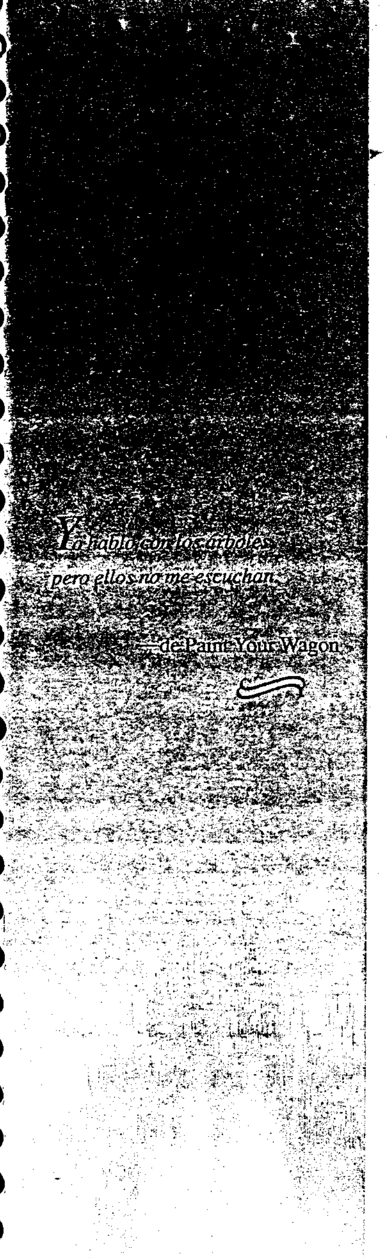


7

ÁRBOLES

Los árboles forman una de las subclases de las gráficas de uso más amplio. En particular, en la computación se hace un uso amplio de los árboles. En este terreno, los árboles sirven para organizar y relacionar los datos en una base de datos (véase el ejemplo 7.1.6). También surgen en problemas teóricos, como el tiempo óptimo para el ordenamiento (véase la sección 7.7).

En este capítulo estudiaremos primero la terminología necesaria. Veremos algunas subclases de árboles (por ejemplo, árboles con raíz y árboles binarios) y muchas aplicaciones de los árboles (por ejemplo, árboles de expansión o generadores, árboles de decisión y árboles de juegos). Nuestro análisis de los isomorfismos de árboles amplía el análisis de la sección 6.6 acerca de los isomorfismos de gráficas.

† Esta sección se puede omitir sin pérdida de continuidad.

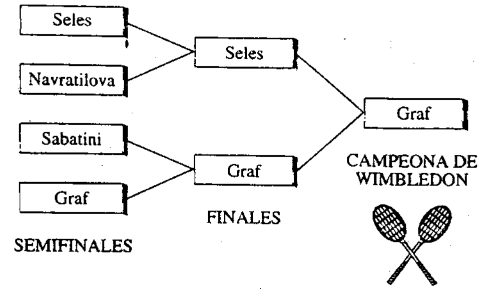


FIGURA 7.1.1 Semifinales y finales en Wimbledon.

7.1 INTRODUCCIÓN

La figura 7.1.1 muestra los resultados de las semifinales y finales de una competencia de tenis en Wimbledon, en la cual participaron cuatro de las mejores jugadoras en la historia del tenis. En Wimbledon, cuando una jugadora pierde, queda fuera del torneo. Las ganadoras continúan jugando hasta que sólo queda una, la campeona. (Esta competencia se llama *torneo de eliminación simple*.) La figura 7.1.1 muestra que en las semifinales, Mónica Seles derrotó a Martina Navratilova y Steffi Graf derrotó a Gabriela Sabatini. Luego jugaron las ganadoras, Seles y Graf, y Graf derrotó a Seles. Steffi Graf, al ser la única jugadora que no fue derrotada, se convirtió en campeona de Wimbledon.

Si consideramos el torneo de eliminación simple de la figura 7.1.1 como una gráfica (véase la figura 7.1.2), obtenemos un **árbol**. Si giramos la figura 7.1.2, se ve como un árbol natural (véase la figura 7.1.3). A continuación damos la definición formal.

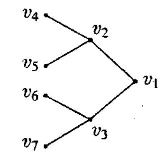


FIGURA 7.1.2 El torneo de la figura 7.1.1 como un árbol.

DEFINICIÓN 7.1.1

Un *árbol (libre)* T es una gráfica simple que satisface: Si v y w son vértices en T, entonces existe un único camino simple de v a w.

Un *árbol con raíz* es un árbol en el cual un vértice particular se designa como la raíz.

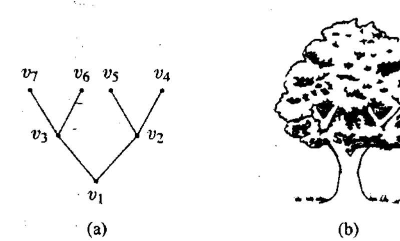


FIGURA 7.1.3 El árbol de la figura 7.1.2 girado (a) comparado con un árbol natural (b).

EJEMPLO 7.1.2

Si designamos a la ganadora como la raíz, el torneo de eliminación simple de la figura 7.1.1 (o la figura 7.1.2) es un árbol con raíz. Observe que si v y w son vértices de esta gráfica, existe un único camino simple de v a w. Por ejemplo, el único camino simple de v_2 a v_7 es (v_2, v_1, v_3, v_7). □

En contraste con los árboles naturales, que tienen su raíz en la parte inferior, en la teoría de gráficas se acostumbra trazar los árboles con raíz con su raíz en la parte superior. La figura 7.1.4 muestra la forma en que se trazaría el árbol de la figura 7.1.2 (con v_1 como raíz). En primer lugar, colocamos la raíz v_1 en la parte superior. Debajo de la raíz y en el mismo nivel, colocamos los vértices v_2 y v_3, los cuales se pueden alcanzar desde la raíz mediante un camino simple de longitud 1. Debajo de estos vértices y en el mismo nivel, colocamos los vértices v_4, v_5, v_6 y v_7, los cuales se pueden alcanzar desde la raíz mediante un camino simple de longitud 2. Continuamos de esta forma hasta trazar todo el árbol. Como el camino simple de la raíz a cualquier vértice dado es único, cada vértice está en un nivel determinado de manera única. Decimos que el nivel de la raíz es el nivel 0. Los vértices debajo de la raíz están en el nivel 1, y así sucesivamente. Así, el **nivel de un vértice** v es la longitud del camino simple de la raíz a v. La **altura** de un árbol con raíz es el número máximo de nivel que aparece en dicho árbol.

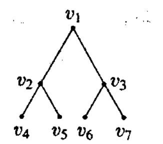


FIGURA 7.1.4
El árbol de la figura 7.1.3(a) con la raíz en la parte superior.

EJEMPLO 7.1.3

Los vértices $v_1, v_2, v_3, v_4, v_5, v_6, v_7$ en el árbol con raíz de la figura 7.1.4 están en los niveles 0, 1, 1, 2, 2, 2, 2, respectivamente. La altura del árbol es 2. □

EJEMPLO 7.1.4

Si designamos e como la raíz del árbol T de la figura 7.1.5, obtenemos el árbol con raíz T' que aparece en la figura 7.1.5. Los vértices $a, b, c, d, e, f, g, h, i, j$ están en los niveles 2, 1, 2, 1, 0, 1, 1, 2, 2, 3, respectivamente. La altura de T' es 3. □

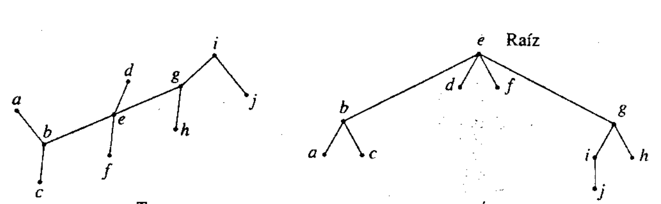


FIGURA 7.1.5 Un árbol T y un árbol con raíz T'. T' se obtiene de T designando a e como la raíz.

EJEMPLO 7.1.5

Con frecuencia, un árbol con raíz se utiliza para especificar relaciones jerárquicas. Cuando un árbol se utiliza de esta manera, si el vértice a está en el siguiente nivel arriba del vértice b y a y b son adyacentes, entonces a está "justo arriba" de b y existe una relación lógica entre a y b: a domina a b o b está subordinado a a de alguna manera. La figura 7.1.6 exhibe un ejemplo de este tipo de árbol, el organigrama de una universidad hipotética. □

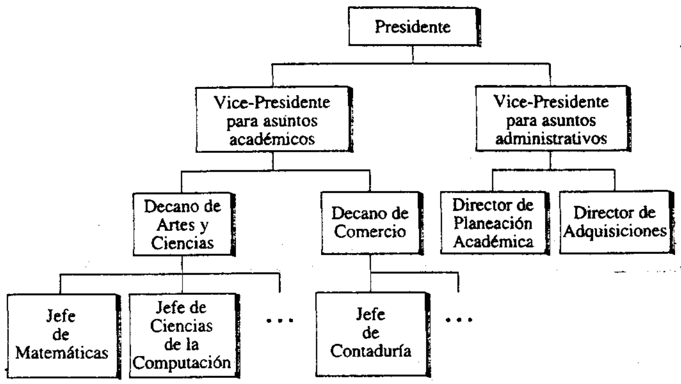


FIGURA 7.1.6 Ejemplo de un organigrama.

EJEMPLO 7.1.6 *Árboles de definición jerárquica*

La figura 7.1.7 es un ejemplo de un **árbol de definición jerárquica**. Tales árboles se utilizan para mostrar las relaciones lógicas entre los registros en una base de datos. [Recuerde (véase la sección 2.7) que una base de datos es una colección de registros controlados por una computadora.] El árbol de la figura 7.1.7 se podría utilizar como modelo para configurar una base de datos para los registros de libros existentes en diversas bibliotecas. □

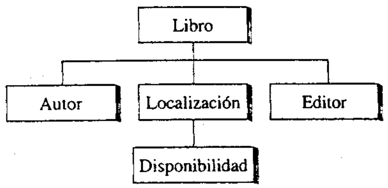


FIGURA 7.1.7 Un árbol de definición jerárquica.

EJEMPLO 7.1.7 *Códigos de Huffman*

La forma más común de representar los caracteres internamente en una computadora es utilizar cadenas de bits de longitud fija. Por ejemplo, ASCII (Código americano estándar para el intercambio de información) representa cada carácter mediante una cadena de siete bits. Algunos ejemplos aparecen en la tabla 7.1.1.

TABLA 7.1.1
Una parte de la tabla ASCII

Carácter	*Código ASCII*
A	100 0001
B	100 0010
C	100 0011
1	011 0001
2	011 0010
!	010 0001
*	010 1010

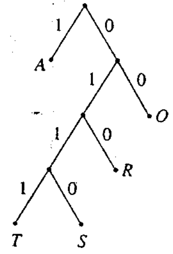


FIGURA 7.1.8
Un código de Huffman.

Los **códigos de Huffman** representan los caracteres mediante cadenas de bits de longitud variable y proporcionan una alternativa para ASCII y otros códigos de longitud fija. La idea consiste en utilizar cadenas cortas de bits para los caracteres de uso frecuente y utilizar cadenas de bits de mayor tamaño para representar caracteres de uso menos frecuente. De esta forma, por lo general es posible representar cadenas de caracteres, como texto y programas, en un menor espacio comparado con el espacio necesario al utilizar ASCII. VCR Plus+, un dispositivo que programa de manera automática una grabadora de videocasetes, utiliza un código de Huffman para generar los números mediante los cuales el usuario elige los programas que desea grabar. Los números se publican en la programación de la televisión.

Un código de Huffman se define fácilmente mediante un árbol con raíz (véase la figura 7.1.8). Para decodificar una cadena de bits, comenzamos en la raíz y nos movemos hacia abajo en el árbol hasta encontrar un carácter. El bit 0 o 1 nos indica si debemos movernos a la izquierda o a la derecha. Por ejemplo, decodifiquemos la cadena

$$01010111. \qquad (7.1.1)$$

Comenzamos en la raíz. Como el primer bit es 0, el primer movimiento es hacia la derecha. A continuación, nos movemos a la izquierda y luego a la derecha. En este momento, encontramos el primer carácter *R*. Para decodificar el siguiente carácter, comenzamos de nuevo en la raíz. El siguiente bit es 1, de modo que nos movemos hacia la izquierda y encontramos el siguiente carácter *A*. Los últimos bits 0111 se decodifican como *T*. Por tanto, la cadena de bits (7.1.1) representan la palabra *RAT*.

Dado un árbol que define un código de Huffman, como la figura 7.1.8, cualquier cadena de bits [por ejemplo, (7.1.1)] se puede decodificar de manera única, aunque los caracteres sean representados mediante cadenas de bits de longitud variable. Para el código de Huffman definido mediante el árbol de la figura 7.1.8, el carácter *A* se representa mediante una cadena de bits de longitud 1, mientras que *S* y *T* se representan mediante cadenas de bits de longitud 4. (*A* se representa como 1, *S* como 0110 y *T* como 0111.) □

Huffman dio un algoritmo (algoritmo 7.1.8) para construir un código de Huffman a partir de una tabla con la frecuencia de aparición de los caracteres que se desea representar, de modo que el código represente cadenas de caracteres en un mínimo espacio, siempre que las cadenas por representar tengan la frecuencia de los caracteres idéntica a la frecuencia de los caracteres en la tabla. Una demostración de que el código construido es óptimo aparece en [Cormen].

ALGORITMO 7.1.8 ***Construcción de un código de Huffman óptimo***

Este algoritmo construye un código de Huffman óptimo a partir de una tabla que contiene la frecuencia de aparición de los caracteres por representar. La salida es un árbol con raíz tal que los vértices en los niveles más bajos se etiquetan con las frecuencias y las aristas se etiquetan con bits, como en la figura 7.1.8. El árbol de codificación se obtiene reemplazando cada frecuencia por un carácter que tenga esa frecuencia.

Entrada: Una serie de n frecuencias, $n \geq 2$

Salida: Un árbol con raíz que define un código de Huffman óptimo

procedure *huffman* (f, n)
  if $n = 2$ **then**
    begin
    sean f_1 y f_2 las frecuencias
    sea T como en la figura 7.1.9
    return(T)
    end
  sean f_i y f_j las frecuencias menores
  reemplazar f_i y f_j en la lista f por $f_i + f_j$
  $T' := huffman(f, n - 1)$
  reemplazar un vértice en T' etiquetado con $f_i + f_j$ por el árbol que aparece en la figura 7.1.10 para obtener el árbol T
  return(T)
end *huffman*

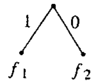


FIGURA 7.1.9
El caso $n = 2$ para el algoritmo 7.1.8.

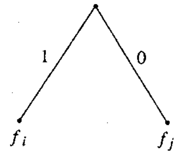


FIGURA 7.1.10 El caso $n > 2$ para el algoritmo 7.1.8.

EJEMPLO 7.1.9

Ahora mostraremos la forma en que el algoritmo 7.1.8 construye un código de Huffman óptimo para la tabla 7.1.2.

TABLA 7.1.2
Una parte de la tabla ASCII

Carácter	*Frecuencia*
!	2
@	3
#	7
$	8
%	12

El algoritmo comienza reemplazando varias veces las dos frecuencias menores con su suma, hasta obtener una sucesión de dos elementos:

$$
\begin{aligned}
2, 3, 7, 8, 12 &\rightarrow 2+3, 7, 8, 12\\
5, 7, 8, 12 &\rightarrow 5+7, 8, 12\\
8, 12, 12 &\rightarrow 8+12, 12\\
12, 20 &
\end{aligned}
$$

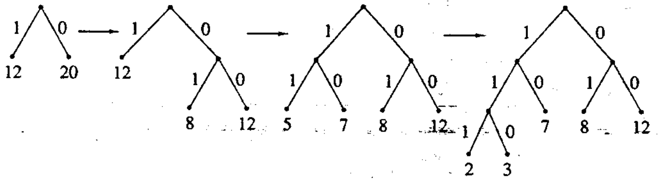


FIGURA 7.1.11 Construcción de un código de Huffman óptimo.

Luego, el algoritmo construye árboles trabajando hacia atrás, comenzando con la sucesión de dos elementos 12, 20 que aparece en la figura 7.1.11. Por ejemplo, el segundo árbol se obtiene del primero reemplazando el vértice con la etiqueta 20 por el árbol de la figura 7.1.12, pues 20 aparece como la suma de 8 y 12. Por último, para obtener el árbol de codificación de Huffman óptimo, reemplazamos cada frecuencia por un carácter que tenga esa frecuencia (véase la figura 7.1.13).

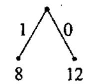


FIGURA 7.1.12
El árbol que reemplaza al vértice con la etiqueta 20 en la figura 7.1.11.

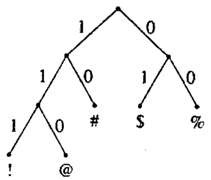


FIGURA 7.1.13
El árbol final de la figura 7.1.11, donde cada frecuencia ha sido reemplazada por un carácter que tenga esa frecuencia.

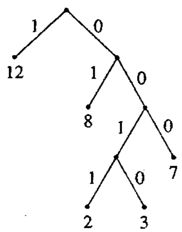
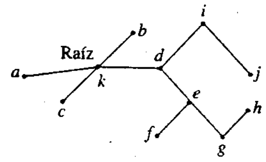


FIGURA 7.1.14
Otro árbol de Huffman óptimo para el ejemplo 7.1.9.

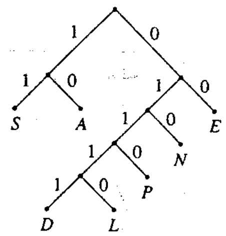
Observe que el árbol de Huffman para la tabla 7.1.2 no es único. Al reemplazar 12 por 5, 7, pues hay dos vértices etiquetados 12, hay que tomar una decisión. En la figura 7.1.11, elegimos de manera arbitraria uno de los vértices etiquetados con 12. Si elegimos el otro vértice etiquetado con 12, obtenemos el árbol de la figura 7.1.14. Cualquier árbol de Huffman proporciona un código óptimo; es decir, cualquier árbol de Huffman codificará un texto que tenga las frecuencias de la tabla 7.1.2 exactamente en el mismo espacio (óptimo). □

Ejercicios



1. Determine el nivel de cada vértice en el árbol de la figura anexa.
2. Determine la altura del árbol de la figura anexa.
3. Trace el árbol T de la figura 7.1.5 como un árbol con raíz, con a como raíz. ¿Cuál es la altura del árbol resultante?
4. Trace el árbol T de la figura 7.1.5 como un árbol con raíz, con b como raíz. ¿Cuál es la altura del árbol resultante?
5. Proporcione un ejemplo similar al ejemplo 7.1.5 de un árbol que se utilice para especificar relaciones jerárquicas.
6. Proporcione un ejemplo distinto al ejemplo 7.1.6 de un árbol de definición jerárquica.

Decodifique cada cadena de bits mediante el código de Huffman de la figura anexa.



7. 011000010
8. 01110100110
9. 01111001001110
10. 1110011101001111

Codifique cada palabra mediante el código de Huffman anterior.

11. DEN 12. NEED 13. LEADEN 14. PENNED

15. ¿Cuáles factores (además de la cantidad de memoria utilizada) deben tomarse en cuenta al elegir un código, como el ASCII o un código de Huffman, para representar los caracteres en una computadora?
16. ¿Cuáles técnicas, además del uso de códigos de Huffman, se podrían utilizar para ahorrar memoria al guardar un texto?
17. Construya un código de Huffman óptimo para el conjunto de letras en la tabla.

Letra	*Frecuencia*
α	5
β	6
γ	6
δ	11
ε	20

18. Construya un código de Huffman óptimo para el conjunto de letras en la tabla.

Letra	*Frecuencia*
I	7.5
U	20.0
B	2.5
S	27.5
C	5.0
H	10.0
M	2.5
P	25.0

19. Utilice el código desarrollado en el ejercicio 18 para codificar las siguientes palabras (las cuales tienen frecuencias consistentes con la tabla del ejercicio 18):

BUS, CUPS, MUSH, PUSS, SIP, PUSH, CUSS, HIP, PUP, PUPS, HIPS.

20. Construya dos árboles de codificación de Huffman óptimos para la tabla del ejercicio 17, de diferentes alturas.
21. El profesor Gig A. Byte necesitaba guardar texto formado mediante los caracteres A, B, C, D, E, los cuales aparecen con las siguientes frecuencias:

Carácter	*Frecuencia*
A	6
B	2
C	3
D	2
E	8

El profesor Byte sugiere el uso de los códigos de longitud variable

Carácter	*Código*
A	1
B	00
C	01
D	10
E	0

los cuales, argumenta, guardan el texto en un espacio menor que el utilizado por un código de Huffman óptimo. ¿Tiene razón el profesor? Explique.
22. Muestre que cualquier árbol con dos o más vértices tiene un vértice de grado 1.
23. Muestre que un árbol es una gráfica plana.
24. Muestre que un árbol es una gráfica bipartita.
25. Muestre que los vértices de un árbol pueden distinguirse con dos colores, de modo que cada arista sea incidente en vértices de colores distintos.

