CAPÍTULO VI

ÁRBOLES Y ARBORESCENCIAS

INTRODUCCIÓN

En este Capítulo se trata el concepto de árbol y el de arborescencia. Se determinan sus propiedades y se utilizan para definir otras estructuras como son los árboles y arborescencias planas, los árboles binarios y los árboles de juego. Se discute sobre estructuras de datos para representar árboles y arborescencias en el computador, así como los tipos de recorridos en arborescencias planas.

VI.1 ÁRBOLES. PROPIEDADES

Como se definió en el capitulo III, un *árbol* es un grafo conexo sin ciclos. Definiremos también una *foresta o bosque* como un grafo donde cada componente conexa es un árbol.

A continuación se hará un desarrollo teórico que facilitará el camino hacia la obtención de una serie de resultados básicos sobre árboles.

Sea G = (V,E) un grafo con E = $\{e_1, e_2, \dots, e_m\}$, un ciclo C en G lo podemos representar como una m-tupla $(c_1, c_2, \dots, c_m)$ donde

$$c_i = \begin{cases} 0 & \text{si } e_i \notin C \\ 1 & \text{si } e_i \in C \end{cases}$$

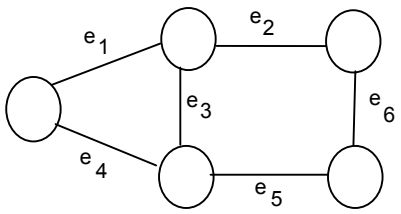
Si C y C' representan dos ciclos de G, la suma $C \oplus C'$ se obtiene de la siguiente manera:

$C \oplus C' = (c_1, c_2, \dots, c_m) \oplus (c'_1, c'_2, \dots, c'_m) = (d_1, d_2, \dots, d_m)$

$$d_i = \begin{cases} 0 & \text{si } c_i = c'_i \\ 1 & \text{si } c_i \neq c'_i \end{cases}$$

en otras palabras, $C \oplus C'$ no es más que la m-tupla que representa los lados no comunes de C y C'.

La figura VI.1 ilustra varios ciclos de un grafo.



$C_1 = < e_2, e_3, e_5, e_6 > \Rightarrow C^1 = (0, 1, 1, 0, 1, 1)$

$C_2 = < e_1, e_3, e_4 > \Rightarrow C^2 = (1, 0, 1, 1, 0, 0)$

$C_3 = < e_1, e_2, e_6, e_5, e_4 > \Rightarrow C^3 = (1, 1, 0, 1, 1, 1)$

figura VI.1

Es de notar que la operación $\oplus$ es conmutativa y asociativa.

Definición VI.1.1

Decimos que un conjunto $\boldsymbol{C}$ de ciclos de un grafo G=(V,E) es *independiente* si no existe ningún sub-conjunto $\boldsymbol{C'} = \{C_1, \dots, C_s\}$ de $\boldsymbol{C}$ tal que $C^1 \oplus C^2 \oplus \dots \oplus C^s = (0, 0, \dots, 0)$, de otro modo el conjunto se llama *dependiente.*

Intuitivamente, un ciclo es dependiente de otros si se puede formar a partir de ellos.

Por ejemplo, el ciclo C_3 de la figura VI.1 se puede formar a partir de los ciclos C_1 y C_2, es decir $C^3 = C^1 \oplus C^2$, por lo que $\boldsymbol{C} = (C_1, C_2, C_3)$ es un conjunto dependiente.

Observación: Un conjunto de ciclos $\boldsymbol{C}$ que consta sólo de uno o dos ciclos será siempre independiente.

En lo que sigue y a menos que se especifique lo contrario, identificaremos un ciclo C con la m-tupla que lo representa.

Definición VI.1.2

Un conjunto independiente de ciclos $\boldsymbol{C}$, de un grafo G=(V,E), es *maximal* si no existe otro conjunto independiente $\boldsymbol{C'}$ de G tal que $\boldsymbol{C} \subset \boldsymbol{C'}$.

Proposición VI.1.1

Sea $\boldsymbol{C}$ un conjunto independiente de ciclos de G=(V,E), $\boldsymbol{C}$ es maximal si y sólo si para todo ciclo C de G tal que $C \notin \boldsymbol{C}$:

$\exists$ ciclos $C_1, \dots, C_k \in \boldsymbol{C}$ con $C = C_1 \oplus \dots \oplus C_k$. Además esta combinación es única.

Demostración:

Sea $\boldsymbol{C}$ maximal y $C \notin \boldsymbol{C}$. Si C no puede expresarse como suma o combinación de un sub-conjunto de $\boldsymbol{C}$, entonces $\boldsymbol{C} \cup \{C\}$ es independiente, lo cual contradice la hipótesis de maximalidad de $\boldsymbol{C}$.

Por otro lado, sea $\boldsymbol{C}$ es un conjunto independiente tal que $\forall C \in \boldsymbol{C}$: existe un sub-conjunto de $\boldsymbol{C}$ cuya combinación es C. Si este conjunto $\boldsymbol{C}$ no fuera maximal, entonces existiría $C' \notin \boldsymbol{C}$ tal que $\boldsymbol{C} \cup \{C'\}$ es independiente, lo cual implica que C' no se puede expresar como combinación de un sub-conjunto de $\boldsymbol{C}$, contradiciendo la hipótesis.

Para ver que la combinación es única, basta con suponer que existe otra combinación de C, es decir

$C = D_1 \oplus \dots \oplus D_l$, con $\{C_1, \dots, C_k\} \neq \{D_1, \dots, D_p\}$.

Entonces $C_1 \oplus \dots \oplus C_k \oplus D_1 \oplus \dots \oplus D_p = 0$. (1)

Sean $C_1,\dots,C_i,D_1,\dots,D_i$ los ciclos comunes a ambas colecciones, entonces también $C_1 \oplus \dots \oplus C_i \oplus D_1 \oplus \dots \oplus D_i = 0$, de donde (1) se convierte en:

$0 \oplus C_{i+1} \oplus \dots \oplus C_k \oplus D_{i+1} \oplus \dots \oplus D_p = 0$,

lo cual implica que

$C_{i+1} \oplus \dots \oplus C_k \oplus D_{i+1} \oplus \dots \oplus D_p = 0$,

lo cual es una contradicción pues $\{C_{i+1},\dots,C_k, D_{i+1},\dots,D_p\}$ es sub-conjunto de $\boldsymbol{C}$.

¤

Esta proposición sirve para caracterizar todos los ciclos de un grafo a partir de un conjunto independiente maximal del mismo.

Proposición VI.1.2

Todas las colecciones independientes maximales de ciclos de un grafo G tienen la misma cardinalidad.

Demostración:

Sean $\boldsymbol{C}$ y $\boldsymbol{D}$ dos colecciones independientes maximales de G:

$\boldsymbol{C} = \{C_1, \dots, C_k\}$, $\boldsymbol{D} = \{D_1, \dots, D_s\}$, con $k < s$.

Si $\boldsymbol{D}$ es independiente entonces D_1 no puede expresarse como combinación de ningún sub-conjunto de $\boldsymbol{D} - \{D_1\}$, pero sí en cambio existe un único sub-conjunto $\{C_i^1\}$ de $\boldsymbol{C}$ cuya suma es D_1. Por lo tanto, al menos uno de los ciclos en $\{C_i^1\}$ no es combinación de ciclos en $\boldsymbol{D} - \{D_1\}$.

Reordenando, podemos suponer que C_1 es tal ciclo y así, el conjunto $\boldsymbol{D_1'} = \{C_1, D_2, \ldots, D_s\}$ es independiente.

De igual manera, D_2 no puede expresarse como combinación de un sub-conjunto en $\boldsymbol{D_1'}$, pero sí de un sub-conjunto $\{C_i^2\}$ de $\boldsymbol{C}$. Existirá entonces en $\{C_i^2\}$ un ciclo $C_i^2 \neq C_1$ que no es combinación de ciclos en $\boldsymbol{D_1'}$ y podemos suponer, reordenando, que $C_i^2 = C_2$.

Así $\boldsymbol{D_2'} = \boldsymbol{D_1'} - \{D_2\} \cup \{C_2\}$ es independiente.

Continuando con el mismo razonamiento, llegamos a construir

$\boldsymbol{D_k'} = \{ C_1, \ldots, C_k, D_{k+1}, \ldots, D_s\}$ independiente, lo cual contradice la maximalidad de $\boldsymbol{C}$, de donde concluimos entonces que k=s.

¤

Definición VI.1.3

Llamamos *rango* de los ciclos de G, denotado r(G), al número máximo de ciclos independientes de G, es decir, la cardinalidad de cualquier conjunto independiente maximal.

Proposición VI.1.3

Sea G=(V,E) un grafo y $\boldsymbol{C}$ un conjunto independiente maximal de G. Si añadimos a G una nueva arista **e**={v,w}, entonces si **e** forma un ciclo C en G'= (V,E U {**e**}), el rango de los ciclos de G', r(G') es r(G) + 1. Si **e** no forma un ciclo en G' entonces r(G') = r(G).

Demostración:

Si se forma el ciclo C en G', este posee un lado {v,w} que no está presente en ninguno de los ciclos de $\boldsymbol{C}$, de donde $\boldsymbol{C'} = \boldsymbol{C} \cup \{C\}$ es un conjunto independiente en G' de cardinalidad r(G)+1.

Veamos que $\boldsymbol{C'}$ es maximal, es decir que si C' es un ciclo en G' tal que $C' \notin \boldsymbol{C'}$, entonces C' es una combinación de ciclos en $\boldsymbol{C'}$.

Si $e \notin C'$ entonces C' es un ciclo de G, por lo tanto se puede expresar como combinación de ciclos en $\boldsymbol{C} \subset \boldsymbol{C'}$.

Si $e \in C'$, obsérvese que :

$C' = (C \oplus C) \oplus C'$ pues $(C \oplus C) = (0, \ldots, 0)$, de donde $C' = (C \oplus C') \oplus C$.

Además, nótese que $e \notin C \oplus C'$ por ser común a ambos. Por lo tanto $C \oplus C'$ se puede expresar como combinación en $\boldsymbol{C}$ ($C \oplus C'$ representa a un conjunto de ciclos disjuntos en G', ninguno de los cuales contiene a e, ver proposición III.1.3.7).

Así $C \oplus C' = \sum_i C_i$ con $\{Ci\} \subseteq \boldsymbol{C} \subset \boldsymbol{C'}$, de

donde $C' = \sum_i C_i \oplus C$ es una combinación de ciclos en $\boldsymbol{C'}$.

Concluimos entonces que C' es maximal y por lo tanto

r(G') = r(G) + 1.

Proposición VI.1.4

Sea G=(V,E) un grafo. Sea n=|V|, m=|E| y p(G) el número de componentes conexas de G. El rango de los ciclos de G es

$$r(G) = m - n + p(G).$$

Demostración:

Usaremos inducción sobre el número de lados de G .

Si G = (V, {}) entonces r(G) = 0 = |E| – |V| + p(G).

Supongamos que la propiedad es cierta para todo grafo con a lo sumo m lados. Veamos que también es cierta para grafos con m+1 lados.

Sea G=(V,E) un grafo con m+1 lados (m≥0), sea e={v,w} un lado cualquiera en G, y sea G'=(V,E-{e})

Sabemos que existe una cadena entre v y w en G' si al agregar e ={v,w} a G', se forma un nuevo ciclo C .

En este caso r(G) = r(G') + 1, según la proposición VI.1.3.

- si existe cadena entre v y w en G', tenemos

$|E| = |E(G')| + 1$ y $p(G') = p(G)$ y así

$r(G) = r(G') + 1 = |E(G')| - |V| + p(G') + 1 = |E| - |V| + p(G)$

- si no existe cadena entre v y w en G', entonces e no genera un nuevo ciclo en G, de donde:

$r(G) = r(G')$, $|E| = |E(G')| + 1$, $p(G) = p(G') - 1$,

y así de nuevo

$r(G) = r(G') = |E(G')| - |V| + p(G') + 1 - 1$

$= |E| - |V| + p(G)$.

¤

Definición VI.1.4

Decimos que un grafo G es *arista maximal* (respectivamente *minimal*) con respecto a una propiedad si G cumple dicha propiedad pero la pierde siempre que una arista es agregada al (respectivamente eliminada del) grafo.

Podemos ahora enunciar el siguiente resultado sobre árboles:

Proposición VI.1.5

Los siguientes enunciados son equivalentes para un grafo G=(V,E) de orden n≥1.
(1) G es un árbol.
(2) G tiene (n–1) lados y no tiene ciclos.
(3) G es conexo y tiene (n–1) lados.
(4) G es arista maximal con respecto a la propiedad "no tiene ciclos".
(5) G es arista minimal con respecto a la propiedad "es conexo".
(6) Para cada par de vértices **x**, **y** en V(G) existe una única cadena simple de **x** a **y**.

Demostración:

Mostraremos (1) ⇒(2)⇒(3)⇒(4)⇒(5)⇒(6)⇒(1).

Sea |E|=m, n=|V| = orden de G, p(G)=Nº de componentes conexas de G.

(1) ⇒ (2): Si G es conexo entonces p(G) = 1. Por ser G un árbol, no tiene ciclos, luego
$r(G) = 0 = m - n + 1$, de donde $|E| = m = n-1$.

(2) ⇒ (3): Si G no tiene ciclos entonces $r(G) = 0 = m - n + p(G)$. Pero $m = n - 1$, de donde $p(G) = n - (n-1) = 1$, luego G es conexo.

(3) ⇒ (4): Si G conexo entonces p(G) = 1, además $m = n-1 \Rightarrow r(G) = (n-1) - n + 1 = 0$ luego G no tiene ciclos. Si agregamos una arista **e** a G, el grafo resultante

G' = (V, E') tendrá | E'| = n, p(G') = 1, luego r(G')=n – n + 1 = 1 y por lo tanto G' tiene un ciclo.

(4) ⇒ (5): Veamos que G es conexo. Supongamos que no lo es, entonces G tendrá por lo menos dos componentes conexas C_1 y C_2. Sea $v \in C_1$, $w \in C_2$. Si agregamos e={v,w} a E(G), este lado no formará un ciclo en G, lo cual contradice la hipótesis que G es arista maximal sin ciclos.

Tenemos entonces que r(G) = 0 y p(G) = 1 ya que G es conexo. Así $0 = r(G) = m - n + 1$ lo cual implica que $m = n - 1$.

Si quitamos una arista **e** a G, el grafo resultante G"=(V, E") no posee ciclos y

$|E"| = m" = m - 1$, luego

$0 = r(G'') = m'' - n + p(G'') = m - 1 - n + p(G'')$

$= n - 2 - n + p(G'') = p(G'') - 2$, de donde $p(G'') = 2$ y por lo tanto G" no es conexo.

