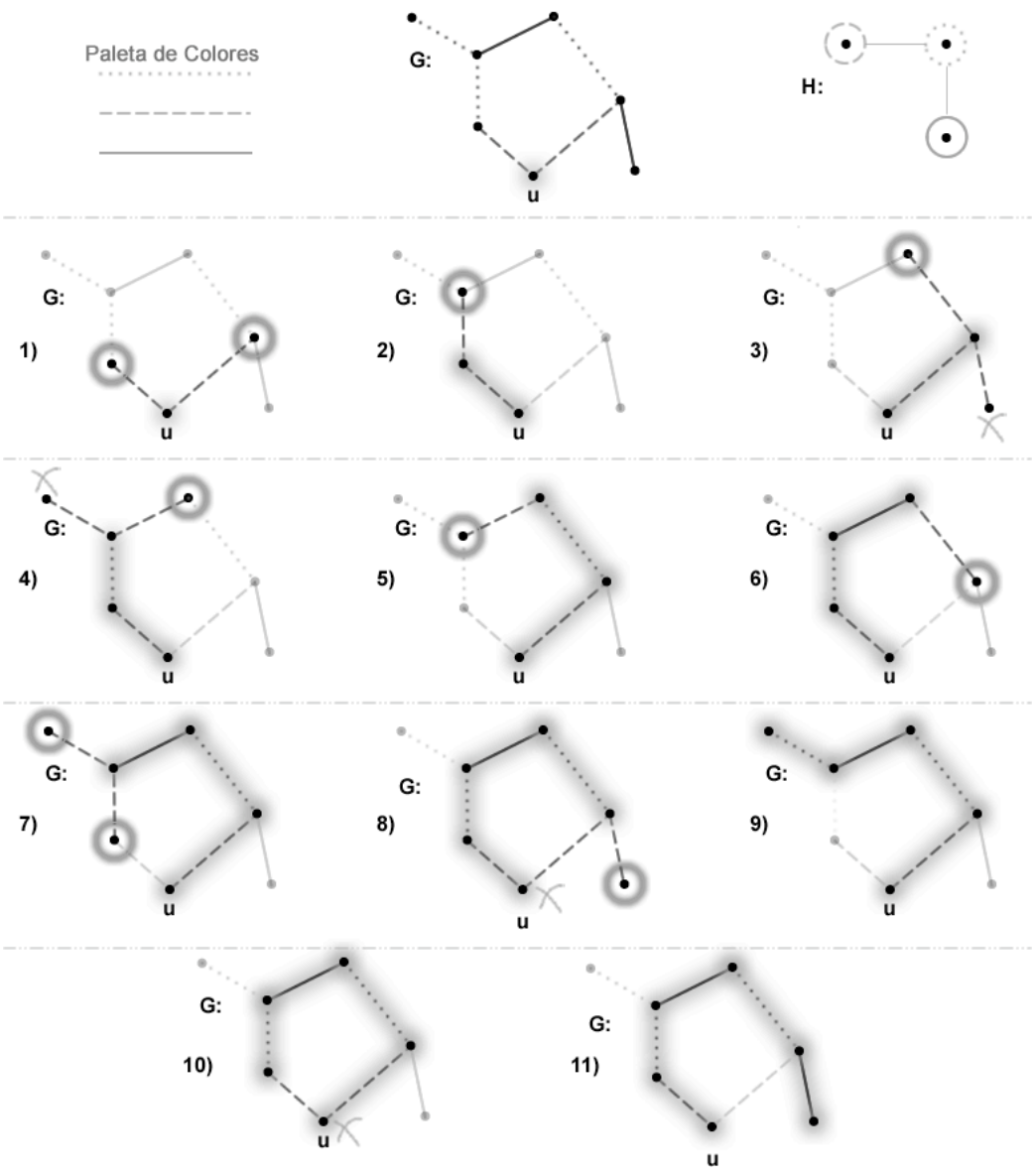


Figura 5.2:
En el primer renglón observemos a la gráfica G coloreada por aristas, la paleta de colores y la gráfica guía H. El algoritmo recorre los vértices de la gráfica analizando trayectorias monótonas extraídas de Q y extendiéndolas a nuevas de ser posible. En esta figura tendremos resaltadas con un halo gris a las trayectorias analizadas recién removidas de Q, circulados en color girs oscuro a los vértices con los que se generan nuevas trayectorias monótonas que se agregarán a Q, y con una cruz a su lado a los vértices con los que no se pudo construir una nueva ditrayectoria monótona.

Dada una gráfica G coloreada por aristas con una digráfica guía H. Si $D = G^*$, definimos una coloración por flechas en D de modo que $\forall e = uv \in E\langle G\rangle$, el par conjugado de uv, (u, v) y (v, u), tenga el mismo color que e. Notemos entonces que H también será digráfica guía en D, por lo que si $u \in V\langle D\rangle$ podemos correr el algoritmo 5.1.1 y obtener para cada $v \in V\langle D\rangle$ una u-v-ditrayectoria monótona mínima guardada en el arreglo $tray$.