La *excentricidad* de un vértice v en un árbol T es la longitud máxima de un camino simple que comienza en v.

26. Determine la excentricidad de cada vértice en el árbol de la figura 7.1.5.

Un vértice v en un árbol T es un *centro* para T si la excentricidad de v es mínima.

27. Determine el centro (o centros) del árbol de la figura 7.1.5.
☆ 28. Muestre que un árbol tiene uno o dos centros.
☆ 29. Muestre que si un árbol tiene dos centros, éstos son adyacentes.
30. Defina el radio r de un árbol mediante los conceptos de excentricidad y centro. El diámetro d de cualquier gráfica se definió antes del ejercicio 70 de la sección 6.2. ¿Es siempre cierto, de acuerdo con su definición de radio, que $2r = d$? Explique.
31. Proporcione un ejemplo de árbol T que no satisfaga la propiedad: Si v y w son vértices en T, existe un único camino de v a w.

7.2 *TERMINOLOGÍA Y CARACTERIZACIONES DE LOS ÁRBOLES*

Una parte del árbol genealógico de los antiguos dioses griegos aparece en la figura 7.2.1. (No aparecen todos los hijos.) Como se muestra, un árbol genealógico se puede considerar como un árbol con raíz. Los vértices adyacentes a un vértice v y en el siguiente nivel hacia abajo son los hijos de v. Por ejemplo, los hijos de Cronos son Zeus, Poseidón, Hades y Ares. La terminología adaptada de un árbol genealógico se utiliza de manera normal en cualquier árbol con raíz. A continuación damos las definiciones formales.

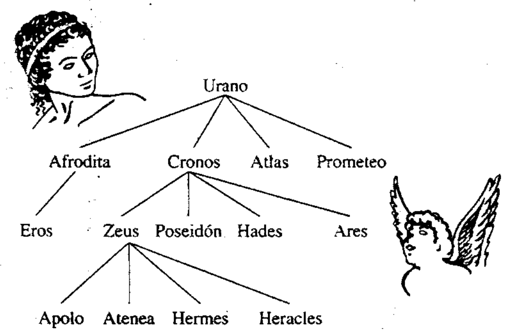


FIGURA 7.2.1 Una parte del árbol genealógico de los antiguos dioses griegos.

DEFINICIÓN 7.2.1

Sea T un árbol con raíz v_0. Suponga que x, y y z son vértices en T y que $(v_0, v_1, \ldots, v_n)$ es un camino simple en T. Entonces

(a) v_{n-1} es el *padre* de v_n.

(b) $v_0, \ldots, v_{n-1}$ son *ancestros* de v_n.

(c) v_n es un *hijo* de v_{n-1}.

(d) Si x es un ancestro de y, y es un *descendiente* de x.

(e) Si x y y son hijos de z, x y y son *hermanos*.

(f) Si x no tiene hijos, x es un *vértice terminal* (o una *hoja*).

(g) Si x no es un vértice terminal, x es un *vértice interno* (o una *rama*).

(h) El *subárbol de T con raíz en x* es la gráfica con conjunto de vértices V y conjunto de aristas E, donde V es x junto con los descendientes de x y

$$E = \{e \mid e \text{ es una arista en un camino simple de } x \text{ a algún vértice en } V\}.$$

EJEMPLO 7.2.2

En el árbol con raíz de la figura 7.2.1,

(a) El padre de Eros es Afrodita.

(b) Los ancestros de Hermes son Zeus, Cronos y Urano.

(c) Los hijos de Zeus son Apolo, Atenea, Hermes y Heracles.

(d) Los descendientes de Cronos son Zeus, Poseidón, Hades, Ares, Apolo, Atenea, Hermes y Heracles.

(e) Afrodita y Prometeo son hermanos.

(f) Los vértices terminales son Eros, Apolo, Atenea, Hermes, Heracles, Poseidón, Hades, Ares, Atlas y Prometeo.

(g) Los vértices internos son Urano, Afrodita, Cronos y Zeus.

(h) El subárbol con raíz en Cronos aparece en la figura 7.2.2. □

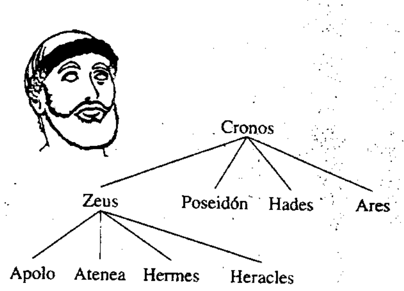


FIGURA 7.2.2 El subárbol con raíz en Cronos del árbol de la figura 7.2.1.

Dedicamos el resto de esta sección a algunas caracterizaciones alternativas de los árboles. Sea T un árbol. Observamos que T es conexo, pues existe un camino simple de cualquier vértice a cualquier otro vértice. Además, podemos mostrar que T no tiene ciclos. Para ver esto, suponga que T tiene un ciclo C'. Por el teorema 6.2.24, T tiene un ciclo simple (véase la figura 7.2.3)

$$C = (v_0, \ldots, v_n),$$

con $v_0 = v_n$. Como T es una gráfica simple, C no puede ser un lazo; así, C contiene al menos dos vértices distintos v_i y v_j, con $i < j$. Ahora,

$$(v_i, v_{i+1}, \ldots, v_j), \quad (v_i, v_{i-1}, \ldots, v_0, v_{n-1}, \ldots, v_j)$$

son caminos simples distintos de v_i a v_j, lo cual contradice la definición de árbol. Por tanto, un árbol no puede contener un ciclo.

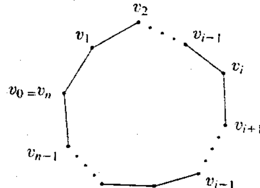


FIGURA 7.2.3
Un ciclo simple.

Una gráfica sin ciclos es una **gráfica acíclica**. Hemos mostrado que un árbol es una gráfica conexa acíclica. El recíproco también es cierto; toda gráfica conexa acíclica es un árbol. El siguiente teorema da ésta y otras caracterizaciones de los árboles.

TEOREMA 7.2.3

Sea T una gráfica con n vértices. Las siguientes afirmaciones son equivalentes.

(a) T es un árbol.

(b) T es conexa y acíclica.

(c) T es conexa y tiene $n - 1$ aristas.

(d) T es acíclica y tiene $n - 1$ aristas.

Demostración. Para mostrar que (a) − (d) son equivalentes, demostraremos cuatro resultados: si (a), entonces (b); si (b), entonces (c); si (c), entonces (d); y si (d), entonces (a).

[Si (a), entonces (b).] La demostración de este resultado se dio antes del enunciado del teorema.

[Si (b), entonces (c).] Suponga que T es conexa y acíclica. Demostraremos que T tiene $n - 1$ aristas por inducción sobre n.

Si $n = 1$, T consta de un vértice y cero aristas, por lo que el resultado es verdadero si $n = 1$.

Ahora suponga que el resultado es válido para una gráfica conexa, acíclica con n vértices. Sea T una gráfica conexa, acíclica, con $n + 1$ vértices. Sea P un camino simple de longitud máxima. Como T es acíclica, P no es un ciclo. Por tanto, P contiene un vértice v de grado 1 (véase la figura 7.2.4). Sea T^* el árbol obtenido de T al eliminar v y la arista incidente en v. Entonces T^* es conexa y acíclica, y como T^* contiene n vértices, por la hipótesis de inducción T^* contiene $n - 1$ aristas. Por tanto, T tiene n aristas. El argumento inductivo está completo y concluye esta parte de la demostración.

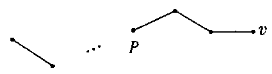


FIGURA 7.2.4
La demostración del teorema 7.2.3 [Si (b), entonces (c).]. P es un camino simple v y la arista incidente en v se eliminan, de modo que se pueda utilizar la hipótesis inductiva.

[Si (c), entonces (d).] Suponga que T es conexa y que tiene $n - 1$ aristas. Debemos mostrar que T es acíclica.

Supongamos que T tiene al menos un ciclo. Como la eliminación de una arista de un ciclo no desconecta una gráfica, podemos eliminar las aristas, pero no los vértices, de los ciclos de T hasta que la gráfica resultante T^* sea conexa y acíclica. Ahora, T^* es una gráfica conexa acíclica con n vértices. Podemos utilizar el resultado recién demostrado, (b) implica (c), para concluir que T^* tiene $n - 1$ aristas. Pero entonces T tiene más de $n - 1$ aristas. Esto es una contradicción. Por tanto, T es acíclica y concluye esta parte de la demostración.

[Si (d), entonces (a).] Suponga que T es acíclica y tiene $n - 1$ aristas. Debemos mostrar que T es un árbol; es decir, que T es una gráfica simple y que T tiene un único camino simple desde cualquier vértice hasta cualquier otro vértice.

La gráfica T no puede contener lazos, pues los lazos son ciclos y T es acíclica. De manera análoga, T no puede tener aristas distintas e_1 y e_2 incidentes en v y w, pues entonces tendríamos el ciclo (v, e_1, w, e_2, v). Por tanto, T es una gráfica simple.

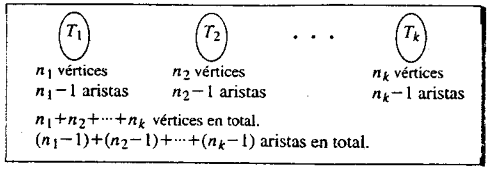


FIGURA 7.2.5 La demostración del teorema 7.2.3 [Si (d), entonces (a).]. Los T_i son componentes de T. T_i tiene n_i vértices y n_i-1 aristas. Surge una contradicción del hecho de que el número total de aristas debe ser igual a $n-1$.

Supongamos, por contradicción, que T no es conexa (véase la figura 7.2.5). Sean

$$T_1, \quad T_2, \quad \ldots, \quad T_k$$

los componentes de T. Como T no es conexa, $k > 1$. Supongamos que T_i tiene n_i vértices. Cada T_i es conexa y acíclica, de modo que podemos utilizar el resultado ya demostrado, (b) implica (c), para concluir que T_i tiene n_i-1 aristas. Entonces

$$\begin{aligned} n-1 &= (n_1-1)+(n_2-1)+\cdots+(n_k-1) && \text{(contando las aristas)}\\ &< (n_1+n_2)+\cdots+n_k)-1 && \text{(pues } k>1)\\ &= n-1, && \text{(contando los vértices)} \end{aligned}$$

lo cual es imposible. Por tanto, T es conexa.

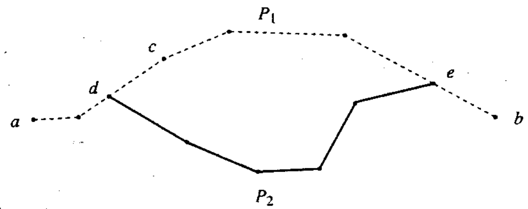


FIGURA 7.2.6 La demostración del teorema 7.2.3 [Si (d), entonces (a).]. P_1 (que aparece punteado) y P_2 (que aparece en línea continua) son caminos simples distintos de a a b. c es el primer vértice posterior a a en P_1 que no está en P_2. d es el vértice que precede a c en P_1. e es el primer vértice posterior a d en P_1 que también está en P_2. Como se muestra, aparece un ciclo, lo cual da una contradicción.

Suponga que existen caminos simples distintos P_1 y P_2 de a a b en T (véase la figura 7.2.6). Sea c el primer vértice posterior a a en P_1 que no está en P_2; sea d el vértice que precede a c en P_1; y sea e el primer vértice posterior a d en P_1 que también está en P_2. Sea

$$(v_0, v_1, \ldots, v_{n-1}, v_n)$$

la parte de P_1 de $d = v_0$ a $e = v_n$. Sea

$$(w_0, w_1, \ldots, w_{m-1}, w_m)$$

la parte de P_2 de $d = w_0$ a $e = w_m$. Entonces

$$(v_0, \ldots, v_n = w_m, w_{m-1}, \ldots, w_1, w_0) \tag{7.2.1}$$

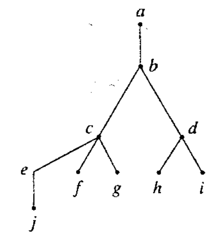
es un ciclo en T, lo cual es una contradicción. [De hecho, (7.2.1) es un ciclo simple, pues ningún vértice está repetido, excepto v_0 y w_0.] Así, existe un único camino simple desde cualquier vértice hasta cualquier otro vértice de T. Por tanto, T es un árbol. Esto concluye la demostración. ■

Ejercicios

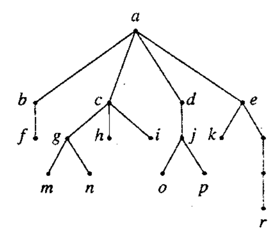
Responda las preguntas de los ejercicios 1-6 para el árbol de la figura 7.2.1.

1. Determine el padre de Poseidón.
2. Determine los ancestros de Eros.
3. Determine los hijos de Urano.
4. Determine los descendientes de Zeus.
5. Determine los hermanos de Ares.
6. Trace el subárbol con raíz en Afrodita.

Responda las preguntas de los ejercicios 7-15 para el árbol anexo.



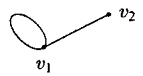
7. Determine los padres de c y de h.
8. Determine los ancestros de c y de j.
9. Determine los hijos de d y de e.
10. Determine los descendientes de c y de e.
11. Determine los hermanos de f y de h.
12. Determine los vértices terminales.
13. Determine los vértices internos.
14. Trace el subárbol con raíz en j.
15. Trace el subárbol con raíz en e.
16. Responda las preguntas de los ejercicios 7-15 para el árbol anexo.



17. ¿Qué podría decir acerca de dos vértices en un árbol con raíz que tengan el mismo padre?
18. ¿Qué podría decir acerca de dos vértices en un árbol con raíz que tengan los mismos ancestros?
19. ¿Qué podría decir acerca de un vértice en un árbol con raíz que no tenga ancestros?
20. ¿Qué podría decir acerca de dos vértices en un árbol con raíz que tengan un descendiente común?
21. ¿Qué podría decir acerca de un vértice en un árbol con raíz que no tenga descendientes?

En los ejercicios 22-26, trace una gráfica con las propiedades dadas o explique por qué no existe tal gráfica.

22. Seis aristas; ocho vértices
23. Acíclica; cuatro aristas, seis vértices
24. Árbol; todos los vértices de grado 2
25. Árbol; seis vértices con grados 1, 1, 1, 1, 3, 3
26. Árbol; cuatro vértices internos; seis vértices terminales
27. Explique por qué si permitimos la existencia de ciclos de longitud 0, una gráfica que consta de un único vértice, sin aristas, no es acíclica.
28. Explique por qué si permitimos que se repitan aristas en los ciclos, una gráfica que consta de una única arista y dos vértices no es acíclica.



29. La gráfica conexa que se muestra aquí tiene un único camino simple de cualquier vértice a cualquier otro vértice pero no es un árbol. Explique.

Un *bosque* es una gráfica simple sin ciclos.

30. Explique por qué un bosque es una unión de árboles.
31. Si un bosque F consta de m árboles y n vértices, ¿cuántas aristas tiene F?
32. Si $P_1 = (v_0, \ldots, v_n)$ y $P_2 = (w_0, \ldots, w_m)$ son caminos simples distintos de a a b en una gráfica simple G, ¿es

$$(v_0, \ldots, v_n = w_m, w_{m-1}, \ldots, w_1, w_0)$$

necesariamente un ciclo? Explique. (Este ejercicio es importante para el último párrafo de la demostración del teorema 7.2.3.)
33. Muestre que una gráfica G con n vértices y menos de $n - 1$ aristas no es conexa.
☆ 34. Demuestre que T es un árbol si y sólo si T es conexa y tal que al agregar una arista entre cualesquiera dos vértices, se crea exactamente un ciclo.
35. Muestre que si G es un árbol, cada vértice de grado mayor o igual que 2 es un punto de articulación. ("Punto de articulación" se definió antes del ejercicio 55 de la sección 6.2.)
36. Proporcione un ejemplo para mostrar que el recíproco del ejercicio 35 es falso, incluso suponiendo que G es conexa.

RINCÓN DE SOLUCIÓN DE PROBLEMAS: ÁRBOLES

Problema

Sea T una gráfica simple. Demuestre que T es un árbol si y sólo si T es conexa, pero al eliminar cualquier arista de T (pero no los vértices), T se desconecta.

Para enfrentar el problema

Debemos saber claramente lo que debemos demostrar. Como la afirmación es "si y sólo si", debemos demostrar dos afirmaciones:

Si T es un árbol, entonces T es conexa pero al eliminar cualquier arista de T (pero no los vértices), T se desconecta. (1)

Si T es conexa pero al eliminar cualquier arista de T (pero no los vértices), T se desconecta, entonces T es un árbol. (2)

En (1), partiendo de la hipótesis de que T es un árbol, debemos demostrar que T es conexa, pero al eliminar cualquier arista de T (pero no los vértices), T se desconecta. En (2), a partir de la hipótesis de que T es conexa pero al eliminar cualquier arista de T (pero no los vértices), T se desconecta, debemos deducir que T es un árbol.

Al desarrollar una demostración, por lo general es útil revisar las definiciones y otros resultados relacionados con las afirmaciones por demostrar. En este caso, la definición de árbol y el teorema 7.2.3, el cual proporciona condiciones equivalentes para que una gráfica sea un árbol, tienen una relación directa con lo que debemos demostrar.

La definición 7.1.1 establece:

Un árbol T es una gráfica simple que satisface: Si v y w son vértices en T, existe un único camino simple de v a w. (3)

El teorema 7.2.3 establece que las siguientes afirmaciones son equivalentes para una gráfica T con n vértices:

T es un árbol. (4)

T es conexa y acíclica. (5)

T es conexa y tiene $n - 1$ aristas. (6)

T es acíclica y tiene $n - 1$ aristas. (7)

Determinación de una solución

Primero intentaremos demostrar (1). Suponemos que T es un árbol. Debemos demostrar dos cosas: que T es conexa y que al eliminar cualquier arista de T (pero no los vértices), T se desconecta.

Las afirmaciones (5) y (6) nos indican de inmediato que T es conexa. Ninguna de las afirmaciones (3) a (7) nos dicen de manera directa algo acerca de la eliminación de aristas o de una gráfica que no sea conexa. Sin embargo, si argumentamos por contradicción y suponemos que al eliminar cualquier arista de T (pero no los vértices), T no se desconecta, entonces al eliminar esa arista de T, la gráfica resultante T' es conexa. En este caso, para la gráfica T', (5) es verdadera, pero (6) y (7) son falsas, lo cual es una contradicción, pues (5), (6) y (7) son todas verdaderas (y la gráfica es un árbol) o todas falsas (y la gráfica no es un árbol).

Ahora, intentemos demostrar (2). Suponemos que T es conexa y que al eliminar cualquier arista de T (pero no los vértices), T se desconecta. Debemos mostrar que T es un árbol. Intentemos mostrar que T es conexa y acíclica. Entonces podemos apelar a (5) para concluir que T es un árbol.

Como T es conexa, lo único que debemos hacer es mostrar que T es acíclica. De nuevo, nuestro método será por contradicción. Supongamos que T tiene un ciclo. Recordando nuestra hipótesis (la eliminación de cualquier arista de T desconecta a T), el lector debe tratar de ver cuál será la contradicción por el hecho de suponer que T tiene un ciclo antes de continuar.

Si eliminamos una arista del ciclo de T, T seguirá siendo conexa. Esta contradicción muestra que T es acíclica. Por (5), T es un árbol.

Solución formal

Suponga que T tiene n vértices.

Suponga que T es un árbol. Entonces, por el teorema 7.2.3, T es conexa y tiene $n - 1$ aristas. Suponga que podemos eliminar una arista de T para obtener T', de modo que T' sea conexa. Como T no tiene ciclos, T' tampoco tiene ciclos. Por el teorema 7.2.3, T' es un árbol. De nuevo por el teorema 7.2.3, T' tiene $n - 1$ aristas. Esto es una contradicción. Por tanto, T es conexa pero al eliminar cualquier arista de T (pero no los vértices), T se desconecta.

Si T es conexa y al eliminar cualquier arista de T (pero no los vértices), T se desconecta, entonces T no contiene ciclos. Por el teorema 7.2.3, T es un árbol.

Resumen de técnicas para resolver problemas.

- Al intentar construir una demostración, escriba con cuidado lo que se está suponiendo y lo que se debe demostrar.
- Al intentar construir una demostración, considere el uso de definiciones y teoremas relacionados con aquello que se debe demostrar.
- Al intentar construir una demostración, revise las demostraciones de teoremas similares y relacionados con lo que se desea demostrar.
- Si ninguna de las condiciones de las definiciones y teoremas potencialmente útiles se puede aplicar, intente la demostración por contradicción. Al suponer la negación de las hipótesis, dispondrá de afirmaciones adicionales que podrían volver aplicables las condiciones de las definiciones y los teoremas.

7.3 *ÁRBOLES DE EXPANSIÓN*

En esta sección consideraremos el problema de determinar una subgráfica T de una gráfica G de modo que T sea un árbol con todos los vértices de G; es decir, un **árbol de expansión**. Veremos que los métodos para determinar árboles de expansión se pueden aplicar también a otros problemas.