(5) $\Rightarrow$ (6): Supongamos que existen dos cadenas simples diferentes C y C' entre dos vértices $v,w \in V$, entonces por la proposición III.1.3.8 existe un ciclo en G. Sea $e=\{x,y\}$ una arista en ese ciclo,entonces en $G =(V,E-\{e\})$ existe aún una cadena entre cualquier par de vértices, es decir que es conexo, lo cual contradice la arista minimalidad de G con respecto a la propiedad de ser conexo.

(6) $\Rightarrow$ (1): Si existe cadena entre cualquier par de vértices, entonces G es conexo, de donde $p(G) = 1$. Si G tuviera un ciclo, entonces existiría al menos un par de vértices con al menos dos cadenas distintas entre ellos, lo cual contradice nuestra hipotesis de unicidad de cadenas entre cada par de vértices.

¤

Proposición VI.1.6

Un árbol $G = (V,E)$ con $|V| \geq 2$ tiene al menos dos vértices de grado 1.

Demostración:

Sea $n = |V|$, por la proposición VI.1.5 sabemos que $|E| = n-1$. por ser G conexo : $d(x) \geq 1$, $\forall x \in V$. Si para todos los vértices menos quizá uno $d(x) \geq 2$ entonces:

$$2|E| = \sum_{x \in V} d(x) \geq 2n - 1 > 2(n-1)$$

lo cual es una contradicción, por lo tanto deben existir al menos 2 vértices de grado 1.

¤

Proposición VI.1.7

Un grafo $G = (V,E)$ es conexo si y sólo si contiene un grafo parcial que sea un árbol.

Demostración:

Basta recordar cualquiera de los métodos para recorrer grafos, DFS o BFS, vistos en el capitulo IV, ya que con cualquiera de ellos obtenemos un grafo parcial que es un árbol si y sólo si el grafo es conexo.

¤

VI.1.1. ÁRBOL COBERTOR DE COSTO MÍNIMO

Sea G=(V,E) un grafo conexo y sea c una función definida sobre los lados de G, $c: E(G) \rightarrow \Re$. Un grafo como el anterior nos permite representar diversas situaciones como por ejemplo las que se presentan en el estudio de redes de comunicación (vial, teléfonica, eléctrica, por ejemplo).

Supongamos que los vértices representan ciudades y los lados posibles interconexiones entre ellas. La función $c:E \rightarrow \Re$ podría representar los costos asociados a la instalación de cada interconexión. El problema en cuestión será buscar una red que conecte todas las ciudades al menor costo posible. Si además suponemos todos los costos positivos, será obvio que mientras menos lados tenga la red, menor será su costo.

Definición VI.1.1.1

Sea G un grafo conexo. Un grafo parcial de G conexo que es a su vez un árbol se llama un *Árbol Cobertor de G.*.

En vista de lo anterior y puesto que debemos conservar la conexidad del grafo, es evidente que necesitamos un árbol cobertor del grafo. Pero hay aún más, el árbol cobertor T que se busca debe ser el de menor costo, es decir, se busca un grafo parcial T de G que sea un árbol y tal que $\sum_{e \in T} c(e)$ sea mínima.

Un árbol que cumpla con la propiedad anterior se llama un *Árbol Mínimo Cobertor de G.*

Proposición VI.1.1.1

Sea G=(V,E) un grafo conexo y c una función c: E$\to\Re$. Sea U un subconjunto propio de V. Sea e={x,y} un lado en Ω(U) (el cociclo definido por U) tal que , $\forall$ e'$\in \Omega$(U) : c(e) $\le$ c(e'). Entonces existe un árbol mínimo cobertor T de G que contiene a e.

Demostración:

Sea T=(V,E(T)) un árbol mínimo cobertor de G. Si e está en T, la propiedad es cierta. Si e no está en T, entonces agregando e a T se formará un ciclo en T'=(V,E(T)$\cup${e}) que lo incluye (Prop. VI.1.5 (4)). Esto implica que existe otro lado e' = {u,v} en T' tal que e'$\in \Omega$(U). Si eliminamos de T' el lado e', obtenemos entonces un nuevo árbol cobertor T" de G, tal que $\sum_{e\in T"} c(e) \le \sum_{e\in T} c(e)$ puesto que por hipótesis, c(e) $\le$ c(e'). Por lo tanto T" es un árbol mínimo cobertor y la propiedad queda demostrada.

¤

Los dos algoritmos que veremos a continuación nos permiten encontrar un árbol mínimo cobertor en un grafo G conexo, y ambos están basados en la propiedad anterior. La diferencia entre ellos radica en la forma cómo se escoge el conjunto U de la proposición VI. 1.1.1.

Algoritmo de Prim:

La idea básica de este algoritmo consiste en inicializar como vacío al conjunto que contendrá los lados del árbol T y con un vértice cualquiera **u** al conjunto U **.** Luego se selecciona el lado {**u,v**} en Ω(U) , como se indica en la proposición VI.1.1.1. El vértice **v** se agrega al conjunto U. El árbol T buscado se irá formando al agregar, uno a uno, los lados que cumplan la propiedad, al mismo tiempo que crece U al incorporársele el otro extremo del lado recién agregado a T. Este proceso se repite hasta que los lados en T formen un árbol cobertor de G, o lo que es lo mismo hasta que U sea igual a V, es decir n -1 veces.

Supongamos que se numera el conjunto V de vértices de G, V={$v_1, v_2, \dots , v_n$}

El algoritmo es el siguiente:

Algoritmo de Prim (Versión 1):
{Entrada: un grafo G=(V,E) no orientado conexo y una función de costos en las aristas.
Salida: Un conjunto T de lados tales que G(T) es un árbol mínimo cobertor de G.}
Variable U: conjunto de vértices .
Comienzo
(0) T $\leftarrow$ {}; U $\leftarrow$ {v_1};
(1) Mientras U $\neq$ V hacer
{ Invariante : |T|=|U|-1 y T está contenido en un árbol cobertor de costo mínimo}
Comienzo
1.1 Escoger e={u,v}, el lado de costo mínimo en Ω(U),
es decir, con u$\in$U, v$\in$V – U;
1.2 T $\leftarrow$ T $\cup$ {e};
1.3 U $\leftarrow$ U $\cup$ {v};
Fin;
Fin;

Proposición VI.1.1.2

El algoritmo de Prim construye un árbol mínimo cobertor.

Demostración:

Para demostrar esta proposición basta con demostrar el siguiente invariante: al comienzo de una iteración cualquiera existe un árbol mínimo cobertor que contiene a T.

En la primera iteración es evidente que se cumple el invariante.

Supongamos que se cumple al inicio de una iteración k, veamos que también se cumplirá al comienzo de la iteración k+1:

Al comienzo de la iteración k: sea Tmin el árbol mínimo cobertor que contiene a T y e* el lado de costo mínimo en Ω(U). Si e* está en Tmin, entonces al comienzo de la iteración k+1, T estará contenido en Tmin. Si e* no está en Tmin, el grafo formado al agregar e* a Tmin posee un ciclo C. Entonces existirá un lado e' en C $\cap \Omega$(U) (nótese que e' no puede

estar en T). El árbol que se obtiene al agregar e* y eliminar e' de Tmin es un árbol mínimo cobertor (ver Proposición VI.1.1.1) que contiene a T∪{e*}, el cual es el conjunto T al comienzo de la iteración k+1.

¤

A efectos de implementación, una forma sencilla de encontrar a cada paso el lado {u,v} de menor costo que conecte U con V–U es la siguiente:

- en un arreglo VECINO[i] mantenemos para cada vértice v_i en V–U el número del vértice en U que esté conectado a v_i por el lado de menor costo;

- en otro arreglo COSTOMIN[i] tenemos el costo del lado $\{v_i, VECINO[i]\}$.

El arreglo COSTOMIN se inicializa de la siguiente forma:

- para cada vértice vi adyacente a v1, $COSTOMIN[i] = c(\{v_i,v_1\})$.

- para los vértices v_i no adyacentes a v_1 a COSTOMIN[i] se le asigna un valor suficientemente grande.

En cada iteración del algoritmo, recorriendo COSTOMIN encontramos el vértice v_k en V–U conectado por un lado de costo mínimo en Ω(U) a un vértice en U y se lo anexamos al conjunto U. Al final, los lados {i, VECINO[i]} serán los lados del árbol. Será necesario en cada iteración actualizar ambos arreglos, teniendo en cuenta que el vértice v_k entra en U y que alguno de los vértices que restan en V–U pudiera estar conectado a v_k por un lado de costo mínimo en Ω(U). Será necesario tener un arreglo que indique si un vértice está o no en V-U. Cada vez que un vértice v_k entra en U, a COSTOMIN[k] se le asigna un valor lo suficientemente grande para evitar que sea considerado de nuevo.

Con una implementación como ésta, la complejidad del algoritmo de Prim es $O(n^2)$, puesto que se debe recorrer n–1 veces un arreglo de tamaño n.

Podemos ver además, que este algoritmo de Prim es otro caso particular del algoritmo general de etiquetamiento. Para ello, asignaremos como atributo de cada cadena abierta del algoritmo de etiquetamiento el costo de su último lado. En cada iteración se escoge la cadena abierta que tenga menor atributo y se cierra esa cadena. Las cadenas cerradas no se reemplazan y serán las que forman el árbol mínimo cobertor. Como justificación de esta versión del algoritmo de Prim podemos considerar el conjunto U formado por los vértices finales de las cadenas cerradas, de modo que el último lado de cada cadena abierta es un lado en Ω(U). Al escoger la cadena abierta con menor atributo estamos escogiendo un lado como lo señala la proposición VI.1.1.1.

Algoritmo de Prim (Versión 2):
{Entrada: un grafo G=(V,E) no orientado.
Salida: una lista de cadenas en G tales que el sub-grafo G(T), inducido por el conjunto de lados de las cadenas es un árbol mínimo cobertor de G.}
Comienzo
(0) Escoja un vértice v_1 en V, abra el camino $P_o = < v_1 >$ y asígnele su atributo $A(P_o) = 0$;
(1) Mientras existan cadenas abiertas hacer:
Comienzo
1.1 Escoja el camino abierto $P_s = <v_1, \dots, v_s>$ con menor atributo.
1.2 Cierre P_s.
1.3 Obtenga los vecinos $v^1, \dots, v^q$ de v_s (el vértice terminal de P_s).
1.4 Construya los caminos expandidos $P^i = Exp(P_s, v^i)$, $1 \le i \le q$, y calcule sus atributos $A(P^i) = c(\{v_s, v_i\})$.
1.5 Aplique la rutina de eliminación de cadenas.
Fin;
Donde
Rutina de Eliminación de Cadenas:
Comienzo
1. Para cada $P^i = Exp(P_s, v^i)$, $1 \le i \le q$, hacer:

```
        Comienzo
         1.1Si existe un camino listado con vértice terminal
              igual a v^i entonces
               Comienzo
               Si existe un camino abierto P_k con vértice terminal
              igual a v^i y A (P_k) >  A( P^i ) entonces
                    Comienzo
                      1.1.2     Eliminar P^k de la lista;
                      1.1.3     Abrir P^i;
                    Fin;
               Fin;
        1.2  Si no abra P^i
         Fin;
  Fin;  {Rutina de eliminación}
Fin.   { Prim }
```

A continuación se muestra un ejemplo de aplicación del algoritmo de Prim (Versión 2) al grafo de la figura VI.2.

Iter 0:	P_o = < a >	Abierto	A (Po) = 0
Iter 1:	P_o = < a >	Cerrado.	
	P_1 = < a, b >	Abierto	A (P_1) = 1
	P_2 = < a, d >	Abierto	A (P_2) = 2
Iter 2:	P_o = < a >	Cerrado.	
	P_1 = < a, b >	Cerrado.	
	P_3 = < a, b, c >	Abierto	A (P_3) =2
	P_4 = < a, b, d >	Abierto	A (P_4) = 1

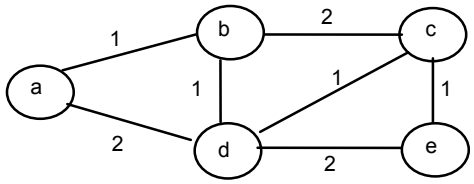


figura VI.2

Nota: El camino P_2 = < a, d > fue eliminado para abrir el camino P_4 = < a, b, d >.

Iter 3:	P_o = < a >	Cerrado.	
	P_1 = < a, b >	Cerrado.	
	P_4 = < a, b, d >	Cerrado.	
	P_5 = < a, b, d, c >	Abierto	A (P_5) = 1
	P_6 = < a, b, d, e >	Abierto	A (P_6) = 2
Iter 4:	P_o = < a >	Cerrado.	
	P_1 = < a, b >	Cerrado.	
	P_4 = < a, b, d >	Cerrado.	
	P_5 = < a, b, d, c >	Cerrado.	
	P_7 = < a, b, d, c, e >	Abierto	A (P_7) = 1
Iter 5:	P_o = < a >	Cerrado.	
	P_1 = < a, b >	Cerrado.	
	P_4 = < a, b, d >	Cerrado.	
	P_5 = < a, b, d, c >	Cerrado.	

P_7 = < a, b, d, c, e > Cerrado.

Algoritmo de Kruskal;

Otra forma de construir un árbol mínimo cobertor de un grafo G=(V,E) es comenzando con un grafo T= (V,{}) donde V es el conjunto de vértices de G. De esta forma cada vértice v en T forma una componente conexa. A medida que avanza el algoritmo iremos agregando lados a T y se formarán nuevas componentes a partir de las anteriores. Para decidir qué lados agregar, se examinan los lados de E en orden creciente de costo. Si el lado en consideración NO forma ciclo con los otros lados ya en T, es decir, si conecta dos vértices en componentes distintas, entonces se agrega ese lado a T, en caso contrario se descarta y se examina el siguiente lado. Cuando todos los vértices forman una sola componente en T, es decir cuando ya han sido seleccionados los |V|–1 lados requeridos, T es un árbol mínimo cobertor.

El algortimo en cuestión es el siguiente:

Algoritmo de Kruskal (Versión 1):
{Entrada: un grafo G = (V,E) conexo no orientado.
Salida: un conjunto T de lados tal que G(T) es un árbol mínimo cobertor de G.}
Variables : Numcomp: entero; e : lado;
{NumComp representa el Nº de componentes conexas del grafo en cada iteración }
Comienzo
(0) T ← {}; NumComp ← |V|; E' ← E;
(1) Mientras NumComp ≠ 1 hacer
{Invariante: existe un árbol mínimo cobertor que contiene a T y NumComp = al número de componentes conexas del grafo inducido por T = número de componentes conexas de G(E-E')}
Comienzo
1.1 e* ← lado de menor costo en E';
1.2 E' ← E' – {e*};
1.3 Si (G(T ∪ {e*})) no contiene ciclos entonces
Comienzo
T ← T ∪ {e*} ;
NumComp ← NumComp – 1;
Fin;
Fin;
Fin.