Ahora por 1.7.1, sabemos que si $v \in V\langle G\rangle$ y $T_d = tray(v)$, entonces T, la trayectoria subyacente de T_d, es una trayectoria en G, veamos que T es además monótona ya que el color de cada flecha en D es el mismo que el de la arista subyacente asociada a éste, y el patrón de colores en T es por lo tanto el mismo que en T_d.

De aquí que el problema en gráficas sea una caso particular del problema para digráficas. Otros casos particulares similares se dan cuando se utiliza una gráfica guía H en vez de una digráfica, dando así la posibilidad de que las trayectorias monótonas sólo tengan que respetar adyacencias en H y no adyacencias por flechas. Sin embargo este problema será equivalente al ya resuelto usando la digráfica simétrica H^*.

5.3. Trayectorias monótonas en Redes

Una red R se define como una digráfica con peso en los arcos, es decir es aquella digráfica a la cual, mediante una función de asignación $w : E\langle R\rangle \to \mathbb{N}$ se le dan valores numéricos a los arcos, por consiguiente, dada $e \in E\langle R\rangle$ nos referiremos a $w(e)$ como el *peso de* e.

La aproximación a redes dentro de este capítulo no es tan fácil de observar, pero si pensamos que nuestra paleta de colores es un subconjunto de $\mathbb{N}$ y con ésta coloreamos a los arcos de una digráfica R, podremos definir una función de asignación sobre los arcos de R, de manera que cada arco esté asociado con un único valor númerico al que nos referiremos como su peso.

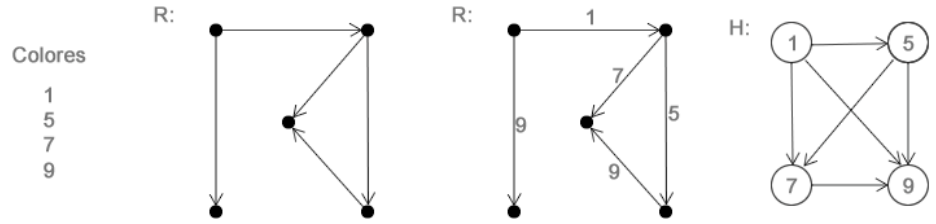


Figura 5.3:
En esta figura podemos observar una digráfica siendo coloreada por una paleta de colores pensada como un subconjunto de $\mathbb{N}$. Así como su digráfica guía H representando el orden heredado de $\mathbb{N}$.

De igual forma podemos definir una digráfica guía como se muestra en el ejemplo, más aún, si definimos a H de modo que represente el orden transitivo heredado por $\mathbb{N}$, es decir que para todo $c_i, c_j \in C(R) \subset \mathbb{N}$ el arco $(c_i, c_j) \in E\langle R\rangle$ si y sólo si $c_i < c_j$. Entonces, al momento de correr el algoritmo 5.1.1 obtendremos trayectorias monótonas estríctamente crecientes en R, de igual forma se puede modificar a H invirtiendo la dirección de sus arcos, de forma que obtengamos trayectorias monótonas estríctamente decrecientes en R. Si además en H aceptamos bucles en todos los vértices, entonces al correr el algoritmo obtendremos trayectorias monótonas crecientes o decrecientes.

Cabe resaltar que el nombre de "trayectorias monótonas" proviene de esta aplicación en el área de redes, lugar dónde el mayor número de aplicaciones y resultados se han presentado.

5.4. Problemas para el futuro

Una vez introducido el concepto de trayectorias monótonas, uno de los problemas que se pueden plantear de manera natural es el de la existencia de algoritmos óptimos que encuentren trayectorias de peso mínimo en gráficas o digráficas coloreadas, con peso en las aristas. Este problema será el siguiente paso a dar, tratando de utilizar las características ya identificadas de las gráficas con coloraciones y el comportamiento de las trayectorias monótonas dentro de algoritmos que visiten todos los vértices de la gráfica de manera sistemática, como el encontrado en este capítulo final.

Otro tema que me quedaría por abordar más adelante es el de la existencia de subconjuntos mínimos de vértices, dentro de digráficas coloreadas por aristas, desde los cuales se puedan alcanzar todos los demás vértices de la gráfica mediante trayectorias monótonas. Es claro que el total de los vértices cumple esta propiedad, pero sin duda el problema será encontrar un algoritmo óptimo que los minimice.

Así como se plantean este par de problemas existen muchos otros que seguramente serán abordados más adelante y este trabajo es sólo el comienzo de esta investigación.

Bibliografía

[1] Arpin, P. and Linek V. *Reachability Problems in Edge-Coloured Digraphs*, Department of Mathematics and Statistics, University of Winnipeg, 2004

[2] Reid K. B. *Monotone reachability in arc-colored tournaments*, California State University San Marco, San Marcos, CA, 2000.

[3] Chartrand, Gary y Oellermann, Ortrud R. *Applied and Algorithmic Graph Theory*, McGraw-Hill, Inc., United States Of America 1976.

[4] Thomas H. Cormen, Charles E. Leiserson, y Ronald L. Rivest *Introduction to Algorithms*, MIT Press, United States Of America 1990.