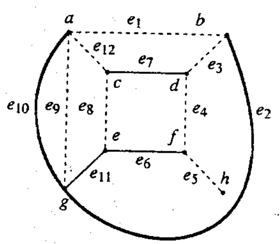


FIGURA 7.3.1
Una gráfica y un árbol de expansión en líneas punteadas.

DEFINICIÓN 7.3.1

Un árbol T es un *árbol de expansión* de una gráfica G si T es una subgráfica de G que contiene a todos los vértices de G.

EJEMPLO 7.3.2

Un árbol de expansión de la gráfica G de la figura 7.3.1 aparece en líneas punteadas. □

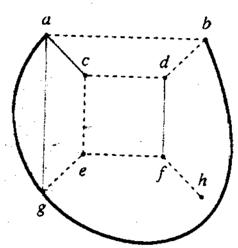


FIGURA 7.3.2
Otro árbol de expansión (en líneas punteadas) de la gráfica de la figura 7.3.1.

EJEMPLO 7.3.3

En general, una gráfica tendrá varios árboles de expansión. Otro árbol de expansión de la gráfica G de la figura 7.3.1 aparece en la figura 7.3.2. □

Suponga que una gráfica G tiene un árbol de expansión T. Sean a y b vértices de G. Como a y b son también vértices de T y T es un árbol, existe un camino P de a a b. Sin embargo, P también sirve como un camino de a a b en G; así, G es conexa. El recíproco también es verdadero.

TEOREMA 7.3.4

Una gráfica G tiene un árbol de expansión si y sólo si G es conexa.

Demostración. Ya hemos mostrado que si G tiene un árbol de expansión, entonces G es conexa. Suponga que G es conexa. Si G es acíclica, por el teorema 7.2.3, G es un árbol.

Suponga que G contiene un ciclo. Eliminamos una arista (pero no los vértices) de este ciclo. La gráfica producida sigue siendo conexa. Si es acíclica, nos detenemos. Si contiene un ciclo, eliminamos una arista de este ciclo. Continuamos de esta manera hasta obtener una subgráfica acíclica, conexa. Por el teorema 7.2.3, T es un árbol. Como T contiene a todos los vértices de G, T es un árbol de expansión de G. ■

Un algoritmo para determinar un árbol de expansión con base en la demostración del teorema 7.3.4 no sería muy eficiente; implicaría el lento proceso de determinación de ciclos. Pero podemos hacer algo mucho mejor. Ilustraremos el primer algoritmo para determinar un árbol de expansión por medio de un ejemplo y luego enunciaremos el algoritmo.

EJEMPLO 7.3.5

Determinar un árbol de expansión para la gráfica G de la figura 7.3.1.

Utilizaremos un método llamado **búsqueda a lo ancho** (algoritmo 7.3.6). La idea de la búsqueda a lo ancho es procesar todos los vértices de un nivel dado antes de pasar al siguiente nivel superior.

Primero, elegimos un orden, digamos *abcdefgh*, de los vértices de G. Elegimos el primer vértice a y lo etiquetamos como la raíz. Sea T la gráfica formada por el único vértice a, sin aristas. Agregamos a T todas las aristas (a, x) y los vértices sobre los cuales son incidentes, desde $x = b$ hasta h, que no produzcan un ciclo al agregarse a T. Así, agregamos a T las aristas (a, b), (a, c) y (a, g). (Podemos usar cualesquiera de las aristas paralelas incidentes en a y g.) Repetimos este procedimiento con los vértices del nivel 1, examinándolos en orden:

b: Incluir (b, d).
c: Incluir (c, e).
g: Ninguno.

Repetimos el procedimiento con los vértices del nivel 2:

d: Incluir (d, f).
e: Ninguno.

Repetimos el procedimiento con los vértices del nivel 3:

f: Incluir (f, h).

Como no se pueden agregar más aristas a h, el único vértice del nivel 4, el procedimiento termina. Hemos determinado el árbol de expansión de la figura 7.3.1. □

Formalizamos el método del ejemplo 7.3.5 como el algoritmo 7.3.6.

ALGORITMO 7.3.6 ***Búsqueda a lo ancho para obtener un árbol de expansión***

Este algoritmo determina un árbol de expansión mediante el método de búsqueda a lo ancho.

Entrada: Una gráfica conexa G con los vértices ordenados

$$v_1, \quad v_2, \quad \ldots, \quad v_n$$

Salida: Un árbol de expansión T

```
procedure bfs (V, E)
 // V = vértices ordenados v1, . . . , vn; E = aristas
 // V' = vértices del árbol de expansión T; E' = aristas del árbol de expansión T
 // v1 es la raíz del árbol de expansión
 // S es una lista ordenada
 S := (v1)
 V' := {v1}
 E' := ∅
  while true do
   begin
   for cada x ∈ S, en orden, do
    for cada y ∈ V − V', en orden, do
     if (x, y) es una arista then
      agregar arista (x, y) a E' y y a V'
   if no se agregó arista alguna then
    return (T)
   S := hijos de S ordenados de manera consistente, con el orden original
      de los vértices
   end
end bfs
```

El ejercicio 16 consiste en dar un argumento para mostrar que el algoritmo 7.3.6 determina de manera correcta un árbol de expansión.

La búsqueda a lo ancho se puede utilizar para verificar si una gráfica arbitraria G con n vértices es conexa (véase el ejercicio 26). Utilizamos el método del algoritmo 7.3.6 para producir un árbol T. Entonces G es conexa si y sólo si T tiene n vértices.

La búsqueda a lo ancho también se puede utilizar para determinar caminos de longitud mínima en una gráfica sin pesos, que vayan de un vértice dado v a todos los demás vértices (véase el ejercicio 20). Utilizamos el método del algoritmo 7.3.6 para obtener un árbol de expansión con raíz en v. Observamos que la longitud de un camino más corto de v a un vértice en le nivel i del árbol de expansión es i. El algoritmo para el camino (o ruta) más corto de Dijkstra para gráficas con pesos (algoritmo 6.4.1) se puede considerar como una generalización de la búsqueda a lo ancho (véase el ejercicio 21).

Una alternativa a la búsqueda a lo ancho es la **búsqueda a profundidad**, la cual pasa a los niveles sucesivos de un árbol en la primera oportunidad posible.

ALGORITMO 7.3.7 ***Búsqueda a profundidad de un árbol de expansión***

Este algoritmo determina un árbol de expansión mediante el método de búsqueda a profundidad.

Entrada: Una gráfica conexa G con los vértices ordenados

$$v_1, \quad v_2, \quad \ldots, \quad v_n$$

Salida: Un árbol de expansión T

```
procedure dfs(V, E)
 // V' = vértices del árbol de expansión T; E' = aristas del árbol de expansión T
 // v1 es la raíz del árbol de expansión
 V' := {v1}
 E' := ∅
 w := v1
 while true do
   begin
   while existe una arista (w, v) tal que al agregarse a T no se crea un ciclo en T do
   begin
   se elige la arista (w, vk) con k mínimo tal que al agregarse a T no crea un ciclo en T
   se agrega (w, vk) a E'
   se agrega vk a V'
   w := vk
   end
  if w = v1 then
   return(T)
  w := padre de w en T // retroceso
  end
end dfs
```

El ejercicio 17 consiste en dar un argumento para mostrar que el algoritmo 7.3.7 determina de manera correcta un árbol de expansión.

EJEMPLO 7.3.8

Utilizar la búsqueda a profundidad (algoritmo 7.3.7) para determinar un árbol de expansión para la gráfica de la figura 7.3.2 con el orden *abcdefgh* de los vértices.

Elegimos el primer vértice a y lo llamamos la raíz (véase la figura 7.3.2). A continuación, agregamos a nuestro árbol la arista (a, x), con x mínimo. En nuestro caso, agregamos la arista (a, b).

Repetimos este proceso. Agregamos las aristas (b, d), (d, c), (c, e), (e, f) y (f, h). En este momento no podemos agregar una arista de la forma (h, x), así que retrocedemos al padre f de h e intentamos agregar una arista de la forma (f, x). De nuevo, no podemos lograr esto, de modo que regresamos al padre e de f. Esta vez podemos agregar la arista (e, g). En este momento no se pueden agregar más aristas, de modo que finalmente retrocedemos hasta la raíz y el procedimiento concluye. □

Debido a la línea en el algoritmo 7.3.7 donde regresamos a lo largo de una arista hacia la raíz elegida al principio, la búsqueda a profundidad también se llama **retroceso**. En el siguiente ejemplo utilizamos el retroceso para resolver un juego.

EJEMPLO 7.3.9 *El problema de las cuatro reinas*

El problema de las cuatro reinas consiste en colocar cuatro fichas en una cuadrícula 4×4 de modo que no haya dos fichas en el mismo renglón, la misma columna o en diagonal. Construir un algoritmo de retroceso para resolver el problema de las cuatro reinas. (Para utilizar la terminología del ajedrez, éste es el problema de colocar cuatro reinas en un tablero 4×4 de modo que las reinas no se ataquen entre sí.)

La idea del algoritmo consiste en colocar las fichas de manera sucesiva en las columnas. Cuando no sea posible colocar una ficha en una columna, retrocedemos y ajustamos la ficha de la columna anterior. □

ALGORITMO 7.3.10 *Resolución del problema de las cuatro reinas mediante retroceso*

Este algoritmo utiliza el retroceso para buscar un arreglo de cuatro fichas en una cuadrícula 4×4 de modo que no haya dos fichas en el mismo renglón, la misma columna, o en la diagonal.

Entrada: Un arreglo *row* de tamaño 4

Salida: **true**, si existe una solución
false, si no existe solución
[Si existe una solución, la k-ésima reina está en la columna k, y el renglón *row* (k).]

```
procedure four_queens (row)
  k := 1     // se comienza en la columna 1
  // row (k) se incrementa antes de utilizarse, de modo que comenzamos en el renglón 1
  row (1) := 0
  while k > 0 do
    begin
    row (k) := row (k) + 1
    // se busca un movimiento válido en la columna k
    while row (k) ≤ 4 and hay conflicto en la columna k, renglón row (k) do
      // se intenta con el siguiente renglón
      row (k) := row (k) + 1
    if row (k) ≤ 4 then     // se ha determinado un movimiento válido en la columna k
      if k = 4 then     // solución concluida
        return(true)
      else     // siguiente columna
        begin
        k := k + 1
        row (k) := 0
        end
    else     // se retrocede a la columna anterior
      k := k − 1
    end
  return(false)     // no existe solución
end four_queens
```

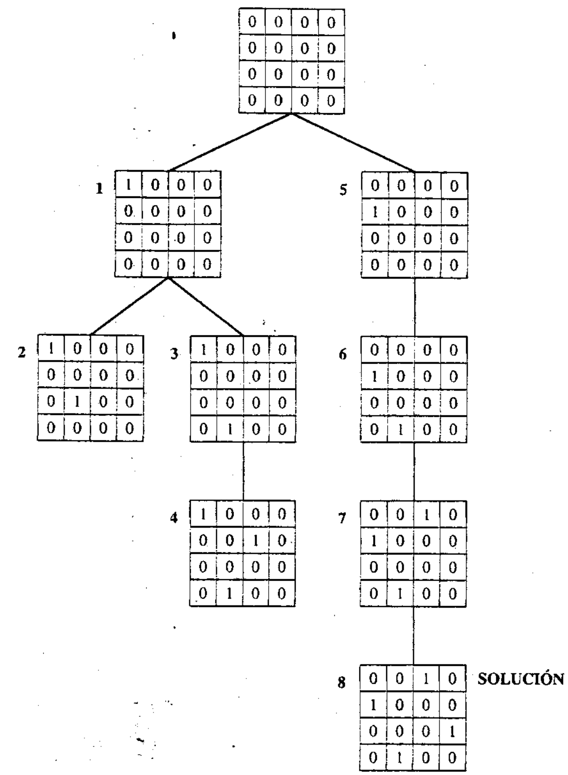


FIGURA 7.3.3 El árbol generado por el algoritmo de retroceso (algoritmo 7.3.10) en búsqueda de una solución al problema de las cuatro reinas.

El árbol generado por el algoritmo 7.3.10 aparece en la figura 7.3.3. Los números indican el orden en que se generan los vértices. La solución aparece en el vértice 8.

El problema de las n reinas consiste en colocar n fichas en una cuadrícula $n \times n$ de modo que no haya dos fichas en el mismo renglón, la misma columna, o la misma diagonal. Se puede verificar directamente que no existe una solución para el problema de dos o tres reinas (véase el ejercicio 10). Acabamos de ver que el algoritmo 7.3.10 genera una solución al problema de las cuatro reinas. Se han dado muchas construcciones para generar soluciones al problema de las n reinas para toda $n \geq 4$ (véase, por ejemplo, [Erbas]).

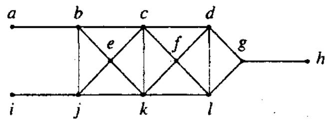
El retroceso o búsqueda a profundidad es de particular interés en un problema como el del ejemplo 7.3.9, donde lo único que se quería era una solución. Como una solución (si existe) aparece en un vértice terminal, al pasar a los vértices terminales lo más pronto posible, en general se evita la generación de muchos vértices innecesarios.

Ejercicios

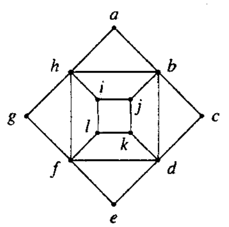
1. Utilice la búsqueda a lo ancho (algoritmo 7.3.6) con el orden *hgfedcba* de los vértices para determinar un árbol de expansión de la gráfica G de la figura 7.3.1.
2. Utilice la búsqueda a lo ancho (algoritmo 7.3.6) con el orden *hfdbgeca* de los vértices para determinar un árbol de expansión de la gráfica G de la figura 7.3.1.
3. Utilice la búsqueda a lo ancho (algoritmo 7.3.6) con el orden *chbgadfe* de los vértices para determinar un árbol de expansión de la gráfica G de la figura 7.3.1.
4. Utilice la búsqueda a profundidad (algoritmo 7.3.7) con el orden *hgfedcba* de los vértices para determinar un árbol de expansión de la gráfica G de la figura 7.3.1.
5. Utilice la búsqueda a profundidad (algoritmo 7.3.7) con el orden *hfdbgeca* de los vértices para determinar un árbol de expansión de la gráfica G de la figura 7.3.1.
6. Utilice la búsqueda a profundidad (algoritmo 7.3.7) con el orden *dhcbefag* de los vértices para determinar un árbol de expansión de la gráfica G de la figura 7.3.1.

En los ejercicios 7-9, determine un árbol de expansión para cada gráfica.

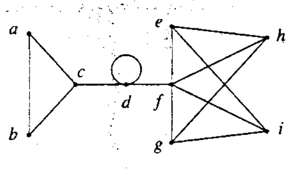
7.



8.

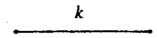


9.

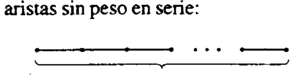


10. Muestre que no existe solución para los problemas de dos o tres reinas.
11. Determine una solución para los problemas de cinco y seis reinas.
12. ¿Verdadero o falso? Si G es una gráfica conexa y T es un árbol de expansión para G, existe un orden de los vértices de G tal que el algoritmo 7.3.6 produce a T como árbol de expansión. De ser cierto, demuestre este hecho; en caso contrario, proporcione un contraejemplo.
13. ¿Verdadero o falso? Si G es una gráfica conexa y T es un árbol de expansión para G, existe un orden de los vértices de G tal que el algoritmo 7.3.7 produce a T como árbol de expansión. De ser cierto, demuestre este hecho; en caso contrario, proporcione un contraejemplo.
14. Muestre, mediante un ejemplo, que el algoritmo 7.3.6 puede producir árboles de expansión idénticos para una gráfica conexa G partiendo de dos órdenes distintos de los vértices de G.
15. Muestre, mediante un ejemplo, que el algoritmo 7.3.7 puede producir árboles de expansión idénticos para una gráfica conexa G partiendo de dos órdenes distintos de los vértices de G.

16. Demuestre que el algoritmo 7.3.6 es correcto.
17. Demuestre que el algoritmo 7.3.7 es correcto.
18. ¿Bajo qué condiciones está una arista de una gráfica conexa G contenida en todo árbol de expansión de G?
19. Sean T y T' dos árboles de expansión de una gráfica conexa G. Suponga que una arista x está en T pero no en T'. Muestre que existe una arista y en T' pero no en T tal que $(T - \{x\}) \cup \{y\}$ y $(T' - \{y\}) \cup \{x\}$ son árboles de expansión de G.
20. Escriba un algoritmo con base en la búsqueda a lo ancho que determine la longitud mínima de cada camino en una gráfica sin pesos que parta de un vértice fijo v a los demás vértices.
21. Sea G una gráfica con pesos en la que el peso de cada arista es un entero positivo. Sea G' la gráfica obtenida de G reemplazando cada arista



en G de peso k por k aristas sin peso en serie:



Muestre que el algoritmo de Dijkstra para determinar la longitud mínima de cada camino en la gráfica con pesos G partiendo de un vértice fijo v a los demás vértices (algoritmo 6.4.1) y la realización de una búsqueda a lo ancho en la gráfica sin pesos G' partiendo del vértice v son, en realidad, el mismo proceso.

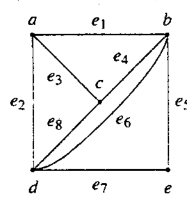
22. Sea T un árbol de expansión para una gráfica G. Muestre que si una arista en G, pero no en T, se agrega a T, se produce un único ciclo.

Un ciclo como el descrito en el ejercicio 22 es un *ciclo fundamental*. La *matriz de ciclos fundamentales* de una gráfica G tiene sus renglones indicados por los ciclos fundamentales de G con respecto de un árbol de expansión T de G y sus columnas indicadas por las aristas de G. La entrada ij es 1 si la arista j está en el i-ésimo ciclo fundamental y 0 en caso contrario. Por ejemplo, la matriz de ciclos fundamentales de la gráfica G de la figura 7.3.1 con respecto del árbol de expansión que aparece en la figura 7.3.1 es

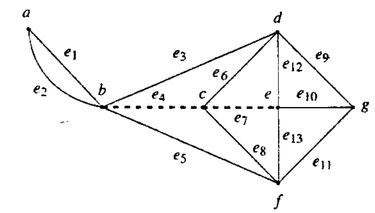
$$
\begin{array}{cc}
 & \begin{array}{cccccccccccc} e_7 & e_6 & e_{11} & e_{10} & e_2 & e_1 & e_3 & e_4 & e_5 & e_8 & e_9 & e_{12} \end{array} \\
\begin{array}{c} (abdca) \\ (efdbace) \\ (ageca) \\ (aga) \\ (abga) \end{array} &
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0
\end{pmatrix}
\end{array}
$$

Determine la matriz de ciclos fundamentales de cada gráfica. El árbol de expansión por utilizar aparece en líneas punteadas.

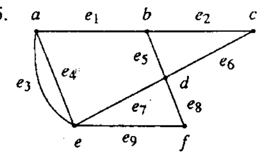
23.



24.



25.



26. Escriba un algoritmo de búsqueda a lo ancho para verificar si una gráfica es conexa.

27. Escriba un algoritmo de búsqueda a profundidad para verificar si una gráfica es conexa.

28. Escriba un algoritmo de búsqueda a profundidad que determine todas las soluciones al problema de cuatro reinas.

7.4 *ÁRBOLES DE EXPANSIÓN MÍNIMOS*

La gráfica con pesos, G, de la figura 7.4.1 muestra seis ciudades y los costos de construcción de carreteras entre ciertos pares de ellas. Queremos construir el sistema de carreteras de menor costo que una estas seis ciudades. La solución se puede representar mediante una subgráfica. Esta subgráfica debe ser un árbol de expansión, pues debe contener a todos los vértices (de modo que cada ciudad esté en el sistema de carreteras), debe ser conexa (de modo que se pueda llegar a cualquier ciudad partiendo de otra) y debe tener un único camino simple entre cada par de vértices (pues una gráfica con varios caminos simples entre un par de vértices no podría representar un sistema con costo mínimo). Así, lo que se necesita es un árbol de expansión cuya suma de pesos sea mínima. Tal árbol es un **árbol de expansión mínimo**.

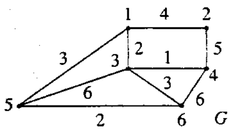


FIGURA 7.4.1 Seis ciudades (a, b, c, d, e, f) y los costos de construcción de carreteras entre ciertos pares de ellas.

DEFINICIÓN 7.4.1

Sea G una gráfica con pesos. Un *árbol de expansión mínimo* de G es un árbol de expansión de G con peso mínimo.