Proposición VI.1.1.3

El algoritmo de Kruskal construye un árbol mínimo cobertor.

Demostración:

Para demostrar esta proposición basta con demostrar el siguiente invariante: al comienzo de una iteración cualquiera existe un árbol mínimo cobertor que contiene a T y NumComp es igual al número de componentes conexas de G(E-E').

En la primera iteración es evidente que se cumple el invariante.

Supongamos que se cumple al inicio de una iteración k, veamos que también se cumplirá al comienzo de la iteración k+1:

Al comienzo de la iteración k: sea Tmin el árbol mínimo cobertor que contiene a T. Si e* forma ciclo en G(T∪{e*}) entonces el conjunto T al comienzo de la iteración k+1 será igual al conjunto T al comienzo de la iteración k. En caso contrario, sea U el conjunto de vértices de una de las componentes conexas que contiene un extremo de e*. Así, e* está en Ω(U) y es el de menor costo entre todos los lados en Ω(U). Si e* está en Tmin, entonces al comienzo de la iteración k+1, T estará contenido en Tmin. Si e* no está en Tmin, el grafo formado al agregar e* a Tmin posee un ciclo C. Entonces existirá un lado e' en C ∩ Ω(U) (nótese que e' no puede estar en T). El árbol que se obtiene al agregar e* y eliminar e' de Tmin es un árbol mínimo cobertor (ver Proposición VI.1.1.1) que contiene a T∪{e*}, el cual es el conjunto T al comienzo de la iteración k+1.

Por otra parte es evidente que NumComp es igual al número de componentes conexas de G(E-E'). Por lo que podemos garantizar que el While termina ya que NumComp crece de iteración a iteración (aunque no estrictamente), y como la cardinalidad de E' decrece estrictamente en 1 en cada iteración, en alguna iteración el número de componentes conexas de G(E-E') será igual a 1 porque G es conexo.

¤

Este algoritmo resulta sumamente sencillo a simple vista, sin embargo, notaremos que es necesario revisar ciertos detalles en cuanto a la implementación del mismo. Lo primero es con respecto a la forma de seleccionar los lados en orden creciente de costo. Ordenar desde un principio todos los lados resulta bastante inconveniente pues normalmente se estará haciendo mucho más trabajo del necesario, ya que rara vez se llegan a examinar todos los lados antes de encontrar los |V|–1 requeridos. Lo que se necesita entonces es disponer de un operador que permita encontrar en cada iteración el lado de menor costo entre los que aún no han sido estudiados y que lo excluya de E(G). Para lograr lo anterior podemos colocar los lados en un árbol parcialmente ordenado o heap y utilizar un operador que llamaremos LADOMIN. Con este operador podemos extraer el lado de menor distancia y reordenar el heap en un tiempo *O* (log|E|). Igualmente podemos ordenar parcialmente los lados antes de comenzar en un tiempo *O* (|E|), esto lo realizará el operador ORDENAR, similar al procedimiento CREAR de la sección VII.4.

Lo segundo que se debe revisar detalladamente es como se almacenarán los vértices de cada componente, de forma que pueda revisarse de forma eficiente la posible formación de ciclos cada vez que se examina un nuevo lado. Una solución es formar conjuntos con los vértices que pertenecen a una misma componente y de esta manera, con una representación adecuada para conjuntos (AHO et al, [1]), se puede determinar en un tiempo constante a qué componente pertenecen los extremos de un lado, mediante un operador ENCONTRAR aplicado a cada uno de sus extremos para luego verificar si esas componentes son distintas. En caso de ser distintas se deben unir las componentes en cuestión y agregar el lado a T; en caso contrario se desecha el lado examinado. Para unir las dos componentes se requerirá de un operador UNION.

Las n-1 uniones que serán requeridas se pueden lograr en un tiempo *O* (nlogn) con la misma representación de conjuntos.

El operador INICIALIZAR crea las componentes conexas del grafo G=(V,{}), las cuales constan de un solo vértice cada una y serán identificadas con un número de 1 a |V|.

Utilizando los operadores descritos arriba, el algoritmo resulta:

```
Algoritmo de Kruskal  (Versión 2):
{Entrada:  Un grafo G = (V,E) no orientado conexo y una función de costos en las aristas.
 Salida :  Un conjunto de lados T tal que G(T) es un árbol mínimo cobertor del grafo G.}
Variables  Comp, NumComp: entero;
           e: registro x,y:vértice Fin;
           Xcomp,Ycomp : entero;
{ Comp es usado para etiquetar las componentes. NumComp
representa el Nº de componentes conexas del grafo en cada iteración};
Comienzo
(0)  T ← {};  NumComp ← |V|;
(1)  Para i ← 1 hasta |V| hacer
     Comienzo
       INICIALIZAR ( i, i );
     Fin;
(2)  ORDENAR (E);
(3)  Mientras NumComp≠1 hacer
     Comienzo
       LADOMIN (E, e);  { e={e.x, e.y} }
       ENCONTRAR (e.x, Xcomp);
       ENCONTRAR (e.y, Ycomp);
       Si Xcomp ≠ Ycomp  entonces
            Comienzo
              UNION (Xcomp, Ycomp);
            T ← T ∪ {e};
            NumComp ← NumComp -1;
            Fin;
     Fin;
Fin.
```

Con una implementación como ésta, el tiempo de ejecución del algoritmo será *O* (|E|log|E|), pues la inicialización es *O* (|V|), ORDENAR es *O* (|E|), las n-1 veces que se realiza la UNION se logra en un tiempo *O* (|V|log|V|) y LADOMIN es

O (log|E|) y será repetida p veces, donde p es el número de lados que serán examinados antes de formar el árbol (normalmente considerablemente menor que |E|), siendo esto lo que consume el mayor de tiempo del algoritmo.

VI.2 ARBORESCENCIAS. PROPIEDADES

En el Capítulo III hemos definido una raíz de un árbol orientado A como un vértice desde el cual son alcanzables todos los demás vértices del árbol. Se definió así mismo una *arborescencia* como un árbol orientado con un vértice raíz r. Se vieron también algunas consecuencias directas de esta definición que dan lugar a la siguiente proposición.

Proposición VI.2.1

Sea G = (V,E) un grafo orientado de orden n. Las condiciones siguientes son equivalentes y caracterizan una *arborescencia* de raíz **r**.

(i) G es un árbol con una raíz **r**.

(ii) G tiene un vértice raíz **r** y $\forall x \in V$, $x \neq \mathbf{r}$: existe un único camino de **r** a x.

(iii) G admite **r** como raíz y es arco minimal bajo esta propiedad, es decir, si se elimina un arco de G, **r** deja de ser raíz.

(iv) G tiene una raíz **r** y $d^-(\mathbf{r}) = 0$, $d^-(x) = 1$, $\forall x \in V$, $x \neq \mathbf{r}$.

(v) G es conexo con $d^-(\mathbf{r}) = 0$ y $d^-(x) = 1$, $\forall x \in V$, $x \neq \mathbf{r}$.

(vi) G no tiene ciclos y $d^-(\mathbf{r}) = 0$, $d^-(x) = 1$, $\forall x \in V$, $x \neq \mathbf{r}$.

(vii) G tiene una raíz **r** y no tiene ciclos.

(viii) G tiene una raíz **r** y $|E| = n - 1$.

Demostración:

(i) $\Rightarrow$ (ii) G tiene una raíz **r** y por lo tanto existen caminos de **r** a x, $\forall x \in V$. Por ser G un árbol no tiene ciclos y en consecuencia tampoco circuitos, por lo que todos los caminos son elementales.

Supongamos que existe $y \in V$ tal que existen 2 caminos distintos de **r** a y, entonces en el grafo subyacente de G existen dos cadenas elementales, y por tanto simples, distintas de **r** a x, lo cual contradice la hipótesis que G es un árbol (Prop. VI.1.5 (6)).

(ii) $\Rightarrow$ (iii) Sea $e \in E$ tal que **r** es raíz de $G' = (V, E - \{e\})$, entonces existirían dos caminos distintos de **r** al extremo final de e, contradiciendo (ii).

(iii) $\Rightarrow$ (iv) Por ser **r** una raíz sabemos que existe camino de **r** a x, por lo tanto $\forall x \in V, x \neq \mathbf{r}$, $d^-(x) \geq 1$. Supongamos que $\exists\, y \in V$ con $d^-(y) \geq 2$; entonces $\exists\, v_1 \neq v_2$ en V tales que $(v_1,y), (v_2,y) \in E$. Sea $G' = (V, E - \{(v_1,y)\})$, **r** es raíz de G' contradiciendo (iii).

Por otra parte, si $d^-(r) \geq 1$ entonces existiría $x \neq r$ tal que $(x,r) \in E$ y si eliminamos este arco de G **r** seguiría siendo raíz, contradiciendo de nuevo la arco minimalidad de G con respecto a esta propiedad.

(iv) $\Rightarrow$ (v) G es conexo puesto que admite una raíz.

(v) $\Rightarrow$ (vi) Si $d^-(r)=0$ y $d^-(x)=1$, $\forall x \in V$, $x \neq r$ entonces $|E| = \sum_{x \in V} d^-(x) = n-1$

por lo tanto, siendo G conexo, no tiene ciclos.

(vi) $\Rightarrow$ (vii) Si G no tiene ciclos tampoco tiene circuitos y por lo tanto es una grafo de precedencias cuya única fuente es el vértice **r**. Por la Prop. V.5.1.4 todos los caminos más largos partirán de **r** .

$d^-(r)=0$ $d^-(x) =1$ $\forall x \in V$, $x \neq \mathbf{r} \Rightarrow$ todo vértice menos **r** tiene un predecesor, y por lo tanto algún camino que le llega, el más largo de los cuales parte de r, luego **r** es raíz de G.

(vii) $\Rightarrow$ (viii) **r** raíz y G sin ciclos $\Rightarrow$ G conexo y sin ciclos, luego G es un árbol, por lo tanto $|E| = n - 1$.

(viii) $\Rightarrow$ (i) G con raíz $\Rightarrow$ G conexo. $|E| = n - 1$ y G conexo $\Rightarrow$ G es conexo y sin ciclos y por lo tanto es un árbol.

¤

Observación:

Resulta inmediato de la proposición anterior que una arborescencia no tiene circuitos y por lo tanto, todos los caminos son elementales. Además, toda arborescencia tiene un único vértice fuente, la raíz, y al menos un vértice sumidero, una hoja.

VI.2.1 ARBORESCENCIAS COBERTORAS ÓPTIMAS

Un problema similar al de conseguir un árbol óptimo (mínimo o máximo) cobertor se presenta en grafos o redes orientados. Son muchos los problemas que se pueden plantear en los cuales se requiere determinar la forma de encontrar un punto desde el cual comunicarse con todos los demás de un grafo, respetando la orientación de los arcos (vías de comunicación, por ejemplo) y al menor costo global posible. Nuevamente se entiende por costo global de la red la suma de los costos de todos los arcos utilizados. En este caso lo que se desea obtener es una arborescencia mínima cobertora en el grafo asociado al problema.

Definiremos un *Bosque Orientado* como un bosque en el cual cada componente conexa es una arborescencia.

Sea G =(V,E) un digrafo y sea c: E $\rightarrow \Re$ una función que asocia a cada arco e$\in$E un número c(e)$\in\Re$. El problema que tratamos aquí consiste en encontrar un grafo parcial B de G tal que B sea un Bosque orientado y la suma $\sum_{e\in B} c(e)$ sea máxima.

Un subgrafo que cumpla estas condiciones se llama *Bosque Orientado de Peso Máximo de G*. El problema que planteamos al principio de esta sección y algunos otros que mencionaremos, se reducen a variaciones del problema de buscar un Bosque Orientado de Peso Máximo.

Es fácil observar que un bosque orientado B de un grafo G es una arborescencia cobertora si y sólo si el número de arcos de B es igual al número de vértices de G menos uno, y este es el máximo de arcos que puede tener un bosque.

Un bosque orientado de peso máximo de G no incluye ningún arco e con c(e)<0 , por lo tanto, si c(e)≤o, $\forall$ e$\in$E, el bosque orientado de peso máximo tendrá un conjunto vacío de arcos. Más aún, G puede tener un subgrafo que sea una arborescencia y c(e)>0, $\forall$ e$\in$E, y sin embargo, el bosque orientado de peso máximo de G no ser una arborescencia, como se muestra en el ejemplo de la figura VI.3.

Si un grafo G=(V,E) con c:E $\rightarrow\Re$ tiene un subgrafo parcial que es una arborescencia, entonces una arborescencia cobertora de peso máximo de G se puede conseguir basándonos en la siguiente propiedad y en el corolario que le sigue.

Proposición VI.2.1.1

Sea G=(V,E) un digrafo y c:E $\rightarrow\Re$.

Sea c':E $\rightarrow\Re$ con c'(e) = c(e) + k, $\forall$e$\in$E, y $k > \sum_{e\in E}|c(e)|$.

Si B es un bosque orientado de peso máximo para c', entonces B es el bosque con mayor peso con respecto a c, entre todos los bosques orientados cobertores de G que poseen el mayor número de arcos posible.

Demostración:

Veremos primero que un bosque orientado de peso máximo en G con c'j = cj + k, $k > \sum_{e\in E}|c(e)|$, tiene un número máximo de arcos.

Sean B1 y B2 dos bosques orientados cobertores de G con pesos P'(B1) y P'(B2), tales que:

|E(B1)| > |E(B2)|. Entonces

$$P'(B1) = \sum_{e\in E(B1)} (c(e)+k) = \sum_{e\in E(B1)} c(e) + |E(B1)|*k$$

$$\geq \sum_{e\in E(B1)} c(e) + (|E(B2)| + 1)*k = \sum_{e\in E(B1)} c(e) - \sum_{e\in E(B2)} c(e) + \sum_{e\in E(B2)} c(e) + (|E(B2)| + 1)*k >$$

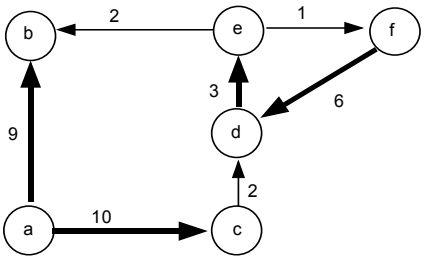
$$\sum_{e\in E(B1)} c(e) - \sum_{e\in E(B2)} c(e) + \sum_{e\in E}|c(e)| + \sum_{e\in E(B2)} c(e) + |E(B2)| *k =$$

$$\sum_{e\in E(B1)-E(B2)} c(e) - \sum_{e\in E(B2)-E(B1)} c(e) + \sum_{e\in E} |c(e)| + \sum_{e\in E(B2)} c(e) + |E(B2)| * k$$

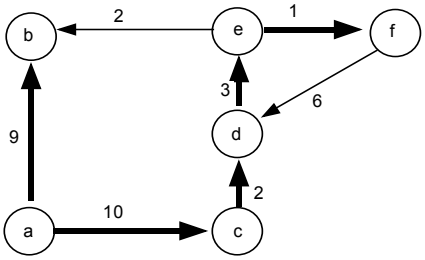
$$\geq \sum_{e\in E(B2)} c(e) + |E(B2)| * k = P'(B2)$$

De lo anterior se deduce que si B es un bosque orientado cobertor de G con respecto a c', entonces B tiene número máximo de arcos. Además, si B_1, B_2 son dos bosques orientados cobertores de G con $|E(B_1)| = |E(B_2)|$ y $P'(B_1) > P'(B_2)$ entonces $\sum_{e\in E(B1)} c(e) = P(B1) > P(B2) = \sum_{e\in E(B2)} c(e)$, lo cual completa la demostración.