EJEMPLO 7.4.2

El árbol T' de la figura 7.4.2 es un árbol de expansión de la gráfica G de la figura 7.4.1. El peso de T' es 20. Éste no es un árbol de expansión mínimo, pues el árbol de expansión T de la figura 7.4.3 tiene peso 12. Veremos más adelante que T es un árbol de expansión mínimo de G. □

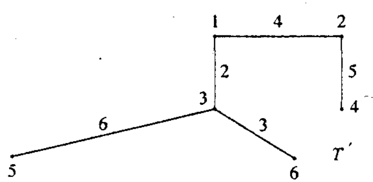


FIGURA 7.4.2 Un árbol de expansión de peso 20 para la gráfica de la figura 7.4.1.

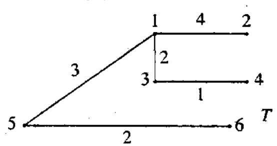


FIGURA 7.4.3 Un árbol de expansión de peso 12 para la gráfica de la figura 7.4.1.

Analizaremos un algoritmo para determinar un árbol de expansión mínimo conocido como **algoritmo de Prim** (algoritmo 7.4.3). Este algoritmo construye un árbol agregando aristas de manera iterativa hasta obtener un árbol de expansión mínimo. En cada iteración agregamos una arista de peso mínimo que no forme un ciclo en el árbol en cuestión. Otro algoritmo para determinar un árbol de expansión mínimo, el **algoritmo de Kruskal**, se presenta en los ejercicios (véanse los ejercicios 20-22).

ALGORITMO 7.4.3 ***Algoritmo de Prim***

Este algoritmo determina un árbol de expansión mínimo en una gráfica conexa con pesos.

Entrada: Una gráfica conexa con pesos, con vértices $1, \ldots, n$ y vértice inicial s. Si (i, j) es una arista, $w(i, j)$ es igual al peso de (i, j); si (i, j) no es una arista, entonces $w(i, j)$ es igual a ∞ (un valor mayor que cualquier peso real).

Salida: El conjunto de aristas E en un árbol de expansión mínimo.

```
procedure prim (w, n, s)
 // v(i) = 1 si el vértice i se ha agregado al árbol de expansión mínimo
 // v(i) = 0 si el vértice i no se ha agregado al árbol de expansión mínimo
 for i := 1 to n do
   v(i) := 0
 // se agrega el vértice inicial al árbol de expansión mínimo
 v(s) := 1
 // se comienza con un conjunto de aristas vacío
 E := ∅
 // se colocan n − 1 aristas en el árbol de expansión mínimo
 for i := 1 to n − 1 do
  begin
  // se agrega la arista de peso mínimo que tenga un vértice en el árbol de expansión
     mínimo
  // y otro vértice que no esté en el árbol de expansión mínimo
  min := ∞
  for j := 1 to n do
   if v(j) = 1 then     // j es un vértice en el árbol de expansión mínimo
    for k = 1 to n do
     if v(k) = 0 and w(j, k) , < min then
      begin
      add_vertex := k
      e := (j, k)
      min := w(j, k)
      end
  // se agregan el vértice y la arista al árbol de expansión mínimo
  v(add_vertex) := 1
  E := E ∪ {e}
  end
 return(E)
end prim
```

EJEMPLO 7.4.4

Mostrar la forma en que el algoritmo de Prim determina un árbol de expansión mínimo para la gráfica de la figura 7.4.1. Suponer que el vértice inicial s es 1.

En la línea 3, agregamos el vértice 1 al árbol de expansión mínimo. La primera vez que ejecutamos el ciclo for en las líneas 8–16, las aristas con un vértice en el árbol y un vértice que no está en el árbol son

Arista	*Peso*
(1, 2)	4
(1, 3)	2
(1, 5)	3

Se elige la arista con menor peso, (1, 3). En las líneas 17 y 18, el vértice 3 se agrega al árbol de expansión mínimo y la arista (1, 3) se agrega a E.

La siguiente vez que ejecutamos el ciclo for en las líneas 8–16, las aristas con un vértice en el árbol y un vértice que no está en el árbol son

Arista	*Peso*
(1, 2)	4
(1, 5)	3
(3, 4)	1
(3, 5)	6
(3, 6)	3

Se elige la arista con menor peso, (3, 4). En las líneas 17 y 18, el vértice 4 se agrega al árbol de expansión mínimo y la arista (3, 4) se agrega a E.

La siguiente vez que ejecutamos el ciclo for en las líneas 8–16, las aristas con un vértice en el árbol y un vértice que no está en el árbol son

Arista	*Peso*
(1, 2)	4
(1, 5)	3
(2, 4)	5
(3, 5)	6
(3, 6)	3
(4, 6)	6

Esta vez, dos aristas tienen el peso mínimo 3. Sin importar la arista elegida, se obtendrá un árbol de expansión mínimo. En esta versión, se elige la arista (1, 5). En las líneas 17 y 18, el vértice 5 se agrega al árbol de expansión mínimo y la arista (1, 5) se agrega a E.

La siguiente vez que ejecutamos el ciclo for en las líneas 8–16, las aristas con un vértice en el árbol y un vértice que no está en el árbol son

Arista	*Peso*
(1, 2)	4
(2, 4)	5
(3, 6)	3
(4, 6)	6
(5, 6)	2

Se elige la arista con menor peso, (5, 6). En las líneas 17 y 18, el vértice 6 se agrega al árbol de expansión mínimo y la arista (5, 6) se agrega a E.

La última vez que ejecutamos el ciclo for en las líneas 8–16, las aristas con un vértice en el árbol y un vértice que no está en el árbol son

Arista	*Peso*
(1, 2)	4
(2, 4)	5

Se elige la arista con menor peso, (1, 2). En las líneas 17 y 18, el vértice 2 se agrega al árbol de expansión mínimo y la arista (1, 2) se agrega a E. El árbol de expansión mínimo construido de esta manera aparece en la figura 7.4.3. □

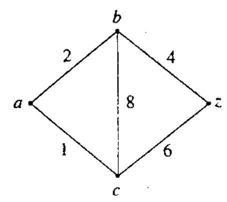


FIGURA 7.4.4
Esta gráfica muestra que la elección de una arista de peso mínimo incidente en el vértice agregado de manera más reciente *no* necesariamente produce un camino más corto. Al partir de a, obtenemos (a, c, z), pero el camino más corto de a a z es (a, b, z).

El algoritmo de Prim es un ejemplo de **algoritmo codicioso**; es decir, aquel que optima la elección de cada iteración, sin considerar las elecciones anteriores. El principio se puede resumir como "hacer lo mejor localmente". En el algoritmo de Prim, como queremos determinar un árbol de expansión mínimo, en cada iteración sólo agregamos una arista disponible con peso mínimo.

La optimación en cada iteración no necesariamente proporciona una solución óptima del problema original. Más adelante (teorema 7.4.5) mostraremos que el algoritmo de Prim es correcto: en realidad obtenemos un árbol de expansión mínimo. Como ejemplo de un algoritmo codicioso que no conduce a una solución óptima, consideremos un "algoritmo del camino más corto", en el cual en cada paso elegimos una arista disponible con peso mínimo, incidente en el vértice agregado de manera más reciente. Si aplicamos este algoritmo a la gráfica con pesos de la figura 7.4.4 para determinar un camino más corto de a a z, elegiríamos la arista (a, c) y luego la arista (c, z). Por desgracia, éste no es el camino más corto de a a z.

Ahora mostraremos que el algoritmo de Prim es correcto.

TEOREMA 7.4.5

El algoritmo de Prim (algoritmo 7.4.3) es correcto; es decir, al concluir el algoritmo 7.4.3, T es un árbol de expansión mínimo.

Demostración. Sea T_i la gráfica construida por el algoritmo 7.4.3 después de la i-ésima iteración del ciclo for, líneas 5-19. Más precisamente, el conjunto de aristas de T_i es el conjunto E construido después de la i-ésima iteración del ciclo for, líneas 5–19, y el conjunto de vértices de T_i es el conjunto de vértices donde inciden las aristas de E. Sea T_0 la gráfica construida por el algoritmo 7.4.3 justo antes de entrar al ciclo for en la línea 5 por primera vez; T_0 consta del único vértice s, sin aristas. En esta demostración, eliminaremos el conjunto de vértices y haremos referencia a una gráfica especificando su conjunto de aristas.

Por construcción, al concluir el algoritmo 7.4.3, la gráfica construida T_{n-1} es una subgráfica conexa, acíclica, de la gráfica dada G y que contiene a todos los vértices de G; por tanto, T_{n-1} es un árbol de expansión de G.

Utilizamos inducción para mostrar que para toda $i = 0, \ldots, n-1$, T_i está contenida en un árbol de expansión mínimo. Esto implica que, al concluir, T_{n-1} es un árbol de expansión mínimo.

Si $i = 0$, T_0 consta de un único vértice. En este caso, T_0 está contenida en todo árbol de expansión mínimo. Hemos verificado el paso base.

A continuación, supongamos que T_i está contenida en un árbol de expansión mínimo T'. Sea V el conjunto de vértices en T_i. El algoritmo 7.4.3 elige una arista (j, k) de peso mínimo, tal que $j \in V$ y $k \notin V$ y la agrega a T_i para producir T_{i+1}. Si (j, k) está en T', entonces T_{i+1} está contenida en el árbol de expansión mínimo T'. Si (j, k) no está en T', $T' \cup \{(j, k)\}$ contiene un ciclo C. Se elige una arista (x, y) en C, distinta de (j, k), con $x \in$ y $\notin V$. Entonces

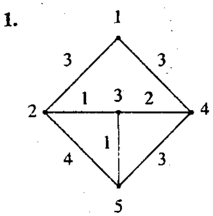
$$w(x, y) \geq w(j, k). \qquad (7.4.1)$$

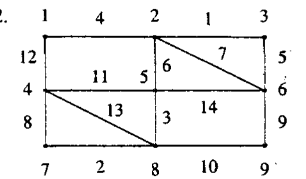
Debido a (7.4.1), la gráfica $T'' = [T' \cup \{(j, k)\}] - \{(x, y)\}$ tiene peso menor o igual al peso de T'. Como T'' es un árbol de expansión, T'' es un árbol de expansión mínimo. Como T_{i+1} está contenido en T'', hemos verificado el paso inductivo y hemos concluido la demostración. ■

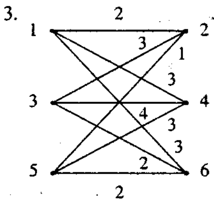
Nuestra versión del algoritmo de Prim examina $\Theta(n^3)$ aristas en el peor de los casos (véase el ejercicio 6) para determinar un árbol de expansión mínimo para una gráfica con n vértices. Es posible (véase el ejercicio 8) implantar el algoritmo de Prim de modo que sólo examine $\Theta(n^2)$ aristas en el peor de los casos. Como K_n tiene $\Theta(n^2)$ aristas, esta última versión es óptima.

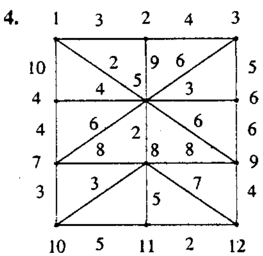
Ejercicios

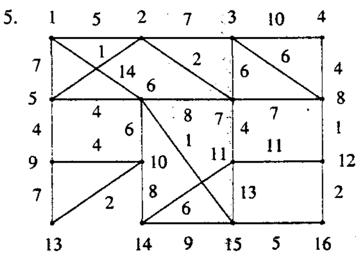
En los ejercicios 1-5, determine el árbol de expansión mínimo dado por el algoritmo 7.4.3 para cada gráfica.

1.


2.


3.


4.


5.


6. Muestre que el algoritmo 7.4.3 examina $\Theta(n^3)$ aristas en el peor de los casos.

Los ejercicios 7-9 se refieren a una versión alternativa del algoritmo de Prim (algoritmo 7.4.6).

ALGORITMO 7.4.6 ***Versión alternativa del algoritmo de Prim***

Este algoritmo determina un árbol de expansión mínimo en una gráfica conexa, con pesos, G. En cada paso, algunos vértices tienen etiquetas temporales y otros tienen etiquetas permanentes. La etiqueta del vértice i se denota L_i.

Entrada: Una gráfica conexa con pesos, con vértices $1, \ldots, n$ y vértice inicial s. Si (i, j) es una arista, $w(i, j)$ es igual al peso de (i, j); si (i, j) no es una arista, entonces $w(i, j)$ es igual a ∞ (un valor mayor que cualquier peso real).

Salida: Un árbol de expansión mínimo T.

```
procedure prim_alternate (w, n, s)
  sea T la gráfica con vértice s, sin aristas
  for j := 1 to n do
    begin
    L_j := w (s, j) // estas etiquetas son temporales
    back (j) := s
    end
  L_s := 0
  hacer L_s permanente
  while existen etiquetas temporales do
    begin
    elegir la menor etiqueta temporal L_i
    hacer L_i permanente
    agregar arista (i, back (i)) a T
    agregar vértice i a T
    for cada etiqueta temporal L_k do
      if w (i, k) < L_k then
        begin
        L_k := w (i, k)
        back(k) := i
        end
    end
  return(T)
end prim_alternate
```

7. Muestre la forma en que el algoritmo 7.4.6 determina un árbol de expansión mínimo para las gráficas de los ejercicios 1-5.
8. Muestre que el algoritmo 7.4.6 examina $O(n^2)$ aristas en el peor de los casos.
9. Demuestre que el algoritmo 7.4.6 es correcto; es decir, que al concluir el algoritmo 7.4.6, T es un árbol de expansión mínimo.
10. Sea G una gráfica conexa con pesos, sea v un vértice en G y e una arista de peso mínimo incidente en v. Muestre que e está contenida en algún árbol de expansión mínimo.
11. Sea G una gráfica conexa con pesos y v un vértice en G. Suponga que los pesos de las aristas incidentes en v son distintos. Sea e la arista de peso mínimo incidente en v. ¿Debe e estar contenida en algún árbol de expansión mínimo?
12. Muestre que cualquier algoritmo que determine un árbol de expansión mínimo en K_n, donde todos los pesos son iguales, debe examinar cada arista en K_n.
13. Muestre que si todos los pesos de una gráfica conexa G son distintos, entonces G tiene un único árbol de expansión mínimo.

En los ejercicios 14-16, decida si la afirmación es verdadera o falsa. Si la afirmación es verdadera, demuéstrela; en caso contrario, proporcione un contraejemplo. En cada ejercicio, G es una gráfica conexa con pesos.

14. Si todos los pesos en G son distintos, los diversos árboles de expansión de G tendrán pesos distintos.
15. Si e es una arista en G cuyo peso es menor que el peso de cualquier otra arista, e está en cada árbol de expansión mínimo de G.

16. Si T es un árbol de expansión mínimo de G, existe un etiquetado de los vértices de G de modo que el algoritmo 7.4.3 produce a T.
17. Sea G una gráfica conexa con pesos. Muestre que si, mientras sea posible, eliminamos una arista de G con peso máximo y cuya eliminación no desconecte a G, el resultado es un árbol de expansión mínimo de G.
☆ 18. Escriba un algoritmo que determine un árbol de expansión máximo en una gráfica conexa con pesos.
19. Demuestre que su algoritmo del ejercicio 18 es correcto.

El *algoritmo de Kruskal* determina un árbol de expansión mínimo en una gráfica conexa con pesos G con n vértices de la manera siguiente. La gráfica T consta inicialmente de los vértices de G y ninguna arista. En cada iteración, agregamos una arista e a T con peso mínimo y que no complete un ciclo en T. Cuando T tenga $n - 1$ aristas, nos detenemos.

20. Enuncie de manera formal el algoritmo de Kruskal.
21. Muestre la forma en que el algoritmo de Kruskal determina árboles de expansión mínimos para las gráficas de los ejercicios 1-5.
22. Muestre que el algoritmo de Kruskal es correcto; es decir, que al concluir el algoritmo de Kruskal, T es un árbol de expansión mínimo.
23. Sea V un conjunto de n vértices y s una "función de disimilaridad" en $V \times V$ (véase el ejemplo 6.1.6). Sea G la gráfica completa con pesos con conjunto de vértices V y pesos $w(v_i, v_j) = s(v_i, v_j)$. Modifique el algoritmo de Kruskal de modo que agrupe los datos en clases. Esta modificación se conoce como el **método de los vecinos más cercanos** (véase [Gose]).

Los ejercicios 24-30 se refieren a la siguiente situación. Suponga que tenemos estampillas de diversas denominaciones y que queremos elegir la menor cantidad de estampillas para pagar una tarifa postal. Consideremos un algoritmo codicioso que selecciona las estampillas eligiendo primero la mayor cantidad posible de estampillas de la mayor denominación, luego la mayor cantidad posible de estampillas de la segunda mayor denominación, etcétera.

24. Muestre que si las denominaciones disponibles son 1, 8 y 10 centavos, el algoritmo no siempre produce la menor cantidad de estampillas para pagar cierta tarifa postal.
☆ 25. Muestre que si las denominaciones disponibles son 1, 5 y 25 centavos, el algoritmo produce la menor cantidad de estampillas para pagar cierta tarifa postal.
26. Determine enteros positivos a_1 y a_2, tales que $a_1 > 2a_2 > 1$, a_2 no divide a a_1, y que el algoritmo con denominaciones $1, a_1, a_2$, no siempre produce la menor cantidad de estampillas para pagar cierta tarifa postal.
★ **27.** Determine enteros positivos a_1 y a_2, tales que $a_1 > 2a_2 > 1$, a_2 no divide a a_1, y que el algoritmo con denominaciones $1, a_1, a_2$, produce la menor cantidad de estampillas para pagar cierta tarifa postal. Demuestre que sus valores proporcionan una solución óptima.
☆ 28. Suponga que las denominaciones posibles son

$$1 = a_1 < a_2 < \cdots < a_n.$$

Muestre, mediante contraejemplos, que la condición

$$a_i \geq 2a_{i-1} - a_{i-2}, \qquad 3 \leq i \leq n$$

no es necesaria ni suficiente para que el algoritmo codicioso sea óptimo.
☆ 29. Muestre que el algoritmo codicioso es óptimo para las denominaciones

$$1 = a_1 < a_2 < a_3$$

si y sólo si el algoritmo codicioso es óptimo para $n = 1, 2, \ldots, a_2 a_3 - 1$.

★ **30.** Muestre que el algoritmo codicioso es óptimo para las denominaciones

$$1 = a_1 < a_2 < \cdots < a_m$$

si y sólo si el algoritmo codicioso es óptimo para $n = 1, 2, \ldots, k$, donde

$$k = \sum_{i=1}^{m-1} \left(\frac{a_{i+1}}{\text{mcd}(a_{i+1}, a_i)} - 1 \right) a_i .$$

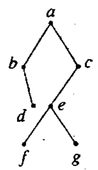


FIGURA 7.5.1
Un árbol binario.

7.5 *ÁRBOLES BINARIOS*

Los **árboles binarios** son de los tipos particulares más importantes de árboles con raíz. Cada vértice de un árbol binario tiene a lo más dos hijos (véase la figura 7.5.1). Además, cada hijo se designa como **hijo izquierdo** o **hijo derecho**. Al trazar un árbol binario, un hijo izquierdo se dibuja a la izquierda y un hijo derecho se dibuja a la derecha. A continuación damos la definición formal.

DEFINICIÓN 7.5.1

Un *árbol binario* es un árbol con raíz en el cual cada vértice tiene cero, uno o dos hijos. Si un vértice tiene un hijo, ese hijo se designa como un hijo izquierdo o un hijo derecho (pero no ambos). Si un vértice tiene dos hijos, uno de ellos de designa como un hijo izquierdo y el otro se designa como un hijo derecho.

EJEMPLO 7.5.2

En el árbol binario de la figura 7.5.1, el vértice b es el hijo izquierdo del vértice a y el vértice c es el hijo derecho del vértice a. El vértice d es el hijo derecho del vértice b; el vértice b no tiene un hijo izquierdo. El vértice e es el hijo izquierdo del vértice c; el vértice c no tiene un hijo derecho. □

EJEMPLO 7.5.3

Un árbol que define un código de Huffman es un árbol binario. Por ejemplo, en el árbol del código de Huffman de la figura 7.1.8, el paso de un vértice a un hijo izquierdo corresponde a utilizar el bit 1 y el paso de un vértice a un hijo derecho corresponde al uso del bit 0. □

Un **árbol binario completo** es un árbol binario en el cual cada vértice tiene dos o cero hijos. Un resultado fundamental acerca de los árboles binarios completos es el siguiente teorema.

TEOREMA 7.5.4

Si T es un árbol binario completo con i vértices internos, entonces T tiene $i + 1$ vértices terminales y $2i + 1$ vértices en total.

Demostración. Los vértices de T constan de los vértices que son hijos (de algún padre) y los vértices que no son hijos (de ningún padre). Existe un vértice que no es hijo de nadie: la raíz. Como existen i vértices internos, cada uno de los cuales tiene dos hijos, existen $2i$ hijos. Así, la cantidad total de vértices de T es $2i + 1$ y el número de vértices terminales es

$$(2i + 1) - i = i + 1.$$ ■

EJEMPLO 7.5.5

Un torneo de eliminación simple es aquél en que cada competidor se elimina después de perder una vez. La gráfica de un torneo de eliminación simple es un árbol binario completo (véase la figura 7.5.2). Los nombres de los competidores aparecen a la izquierda. Los ganadores siguen avanzando hacia la derecha. En cierto momento, existe un único ganador en la raíz. Si el número de competidores no es una potencia de 2, algunos competidores pueden pasar a la siguiente ronda sin tener que jugar; en la figura 7.5.2, el competidor 7 está en este caso.

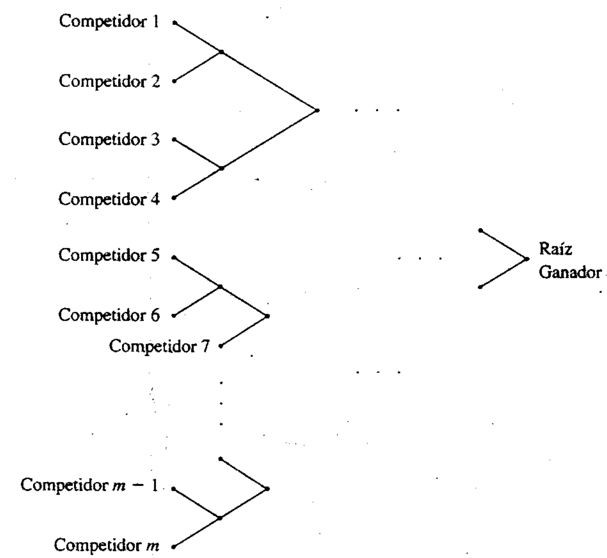


FIGURA 7.5.2 La gráfica (árbol binario completo) de un torneo de eliminación simple.

Mostraremos que si existen n competidores en un torneo de eliminación simple, se juegan un total de $n - 1$ encuentros.

El número de competidores es igual al número de vértices terminales y el número de encuentros i es igual al número de vértices internos. Así, por el teorema 7.5.4,

$$n + i = 2i + 1 ,$$

de modo que $i = n - 1$. □

Nuestro siguiente resultado relativo a los árboles binarios relaciona el número de vértices terminales con la altura.

TEOREMA 7.5.6

Si un árbol binario de altura h tiene t vértices terminales, entonces

$$\lg t \leq h. \tag{7.5.1}$$

Demostración. Demostraremos la desigualdad equivalente

$$t \leq 2^h \tag{7.5.2}$$

por inducción sobre h. La desigualdad (7.5.1) se obtiene de (7.5.2) tomando el logaritmo en base 2 de ambos lados de (7.5.2).

Si $h = 0$, el árbol binario consta de un único vértice. En este caso, $t = 1$ y entonces (7.5.2) es verdadero.

Suponga que el resultado es válido para un árbol binario cuya altura sea menor que h. Sea T un árbol binario de altura $h > 0$ con t vértices terminales. Suponga primero que la raíz de T tiene sólo un hijo. Si eliminamos la raíz y la arista incidente en la raíz, el árbol resultante tiene altura $h - 1$ y el mismo número de vértices terminales que T. Por inducción, $t \leq 2^{h-1}$. Como $2^{h-1} < 2^h$, (7.5.2) queda establecida para este caso.

Ahora supongamos que la raíz de T tiene hijos v_1 y v_2. Sea T_i el subárbol con raíz en v_i y suponga que T_i tiene altura h_i y t_i vértices terminales, $i = 1, 2$. Por inducción,

$$t_i \leq 2^{h_i}, \quad i = 1, 2. \tag{7.5.3}$$

Los vértices terminales de T constan de los vértices terminales de T_1 y T_2. Por tanto,

$$t = t_1 + t_2. \tag{7.5.4}$$

Al combinar (7.5.3) y (7.5.4), obtenemos

$$t = t_1 + t_2 \leq 2^{h_1} + 2^{h_2} \leq 2^{h-1} + 2^{h-1} = 2^h.$$

Hemos verificado el paso inductivo, con lo cual concluye la demostración. ■

EJEMPLO 7.5.7

El árbol binario de la figura 7.5.3 tiene altura $h = 3$ y el número de vértices terminales es $t = 8$. Para este árbol, la desigualdad (7.5.1) se convierte en una igualdad. □

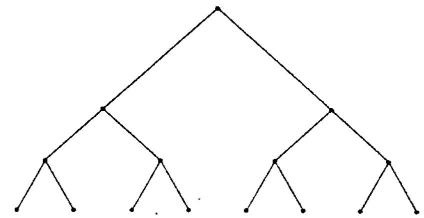


FIGURA 7.5.3 Un árbol binario de altura $h = 3$ con $t = 8$ vértices terminales. Para este árbol binario, $\lg t = h$.

Suponga que tenemos un conjunto S cuyos elementos se pueden ordenar. Por ejemplo, si S consta de números, podemos utilizar el orden común definido sobre los números, y si S consta de cadenas de caracteres alfabéticos, podemos utilizar el orden lexicográfico. Los árboles binarios se utilizan ampliamente en las ciencias de la computación para guardar los elementos de un conjunto ordenado, como un conjunto de número o un conjunto de cadenas. Si el elemento dato $d(v)$ se guarda en el vértice v y el elemento de dato $d(w)$ se guarda en el vértice w, entonces, si v es un hijo izquierdo (o un hijo derecho) de w, se puede garantizar que existe cierta relación de orden entre $d(v)$ y $d(w)$. Un ejemplo es un **árbol de búsqueda binaria**.

DEFINICIÓN 7.5.8

Un *árbol de búsqueda binaria* es un árbol binario T en el cual se asocian ciertos datos con los vértices. Los datos están ordenados de modo que, para *cada* vértice v en T, cada elemento de dato en el subárbol izquierdo de v sea menor que el elemento de dato en v y cada elemento de dato en el subárbol derecho de v es mayor que el elemento de dato en v.

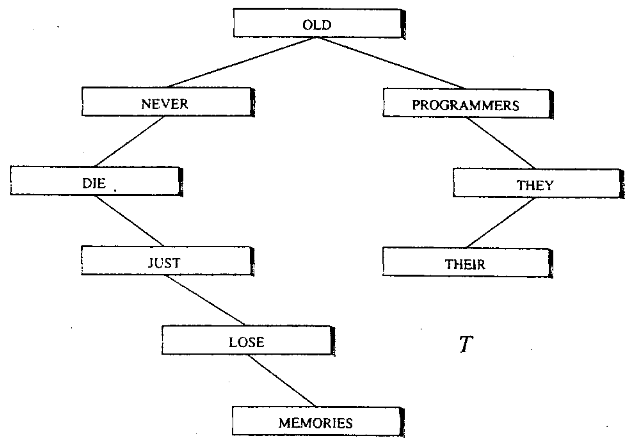


FIGURA 7.5.4 Un árbol de búsqueda binaria.

EJEMPLO 7.5.9

Las palabras

OLD PROGRAMMERS NEVER DIE

THEY JUST LOSE THEIR MEMORIES (7.5.5)

(Los viejos programadores nunca mueren; sólo pierden su memoria) se pueden colocar en un árbol de búsqueda binaria, como muestra la figura 7.5.4. Observe que para cualquier vértice v, cada elemento de dato en el subárbol izquierdo de v es menor (es decir, precede alfabéticamente) que el elemento de dato en v y cada elemento de dato en el subárbol derecho de v es mayor que el elemento de dato en v.

En general, habrá muchas formas de colocar datos en un árbol de búsqueda binaria. La figura 7.5.5 muestra otro árbol de búsqueda binaria que guarda las palabras (7.5.5).

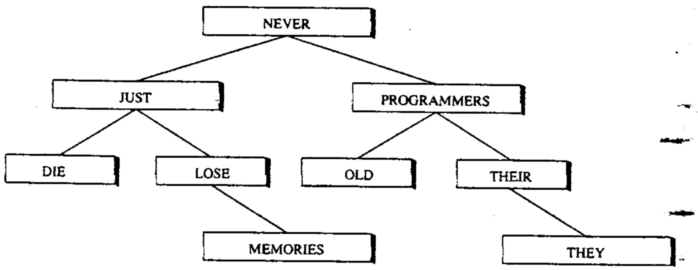


FIGURA 7.5.5 Otro árbol de búsqueda binaria que guarda las mismas palabras que el árbol de la figura 7.5.4.

El árbol de búsqueda binaria T de la figura 7.5.4 se construyó de la siguiente forma. Comenzamos con un **árbol vacío**, es decir, un árbol sin vértices ni aristas. Luego inspeccionamos cada una de las palabras (7.5.5) *en el orden en que aparecen*, comenzando con OLD, luego con PROGRAMMERS, luego NEVER, y así sucesivamente. Para comenzar, creamos un vértice y colocamos la primera palabra OLD en este vértice. Designamos este vértice como la raíz. A partir de este punto, dada una palabra en la lista (7.5.5), agregamos un vértice v y una arista al árbol y colocamos la palabra en el vértice v. Para decidir dónde agregar el vértice y la arista, comenzamos en la raíz. Si la palabra por agregar es menor que la palabra de la raíz (en el orden lexicográfico), pasamos al hijo izquierdo y si la palabra por agregar es mayor que la palabra de la raíz, pasamos al hijo derecho. Si no existe tal hijo, creamos uno, lo colocamos en una arista incidente en la raíz y en el nuevo vértice y colocamos la palabra en el nuevo vértice. Si existe un hijo v, repetimos este proceso. Es decir, comparamos la palabra por agregar con la palabra en v y nos movemos al hijo derecho de v si la palabra por agregar es menor que la palabra en v; en caso contrario, nos movemos al hijo derecho de v. Si no existe un hijo al cual movernos, creamos uno, lo colocamos en una arista incidente en v y el nuevo vértice y colocamos la palabra en el nuevo vértice. Si existe un hijo al cual movernos, repetimos este proceso. En algún momento colocamos la palabra en el árbol. Luego vamos por la siguiente palabra en la lista, la comparamos con la raíz, nos movemos hacia la izquierda o la derecha, la comparamos con el nuevo vértice, nos movemos hacia la izquierda o hacia la derecha, y así sucesivamente, de modo que en algún momento lo guardamos en el árbol. De esta forma, guardamos todas las palabras en el árbol y así creamos un árbol de búsqueda binaria. Establecemos este método de construcción de un árbol de búsqueda binaria de manera formal como el algoritmo 7.5.10.

ALGORITMO 7.5.10 ***Construcción de un árbol de búsqueda binaria***

Este algoritmo construye un árbol de búsqueda binaria. La entrada se lee en el orden con el cual fue enviada. Después de leer cada palabra, ésta se inserta en el árbol.

Entrada: Una sucesión $w_1, \ldots, w_n$ de palabras distintas y la longitud n de la sucesión

Salida: Un árbol de búsqueda binaria T

```
procedure make_bin_search_tree(w, n)
  sea T el árbol con un vértice, root.
  guardar w_1 en root
  for i := 2 to n do
    begin
    v := root
    search := true     // encontrar un lugar para w_i
    while search do
      begin
      s := palabra en v
      if w_i < s then
        if v no tiene hijo izquierdo then
          begin
          agregar un hijo izquierdo l a v
          guardar w_i en l
          search := false     // fin de la búsqueda
          end
        else
          v := hijo izquierdo de v
      else // w_i > s
        if v no tiene hijo derecho then
          begin
          agregar un hijo derecho r a v
          guardar w_i en r
          search := false     // fin de la búsqueda
          end
        else
          v := hijo derecho de v
      end     // mientras
    end     // para
  return(T)
end make_bin_search_tree
```

Los árboles de búsqueda binaria son útiles para localizar datos. Es decir, dado un elemento D, podemos determinar con facilidad si D está en un árbol de búsqueda binaria y, de estar presente, conocer su posición. Para determinar si un elemento de dato D está en un árbol de búsqueda binaria, comenzaríamos en la raíz. Luego compararíamos de manera sucesiva D con el elemento de dato en el vértice en cuestión. Si D es igual al elemento de dato en el vértice en cuestión, hemos encontrado a D, por lo cual habremos concluido. Si D es menor que el elemento de dato en el vértice en cuestión v, nos movemos al hijo izquierdo de v y repetimos el proceso. Si D es mayor que el elemento de dato en el vértice en cuestión v, nos movemos al hijo derecho de v y repetimos el proceso. Si en algún momento no existe un hijo al cual moverse, podemos concluir que D no está en el árbol. (El ejercicio 2 pide establecer de manera formal este proceso.)

El tiempo utilizado en la búsqueda de un elemento en un árbol de búsqueda binaria es máximo cuando el elemento no está presente y seguimos el camino de mayor longitud desde la raíz. Así el tiempo máximo para buscar un elemento en un árbol de búsqueda binaria es aproximadamente proporcional a la altura del árbol. Por tanto, si la altura de un árbol de búsqueda binaria es pequeña, la búsqueda en el árbol será siempre muy rápida (véase el ejercicio 21). Se conocen muchas formas de minimizar la altura de un árbol de búsqueda binaria (véase, por ejemplo, [Cormen]).

Estableceremos un enunciado más preciso acerca de la búsqueda en un árbol de búsqueda binaria en el peor de los casos. Sea T un árbol de búsqueda binaria con n vértices y sea T^* el árbol binario completo obtenido de T al agregar hijos izquierdos y derechos a los vértices existentes de T, cuando esto sea posible. En la figura 7.5.6, mostramos el árbol binario completo resultante de modificar el árbol de búsqueda binaria de la figura 7.5.4. Los vértices agregados se dibujan como cajas. Una búsqueda sin éxito en T corresponde a llegar a un vértice agregado (caja) en T^*. Definiremos el tiempo necesario para realizar el procedimiento de búsqueda en el peor de los casos como la altura h del árbol T^*. Por el teorema 7.5.6, $\lg t \le h$, donde t es el número de vértices terminales en T^*. El árbol binario completo T^* tiene n vértices internos, de modo que por el teorema 7.5.4, $t = n + 1$. Así, en el peor de los casos, el tiempo será igual al menos a $\lg t = \lg(n + 1)$. El ejercicio 3 muestra que si se minimiza la altura de T, el peor de los casos requiere un tiempo igual a $\lceil \lg(n + 1) \rceil$. Por ejemplo, como

$$\lceil \lg(2{,}000{,}000 + 1) \rceil = 21,$$

es posible guardar 2 millones de datos en un árbol de búsqueda binaria y encontrar uno de estos elementos, o determinar que no está presente en el árbol, en a lo más 21 pasos.

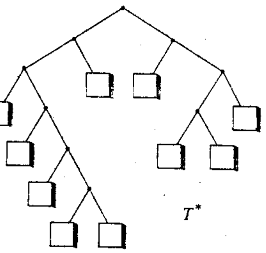


FIGURA 7.5.6
Ampliación de un árbol de búsqueda binaria a un árbol binario completo.

Ejercicios

1. Coloque las palabras FOUR SCORE AND SEVEN YEARS AGO OUR FOREFATHERS BROUGHT FORTH. en orden de aparición, en un árbol de búsqueda binaria.
2. Escriba un algoritmo formal para la búsqueda en un árbol de búsqueda binaria.
3. Escriba un algoritmo que guarde n palabras distintas en un árbol de búsqueda binaria T de altura mínima. Muestre que el árbol derivado T^*, según lo descrito en el texto, tiene altura $\lceil \lg(n + 1) \rceil$.
4. ¿Verdadero o falso? Sea T un árbol binario. Si para cada vértice v en T el elemento de dato en v es mayor que el elemento de dato en el hijo izquierdo de v, y el elemento de dato en v es menor que el elemento de dato en el hijo derecho de v, entonces T es un árbol de búsqueda binaria. Explique.

En los ejercicios 5-7, trace una gráfica con las propiedades dadas o explique por qué no existe tal gráfica.

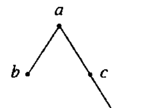
5. Árbol binario completo; cuatro vértices internos; cinco vértices terminales
6. Árbol binario completo; altura = 3; nueve vértices terminales
7. Árbol binario completo; altura = 4; nueve vértices terminales

8. Un **árbol *m*-ario completo** es un árbol con raíz tal que cada padre tiene m hijos ordenados. Si T es un árbol m-ario completo con i vértices internos, ¿cuántos vértices tiene T? ¿Cuántos vértices terminales tiene T? Demuestre sus resultados.
9. Proporcione un algoritmo para construir un árbol binario completo con $n > 1$ vértices terminales.
10. Proporcione un algoritmo recursivo para insertar una palabra en un árbol de búsqueda binaria.
11. Determine la altura máxima de un árbol de búsqueda binaria con t vértices terminales.
12. Escriba un algoritmo que verifique si un árbol binario en cuyos vértices se guardan datos es un árbol de búsqueda binaria.
13. Sea T un árbol binario completo. Sea I la suma de las longitudes de los caminos simples de la raíz a los vértices internos. I es la *longitud interna*. Sea E la suma de las longitudes de los caminos simples de la raíz a los vértices terminales. E es la *longitud externa*. Demuestre que si T tiene n vértices internos, entonces $E = I - 2n$.

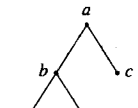
Un árbol binario T está *balanceado* si para cada vértice v en T, las alturas de los subárboles izquierdo y derecho de v difieren a lo más en 1. (En este caso, la altura de un árbol vacío se define como -1.)

Indique si cada árbol de los ejercicios 14-17 es balanceado o no.

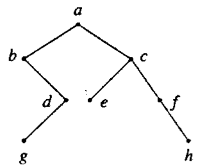
14.



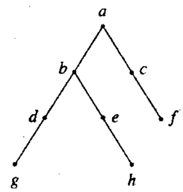
15.



16.



17.



En los ejercicios 18-20, N_h se define como el menor número de vértices en un árbol binario balanceado de altura h y $f_1, f_2, \ldots$ denota la sucesión de Fibonacci.

18. Muestre que $N_0 = 1$, $N_1 = 2$ y $N_2 = 4$.
19. Muestre que $N_h = 1 + N_{h-1} + N_{h-2}$, para $h \ge 0$.
20. Muestre que $N_h = f_{h+2} - 1$, para $h \ge 0$.

★ 21. Muestre que la altura h de un árbol binario balanceado con n vértices satisface $h = O(\lg n)$. Este resultado muestra que el tiempo de búsqueda en un árbol binario balanceado con n vértices es $O(\lg n)$, en el peor de los casos.

☆ 22. Demuestre que si un árbol binario de altura h tiene $n \ge 1$ vértices, entonces $\lg n < h + 1$. Este resultado, junto con el ejercicio 21, muestra que el tiempo de búsqueda en un árbol binario balanceado con n vértices es $\Theta(\lg n)$, en el peor de los casos.

7.6 RECORRIDOS DE UN ÁRBOL

La búsqueda a lo ancho y la búsqueda a profundidad proporcionan formas de "recorrer" un árbol, es decir, de recorrerlo de manera sistemática de modo que cada vértice sea visitado exactamente una vez. En esta sección consideraremos otros tres métodos para recorrer un árbol. Definiremos estos recorridos de manera recursiva.

ALGORITMO 7.6.1 *Recorrido en preorden*

Este algoritmo recursivo procesa los vértices de un árbol binario utilizando el recorrido en preorden.

Entrada: *PT*, la raíz de un árbol binario

Salida: Depende de la interpretación de "procesar" en la línea 3

```
   procedure preorder (PT)
1.   if PT es vacío then
2.      return
3.    procesar PT
4.    l := hijo izquierdo de PT
5.    preorder (l)
6.    r := hijo derecho de PT
7.    preorder (r)
   end preorder
```

Examinaremos el algoritmo 7.6.1 para algunos casos sencillos. Si el árbol binario es vacío, nada se procesa pues, en este caso, el algoritmo simplemente regresa en la línea 2.

Suponga que la entrada consta de un árbol con un único vértice. Hacemos *PT* igual a la raíz y llamamos a *preorder*(*PT*). Como *PT* no es vacío, pasamos a la línea 3, donde procesamos la raíz. En la línea 5, llamamos a *preorder* con *PT* igual al hijo izquierdo (vacío) de la raíz. Sin embargo, hemos visto que cuando la entrada es un árbol vacío, *preorder* no procesa nada. De manera análoga, en la línea 7 utilizamos un árbol vacío como entrada de *preorder* y nada se procesa. Así, cuando la entrada consta de un árbol con un único vértice, procesamos la raíz y regresamos.

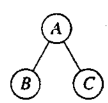


FIGURA 7.6.1
Entrada del algoritmo 7.6.1.

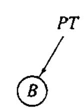


FIGURA 7.6.2
En la línea 5 del algoritmo 7.6.1, donde la entrada es el árbol de la figura 7.6.1.

Ahora, supongamos que la entrada es el árbol de la figura 7.6.1. Hacemos *PT* igual a la raíz y llamamos a *preorder*(*PT*). Como *PT* no es vacío, pasamos a la línea 3, donde procesamos a la raíz. En la línea 5 llamamos a *preorder*, con *PT* igual al hijo izquierdo de la raíz (véase la figura 7.6.2). Acabamos de ver que si la entrada de *preorder* consta de un único vértice, *preorder* procesa ese vértice. Así, a continuación procesamos el vértice *B*. De manera análoga, en la línea 7, procesamos el vértice *C*. Así, los vértices se procesan en el orden *ABC*.