¤



(a) Bosque de peso máximo



(b) Arborescencia cobertora de peso máximo

figura VI.3

Corolario VI.2.1.1

Si G admite una arborescencia, el bosque máximo B con respecto a c' es una arborescencia de peso máximo con respecto a c en G.

Demostración

La prueba se basa en el hecho que ningún bosque tiene más arcos que una arborescencia cobertora y que si A1, A2 son dos arborescencias cobertoras de G con pesos P(A1) > P(A2) con respecto a c entonces

$$P'(A1) = \sum_{e\in E(A1)} (c(e)+k) = P(A1) + (|V| - 1) * k$$

$$> P(A2) + (|V| - 1) * k = P'(A2)$$

donde P'(A1), P'(A2) son los pesos de las arborescencias con respecto a c'.

¤

Sin embargo, no será necesario modificar los pesos de los arcos para buscar una arborescencia máxima cobertora, ya que el algoritmo que estudiaremos se puede modificar fácilmente para tratar directamente este problema.

Una arborescencia A en G de peso mínimo, es decir, una arborescencia con $\sum_{e\in E(A)} c(e)$ mínima es una arborescencia de peso máximo en G' =(V,E) con c'(e) = –c(e), $\forall e\in E$.

Por otra parte, si lo que se busca es una arborescencia cobertora de peso máximo o mínimo cuya raíz sea un vértice dado **x**, basta con agregar al grafo original un vértice ficticio x_o y un arco (x_o, x) con cualquier peso positivo. Si existe una arborescencia en G con raíz **x**, entonces existe una arborescencia con raíz x_o en el grafo modificado.

Veremos a continuación un algoritmo debido a Edmonds (1966) para encontrar un bosque de peso máximo en un grafo G = (V,E) con c: $E\rightarrow\Re$.

Definiremos conjuntos D^i y E^i, inicialmente vacíos y a partir de ellos iremos generando una secuencia de grafos G^i, siendo G^o = G, hasta conseguir en algún G^f un bosque orientado de peso máximo, formado por los arcos en el conjunto E^f. El algoritmo es el siguiente:

Algoritmo de Búsqueda de Bosque Orientado de Peso Máximo de G = (V,E):

Comienzo

Paso 1: Escoja un vértice cualquiera x en G^i tal que $x\notin D^i$. Agregue x a D^i. Si existen arcos en G^i con peso positivo, dirigidos hacia x, entonces tome uno, digamos **e**, con peso máximo y luego inclúyalo en E^i. Repita este paso hasta que:

(a) Los arcos de E^i no formen un bosque en G^i, es decir, hasta que se forme un circuito con la inclusión de e en E^i, o hasta que

(b) Todos los vértices de G^i estén en D^i, y E^i no posea circuitos, en cuyo caso los arcos de E^i forman un bosque cobertor en G^i.

Note que cada arco **e** que se agrega a E^i en el paso 1 es el único en E^i que llega a algún vértice x en D^i, el cual hasta el momento de la inclusión de **e** en E^i era una raíz de alguna componente del bosque $B' = (V(G^i), E^i)$. Si el otro extremo de **e** está en la misma componente que x, entonces al incluir **e** se formará un circuito Q^i, es decir, se presentará el caso (a).

Si ocurre (a) entonces:

Paso 2 : Almacene Q^i junto con el arco $e_o{}^i$ de menor peso en Q^i. Contraiga los vértices en Q^i en un nuevo vértice v^{i+1} para formar el siguiente grafo G^{i+1} cuyos vértices serán $V(G^{i+1}) = (V(G^i) \approx \{v^{i+1}\}) - V(Q^i)$. Los arcos de G^{i+1}son aquellos arcos de G^i que tienen a lo sumo un extremo en Q^i. Por cada **a** arco en G^i con un extremo (y sólo uno) en Q^i se crea un **a'** arco en G^{i+1} que resulta de reemplazar en **a** el extremo en Q^i por el vértice v^{i+1}. Denotemos por F^{i+1} el conjunto de los arcos **a'** con extremo inicial v^{i+1}.

Haga $E^{i+1} = (E^i\cap E(G^{i+1}))\cup F^{i+1}$ y $D^{i+1} = D^i - V(Q^i)$

(note que $v^{i+1} \notin D^{i+1}$ y que en E^{i+1} puede existir arcos paralelos).

Los pesos c^{i+1} de los arcos en G^{i+1} serán los mismos que en G^i salvo para aquellos vértices cuyo extremo final sea v^{i+1}. Para cada uno de estos arcos e_S, sea x_S el extremo de e_S en Q^i y e_k el único otro arco en Q^i que llega a x_S, entonces

$$c^{i+1}(e_S) = c^i(e_S) + c^i(e_o{}^i) - c^i(e_k).$$

Note que: $c^i(e_o) \geq 0$

$$c^i(e_k) \geq c^i(e_o)$$

$$c^i(e_k) \geq c^i(e_S)$$

(ya que siendo ambos, e_S y e_k, incidentes a x_k, e_k fue escogido para entrar en E^i en lugar de e_S).

Repita nuevamente el paso 1 con i = i+1.
Luego de algunas repeticiones de Paso 1, Paso 2, ocurrirá el caso (b).
Al ocurrir (b) ejecute el Paso 3.

Paso 3 : Al llegar a la ejecución de este paso el grafo $(V(G^i), E^i)$, que llamaremos B^i, es un bosque orientado.

Construya H^i a partir de B^i sustituyendo el vértice v_i por el circuito Q^{i-1} (de cuya contracción resultó v^i). Es evidente que H^i tiene un único circuito elemental que es Q^{i-1}.

Ahora construiremos el bosque B^{i-1} de la siguiente manera:

Si v^i es raíz de alguna componente en B^i, entonces elimine de H^i el arco de menor peso en Q^{i-1}; el grafo así obtenido será B^{i-1}.

Si v^i no es raíz en B^i entonces, sea **e** el único arco que llega a v^i en B^i. Este arco **e** corresponde a un arco e' en G^{i-1} que llega a un vértice x_k de Q^{i-1}. Sea e_k el único otro arco en Q^{i-1} que llega a x_k. B^{i-1} se obtiene de eliminar el arco e_k de H^i.

Si $i-1 \neq 0$, vuelva a repetir el paso 3 con $i=i-1$.

Fin;

Es evidente que al detenerse el algortimo, B^o es un bosque orientado cobertor. Solo faltaría mostrar que B^o es además un bosque de peso máximo.

Informalmente, puede verse que cada arco incluido en E^i en el paso 1 es el mejor posible. Al pasar de G^i a G^{i+1} los nuevos costos de los arcos con su extremo final en Q^i se determinan de manera tal que de ser elegido uno de ellos para entrar en E^{i+1}, entonces al construir el bosque B^i en el paso 3, el arco en cuestión resultará en una mayor contribución al peso del bosque que los arcos del circuito que serían incluidos en caso contrario. Esto puede verse de la siguiente manera:

Si e_S es un arco cuyo extremo final es v^{i+1} y se le incluye en E^{i+1} en el paso 1, entonces $c^{i+1}(e) = c^i(e) = c^i(e_S) + c^i(e_o{}^{i-1}) - c^i(e_k) > 0$, donde e_k es el arco de Q^i cuyo vértice final es el mismo de e_S. Es decir que:

$$c^i(e_S) + c^i(e_o{}^i) > c^i(e_k)$$

y justamente e_k es el arco que será incluido en H^i al construir B^i.

Si por el contrario, v^{i+1} es raíz en B^{i+1} es porque $c^{i+1}(e) \leq 0$, para cualquier arco e cuyo extremo final es v^{i+1}, por lo que resulta más conveniente excluir $e_o{}^i$ del circuito. El ejemplo de la figura VI.4 muestra esta situación. Para una demostración más formal de lo anterior el lector interesado puede referirse a Edmonds, J. 1968. "Optimum Branchings"[6].

Para obtener una arborescencia de peso máximo en un grafo que contenga una arborescencia cobertora, la única modificación necesaria es eliminar en el paso 1 la exigencia de tomar sólo arcos positivos. En efecto, en lo que a los cálculos del algoritmo se refiere, la única consecuencia de sumar a todos los arcos una constante positiva lo suficientemente grande es que en todo momento los pesos de los arcos serán positivos.

Esto último es fácil de observar, ya que si e es un arco cuyo vértice final es un vértice del circuito Q^i encontrado en el paso 1, entonces:

$$c^{i+1}(e) = (c^i(e) + k) + (c^i(e_o) + k) - (c'(e_k) + k)$$

$$= c^i(e) + c^i(e_o) + k - c^i(e_k) > 0.$$

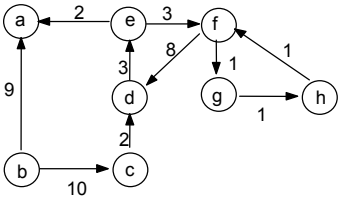
Como resultado, en el paso 1, siempre que para un vértice x, $d^-(x) > 0$, se incluirá un arco con vértice final x en E^i asegurando así que se llegue a una arborescencia de pesos máximos (si el grafo tiene un subgrafo que sea una arborescencia cobertora).

Note que en el ejemplo de la figura VI.4 si sumamos la constante k=100, lo cual es mayor que $\Sigma_{e \in E} c(e)$, el costo del arco (c,v^2) en G^2 sería igual a 97, con lo que sería incluido en en E^2 y así quedaría la arborescencia de la figura VI.5 como resultado final. El mismo resultado se obtendría si en el paso 1 permitimos escoger arcos negativos.

VI.3 EQUIVALENCIA ENTRE ÁRBOLES Y ARBORESCENCIAS. ÁRBOLES ENRAIZADOS

Sea T un árbol no orientado y **r** un vértice cualquiera de T. Por ser T conexo, existe una cadena simple (única) desde **r** hasta todo otro vértice del árbol. Podemos entonces dotar de una orientación a cada uno de los lados de T de forma que

exista un camino (único) desde **r** hasta todos los demás vértices de T. El grafo orientado así obtenido es único y, además, es una arborescencia cuya raíz es el vértice **r**.



(a) $G = G^0$

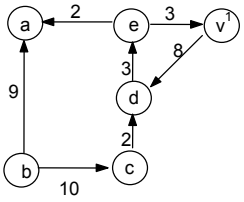
$D^o = \{ a, b, c, g, h, f \}$ $D^1 = \{ a, b, c, d, e, v^1 \}$

$E^o = \{(b,a),(b,c),(f,g),(g,h),(h,f)\}$

$Q^o = \langle f,(f,g),g,(g,h),h,(h,f),f \rangle$

$E_o^0 = (h,f)$, $c^1(e, v^1) = 3 + 1 - 1 = 3$

$e^1 = (e,v^1)$, $c^2 = (c,v^2) = 2+3-8 = -3$

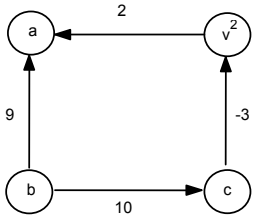


(b) G^1

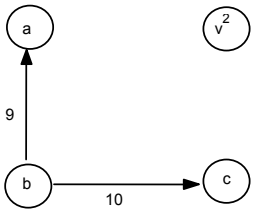
$D^1 = \{ a, b, c, d, e, v^1 \}$

$E^1 = \{(b,a),(b,c),(v^1,d),(d,e),(e,v^1)\}$

$Q^1 = \langle v^1,(v^1,d),d,(d,e),e,(e,v^1),v^1 \rangle$

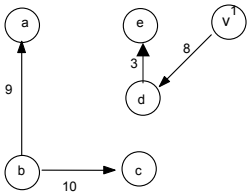


(c) G^2

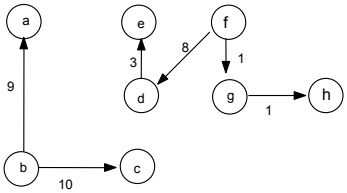


B^2

$D^2 = \{ a, b, c, v^2 \}$ $E^2 = \{ (b,a), (b,c) \}$

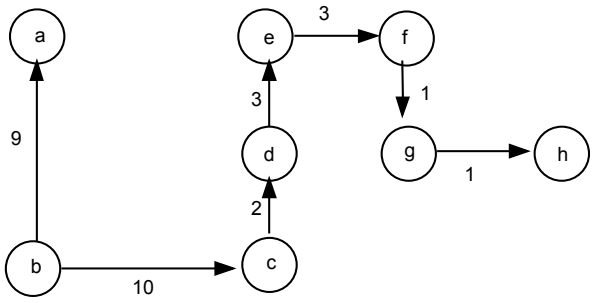


(d) B^1



B^0

figura VI.4



Arborescencia cobertora de peso máximo para el grafo G de la figura VI.4

figura VI.5

Un árbol no orientado en el cual distinguimos un vértice **r** se llama a veces un *árbol enraizado en **r*** y, como vimos anteriormente, existe una correspondencia entre los árboles enraizados y las arborescencias. Esta última afirmación permite a veces utilizar indistintamente los términos arborescencia y árbol, como suele ocurrir cuando se trabaja con árboles en Computación, siempre y cuando se especifique cuál es el vértice distinguido o raíz **r**.

En cuanto a la representación gráfica de los árboles enraizados, estos se suelen dibujar con el vértice raíz **r** en el tope, y el resto de los vértices pendiendo de **r**, con las hojas en la parte inferior de la figura. Este sentido de arriba hacia abajo corresponde a la orientación implícita de los lados.

En base a lo dicho en el punto anterior, a partir de este momento y mientras no se especifique lo contrario, no haremos distinción entre árboles enraizados y arborescencias, siempre que se distinga el vértice raíz.