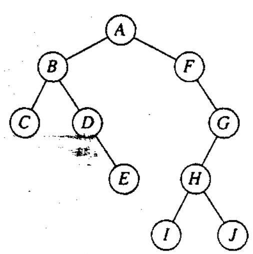


FIGURA 7.6.3
Un árbol binario. El preorden es *A B C D E F G H I J*. El entreorden es *C B D E A F I H J G*.
El posorden es *C E D B I J H G F A*.

EJEMPLO 7.6.2

¿En qué orden se procesan los vértices del árbol de la figura 7.6.3 si se utiliza el recorrido en preorden?

Si seguimos las líneas 3-7 (raíz/izquierdo/derecho) del algoritmo 7.6.1, el recorrido se realiza como muestra la figura 7.6.4. Así, el orden de procesamiento es

A B C D E F G H I J. □

Los recorridos en entreorden y en posorden se obtienen cambiando la posición de la línea 3 (raíz) en el algoritmo 7.6.1. Los prefijos pre, entre y pos se refieren a la posición de la raíz en el recorrido; es decir, "preorden" significa que va primero la raíz; "entreorden", quiere decir que la raíz va en segundo lugar, y "posorden" que la raíz va al final.

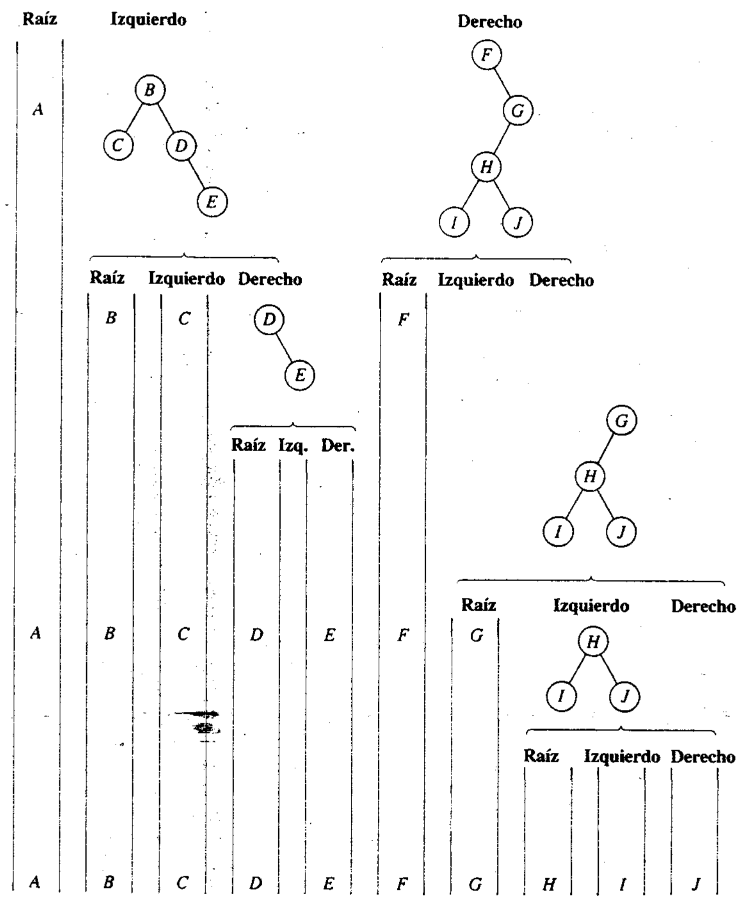


FIGURA 7.6.4 Recorrido en preorden del árbol de la figura 7.6.3.

ALGORITMO 7.6.3 *Recorrido en entreorden*

Este algoritmo recursivo procesa los vértices de un árbol binario utilizando el recorrido en entreorden.

Entrada: *PT*, la raíz de un árbol binario

Salida: Depende de la interpretación de "procesar" en la línea 5

```
   procedure inorder(PT)
1.   if PT es vacío then
2.     return
3.   l := hijo izquierdo de PT
4.   inorder(l)
5.   procesar PT
6.   r := hijo derecho de PT
7.   inorder(r)
   end inorder
```

EJEMPLO 7.6.4

¿En qué orden se procesan los vértices del árbol de la figura 7.6.3 si se utiliza el recorrido en entreorden?

Si seguimos las líneas 3-7 (izquierdo/raíz/derecho) del algoritmo 7.6.3, obtenemos la enumeración en entreorden *CBDEAFIHJG*. □

ALGORITMO 7.6.5 *Recorrido en posorden*

Este algoritmo recursivo procesa los vértices de un árbol binario utilizando el recorrido en posorden.

Entrada: *PT*, la raíz de un árbol binario

Salida: Depende de la interpretación de "procesar" en la línea 7

```
   procedure postorder(PT)
1.   if PT es vacío then
2.     return
3.   l := hijo izquierdo de PT
4.   postorder(l)
5.   r := hijo derecho de PT
6.   postorder(r)
7.   procesar PT
   end postorder
```

EJEMPLO 7.6.6

¿En qué orden se procesan los vértices del árbol de la figura 7.6.3 si se utiliza el recorrido en posorden?

Si seguimos las líneas 3-7 (izquierdo/derecho/raíz) del algoritmo 7.6.5, obtenemos la enumeración en posorden *CEDBIJHGFA*. □

Observe que el recorrido en preorden se puede obtener siguiendo la ruta que aparece en la figura 7.6.5, y que el recorrido en posorden inverso se puede obtener siguiendo la ruta que aparece en la figura 7.6.6.

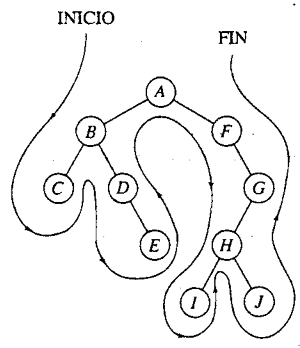


FIGURA 7.6.5
Recorrido en preorden.

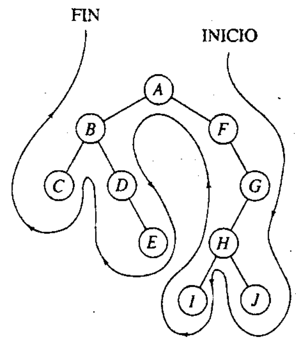


FIGURA 7.6.6
Recorrido en posorden inverso.

Si se guardan datos en un árbol de búsqueda binaria, como se describió en la sección 7.5, el recorrido en entreorden procesará los datos en orden, pues la secuencia izquierdo/raíz/derecho coincide con el orden de los datos en el árbol.

En el resto de esta sección consideraremos la representación de expresiones aritméticas mediante árboles binarios. Tales representaciones facilitan la evaluación de expresiones mediante computadoras.

Restringiremos nuestros operadores a $+$, $-$, $*$ y $/$. Un ejemplo de expresión que utiliza estos operadores es

$$(A + B) * C - D/E. \tag{7.6.1}$$

Esta manera canónica de representar las expresiones es la **forma entrefija de una expresión**. Las variables A, B, C, D y E se conocen como **operandos**. Los **operadores** $+$, $-$, $*$ y $/$ operan sobre pares de operandos o expresiones. En la forma entrefija de una expresión, un operador aparece entre sus operandos.

Una expresión como (7.6.1) se puede representar como un árbol binario. Los vértices terminales corresponden a los operandos, y los vértices internos corresponden a los operadores. La expresión (7.6.1) se representaría como en la figura 7.6.7. En la representación de una expresión mediante un árbol binario, un operador opera sobre sus subárboles izquierdo y derecho. Por ejemplo, en el subárbol cuya raíz es / en la figura 7.6.7, el operador de división opera sobre los operandos D y E; es decir, hay que dividir D entre E. En el subárbol cuya raíz es * en la figura 7.6.7, el operador de multiplicación opera sobre el subárbol encabezado por $+$, que a su vez representa una expresión, y C.

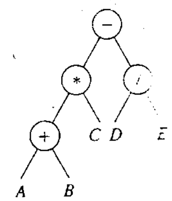


FIGURA 7.6.7
La representación mediante un árbol binario de la expresión $(A + B) * C - D/E$.

En un árbol binario, hacemos una distinción entre los subárboles izquierdo y derecho de un vértice. Los subárboles izquierdo y derecho de un vértice corresponden a los operandos izquierdo y derecho o a expresiones. Esta distinción es importante en las expresiones. Por ejemplo, 4-6 y 6-4 son diferentes.

Si recorremos el árbol binario de la figura 7.6.7 en entreorden, e insertamos un par de paréntesis para cada operación, obtenemos

$$(((A + B) * C) - (D / E)).$$

Esta forma de una expresión se llama la **forma con todos los paréntesis de la expresión**. De esta forma, no tenemos que especificar cuáles operaciones (como la multiplicación) deben realizarse antes que las demás (como la suma), pues los paréntesis indican el orden de las operaciones sin ambigüedad.

Si recorremos el árbol de la figura 7.6.7 en posorden, obtenemos

$$A B + C * D E / -.$$

Esta forma de la expresión es la **forma posfija de la expresión** (o **notación polaca inversa**). En la forma posfija, el operador aparece después de sus operandos. Por ejemplo, los primeros tres símbolos $A\,B+$ indican que A y B deben sumarse. La ventaja de la forma posfija sobre la forma entrefija es que en la notación posfija no se necesitan paréntesis ni convenciones en relación con el orden de las operaciones. La expresión se evaluará sin ambigüedad. Por éstas y otras razones, muchos compiladores traducen las expresiones entrefijas a su forma posfija. Además, algunas calculadoras necesitan que las expresiones sean introducidas en forma posfija.

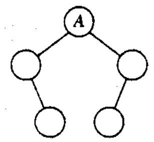
Se puede obtener una tercera forma de una expresión aplicando el recorrido en preorden a una representación de una expresión mediante un árbol binario. En este caso, el resultado es la **forma prefija de la expresión** (o **notación polaca**). Como en la forma posfija, no se necesitan paréntesis ni convenciones acerca del orden de las operaciones. La forma prefija de (7.6.1), obtenida al aplicar el recorrido en preorden al árbol de la figura 7.6.7, es

$$-*+ABC/DE.$$

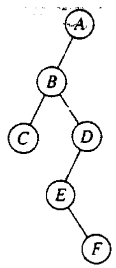
Ejercicios

En los ejercicios 1-5, indique el orden en que se procesan los vértices en un recorrido en preorden, entreorden y posorden.

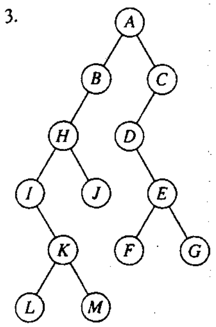
1.



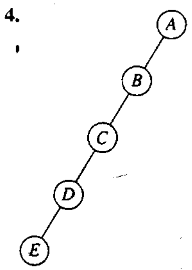
2.



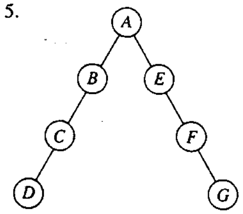
3.



4.



5.



En los ejercicios 6-10, represente la expresión como un árbol binario y escriba las formas prefija y posfija de la expresión.

6. $(A + B) * (C - D)$
7. $((A - C) * D) / (A + (B + D))$
8. $(A * B + C * D) - (A/B - (D + E))$
9. $(((A + B) * C + D) * E) - ((A + B) * C - D)$
10. $(A * B - C/D + E) + (A - B - C - D * D) / (A + B + C)$

En los ejercicios 11-15, represente la expresión posfija como un árbol binario y escriba la forma prefija, la forma entrefija usual y la forma entrefija con todos los paréntesis de la expresión.

11. $A B + C -$
12. $ABC+-$
13. $ABCD+*/E-$
14. $ABC**CDE+/-$
15. $AB + CD*EF/--A*$

En los ejercicios 16-21, determine el valor de la expresión posfija si $A = 1$, $B = 2$, $C = 3$ y $D = 4$.

16. $ABC+-$
17. $AB+C-$
18. $AB+CD*AA/--B*$
19. $ABC**ABC++-$
20. $ABAB*+*D*$
21. $ADBCD*-+*$

22. Muestre, mediante un ejemplo, que distintos árboles binarios con vértices A, B y C pueden tener la misma enumeración en preorden $A\,B\,C$.

23. Muestre que existe un único árbol binario con seis vértices cuya enumeración de los vértices en preorden es $A\,B\,C\,E\,F\,D$ y cuya enumeración de los vértices en entreorden es $A\,C\,F\,E\,B\,D$.

☆ 24. Escriba un algoritmo que reconstruya el árbol binario, dadas sus enumeraciones de los vértices en preorden y en entreorden.

25. Proporcione ejemplos de árboles binarios distintos, B_1 y B_2, cada uno con dos vértices, tales que la enumeración de los vértices de B_1 en preorden sea igual a la enumeración de los vértices de B_2 en preorden y que la enumeración de los vértices de B_1 en posorden sea igual a la enumeración de los vértices de B_2 en posorden.

26. Sean P_1 y P_2 permutaciones de $A\,B\,C\,D\,E\,F$. ¿Existe un árbol binario con vértices A, B, C, D, E y F cuya enumeración en preorden sea P_1 y cuya enumeración en entreorden sea P_2? Explique.

27. Escriba un algoritmo recursivo que imprima el contenido de los vértices terminales de un árbol binario de izquierda a derecha.

28. Escriba un algoritmo recursivo que intercambie todos los hijos izquierdos y derechos de un árbol binario.

29. Escriba un algoritmo recursivo que inicialice cada vértice de un árbol binario con el número de sus descendientes.

En los ejercicios 30 y 31, cada expresión utiliza sólo los operandos $A, B, \ldots, Z$ y los operadores $+, -, *, /$.

★ **30.** Proporcione una condición necesaria y suficiente para que una cadena de símbolos sea una expresión posfija válida.

31. Escriba un algoritmo tal que, dada la representación de una expresión mediante un árbol binario, proporcione como salida la forma entrefija de la expresión con todos sus paréntesis.

32. Escriba un algoritmo que imprima los caracteres y sus códigos dado un árbol de codificación de Huffman (véase el ejemplo 7.1.7). Suponga que cada vértice terminal guarda un carácter y su frecuencia.

★ **33.** Una *cubierta de vértices* de una gráfica $G = (V, E)$ es un subconjunto W de V tal que para cada arista $(v, w) \in E$, entonces $v \in W$ o $w \in W$. El *tamaño* de una cubierta de vértices W es el número de vértices en W. Escriba un algoritmo que determina una cubierta de vértices de tamaño mínimo para un árbol $T = (V, E)$ cuyo tiempo en el peor de los casos sea $\Theta(|E|)$.

7.7 ÁRBOLES DE DECISIÓN Y EL TIEMPO MÍNIMO PARA EL ORDENAMIENTO

El árbol binario de la figura 7.7.1 proporciona un algoritmo para elegir un restaurante. Cada vértice interno pregunta algo. Si partimos de la raíz, respondemos cada pregunta, y seguimos la arista adecuada, llegaremos a un vértice terminal donde se elige un restaurante. Tal árbol es un **árbol de decisión**. En esta sección utilizaremos los árboles de decisión para especificar algoritmos y para obtener cotas inferiores para un ordenamiento en el peor de los casos, así como para resolver ciertos juegos con monedas. Comenzaremos con los juegos de monedas.

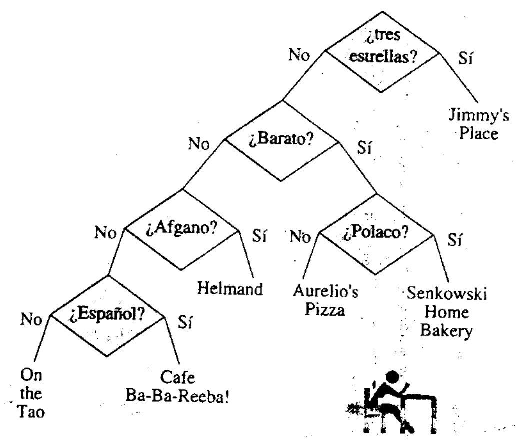


FIGURA 7.7.1 Un árbol de decisión.

EJEMPLO 7.7.1 ***Juego de cinco monedas***

Cinco monedas tienen apariencia idéntica, pero una de ellas es más pesada o más ligera que las demás, las cuales pesan lo mismo. El problema es identificar la moneda distinta y si es más pesada o más ligera que las demás, utilizando una balanza de platillos (véase la figura 7.7.2), la cual compara el peso de dos conjuntos de monedas.

En la figura 7.7.3 aparece un algoritmo para resolver este problema en forma de árbol de decisión. Las monedas se etiquetan como C_1, C_2, C_3, C_4, C_5. Como se muestra, comenzamos en la raíz y colocamos la moneda C_1 en el platillo izquierdo y la moneda C_2 en el platillo derecho. Una arista etiquetada indica que el lado izquierdo de la balanza es más pesado que el lado derecho. De manera análoga, una arista etiquetada indica que el lado derecho de la balanza es más pesado que el lado izquierdo y una arista etiquetada indica que los dos lados están en equilibrio. Por ejemplo, en la raíz, al comparar C_1 con C_2, si el lado izquierdo es más pesado que el lado derecho, sabemos que C_1 es la moneda pesada o que C_2 es la moneda ligera. En este caso, como muestra el árbol de decisión, continuamos comparando C_1 con C_5 (que sabemos es una moneda buena) y de inmediato determinamos si la moneda mala es C_1 o C_2 y si es más ligera o más pesada. Los vértices terminales proporcionan la solución. Por ejemplo, si al comparar C_1 con C_5 los platillos se equilibran, seguimos la arista hasta el vértice terminal etiquetado C_2, L, lo cual nos dice que la moneda mala es C_2 y que es más ligera que las demás. □

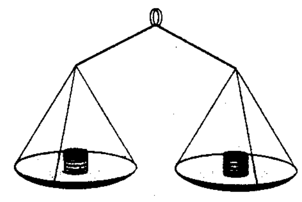


FIGURA 7.7.2 Una balanza de platillos para comparar pesos de monedas.

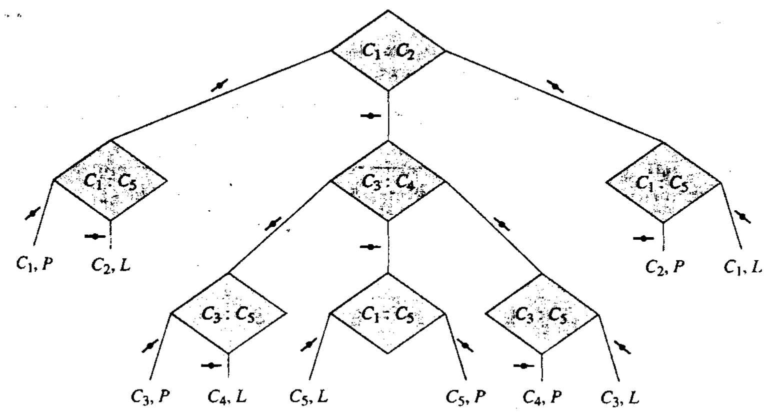


FIGURA 7.7.3 Un algoritmo para resolver el problema de las cinco monedas.

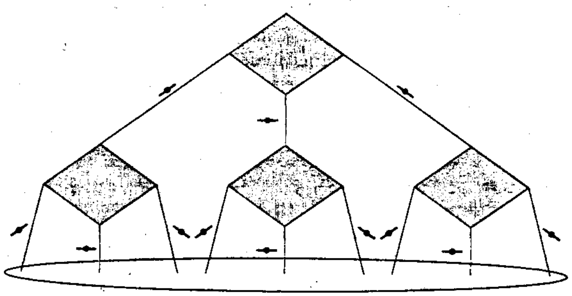
Si definimos el tiempo en el peor de los casos para el problema de pesar las monedas como el número de pesadas necesarias en el peor de los casos, es fácil determinar el tiempo en el peor de los casos a partir del árbol de decisión; el tiempo en el peor de los casos es igual a la altura del árbol. Por ejemplo, la altura del árbol de decisión de la figura 7.7.3 es 3, de modo que el tiempo en el peor de los casos para este algoritmo es igual a 3.

Podemos utilizar árboles de decisión para mostrar que el algoritmo dado en la figura 7.7.3 para resolver el juego de las cinco monedas es óptimo, es decir, que *ningún* algoritmo que resuelva el juego de las cinco monedas tiene un tiempo en el peor de los casos menor que 3.

Argumentaremos por contradicción para mostrar que ningún algoritmo que resuelva el juego de las cinco monedas tiene un tiempo en el peor de los casos menor que 3. Supongamos que existe un algoritmo que resuelve el juego de las cinco monedas en el peor de los casos en dos o menos pesadas. El algoritmo se puede describir mediante un árbol de decisión, y como el tiempo en el peor de los casos es 2 o menos, la altura del árbol de decisión es dos o menos. Como cada vértice interno tiene a lo más tres hijos, tal árbol debe tener a lo más nueve vértices terminales (véase la figura 7.7.4). Ahora, los vértices terminales corresponden a los posibles resultados. Así, un árbol de decisión de altura 2 o menos puede tener a lo más nueve resultados. Pero el juego de las cinco monedas tiene 10 resultados:

$$C_1, L, \quad C_1, P, \quad C_2, L, \quad C_2, P, \quad C_3, L,$$
$$C_3, P, \quad C_4, L, \quad C_4, P, \quad C_5, L, \quad C_5, P.$$

Esto es una contradicción. Por tanto, ningún algoritmo que resuelva el juego de las cinco monedas tiene un tiempo en el peor de los casos menor que 3, y el algoritmo de la figura 7.7.3 es óptimo.