VI.4 TERMINOLOGÍA: NIVEL, ALTURA, LONGITUD

Sea A un árbol y **r** un vértice distinguido o raíz. Los vértices adyacentes a **r** se llaman sus *hijos* o *sucesores* y recíprocamente, **r** es su *padre* o *predecesor*. Sea B el bosque generado por los vértices de A menos la raíz. Se llama *sub-árbol* de **r** a cada uno de los árboles de B, cuyas raíces son los hijos de **r**. Se dice que dos vértices son hermanos si tienen el mismo padre. Un vértice sin sucesores se llama *vértice terminal* u *hoja*. Todo vértice que no sea una hoja se llama *vértice interior*. En consecuencia, todos los vértices interiores son raíces de algún sub-árbol no vacío de A.

Vemos entonces que es posible dar una definición equivalente y recursiva para árboles enraizados (extensible a arborescencias de manera natural) de la siguiente manera:

Definición VI.4.1

Un *árbol enraizado* es un grafo conexo A = (V,E) en el cual:

a) Hay un único vértice distinguido **r**, llamado raíz del árbol

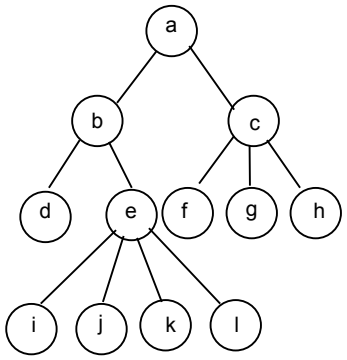
b) El resto de los vértices (excluyendo la raíz) están particionados en $m \geq 0$ subconjuntos (disjuntos) $V_1, V_2, \ldots, V_m$ tales que los subgrafos inducidos por cada V_i, $i = 1, \ldots, m$ son a su vez árboles enraizados. Estos subgrafos inducidos se llaman *sub-árboles* de **r** , y se denotan $A_1, A_2, \ldots, A_m$. La raíz de cada sub-árbol A_i es el único vértice en A_i que es adyacente a **r** en A, es decir, el único hijo de **r** en A_i. El conjunto de lados E es la unión disjunta de los lados en los sub-árboles $A_1, \ldots, A_m$ más los lados incidentes en **r**.

Esta nueva definición excluye la existencia de ciclos desde el momento que vemos que no hay lados entre sub-árboles de un mismo vértice.

Trabajaremos ahora con los niveles de los vértices en un árbol. Como se definió en la Sec. V.5, el nivel $\eta(x)$ de un vértice x en un grafo es el largo máximo de un camino elemental terminado en x. Si A es un árbol enraizado, los caminos más largos parten de la raíz, por lo que la raíz tiene nivel 0, y todos los demás vértices tienen nivel igual al de su padre más 1, o lo que es lo mismo, para todo vértice no terminal con nivel i, el nivel de sus hijos es i+1.

Una forma de obtener los niveles de los vértices en un árbol A es utilizando el algoritmo de búsqueda en amplitud BFS, comenzando por la raíz **r**, a la cual se le asigna nivel(r) = 0, y asignando nivel(x) + 1 a todos los vértices visitados desde un vértice x. De esta misma manera obtenemos la longitud o largo del camino (o cadena) desde la raíz hasta cada vértice, que será igual a su nivel. Definimos la *Altura de un vértice x* como la longitud del camino más largo desde x hasta

una hoja. Se define la *Altura de un Árbol* como la longitud del camino más largo en el árbol, es decir, la altura de la raíz, o lo que es lo mismo, el máximo de los niveles de los vértices del árbol.



Árbol enraizado con raíz a

Hojas: d, f, g, h, i, j, k, l.
Nodos interiores: a, b, c, e.
Raíces de los sub-árboles de a: b, c.
Nivel del nodo f: 2
Altura del nodo b: 2
Altura del árbol: 3

figura VI.6

Consideremos a continuación un árbol A de altura **a** en el cual cada vértice puede tener a lo sumo **g** hijos. Se denota $N_a(g)$ al número máximo de vértices que puede tener un árbol con las características anteriores. Este número $N_a(g)$ se puede determinar a partir del número máximo de vértices que puede haber en cada nivel del árbol, así:

Nivel 0: 1 vértice
Nivel 1: máximo g vértices
Nivel 2: máximo $g.g = g^2$ vértices
…
Nivel a: máximo $g^{a-1}.g = g^a$ vértices

de donde $N_a(g) = \sum_{i=0}^{a} g^i = (g^{a+1}-1)/(g-1)$

Un árbol en el cual todas las hojas están en el máximo nivel y todos los vértices interiores tienen el máximo número de hijos se llama un *Árbol completo*.

Es de especial interés el caso en el que g=2, donde $N_a(g) = \sum_{i=0}^{a} 2^i = (2^{a+1}-1)$

Podemos observar que en un árbol A con número máximo de hijos g=2 y k hojas debe tener un camino de longitud al menos $\log_2(k)$. Aplicando un razonamiento similar al anterior también se puede determinar que la altura mínima que puede tener un árbol de n vértices, donde cada uno puede tener a lo sumo 2 hijos es $[\log_2 n]$.

Estos árboles con g=2 han sido estudiados y utilizados ampliamente en computación, en especial los llamados "árboles binarios" que veremos más adelante, debido a las enormes facilidades que ofrecen para el manejo de información. Así mismo, los resultados anteriores han facilitado la obtención de muchos otros en el estudio de la complejidad de algoritmos que trabajan con árboles binarios.

VI.5 ÁRBOLES Y ARBORESCENCIAS PLANAS

Existe una clase de grafos que posee una importante propiedad: se puede dibujarlos sobre un plano (papel, por ejemplo) de forma tal que los vértices sean puntos en el plano y dos lados cualesquiera no tengan más intersecciones que los

vértices que puedan tener como extremos comunes. A estos grafos se les llama *Grafos Planares* . Es evidente que no todos los grafos son planares, como por ejemplo no lo es K_5, ni el grafo de la figura VI.7

Un mismo grafo puede tener más de una representación planar, por ejemplo las dos representaciones planares de K_4 que se muestran en la figura VI.8.

Los árboles son grafos planares debido a la existencia de una única cadena entre cualquier par de vértices.

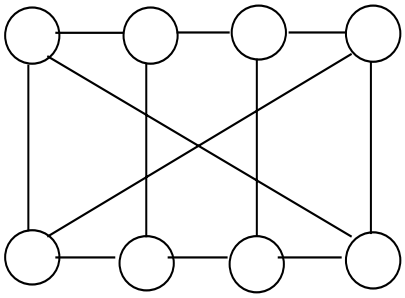
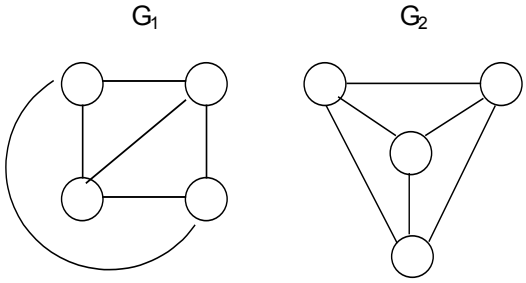


figura VI.7

Dado un árbol A, se puede distinguir diferentes representaciones planares de A: A_1, A_2,… , como se muestra en la figura VI.9 para el árbol A con V={a,b,c,d,e} y E={{a,b}, {a,c}, {c,d}, {c,e}}.

Lo dicho anteriormente para árboles en general, obviamente aplica para árboles enraizados y arborescencias.



Dos representaciones planares del mismo grafo

figura VI.8

VI.5.1 ÁRBOLES ORDENADOS

Si observamos ahora los árboles A_3 y A_4 de la figura VI.9, y distinguimos en ellos el vértice c como raíz, podemos diferenciar uno del otro según las posiciones relativas de sus sub-árboles en cada una de las representaciones planares y podemos entonces numerarlas de izquierda a derecha, de forma que el 1º sub-árbol de c en A_3 es distinto del 1º sub-árbol de c en A_4.

Proposición VI.5.1.1

Toda representación planar de un árbol enraizado (o arborescencia) puede ser caracterizada asignando a cada vértice no terminal una lista ordenada o permutación de sus hijos (raíces de sus sub-árboles).

Para los árboles A_3 y A_4 de la figura VI.9 tenemos:

A_3: P(c) = <a, d, e>, P(a) = <b>

A_4: P(c) = <d, e, a>, P(a) = <b>

Definición VI.5.1.1

Llamamos *Árbol Ordenado* un árbol enraizado donde cada vértice no terminal tiene asociada una permutación de sus hijos.

Si nos remitimos a la definición VI.4.1 de árboles enraizados, vemos que un árbol ordenado es uno en el que es relevante el orden en que aparecen listados los sub-árboles en tal definición.

En este sentido, los árboles A3 y A4 con las permutaciones dadas arriba son dos árboles ordenados diferentes y sin embargo, si los consideramos simplemente como árboles enraizados, son el mismo.

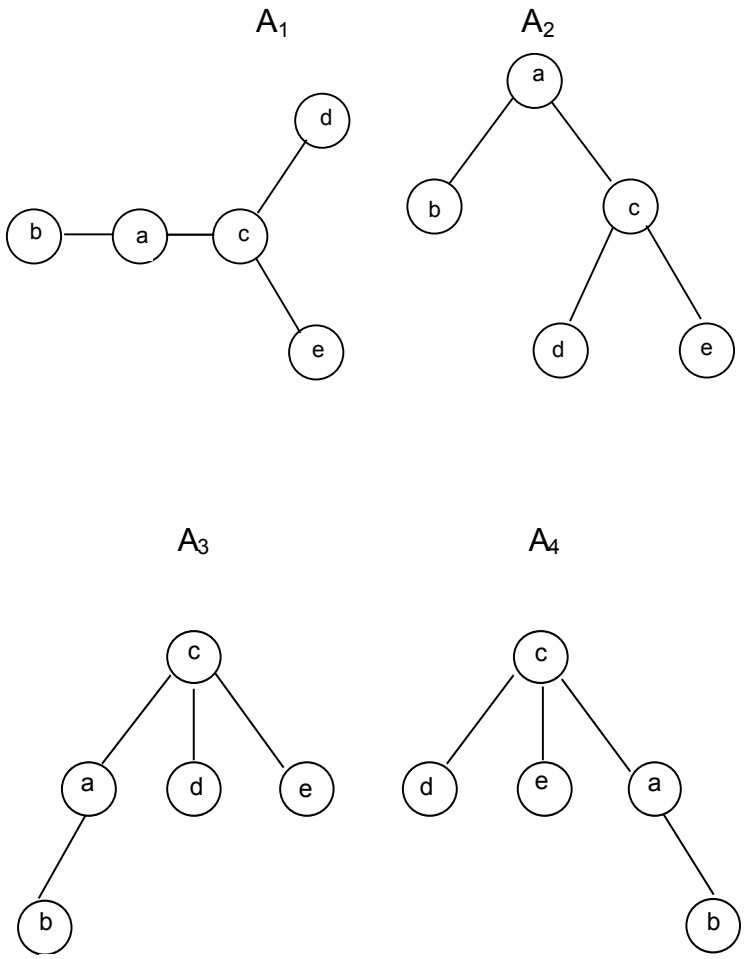


figura VI.9

Vemos entonces que la representación planar de árboles nos lleva de forma natural a los árboles ordenados. Pero no es sólo esto, la utilización de árboles en computación y en consecuencia, la necesidad de representarlos en el computador impone igualmemte la necesidad de ordenar los hijos, debido a la organización secuencial de la memoria y a la ejecución secuencial de las instrucciones.

VI.5.1.1 REPRESENTACIÓN DE EXPRESIONES ALGEBRAICAS

Como ejemplo de utilización de árboles ordenados, consideremos la siguiente expresión algebraica:

(1) $((a + b) * 5) / (9 - (f * \log h)$

En esta expresión aparecen operadores de uno y de dos operandos, y en el caso de los segundos, en particular de los operadores (/) y (–), es importante el orden de los operandos.

Una expresión como ésta puede ser fácilmente representada haciendo uso de árboles ordenados, donde las hojas son los operandos y los operadores están en los vértices internos y en la raíz. El orden de los sub-árboles corresponde al orden de los operandos. Tal representación es la que se muestra en la figura VI.10 para la expresión (1).

VI.5.2 ÁRBOLES BINARIOS

Como ya hemos mencionado antes, existe un tipo de árbol al que se ha dedicado mucho estudio debido a su enorme utilización en computación, son estos los árboles binarios.

Definición VI.5.2.1

Un *Árbol Binario* es un árbol enraizado en el cual cada vértice tiene a lo sumo dos hijos, los cuales se denominan *hijo izquierdo* e *hijo derecho* . De igual manera, los sub-árboles de cada vértice se denominan *sub-árbol izquierdo* y *sub-árbol derecho* respectivamente.

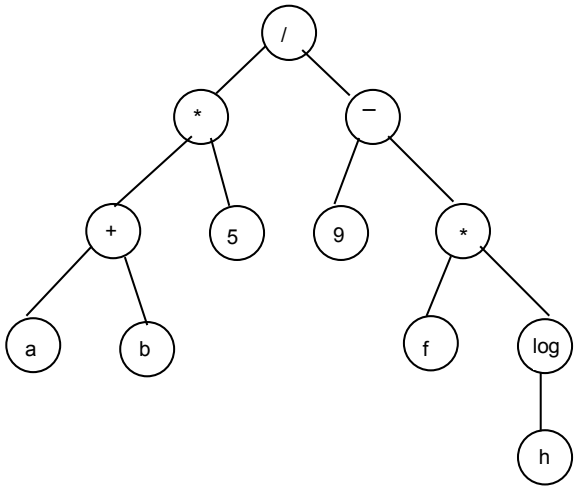


figura VI.10

Es importante notar que un árbol binario no es lo mismo que un árbol ordenado con a lo sumo dos hijos por vértice, pues en este último, si un vértice tiene un solo hijo, éste es simplemente el primero (y único); en el caso de un árbol binario se debe especificar si este hijo es el derecho o el izquierdo, lo cual diferencia un árbol de otro. Los árboles A_1, A_2 y A_3 de la figura VI.11 son iguales si los vemos como simples árboles enraizados, sin embargo, como árboles ordenados A_1 es distinto de A_2 y A_3 que son iguales, y como árboles binarios los tres son distintos.

Los árboles binarios han sido utilizados como estructuras para almacenamiento, ordenamiento y recuperación de información, así como para la representación y solución de problemas de toma de decisiones, entre otros.

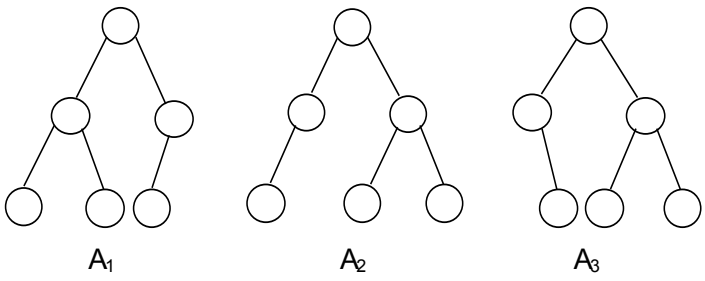


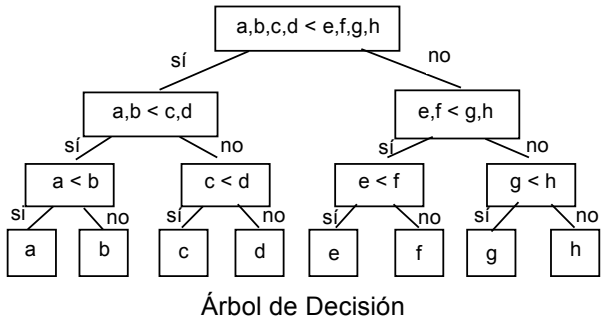
figura VI.11

VI.5.2.1 ÁRBOLES DE DECISIÓN

Son muchos los casos en los cuales la resolución de un determinado problema se facilita si se logra subdividirlo adecuadamente en problemas menores y más sencillos (estrategia conocida como "Divide-and-Conquer").

Consideremos un pequeño ejemplo. Se tiene 8 monedas {a, b, c, d, e, f, g, h} de las cuales una es falsa y de menor peso que las demás. Si disponemos de una balanza podemos determinar cuál es la moneda falsa de la siguiente manera. Se coloca la mitad de las monedas de un lado de la balanza y la otra mitad del otro lado. Del lado que resulte menos pesado sabremos que está la moneda falsa. Tomamos las monedas de este lado y nuevamente las repartimos equitativamente a ambos lados de la balanza. Una de las dos monedas del lado más liviano es la falsa, luego, volviéndolas a pesar, tendremos la respuesta a nuestro problema.

El árbol de la figura VI.12 es una representación del método empleado para resolver el problema. En este árbol cada vértice interno representa una decisión o comparación y dependiendo del resultado de ésta se puede pasar al hijo izquierdo o al derecho sucesivamente hasta llegar a las hojas que representan cada una de las respuestas posibles al problema original.



Árbol de Decisión

figura VI.12

Este tipo de árboles se conoce con el nombre de *Árbol de Decisión*.

Otro ejemplo frecuente de uso de este tipo de árboles es en el ordenamiento de n números distintos. Puesto que cada una de las n! permutaciones de estos números podría ser el ordenamiento correcto, el árbol de decisión asociado al problema, en el cual cada vértice representa los ordenamientos posibles luego de las comparaciones realizadas para llegar a él y los lados el resultado de la comparación entre dos claves, deberá tener n! hojas, si se excluyen comparaciones innecesarias o extemporáneas.

A partir del resultado visto en VI.4 podemos concluir que para llegar a un ordenamiento son necesarias como mínimo log(n!) comparaciones, lo cual es de orden nlogn [AHO et al., " Data Structutres and Algorithms", pp 284-286]. Este resultado es básico en el análisis de los diferentes métodos de ordenamiento por comparación, pues establece una cota inferior para el mínimo de comparaciones que tendrá que realizar cualquiera de estos algoritmos.

VI.5.2.2 ÁRBOLES BINARIOS PERFECTAMENTE BALANCEADOS

Definimos un *Árbol binario Perfectamente Balanceado* o *Equibrado* como un árbol binario en el cual para cada vértice, el número de vértices en su sub-árbol izquierdo y el número de vértices en su sub-árbol derecho difieren a lo sumo en una unidad.

Un árbol perfectamente balanceado con n vértices será un árbol de áltura mínima en el sentido que no puede existir otro árbol binario con el mísmo número de vértices y de altura menor. En un árbol de este tipo todas las hojas estarán en los dos últimos niveles, es decir, si a es la altura del árbol, entonces las hojas estarán todas en los niveles a y a–1.

La construcción de un árbol perfectamente balanceado con n vértices se puede expresar recursivamente de la siguiente manera:

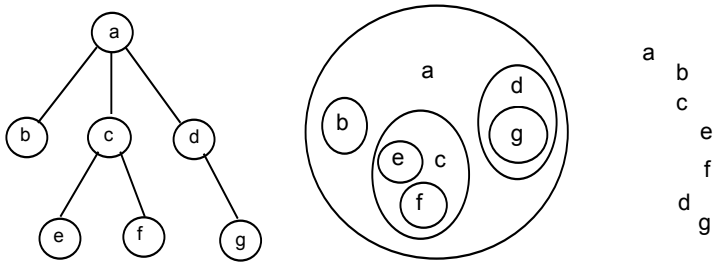
Algoritmo de construcción de un árbol Perfectamente Balanceado:
{Entrada: n vértices.
Salida: el árbol perfectamente balanceado con los n vértices }.
Comienzo
Si n>o entonces:

1) Tomar un vértice como raíz.
2) Construir el sub-árbol izquierdo con $n_i = n$ **div** 2 vértices, siguiendo estos mismos pasos.
3) Construir el sub-árbol derecho con $n_d = n - n_i - 1$ vértices, siguiendo estos mismos pasos.

Fin.

VI.6 REPRESENTACIÓN DE ÁRBOLES Y ARBORESCENCIAS

Como grafos que son, tanto los árboles como las arborescencias admiten cualquiera de las representaciones para grafos presentadas en el capítulo II. Sin embargo, por sus características especiales, es interesante estudiar nuevas formas de representación que se adaptan mejor a estas estructuras, sobre todo para los árboles ordenados en los cuales se conoce el número máximo de hijos por vértice y para árboles binarios.



Distintas representaciones gráficas para un mismo árbol

figura VI.13

En cuanto a la representación gráfica, ya hemos venido utilizando la convención de colocar la raíz en el tope para los árboles enraizados y arborescencias. Sin embargo, los árboles enraizados por su estructura particular son suceptibles de ser representados gráficamente de varias otras formas que nada tienen que ver con la usual representación de grafos, como se muestra en la figura VI.13. Más aún, el método usual utilizado para numerar capítulos y secciones de capítulos en los libros, el mismo que usamos en este libro, es otra manera de representar árboles como se muestra en la figura VI.14.

1: a
- 1.1: b
- 1.2: c
  - 1.2.1: e
  - 1.2.2: f
- 1.3: d
  - 1.3.1: g

Numeración como índice de los vértices del árbol de VI.13

figura VI.14

Dedicaremos ahora nuestra atención a nuevas formas de representación en el computador.

VI.6.1 REPRESENTACIÓN UTILIZANDO LISTAS

VI.6.1.1 REPRESENTACIÓN HOMOGÉNEA ASCENDENTE

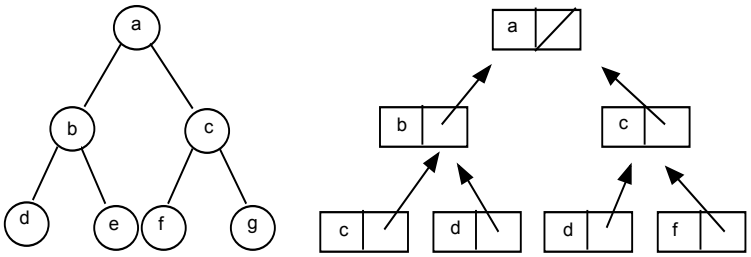
Este tipo de representación para arborescencias, extensible a árboles enraizados, se basa en el hecho de que para todo nodo x que no sea la raíz: $d^-(x) = 1$. De esta manera podemos representar cada nodo como un registro con dos campos: uno para el nombre o etiqueta del nodo y el otro para almacenar un apuntador a su nodo antecesor o padre. Evidentemente, el apuntador al padre de la raíz es NULO.

La declaración para esta estructura sería:

```
tipo  apun_ha = apuntador a  nodo_ha;
tipo  nodo_ha = Registro
                    valor:T;
                    padre: apun_ha
                Fin
```

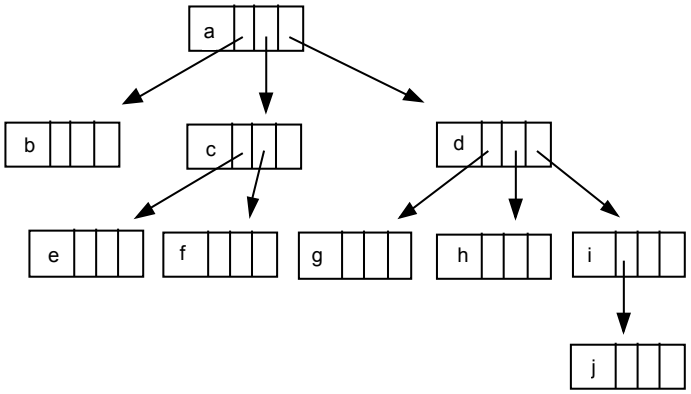
Se le llama homogénea a este tipo de representación por utilizar un único tipo de registro para representar toda la estructura. Una ventaja de esta representación es que es independiente del número de hijos que puedan tener los nodos.

Por otro lado, una desventaja es que esta representación no puede reflejar ningún tipo de orden entre los hijos o sub-árboles de un nodo, por lo que es necesario descartarla cuando el orden es propiedad relevante del árbol que se quiere representar.



Representación homogénea ascendente

figura VI.15



Representación homogénea descendente de un árbol
con máximo 3 hijos por nodo

figura VI.16

VI.6.1.2 REPRESENTACIÓN HOMOGÉNEA DESCENDENTE

Podemos mencionar dos tipos de representaciones homogéneas descendentes, es decir, que representan la relación de un elemento o nodo hacia sus descendientes.

La primera de estas representaciones requiere de un conocimiento previo del número máximo de hijos por nodo. Se puede entonces representar todos los nodos del árbol utilizando una misma estructura base que consiste en un registro con un campo de información para almacenar el valor del nodo, y tantos campos de tipo apuntador como el máximo de hijos que puedan tener los nodos en el árbol a representar.

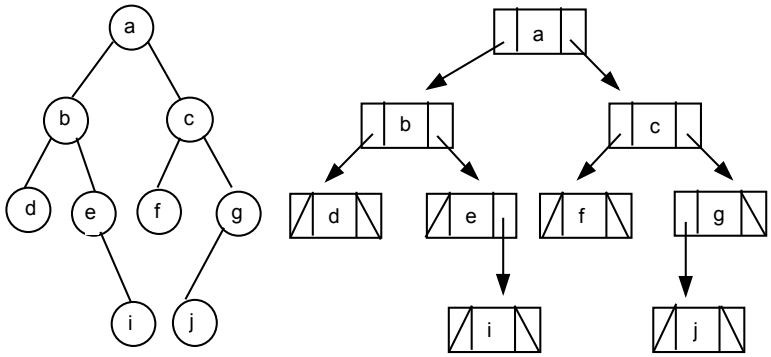
Esta representación permite tomar en consideración el orden relativo entre los hijos si ello fuese necesario. La mayor desventaja que presenta es la cantidad de memoria desperdiciada en apuntadores en todos aquellos nodos con menos hijos que el máximo permitido, en particular en las hojas. Vale la pena sin embargo mencionar que en el caso de árboles con a lo sumo 2 hijos por nodo esta desventaja es mínima, pues pensar en menos campos apuntadores, correspondería ya a las listas lineales. Por esta última razón es esta representación la usualmente seleccionada para este tipo de árboles.

En el caso particular de los árboles binarios, esta es la representación estándar y a los dos apuntadores a los hijos se les distingue como apuntador al hijo izquierdo y apuntador al hijo derecho (ver figura VI.17).

La declaración para estas estructuras sería:

```
tipo  apun_n = apuntador a  árbol_n;
tipo  árbol_n = Registro
                  valor:T;
                  apuntadores: Arreglo [1.. n] de  apun_n
                fin;

tipo  apun2 = apuntador a  binario;
tipo binario = Registro
                  valor:T;
                  izquierdo, derecho: apun2
                fin;
```



Representación homogénea descendente
de árboles binarios

figura VI.17

La segunda de las representaciones homogéneas descendentes que presentaremos, no requiere del conocimiento previo del número máximo de hijos por nodo. Este tipo de representación utiliza un único tipo de registro de tres campos: uno para el valor del nodo, y dos campos de tipo apuntador, uno con la dirección del primero de una lista de los hijos del nodo, y el segundo con la dirección del siguiente hermano del nodo dentro de la lista de hijos de su padre.

La declaración para esta estructura sería:

```
tipo  apun = apuntador a  nodo;
tipo  nodo = Registro
                valor:T;
                hijos, sig_hermano: apun
              fin;
```

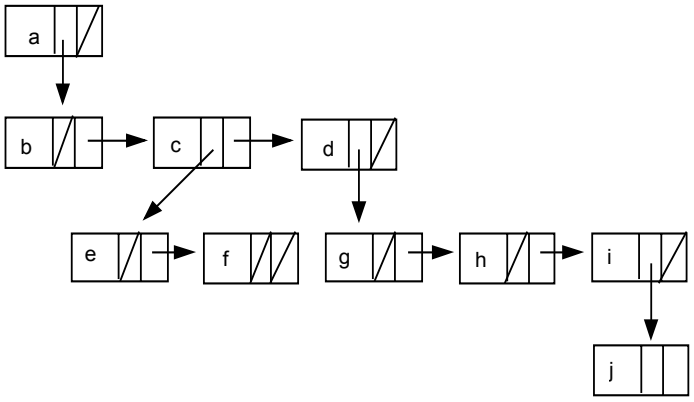
La figura VI.18 muestra la representación del mismo árbol de la figura VI.16 usando esta nueva estructura.

Este último tipo de representación se conoce como *representación de árboles usando árboles binarios.* Es fácil comprender el por qué, basta con cambiar los nombres de los campos "hijos" y "sig_hermano" por "izquierdo" y "derecho" respectivamente (ver figura VI.19).