9 resultados

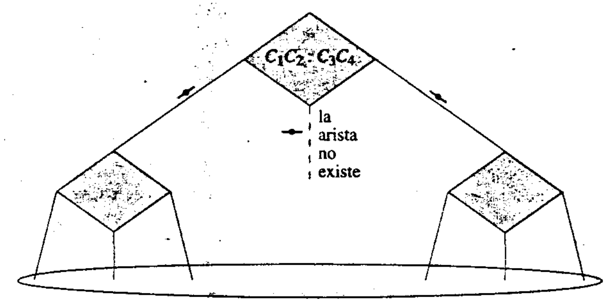
FIGURA 7.7.4 Un algoritmo para el juego de las cinco monedas que utiliza a lo más dos pesadas.

Hemos visto cómo se puede utilizar un árbol de decisión para obtener una cota inferior del tiempo necesario para resolver un problema en el peor de los casos. A veces, la cota inferior no se puede alcanzar.

Consideremos el juego con cuatro monedas (las reglas son iguales a las del juego de las cinco monedas, excepto que el número de monedas se reduce en uno). Como ahora existen ocho resultados en vez de 10, podemos concluir que cualquier algoritmo que resuelva el juego de cuatro monedas requiere al menos dos pesadas en el peor de los casos. (Ahora *no podemos* concluir que se necesitan al menos tres pesadas en el peor de los casos.) Sin embargo, un análisis más cuidadoso muestra que, de hecho, se necesitan tres pesadas.

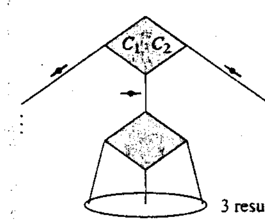
La primera pesada compara dos monedas con dos monedas o una moneda con una moneda. La figura 7.7.5 muestra que si comenzamos comparando dos monedas con dos monedas, el árbol de decisión puede tomar en cuenta seis resultados a lo más. Como existen ocho resultados, ningún algoritmo que comience comparando dos monedas con dos monedas puede resolver el problema con dos pesadas o menos en el peor de los casos. De manera análoga, la figura 7.7.6 muestra que si comenzamos comparando una moneda con otra y las monedas se equilibran, el árbol de decisión puede tomar en cuenta sólo tres resultados. Como existen cuatro resultados posibles después de identificar dos monedas buenas, ningún

algoritmo que comience comparando una moneda con otra puede resolver el problema en dos pesadas o menos en el peor de los casos. Por tanto, cualquier algoritmo que resuelva el juego de las cuatro monedas necesita al menos tres pesadas en el peor de los casos.



6 resultados

FIGURA 7.7.5 Un algoritmo para el juego de cuatro monedas que comienza comparando dos monedas con dos monedas.



3 resultados

FIGURA 7.7.6 Un algoritmo para el juego de cuatro monedas que comienza comparando una moneda con otra.

Si modificamos el juego de las cuatro monedas, pidiendo sólo que identifiquemos la moneda mala (sin determinar si es más pesada o más ligera), podemos resolver el juego en dos pesadas en el peor de los casos (véase el ejercicio 1).

Ahora revisaremos el ordenamiento. Podemos utilizar árboles de decisión para estimar el tiempo necesario para ordenar en el peor de los casos.

El problema de ordenamiento se describe fácilmente: Dados n elementos

$$x_1, \quad \ldots, \quad x_n,$$

hay que ordenarlos en forma ascendente (o descendente). Restringiremos nuestra atención a los algoritmos de ordenamiento que comparan dos elementos y, con base en el resultado de la comparación, modifican la lista original.

EJEMPLO 7.7.2

La figura 7.7.7 muestra un árbol de decisión para el algoritmo de ordenamiento de a_1, a_2, a_3. Cada arista se etiqueta con el arreglo de la lista basado en la pregunta en un vértice interno. Los vértices terminales proporcionan el orden requerido.

Definimos el tiempo necesario para el ordenamiento en el peor de los casos como el número de comparaciones necesarias en el peor de los casos. Al igual que en el caso de los árboles de decisión que resuelven los juegos con monedas, la altura de un árbol de decisión que resuelve un problema de ordenamiento es igual al tiempo en el peor de los casos. Por ejemplo, el tiempo en el peor de los casos para el algoritmo dado por el árbol de decisión de la figura 7.7.7 es igual a 3. Mostraremos que este algoritmo es óptimo, es decir, que *ningún* algoritmo que ordene tres elementos tiene un tiempo menor que 3 en el peor de los casos.

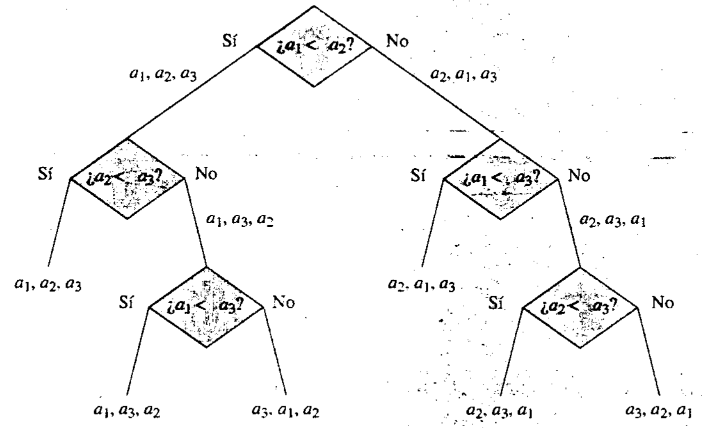


FIGURA 7.7.7 Un algoritmo para ordenar a_1, a_2, a_3.

Argumentaremos por contradicción para mostrar que ningún algoritmo que ordene tres elementos tiene un tiempo menor que 3 en el peor de los casos. Supongamos que existe un algoritmo que ordena tres elementos con dos o menos comparaciones en el peor de los casos. El algoritmo se puede describir mediante un árbol de decisión, y como el tiempo en el peor de los casos es 2 o menos, la altura del árbol de decisión es dos o menos. Como cada vértice interno tiene a lo más dos hijos, tal árbol puede tener a lo más cuatro vértices terminales (véase la figura 7.7.8). Ahora, los vértices terminales corresponden a los resultados posibles. Así, un árbol de decisión con altura 2 o menos sólo puede tomar en cuenta

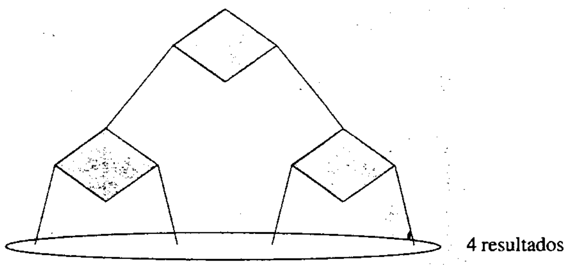
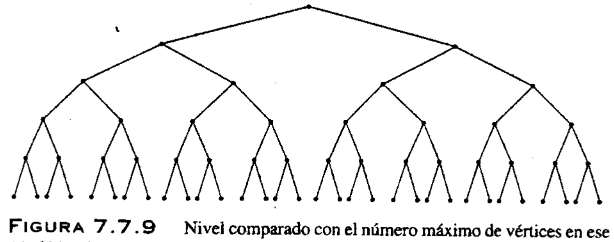


FIGURA 7.7.8 Un algoritmo de ordenamiento que realiza dos comparaciones como máximo.



Nivel	*Número de vértices*
0	1
1	2
2	4
3	8
4	16
5	32

FIGURA 7.7.9 Nivel comparado con el número máximo de vértices en ese nivel en un árbol binario.

cuatro resultados a lo más. Pero el problema de ordenar tres elementos tiene seis resultados posibles, correspondientes a las $3! = 6$ formas en que se pueden ordenar los elementos:

$$s_1, s_2, s_3, \quad s_1, s_3, s_2, \quad s_2, s_1, s_3, \quad s_2, s_3, s_1, \quad s_3, s_1, s_2, \quad s_3, s_2, s_1.$$

Esto es una contradicción. Por tanto, ningún algoritmo que ordene tres elementos tiene un tiempo menor que 3 en el peor de los casos y el algoritmo de la figura 7.7.7 es óptimo. □

Como $4! = 24$, existen 24 resultados posibles para el problema de ordenar cuatro elementos. Para acomodar 24 vértices terminales, debemos tener un árbol por lo menos de altura 5 (véase la figura 7.7.9). Por tanto, cualquier algoritmo que ordene cuatro elementos necesita al menos cinco comparaciones en el peor de los casos. El ejercicio 9 consiste en dar un algoritmo que ordene cuatro elementos utilizando cinco comparaciones en el peor de los casos.

El método del ejemplo 7.7.2 se puede utilizar para dar una cota inferior sobre el número de comparaciones necesarias para ordenar una cantidad arbitraria de elementos en el peor de los casos.

TEOREMA 7.7.3

Si $f(n)$ es el número de comparaciones necesarias para ordenar n elementos en el peor de los casos mediante un algoritmo de ordenamiento, entonces $f(n) = \Omega(n \lg n)$.

Demostración. Sea T el árbol de decisión que representa el algoritmo para una entrada de tamaño n y sea h la altura de T. Entonces el algoritmo necesita h comparaciones en el peor de los casos, de modo que

$$h = f(n). \tag{7.7.1}$$

El árbol T tiene al menos $n!$ vértices terminales, de modo que por el teorema 7.5.6,

$$\lg n! \le h. \tag{7.7.2}$$

El ejemplo 3.5.8 muestra que $\lg n! = \Theta(n \lg n)$; así, para alguna constante positiva C,

$$Cn \lg n \le \lg n! \tag{7.7.3}$$

para todo entero n, salvo un número finito. Al combinar (7.7.1), (7.7.2) y (7.7.3), obtenemos

$$Cn \lg n \le f(n)$$

para toda n, salvo un número finito. Por tanto,

$$f(n) = \Omega(n \lg n).$$ ■

El teorema 5.3.10 establece que el ordenamiento por fusión (algoritmo 5.3.8) utiliza $\Theta(n \lg n)$ en el peor de los casos y, por el teorema 7.7.3, es óptimo. Se sabe que muchos otros algoritmos de ordenamiento también alcanzan la cota óptima $\Theta(n \lg n)$ de comparaciones; uno de ellos, el ordenamiento por torneo, se describe antes del ejercicio 12.

Ejercicios

1. Cuatro monedas tienen apariencia idéntica, pero una de ellas es más pesada o más ligera que las demás, las cuales pesan lo mismo. Trace un árbol de decisión que proporcione un algoritmo para identificar a lo más en dos pesadas a la moneda mala (pero que no necesariamente determine si es más pesada o más ligera que las otras) utilizando sólo una balanza de platillos.

2. Muestre que se necesitan al menos dos pesadas para resolver el problema del ejercicio 1.

3. Ocho monedas tienen apariencia idéntica, pero una de ellas es más pesada o más ligera que las demás, las cuales pesan lo mismo. Trace un árbol de decisión que proporcione un algoritmo para identificar a lo más en tres pesadas a la moneda mala y que determine si es más pesada o más ligera que las otras utilizando sólo una balanza de platillos.

4. Doce monedas tienen apariencia idéntica, pero una de ellas es más pesada o más ligera que las demás, las cuales pesan lo mismo. Trace un árbol de decisión que proporcione un algoritmo para identificar a lo más en tres pesadas a la moneda mala y que determine si es más pesada o más ligera que las otras utilizando sólo una balanza de platillos.

5. ¿Cuál es el error del siguiente argumento, el cual supuestamente muestra que el juego de las doce monedas necesita al menos cuatro pesadas en el peor de los casos si comenzamos pesando cuatro monedas contra cuatro monedas?

Si pesamos cuatro monedas contra cuatro monedas y éstas se equilibran, entonces debemos determinar la moneda mala mediante las cuatro monedas restantes. Pero el análisis de esta sección mostró que la determinación de la moneda mala entre cuatro monedas necesita al menos tres pesadas en el peor de los casos. Por tanto, en el peor de los casos, si comenzamos pesando cuatro monedas contra cuatro, el juego de las doce monedas necesita al menos cuatro pesadas.

☆ 6. Trece monedas tienen apariencia idéntica, pero una de ellas es más pesada o más ligera que las demás, las cuales pesan lo mismo. ¿Cuántas pesadas se necesitan en el peor de los casos para determinar la moneda mala y saber si es más pesada o más ligera que las demás utilizando sólo una balanza de platillos? Demuestre su respuesta.

7. Resuelva el ejercicio 6 para el juego de 14 monedas.

8. $(3^n - 3)/2$, $n \geq 2$, monedas tienen apariencia idéntica, pero una de ellas es más pesada o más ligera que las demás, las cuales pesan lo mismo. [Kurosaka] dio un algoritmo para determinar la moneda mala y saber si es más pesada o más ligera que las demás, utilizando sólo una balanza de platillos, en n pesadas en el peor de los casos. Demuestre que la moneda no se puede determinar e identificar como pesada o ligera en menos de n pesadas.

9. Proporcione un algoritmo que ordene cuatro elementos utilizando cinco comparaciones en el peor de los casos.

10. Utilice árboles de decisión para determinar una cota inferior sobre el número de comparaciones necesarias para ordenar cinco elementos en el peor de los casos. Proporcione un algoritmo que utilice este número de comparaciones para ordenar cinco elementos en el peor de los casos.

11. Utilice árboles de decisión para determinar una cota inferior sobre el número de comparaciones necesarias para ordenar seis elementos en el peor de los casos. Proporcione un algoritmo que utilice este número de comparaciones para ordenar seis elementos en el peor de los casos.

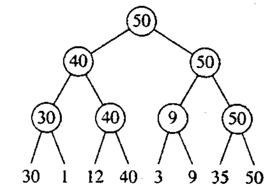
Los ejercicios 12-18 se refieren al *ordenamiento por torneo*.

Ordenamiento por torneo. Se da una sucesión

$$s_1, \quad \ldots, \quad s_{2^k}$$

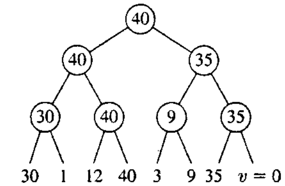
que debe ordenarse en forma creciente.

Formamos un árbol binario con vértices terminales etiquetados $s_1, \ldots, s_{2^k}$. Al margen aparece un ejemplo.



Trabajamos de izquierda a derecha para crear un padre de cada par y lo etiquetamos con el máximo de los hijos. Continuamos de esta forma hasta llegar a la raíz. En este momento se habrá determinado m, el valor máximo.

Para determinar el segundo valor más grande, primero se elige un valor v menor que todos los elementos de la sucesión. Reemplazamos el vértice terminal w que contiene a m con v. Reetiquetamos los vértices siguiendo el camino de w a la raíz, como muestra la figura al margen. En este momento se habrá determinado el segundo valor más grande. Se continúa de esta forma hasta ordenar la sucesión.



12. ¿Por qué es adecuado el nombre "torneo"?

13. Trace los dos árboles que se crearían después del árbol anexo, al aplicar el ordenamiento por torneo.

14. ¿Cuántas comparaciones necesita el ordenamiento por torneo para determinar el elemento máximo?

15. Muestre que cualquier algoritmo que determine el valor máximo entre n elementos necesita al menos $n - 1$ comparaciones.

16. ¿Cuántas comparaciones necesita el ordenamiento por torneo para determinar el segundo elemento más grande?

17. Escriba el ordenamiento por torneo como un algoritmo formal.

18. Muestre que si n es una potencia de 2, el ordenamiento por torneo necesita $\Theta(n \lg n)$ comparaciones.

19. Proporcione un ejemplo de una situación real (como la de la figura 7.7.1) que se pueda modelar mediante un árbol de decisión. Trace el árbol de decisión.

20. Trace un árbol de decisión que se pueda utilizar para determinar las personas que deben recibir un reembolso de impuestos.

21. Trace un árbol de decisión que proporcione una estrategia razonable para jugar blackjack (véase, por ejemplo, [Ainslie]).

7.8 *ISOMORFISMOS DE ÁRBOLES*

En la sección 6.6 definimos lo que quiere decir que dos gráficas sean isomorfas. (Tal vez el lector debería revisar la sección 6.6 antes de continuar.) En esta sección analizamos los árboles isomorfos, los árboles con raíz isomorfos y los árboles binarios isomorfos.

El teorema 6.6.4 establece que las gráficas simples G_1 y G_2 son isomorfas si y sólo si existe una función, f, uno a uno y sobre el conjunto de vértices de G_1 al conjunto de vértices de G_2 que preserva la relación de adyacencia, en el sentido de que los vértices v_i y v_j son adyacentes en G_1 si y sólo si los vértices $f(v_i)$ y $f(v_j)$ son adyacentes en G_2. Como un árbol (libre) es una gráfica simple, los árboles T_1 y T_2 son isomorfos si y sólo si existe una función, f, uno a uno y sobre el conjunto de vértices de T_1 al conjunto de vértices de T_2 que preserva la relación de adyacencia, en el sentido de que los vértices v_i y v_j son adyacentes en T_1 si y sólo si los vértices $f(v_i)$ y $f(v_j)$ son adyacentes en T_2.

EJEMPLO 7.8.1

La función f del conjunto de vértices del árbol T_1 de la figura 7.8.1 al conjunto de vértices del árbol T_2 que aparece en la figura 7.8.2 definida como

$$f(a) = 1, \quad f(b) = 3, \quad f(c) = 2, \quad f(d) = 4, \quad f(e) = 5$$

es una función uno a uno, sobre, que preserva la relación de adyacencia. Así, los árboles T_1 y T_2 son isomorfos. □

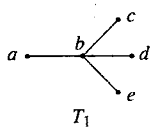


FIGURA 7.8.1 Un árbol.

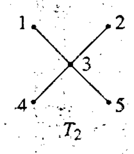


FIGURA 7.8.2 Un árbol isomorfo al árbol de la figura 7.8.1.

Como en el caso de las gráficas, podemos mostrar que dos árboles no son isomorfos si podemos exhibir un invariante no compartido por los árboles.

EJEMPLO 7.8.2

Los árboles T_1 y T_2 de la figura 7.8.3 no son isomorfos, pues T_2 tiene un vértice (x) de grado 3, pero T_1 no tiene un vértice de grado 3. □

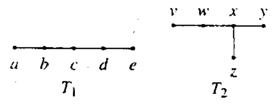


FIGURA 7.8.3 Árboles no isomorfos. T_2 tiene un vértice de grado 3, pero T_1 no.

Podemos mostrar que existen tres árboles no isomorfos con cinco vértices; estos árboles aparecen en las figuras 7.8.1 y 7.8.3.

TEOREMA 7.8.3

Existen tres árboles no isomorfos con cinco vértices.

Demostración. Daremos un argumento para mostrar que cualquier árbol con cinco vértices es isomorfo a alguno de los árboles en las figuras 7.8.1 y 7.8.3.

Si T es un árbol con cinco vértices, por el teorema 7.2.3, T tiene cuatro aristas. Si T tuviera un vértice v de grado mayor que 4, v incidiría en más de cuatro aristas. Esto implica que cada vértice en T tiene grado 4 como máximo.

Primero determinaremos todos los árboles no isomorfos con cinco vértices, en los que el grado máximo de los vértices sea 4. Luego determinaremos todos los árboles no isomorfos con cinco vértices en los cuales el grado máximo de los vértices sea 3, y así sucesivamente.

Sea T un árbol con cinco vértices y supongamos que T tiene un vértice v de grado 4. Entonces existen cuatro aristas incidentes en v y, por el teorema 7.2.3, éstas son todas las aristas. Esto implica que en este caso, T es isomorfo al árbol de la figura 7.8.1.

Supongamos que T es un árbol con cinco vértices y que el grado máximo de los vértices es 3. Sea v un vértice de grado 3. Entonces v es incidente en tres aristas, como muestra la figura 7.8.4. La cuarta arista no puede ser incidente en v, pues entonces v tendría grado 4. Así, la cuarta arista es incidente sobre uno de los vértices v_1, v_2 o v_3. Al agregar una arista incidente en alguno de los vértices v_1, v_2 o v_3, obtenemos un árbol isomorfo al árbol T_2 de la figura 7.8.3.

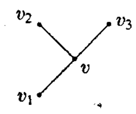


FIGURA 7.8.4
El vértice v tiene grado 3.

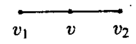


FIGURA 7.8.5
El vértice v tiene grado 2.

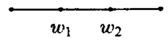


FIGURA 7.8.6
Agregado de una tercera arista a la gráfica de la figura 7.8.5.