VI.6.2 REPRESENTACIÓN USANDO ARREGLOS

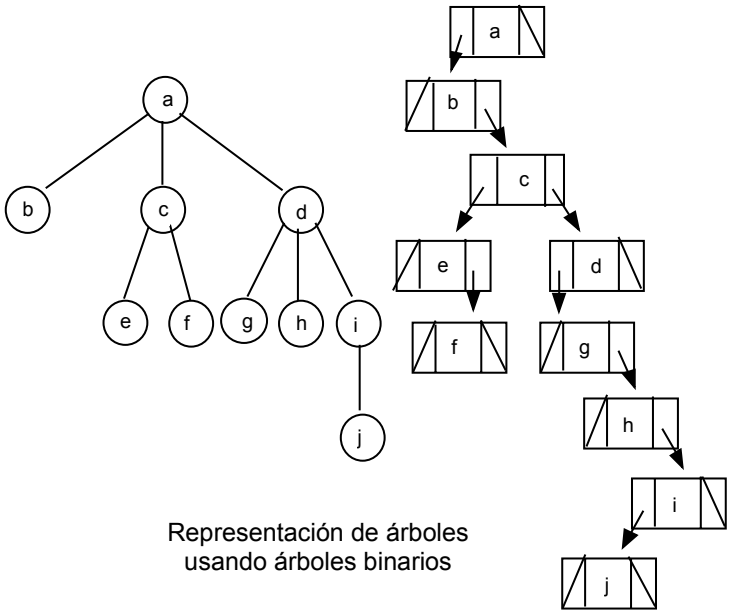
La primera de las representaciones usando arreglos que mencionaremos es equivalente a la homogénea ascendente usando listas. Utiliza dos arreglos, uno para el valor del nodo y otro para guardar las posición de su nodo padre. De esta manera podemos almacenar árboles con cualquier número de hijos. Como desventajas presenta las mismas de la representación equivalente usando listas, además de las ya conocidas asociadas a la declaración estática del tipo arreglo.

Limitaremos las siguientes representaciones con arreglos a árboles con no más de 2 hijos por nodo, en consideración a la cantidad de memoria, que, en otros casos, quedaría frecuentemente sin uso y desperdiciada.



Representación homogénea descendente del árbol de la figura VI.16

figura VI.18



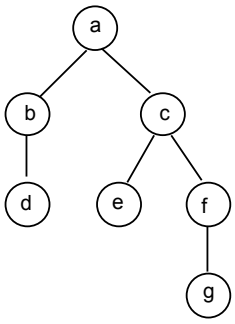
Representación de árboles usando árboles binarios

figura VI.19

La primera de estas representaciones es muy similar a la primera de las representaciones homogéneas descendentes usando listas, en el sentido que utiliza tres arreglos, uno para el valor de cada nodo y los otros dos para almacenar las posiciones de cada uno de los hijos (ver figura VI.20).

La otra representación a la que haremos mención usa un único arreglo unidimensional A[1.. N] en el cual , para cada nodo en posición A[i] sus hijos estarán en las posiciones A[2∗i] y A[2∗i +1], o lo que es lo mismo, para cada nodo en una

posición i, con i≥N/2, su padre será el nodo en la posición i **div** 2. Este tipo de representación, aún pareciendo económica, sólo es recomendable en el caso de árboles completos o de altura mínima, en los cuales las hojas estarán todas en los dos últimos niveles. La última posición no vacía en el arreglo corresponderá a la hoja más a la derecha en el último nivel del árbol.



1	2	3	4	5	6	7
a	b	c	d	e	f	g
2	4	5	0	0	7	0
3	0	6	0	0	0	0

Representación usando Arreglos

figura VI.20

VI.7 RECORRIDOS DE ÁRBOLES

Al utilizar árboles como modelos para representar situaciones o simplemente como una forma de organizar información, a menudo se hace necesario disponer de un método sistemático para recorrer o visitar todos los nodos del árbol con el fin de realizar alguna operación.

Para poder organizar y explicar diferentes recorridos se hace necesario suponer un orden entre los elementos del árbol. En otras palabras, es necesario dotar a cada nodo de un ordenamiento de sus sub-árboles, aún cuando la estructura original no posea dicha organización. Igualmante, visto que todo árbol puede ser considerado como un bosque formado por un único árbol, podemos explicar los tres principales tipos de recorridos, conocidos como *Pre-orden*, *In-orden* u *orden central* y *Post-orden*, referidos a bosques, con lo cual disponemos de métodos más generales y de mayor alcance.

Estos recorridos se pueden expresar recursivamente como sigue:

Pre-Orden:
{ Entrada: Un Bosque B formado por los árboles A_1, …, A_m. Las raíces r_1, …, r_m de cada uno de los árboles.}
Comienzo
(1) Visitar la raíz del primer árbol del bosque B.
(2) Recorrer en Pre-Orden el bosque formado por los sub- árboles de la raíz del primer árbol, si los hay.
(3) Recorrer en Pre-Orden el bosque formado por los árboles restantes, respetando el orden entre ellos.
Fin.

In-Orden u Orden Central:
{ Entrada: Un Bosque B formado por los árboles A_1, …, A_m. Las raíces r_1, …, r_m de cada uno de los árboles.}
Comienzo
(1) Recorrer en In-Orden el bosque formado por los sub-árboles de la raíz del primer árbol, si los hay.
(2) Visitar la raíz del primer árbol del bosque B.
(3) Recorrer en In-Orden el bosque formado por los árboles restantes, respetando el orden entre ellos.
Fin.

Post-orden:
{ Entrada: Un Bosque B formado por los árboles A_1, …, A_m. Las raíces r_1, …, r_m de cada uno de los árboles.}
Comienzo
(1) Recorrer en Post-orden el bosque formado por los sub-árboles de la raíz del primer árbol, si los hay.
(2) Recorrer en Post-orden el bosque formado por el resto de los árboles, respetando el orden entre ellos.
(3) Visitar la raíz del primer árbol del bosque B.
Fin.

Debido a la importancia de los árboles binarios, y a la singular organización de sus sub-árboles, distinguidos como izquierdo y/o derecho, y no primero y segundo, lo cual los distingue de los árboles ordenados, se hace necesario especificar cómo se adaptan los recorridos anteriores a los árboles binarios.

Cuando en un árbol binario no existen sub-árboles, es decir, si un nodo no tiene hijo izquierdo ni derecho entonces "recorrer el sub-árbol" significa "no hacer nada". En caso contrario los recorridos se harán siguiendo los pasos indicados a continuación, según el caso:

Pre-Orden:
{Entrada: Un árbol binario A, el nodo r, raíz de A.}
Comienzo
(1) Visitar la raíz.
(2) Recorrer el sub-árbol izquierdo en Pre-orden.
(3) Recorer el sub-árbol derecho en Pre-orden.
Fin.

In-Orden u Orden Central:
{Entrada: Un árbol binario A, el nodo r, raíz de A.}
Comienzo
(1) Recorrer el sub-árbol izquierdo en In-orden.
(2) Visitar la raíz.
(3) Recorrer el sub-árbol derecho en In-orden.
Fin.

Post-orden:
{Entrada: Un árbol binario A, el nodo r, raíz de A.}
Comienzo
(1) Recorrer el sub-árbol izquierdo en Post-orden.
(2) Recorrer el sub-árbol derecho en Post-orden.
(3) Visitar la raíz.
Fin.

Estos recorridos son de especial interés pues pueden ser aplicados a árboles en general, ya que todos pueden ser representados como árboles binarios.

Es interesante notar que los recorridos del bosque o árbol original y del árbol binario asociado se corresponden uno a uno.

Recorridos del bosque de VI.21(a):

Pre-Orden:	a b c d e f g h i j k l.
In-Orden:	b d e c f a h j k l i g.
Post-orden:	e d f c b l k j i h g a.

Recorridos del árbol binario de VI.21(b)

Pre-Orden:	a b c d e f g h i j k l.
In-Orden:	b d e c f a h j k l i g.
Post-orden:	e d f c b l k j i h g a.

VI.8 ÁRBOLES DE JUEGO

Consideremos algún juego como por ejemplo damas, ajedrez o la "vieja", en el cual intervienen dos jugadores. En juegos como estos, tiene cierta ventaja el jugador con capacidad de prever más jugadas. Suponiendo que el juego tiene un número finito de posibles jugadas y que existe alguna regla que asegura que el juego eventualmente termina, las secuencias de jugadas posibles a partir de una situación o estado inicial, pueden ser representadas utilizando un árbol enraizado o arborescencia llamado *Árbol de Juego* .

La raíz de este árbol representa el estado inicial del juego; los lados incidentes en la raíz representan las posibles jugadas o movimientos del Primer jugador, y los nodos del primer nivel corresponden a los posibles estados del juego luego de cada uno de estos movimientos; los lados entre los nodos del primer nivel y los del segundo representan las jugadas que puede realizar el Segundo jugador según el estado o situación en que encuentre el juego, y así sucesivamente. Los lados que se encuentran entre nodos de un nivel par y los del siguiente corresponderán a jugadas del Primer jugador, y los lados entre nodos de un nivel impar y el siguiente, corresponderán a jugadas del Segundo jugador. Las hojas del árbol representan estados o situaciones a partir de los cuales el juego no puede continuar, bien porque alguno de los jugadores gana, o porque termina el juego con un empate.

El árbol de la figura VI.22 corresponde al juego de NIM en el cual, a partir de un número inicial k de palitos, cada jugador va tomando x palitos, con $1 \leq x \leq 3$, siempre que el número kr de palitos restantes sea mayor que 3, y $1 \leq x \leq kr$ en caso contrario. El jugador que toma el último palito pierde. A cada nodo se asocia un número, igual al número de palitos restantes en ese estado del juego.

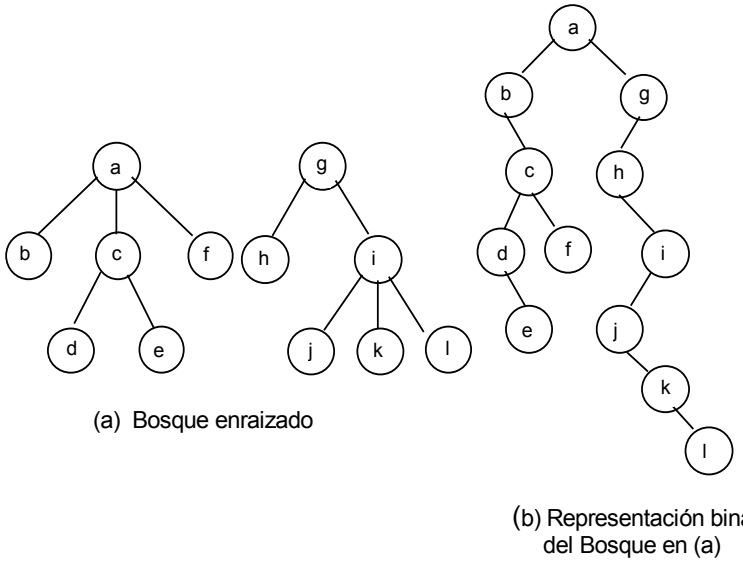


figura VI.21

A cada nodo del árbol se le asocia además un valor, comenzando por las hojas. Estos valores serán: 1, -1 y 0, según que la hoja represente un juego ganado, perdido o empatado para el Primer jugador. Los valores se propagarán luego hacia el resto de los nodos de acuerdo a una estrategia conocida como *min-max*. Note que mientras que el Primer jugador busca ganar, el Segundo busca que el Primero pierda. Esto último en términos de los valores asociados a los nodos significa que el primer jugador hará movimientos que le conduzcan a estados con el máximo valor posible, mientras que el Segundo hará todo lo contrario.

El método *min-max* consiste en asignar a cada nodo no terminal el máximo de los valores de sus hijos, cuando los lados entre ellos corresponden a jugadas del Primer jugador (nodos en nivel par), y el mínimo cuando corresponden a jugadas del Segundo. Esto es fácil de lograr hacieno un recorrido del árbol en Post-orden.

Un jugador tiene una estrategia de juego ganadora, si a partir de su primer turno puede siempre hacer una jugada que le garantice el triunfo, independiente de las jugadas de su adversario. En términos de los valores asociados a los nodos, esto se expresa como sigue: si la raíz tiene valor 1, el Primer jugador tiene una estrategia ganadora; si tiene valor -1, el Segundo jugador tiene una estrategia ganadora; si tiene valor 0, lo mejor que puede hacer el Primer jugador es lograr un empate.

En el árbol de juego de la figura VI.22, el valor 1 corresponde a las hojas en niveles pares y -1 a las hojas en niveles impares (en este juego siempre hay un ganador, por lo que no se asigna el valor 0 a ninguna hoja). Propagando los valores hasta la raíz de acuerdo al método min-max, podemos ver que el juego tiene una estrategia ganadora para el segundo jugador. ¿Qué ocurriría si el juego se iniciara con 6 palitos? ¿Y con 4?

Note sin embargo, que el árbol correspondiente a este juego puede llegar a ser inmensamente grande a medida que aumenta el número de palitos con que se da inicio al juego, es decir, el número en la raíz. En la práctica, incluso juegos

sencillos como la "vieja", suelen dar lugar a árboles sumamente grandes. Esta situación obviamente se repite para la mayoría de los juegos de verdadero interés, en los cuales a veces incluso utilizando un computador es imposible examinar en un tiempo "prudencial" todas las ramas posibles.

La idea de árboles de juego que acabamos de explicar, donde los nodos toman valores 1, -1 y 0, se puede generalizar a árboles en los cuales se deben asignar otros valores a los nodos. Estos valores, que llamaremos *beneficio*, representan un estimado de la probabilidad de que una determinada jugada resulte en un triunfo para el primer jugador. Esta generalización es útil para árboles grandes, como los que mencionamos antes, los cuales son imposibles de examinar en su totalidad.

La mayoría de los juegos que se juegan con el computador generan árboles muy grandes, por lo que en base a lo dicho arriba, lo que se hace es simular el juego hasta un número predefinido, llamado *paso*, de movimientos sub-siguientes.

En general, ocurrirá que las hojas de un árbol de juego, dado un paso pre-fijado, son ambiguas, ya que al no representar un juego terminado, no corresponden a un triunfo, ni a un empate, ni a un juego perdido. Para poder asignar valores a los nodos se deberá disponer entonces de una función, que dependerá del juego, que evaluará el beneficio que puede aportar una determinada jugada al primer jugador. De esta manera, nuevamente aplicando el método min-max, se podrán propagar los valores hasta la raíz. Estas funciones por ser estimaciones, no necesariamente garantizan un triunfo o empate. La figura VI.23 muestra un árbol para el juego de la "vieja" con un paso de 2, con la primera jugada de las X ya realizada (por lo tanto a efectos del árbol, el Primer jugador son las O). La función que se utiliza es la siguiente: número de filas, columnas y diagonales que permanecen abiertas para el primer jugador (las O), menos las que permanecen abiertas para el contrario.

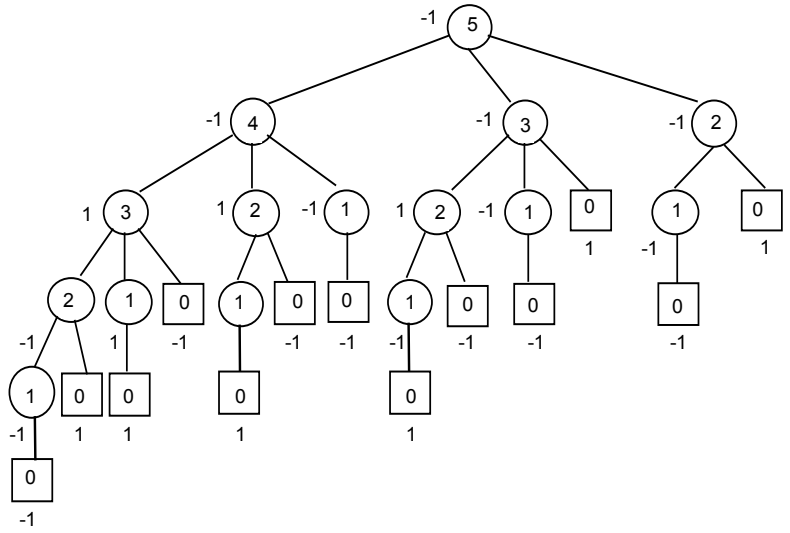


figura VI.22

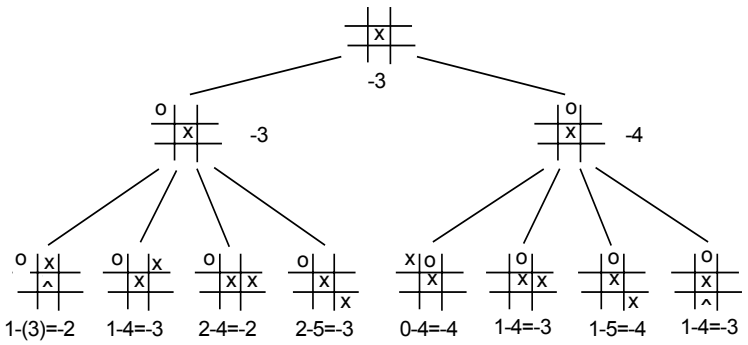


figura VI.23

Observe que en este ejemplo, si el jugador X se encuentra ante la situación de la derecha en el árbol, la función le indica que escoja una de las jugadas con valor -4 y sin embargo, la segunda jugada posible que se le presenta, con un valor de -3, es una jugada con una estrategia ganadora para X. Mientras mayor sea el paso, mejor será la información que proporcionan los valores de los nodos, pero para la escogencia del paso se debe tomar en cuenta la capacidad y velocidad de la computadora, lo que constituye un fuerte factor limitante.

Afortunadamente existe un método que se utiliza en combinación con el min-max y que permite escoger un mayor paso, al permitir descartar de antemano algunas ramas. Este método es conocido como *Poda Alfa-Beta* y se puede utilizar siempre que el paso sea mayor o igual a 2. Para ver mejor cómo funciona lo ilustraremos con el ejemplo de la figura VI.24

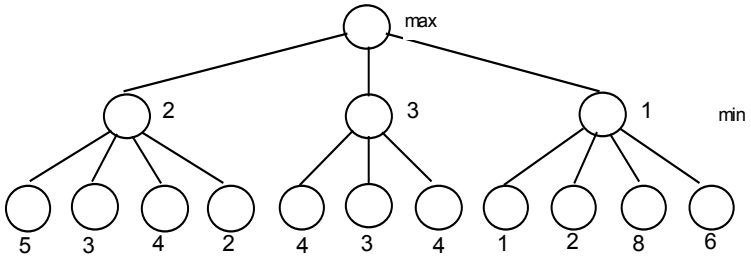


figura VI.24

Supongamos que el Primer jugador analiza las posibles respuestas de su contrincante de izquierda a derecha. Luego de haber examinado las respuestas a sus dos posibles movimientos, al analizar el tercero encuentra que su contrincante puede lograr que solo obtenga un beneficio de 1, con esto sabe que esa jugada, la tercera, no le conviene y no necesita continuar examinando las otras posibles respuestas de su adversario a esa jugada. Se dice que las ramas que no van a ser analizadas son *podadas* del árbol.

Este método se puede acelerar más aún,es decir, recortar más ramas, si se puede disponer de algún criterio que permita de alguna manera distinguir mejores de peores jugadas. Si se analizan las posibles jugadas en base a un tal criterio, es posible que resulten podadas un número mayor de ramas. Por esta razón, conviene tratar los árboles de juego como árboles ordenados.

La mayoría de los juegos por computadora utilizan este método de poda y además van evaluando el árbol a medida que se va generando. De esta forma, las ramas podadas en realidad nunca llegan a existir.

VI.9 EJERCICIOS

1. Hallar todas las colecciones maximales de ciclos independientes del grafo siguiente:

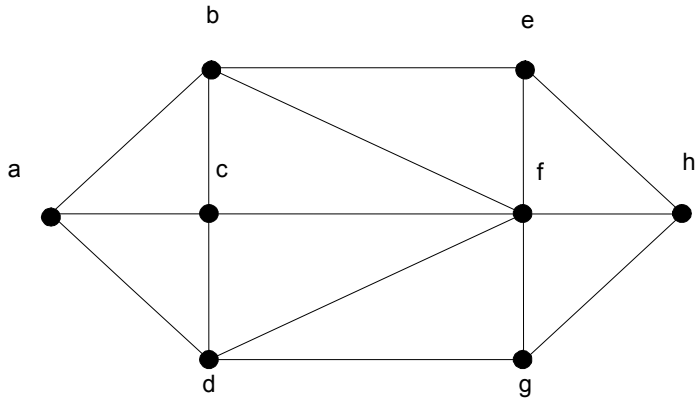


figura VI.25

2. Sea $G=(V,E)$ un grafo no orientado. Demostrar:

a) Si G es conexo entonces G tiene al menos $|E| - |V| + 1$ ciclos distintos.

b) Escriba un algoritmo que encuentre $|E| - |V| + 1$ ciclos distintos de G, si G es conexo.

c) Si G tiene p componentes conexas, entonces G tiene al menos $|E| - |V| + p$ ciclos distintos.

3. Utilizar el algoritmo de Kruskal para construir un árbol mínimo cobertor del grafo $G=(V,E)$ respecto a la función de costos c.

Lado	**Costo**	**Lado**	**Costo**
{a, b}	2	{e, f}	8
{a, c}	1	{e, g}	7
{a, d}	8	{e, h}	1
{b, d}	6	{f, h}	8
{c, d}	4	{g, h}	5
{c, e}	7	{e, i}	7
{c, f}	3	{g, j}	8
{d, f}	2	{h, j}	4
{d, f}	8	{i, j}	8

4. ¿Cómo modificaría el algoritmo de Kruskal para hallar el árbol cobertor de costo máximo?

5. Pruebe que la siguiente afirmación es falsa: Si G es un grafo con una función de costos $c:E\rightarrow\aleph$ tal que G tiene un único árbol mínimo cobertor, entonces G no tiene ciclos.

6. Diseñe un algoritmo que halle las componentes conexas de un grafo basándose en un esquema parecido al algoritmo de PRIM.

7. Sea $T=(V_T,E_T)$ un árbol y v un vértice de T. Podemos inducir una orientación sobre T de forma que el resultado sea una arborescencia T' con raíz v. Decimos que dos vértices $x, y \in V_T$ forman un par descendiente lineal con respecto a T' si existe un camino de x a y o de y a x en T'. Sea $G=(V,E)$ un grafo no orientado, $v \in V$ y T un árbol cobertor de G. Sea T' la arborescencia obtenida al orientar T de forma que v sea raíz de T'. Se dice que T' es un árbol cobertor lineal de G si y sólo si para cada lado $\{x, y\} \in E$, los vértices x, y forman un par descendiente lineal con respecto a T'. Demuestre que dado G y v, existe un árbol cobertor lineal de G con raíz v.

8. En el siguiente grafo:

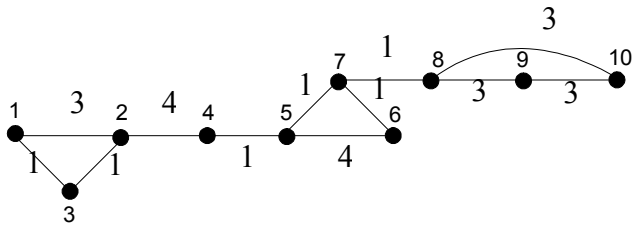


figura VI.26

a) Halle un árbol mínimo cobertor usando Kruskal.

b) Halle un árbol mínimo cobertor usando Prim.

9. La siguiente tabla representa los arcos de un digrafo cuyos vértices son V={a, b, c, d, e, f, g}. Este grafo corresponde a una red de comunicación y los costos de utilización de cada arco aparecen también en la tabla.

N. Inicial	N. Terminal	Costo	N. Inicial	N. Terminal	Costo
a	b	2	e	b	2
b	d	2	f	c	3
c	a	1	f	e	1
d	c	3	g	e	2
d	e	1	g	f	4

Se desea determinar la red que comunique el vértice g con cada uno de los otros vértices de forma que la red resultante tenga el mínimo costo global.

10. Muestre que si $G=(V,E)$ es un grafo no dirigido cuyas aristas tienen costos positivos todos distintos, entonces G tiene un único árbol mínimo cobertor.

11. Sea $G=(V,E)$ un grafo conexo y no orientado y con al menos un istmo e. Sea G_1 y G_2 las dos componentes conexas de $G'=(V, E-\{e\})$, sean T_1 y T_2 los árboles cobertores mínimos de las dos componentes conexas de G'. Sea $T=T_1 \cup T_2 \cup \{e\}$. ¿ T es un árbol mínimo cobertor de G ?.

12. En los campos de golf de LAMUNITA Golf Club se desea instalar un sistema de riego automatizado. Para tal fin se debe instalar una red de tuberías para distribuir agua con fertilizantes para la grama. Los tubos a instalar tienen diferentes diámetros y costos según el terreno por donde deben pasar. Los puntos de riego están marcados con las letras de la **a** a la **f**. Las tuberías posibles de instalar con sus respectivos diámetros y costos son las siguientes:

Tubería	Costo	Diámetro	Tubería	Costo	Diámetro
{a, b}	3	3	{b, d}	5	5
{a, c}	1	5	{b, f}	1	2
{a, d}	1	2	{c, e}	3	5
{a, e}	2	4	{d, e}	2	4
{b, e}	2	2	{d, f}	1	1
{b, e}	2	3	{e, f}	4	3

Sin embargo, para facilitar el paso del agua con fertilizantes, se ha determinado que las tuberías a utilizar deben tener un diámetro mayor o igual a 3. Bajo estas condiciones, determine la red de riego de menor costo. Indique el algoritmo a utilizar y cualquier modificación al mismo si lo considera necesario.

13. El departamento de vialidad del municipio Apié posee un presupuesto limitado para la construcción o reparación de las vías de comunicación del municipio. El municipio Apié consta de diez urbanizaciones y es la intención del departamento proveer a los residentes de las comunidades de un sistema de carreteras que permita interconectar todos los poblados. El cuadro siguiente muestra los costos de conexión directa entre las comunidades, para todas las conexiones posibles:

Conexión	Costo de comunicación
1-2	7
2-3	4
3-7	2
3-8	6
4-5	6
5-6	4
6-7	5
6-10	4
6-9	3
10-9	5
1-9	7

Determine, utilizando sus conocimientos de grafos, cómo deben ser conectados las ciudades de manera que la red de vialidad sea de costo mínimo.

14. Demuestre que el número de hojas de una arborescencia con grado de salida igual a k para todo vértice del grafo (k > 1) es siempre mayor que el número de vértices internos.

15. Encontrar un bosque orientado de peso máximo en el siguiente grafo:

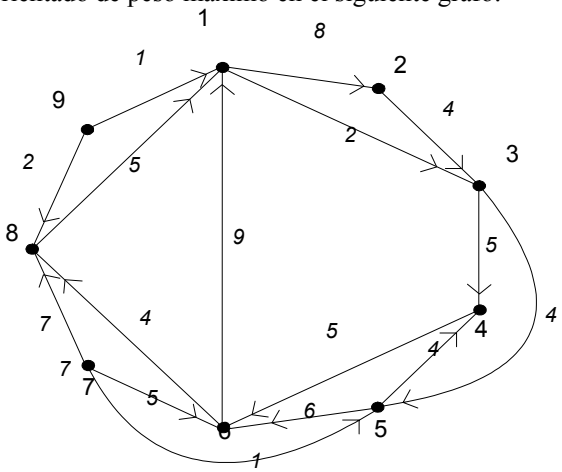


figura VI.27

16. Dadas las siguientes expresiones aritméticas, construya los respectivos árboles planos que las representan:

a) a + b – (c * d)
b) g * j – y / w
c) a – b / c + d
d) w + (x – y) * h
e) x + y / w * h
f) w + (a * d) – (b * c – j)

17. Dada una expresión aritmética con operadores binarios (+, -, *, /):

a) Diseñe el procedimiento que construya el árbol binario correspondiente.
b) Escriba un procedimiento que acepte valores para cada una de las variables en la expresión y luego recorra el árbol para evaluar la expresión al tiempo que imprime la misma notación postfija.

18. Diseñe y codifique una función que reciba un árbol binario y un vértice de ese árbol, y devuelva el nivel en cual se encuentra ese vértice.

19. Se dice que un árbol T_1 es similar a un árbol T_2 si se tiene una de las dos condiciones siguientes:

a) T_1 tiene el mismo dibujo que T_2
b) T_1 es igual a T_2 visto en un espejo.

Diseñe una función que reciba dos árboles T_1 y T_2 y devuelva 1 si los árboles son similares y 0 en caso contrario.

20. Un árbol binario de n hojas es estrictamente binario si tiene $2n$-1 vértices. Diseñe un procedimiento que verifique si un árbol es estrictamente binario.

21. Sea h la altura de un árbol binario T. Determine las dimensiones mínimas de un arreglo en el cual podría almacenarse todo el árbol. ¿Cómo sería el acceso?

22. Suponga que dispone de los siguientes operadores:

- Nivel(T,x): devuelve un entero correspondiente al nivel del vértice cuya clave es x en el árbol T.
- Padre(T,x): devuelve la clave del vértice padre del vértice cuya clave es x.

Usando estos operadores, escriba un procedimiento PAC(T, x, y) que devuelva la clave del vértice correspondiente al primer antecesor común de los vértices cuyas claves son x e y respectivamente.

23. Para una expresión aritmética en preorden en la cual sólo aparecen los operadores binarios (+, -, /, *), diseñe un algoritmo que construya el árbol binario asociado a la expresión.

24. Considere los siguientes recorridos de árboles binarios:

a) (1) Recorrer el subárbol derecho.
(2) Visitar la raíz.
(3) Recorrer el subárbol izquierdo.

b) (1) Visitar la raíz.
(2) Recorrer el subárbol derecho.
(3) Recorrer el subárbol izquierdo.

¿Qué relación existe entre estos dos recorridos y los recorridos pre, pos e inorden?.