Ahora, supongamos que T es un árbol con cinco vértices y que el grado máximo de los vértices es 2. Sea v un vértice de grado 2. Entonces v es incidente en dos aristas, como muestra la figura 7.8.5. Una tercera arista no puede ser incidente en v; así, debe ser incidente en v_1 o en v_2. Al agregar la tercera arista obtenemos la gráfica de la figura 7.8.6. Por la misma razón, la cuarta arista no puede incidir en los vértices w_1 o w_2 de la figura 7.8.6. Al agregar la última arista se obtiene un árbol isomorfo al árbol T_1 de la figura 7.8.3.

Como un árbol con cinco vertices debe tener un vértice de grado 2, hemos determinado todos los árboles no isomorfos con cinco vértices. ■

Para que dos árboles *con raíz* T_1 y T_2 sean isomorfos, debe existir una función, f, uno a uno y sobre T_1 a T_2 que preserve la relación de adyacencia y que preserve la raíz. Esta última condición significa que f(raíz de T_1) = raíz de T_2. A continuación damos la definición formal.

DEFINICIÓN 7.8.4

Sea T_1 un árbol con raíz r_1 y sea T_2 un árbol con raíz r_2. Los árboles con raíz T_1 y T_2 son *isomorfos* si existe una función, f, uno a uno y sobre el conjunto de vértices de T_1 en el conjunto de vértices de T_2 tal que

(a) Los vértices v_i y v_j son adyacentes en T_1 si y sólo si los vértices $f(v_i)$ y $f(v_j)$ son adyacentes en T_2.

(b) $f(r_1) = r_2$.

Decimos que la función f es un *isomorfismo*.

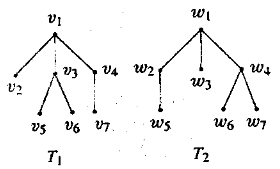


FIGURA 7.8.7
Árboles con raíz isomorfos.

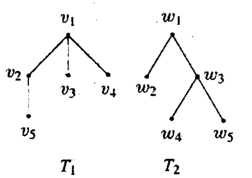


FIGURA 7.8.8
Árboles con raíz no isomorfos. (Los árboles son isomorfos como árboles *libres*.)

EJEMPLO 7.8.5

Los árboles con raíz T_1 y T_2 de la figura 7.8.7 son isomorfos. Un isomorfismo es

$$f(v_1) = w_1,\quad f(v_2) = w_3,\quad f(v_3) = w_4,\quad f(v_4) = w_2,$$
$$f(v_5) = w_7,\quad f(v_6) = w_6,\quad f(v_7) = w_5.$$ □

El isomorfismo del ejemplo 7.8.5 no es único. ¿Podría el lector determinar otro isomorfismo de los árboles con raíz de la figura 7.8.7?

EJEMPLO 7.8.6

Los árboles con raíz T_1 y T_2 de la figura 7.8.8 no son isomorfos, pues la raíz de T_1 tiene grado 3 y la raíz de T_2 tiene grado 2. Estos árboles son isomorfos como árboles *libres*. Cada uno de ellos es isomorfo al árbol T_2 de la figura 7.8.3. □

Al argumentar como en la demostración del teorema 7.8.3, podemos mostrar que existen cuatro árboles con raíz, no isomorfos, con cuatro vértices.

TEOREMA 7.8.7

Existen cuatro árboles con raíz, no isomorfos, con cuatro vértices. Estos árboles aparecen en la figura 7.8.9.

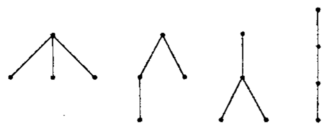


FIGURA 7.8.9 Los cuatro árboles con raíz, no isomorfos, con cuatro vértices.

Demostración. Primero determinaremos todos los árboles con raíz, no isomorfos, con cuatro vértices, en los cuales la raíz tiene grado 3; luego determinaremos los árboles con raíz, no isomorfos, con cuatro vértices en los cuales la raíz tiene grado 2, y así sucesivamente. Observamos que la raíz de un árbol con raíz y cuatro vértices no puede tener grado mayor que 3.

Un árbol con raíz con cuatro vértices en el que la raíz tiene grado 3 debe ser isomorfo al árbol de la figura 7.8.9a.

Un árbol con raíz con cuatro vértices en el que la raíz tiene grado 2 debe ser isomorfo al árbol de la figura 7.8.9b.

Sea T un árbol con raíz con cuatro vértices en el que la raíz tiene grado 1. Entonces la raíz es incidente en una arista. Las dos aristas restantes se pueden agregar de dos formas (véanse las figuras 7.8.9c y d). Por tanto, todos los árboles con raíz, no isomorfos, con cuatro vértices, aparecen en la figura 7.8.9. ■

Los árboles binarios son un tipo especial de árboles con raíz; así, un isomorfismo de árboles binarios debe preservar la relación de adyacencia y las raíces. Sin embargo, en los árboles binarios, un hijo se designa como hijo izquierdo o como hijo derecho. Necesitamos que un isomorfismo de árboles binarios preserve los hijos izquierdo y derecho. A continuación se proporciona la definición formal.

DEFINICIÓN 7.8.8

Sea T_1 un árbol binario con raíz r_1 y sea T_2 un árbol binario con raíz r_2. Los árboles binarios T_1 y T_2 son *isomorfos* si existe una función, f, uno a uno y sobre el conjunto de vértices de T_1 al conjunto de vértices de T_2 tal que

(a) Los vértices v_i y v_j son adyacentes en T_1 si y sólo si los vértices $f(v_i)$ y $f(v_j)$ son adyacentes en T_2.

(b) $f(r_1) = r_2$.

(c) v es un hijo izquierdo de w en T_1 si y sólo si $f(v)$ es un hijo izquierdo de $f(w)$ en T_2.

(d) v es un hijo derecho de w en T_1 si y sólo si $f(v)$ es un hijo derecho de $f(w)$ en T_2.

Decimos que la función f es un *isomorfismo*.

EJEMPLO 7.8.9

Los árboles binarios T_1 y T_2 de la figura 7.8.10 son isomorfos. El isomorfismo es $f(v_i) = w_i$ para $i = 1, \ldots, 4$. □

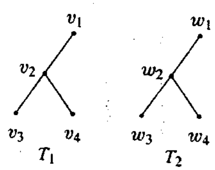


FIGURA 7.8.10 Árboles binarios isomorfos.

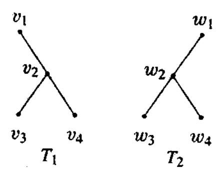


FIGURA 7.8.11 Árboles binarios no isomorfos. (Los árboles son isomorfos como árboles *con raíz* y como árboles *libres*.)

EJEMPLO 7.8.10

Los árboles binarios T_1 y T_2 de la figura 7.8.10 no son isomorfos. La raíz v_1 de T_1 tiene un hijo derecho, pero la raíz w_1 de T_1 no tiene hijo derecho. □

Los árboles T_1 y T_2 de la figura 7.8.10 *son* isomorfos como árboles con raíz y como árboles libres. Como árboles *con raíz*, cualquiera de los árboles de la figura 7.8.11 es isomorfo al árbol con raíz T de la figura 7.8.9c.

Al argumentar como en las demostraciones de los teoremas 7.8.3 y 7.8.7, podemos mostrar que existen cinco árboles binarios, no isomorfos, con tres vértices.

TEOREMA 7.8.11

Existen cinco árboles binarios no isomorfos con tres vértices. Estos cinco árboles binarios aparecen en la figura 7.8.12.

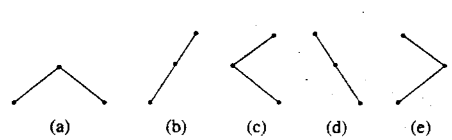


FIGURA 7.8.12 Los cinco árboles binarios, no isomorfos, con tres vértices.

Demostración. Primero determinamos todos los árboles binarios, no isomorfos, con tres vértices, en los cuales la raíz tenga grado 2. Luego determinaremos todos los árboles binarios, no isomorfos, con tres vértices, en los cuales la raíz tenga grado 1. Observemos que la raíz de cualquier árbol binario no puede tener grado mayor que 2.

Un árbol binario con tres vértices en el cual la raíz tenga grado 2 debe ser isomorfo al árbol de la figura 7.8.12a. En un árbol binario con tres vértices en el cual la raíz tenga grado 1, la raíz tiene un hijo izquierdo y no tiene hijo derecho, o bien un hijo derecho y no tiene hijo izquierdo. Si la raíz tiene un hijo izquierdo, este hijo tiene a su vez un hijo izquierdo o derecho. Obtenemos los dos árboles binarios de las figuras 7.8.12b y c. De manera similar, si la raíz tiene un hijo derecho, este hijo tiene a su vez un hijo izquierdo o un hijo derecho. Obtenemos los dos árboles binarios de la figura 7.8.12d y e. Por tanto, todos los árboles binarios no isomorfos con tres vértices aparecen en la figura 7.8.12. ■

Si S es un conjunto de árboles de un tipo particular (por ejemplo, S es un conjunto de árboles libres o S es un conjunto de árboles con raíz o S es un conjunto de árboles binarios) y definimos una relación R en S mediante la regla T_1RT_2 si T_1 y T_2 son isomorfos, entonces R es una relación de equivalencia. Cada clase de equivalencia consta de un conjunto de árboles isomorfos entre sí.

En el teorema 7.8.3 mostramos que existen tres árboles libres no isomorfos con cinco vértices. En el teorema 7.8.7 mostramos que existen cuatro árboles con raíz no isomorfos con cuatro vértices. En el teorema 7.8.11 mostramos que existen cinco árboles binarios no isomorfos con tres vértices. El lector podría preguntarse si existen fórmulas para el número de árboles no isomorfos con n vértices de un tipo particular. Existen fórmulas para el número de árboles libres de n vértices no isomorfos, para el número de árboles con raíz de n vértices no isomorfos, y para el número de árboles binarios de n vértices no isomorfos. Las fórmulas para los dos primeros casos son un poco complicadas. Además, la deducción de estas fórmulas requiere técnicas más avanzadas que las desarrolladas en este libro. Las fórmulas y las demostraciones aparecen en [Deo, sección 10-3]. Deduciremos una fórmula para el número de árboles binarios con n vértices.

TEOREMA 7.8.12

Existen $C(2n, n) / (n + 1)$ *árboles binarios no isomorfos con* n *vértices.*

Demostración. Sea a_n el número de árboles binarios con n vértices. Por ejemplo, $a_0 = 1$, pues existe un árbol binario sin vértices; $a_1 = 1$ pues existe un árbol binario con un vértice; $a_2 = 2$ pues existen dos árboles binarios con dos vértices (véase la figura 7.8.13); y $a_3 = 5$, pues existen cinco árboles binarios con tres vértices (véase la figura 7.8.12).

Deduciremos una relación de recurrencia para la sucesión $a_0, a_1, \ldots$. Consideremos la construcción de un árbol binario con n vértices, $n > 0$. Un vértice debe ser la raíz. Como restan $n - 1$ vértices, si el subárbol izquierdo tiene k vértices, el subárbol derecho debe tener $n - k - 1$ vértices. Construimos un árbol binario de n vértices cuyo subárbol izquierdo tiene k vértices y cuyo subárbol derecho tiene $n - k - 1$ vértices mediante un proceso de dos pasos: Construimos el subárbol izquierdo, construimos el subárbol derecho. (La figura 7.8.14 muestra esta construcción para $n = 6$ y $k = 2$.) Por el principio de multiplicación, esta construcción se puede realizar de $a_k a_{n-k-1}$ formas. Los diversos valores de k proporciona distintos árboles binarios de n vértices, de modo que por el principio de la suma, el número total de árboles binarios de n vértices es

$$\sum_{k=0}^{n-1} a_k a_{n-k-1}.$$

Obtenemos la relación de recurrencia

$$a_n = \sum_{k=0}^{n-1} a_k a_{n-k-1}, \qquad n \geq 1.$$

Pero esta relación de recurrencia y la condición inicial $a_0 = 1$ definen la sucesión de números de Catalan (véanse los ejemplos 4.2.23 y 5.1.7). Así, a_n es igual al número de Catalan $C(2n, n) / (n + 1)$. ■

FIGURA 7.8.13 Los dos árboles binarios no isomorfos con dos vértices.

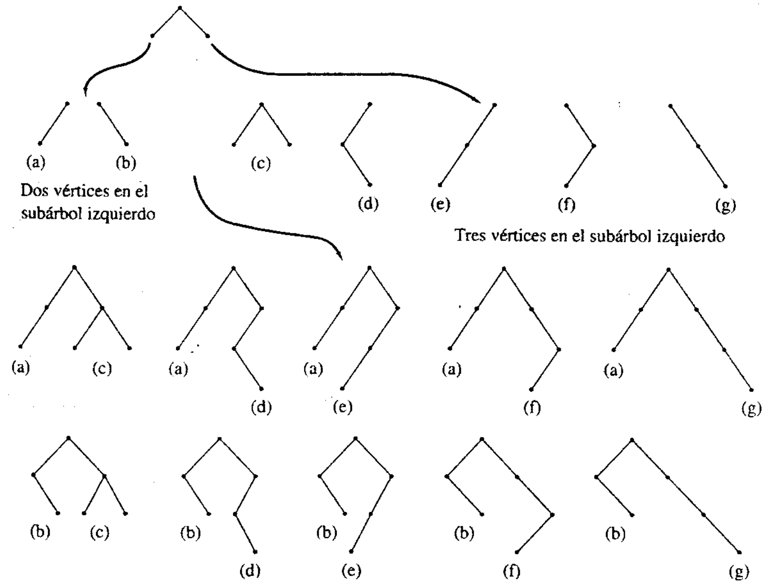


FIGURA 7.8.14 La demostración del teorema 7.8.12 para el caso de $n = 6$ vértices y $k = 2$ vértices en el subárbol izquierdo.

Al analizar los isomorfismos de gráficas en la sección 6.6, mencionamos que no existe un método rápido para decidir si dos gráficas arbitrarias son isomorfas. La situación es distinta para los árboles. Es posible determinar si dos árboles arbitrarios son isomorfos en un tiempo polinomial. Como caso particular, daremos un algoritmo de tiempo lineal para determinar si dos árboles binarios T_1 y T_2 son isomorfos. El algoritmo se basa en el recorrido en preorden (véase la sección 7.6). Primero verificamos que T_1 y T_2 sean no vacíos, después de lo cual verificamos que los subárboles izquierdos de T_1 y T_2 sean isomorfos y que los subárboles derechos de T_1 y T_2 sean isomorfos.

ALGORITMO 7.8.13 ***Verificación de que dos árboles binarios sean isomorfos***

Entrada: Las raíces r_1 y r_2 de dos árboles binarios. (Si el primer árbol es vacío, r_1 tiene el valor especial *null*. Si el segundo árbol es vacío, r_2 tiene el valor especial *null*.)

Salida: **true**, si los árboles son isomorfos
false, si los árboles no son isomorfos

```
   procedure bin_tree_isom (r1, r2)
1.   if r1 = null and r2 = null then
2.     return(true)
     // ahora, una o ambas raíces r1 o r2 no es null
3.   if r1 = null and r2 = null then
4.     return(true)
     // ahora, ninguna de las raíces r1 y r2 es null
5.   lc_r1 := hijo izquierdo de r1
6.   lc_r2 := hijo izquierdo de r2
7.   rc_r1 := hijo derecho de r1
8.   rc_r2 := hijo derecho de r2
9.   return(bin_tree_isom (lc_r1, lc_r2)
       and bin_tree_isom(rc_r1, rc_r2)
   end bin_tree_isom
```

Como medida del tiempo necesario para el algoritmo 7.8.13, contamos el número de comparaciones con *null* en las líneas 1 y 3. Mostraremos que el algoritmo 7.8.13 es un algoritmo de tiempo lineal en el peor de los casos.

TEOREMA 7.8.14

El tiempo del algoritmo 7.8.13 en el peor de los casos es $\Theta(n)$, donde n es la cantidad total de vértices en los dos árboles.

Demostración. Sea a_n el número de comparaciones con *null* en el peor de los casos del algoritmo 7.8.13, donde n es el número total de vértices en los árboles dados como entrada. Utilizaremos la inducción matemática para demostrar que

$$a_n \le 3n + 2, \quad \text{para } n \ge 0.$$

Paso base ($n = 0$). Si $n = 0$, los árboles dados como entrada al algoritmo 7.8.13 son vacíos. En este caso, existen dos comparaciones con *null* en la línea 1, después de lo cual el procedimiento concluye. Así, $a_0 = 2$ y la desigualdad es válida cuando $n = 0$.

Paso inductivo. Suponga que

$$a_k \le 3k + 2,$$

donde $k < n$. Debemos mostrar que

$$a_n \le 3n + 2.$$

Primero determinamos una cota superior para el número de comparaciones en el peor de los casos, cuando el número total de vértices en los árboles dados como entrada al procedimiento es $n > 0$ y ninguno de los árboles es vacío. En este caso, existen cuatro comparaciones en las líneas 1 y 3. Sea L la suma de los números de los vértices en los dos subárboles izquierdos de los árboles dados como entrada y R la suma de los números de los vértices en los dos subárboles derechos de los árboles dados como entrada. Entonces, en la línea 9, existen a lo más $a_L + a_R$ comparaciones adicionales. Por tanto, en el peor de los casos se necesitan a lo más $4 + a_L + a_R$ comparaciones. Por la hipótesis de inducción,

$$a_L \le 3L + 2 \quad \text{y} \quad a_R \le 3R + 2. \tag{7.8.1}$$

Ahora,

$$2 + L + R = n \tag{7.8.2}$$

pues los vértices son precisamente las dos raíces, los vértices en los subárboles izquierdos y los vértices en los subárboles derechos. Al combinar (7.8.1) y (7.8.2), obtenemos

$$4 + a_L + a_R \le 4 + (3L + 2) + (3R + 2) = 3(2 + L + R) + 2 = 3n + 2.$$

Si alguno de los árboles es vacío, se necesitan cuatro comparaciones en las líneas 1 y 3, después de lo cual concluye el procedimiento. Así, sin importar si los árboles son vacíos o no, se necesitan a lo más $3n + 2$ comparaciones en el peor de los casos. Por tanto,

$$a_n \le 3n + 2$$

y esto concluye el paso inductivo. Concluimos que el tiempo necesario en el peor de los casos para el algoritmo 7.8.13 es $O(n)$.

Si n es par, digamos, $n = 2k$, se puede utilizar inducción para mostrar (véase el ejercicio 24) que cuando se introducen como dato del algoritmo 7.8.13 dos árboles binarios isomorfos con k vértices, el número de comparaciones es igual a $3n + 2$. Con este resultado se puede mostrar (véase el ejercicio 25) que si n es impar, digamos $n = 2k + 1$, cuando se introducen como dato del algoritmo 7.8.13 los dos árboles binarios de la figura 7.8.15, el número de comparaciones es igual a $3n + 1$. Así, el tiempo del algoritmo 7.8.13 en el peor de los casos es $\Omega(n)$.

Como el tiempo en el peor de los casos es $O(n)$ y $\Omega(n)$, el tiempo en el peor de los casos del algoritmo 7.8.13 es $\Theta(n)$. ■

[Aho] proporciona un algoritmo para determinar si dos árboles con raíz (no necesariamente binarios) son isomorfos, cuyo tiempo en el peor de los casos es lineal con respecto del número de vértices.

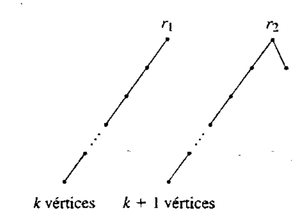
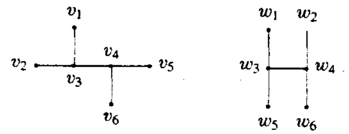


Figura 7.8.15
Dos árboles binarios que proporcionan el tiempo de ejecución en el peor de los casos $3n + 1$ para el algoritmo 7.8.13 cuando $n = 2k + 1$ es impar.

Ejercicios

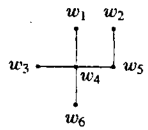
En los ejercicios 1-6, determine si cada par de árboles libres son isomorfos. Si los árboles son isomorfos, especifique un isomorfismo. Si no son isomorfos, mencione un invariante que tenga uno de los árboles y el otro no.

1.



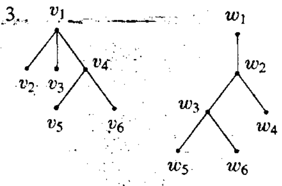
T_1 T_2

2. T_1 como en el ejercicio 1.



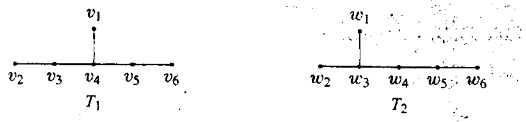
T_2

3.

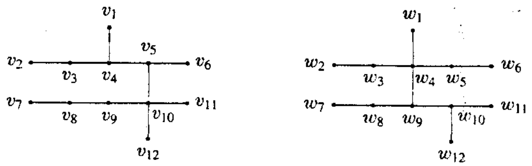


T_1 T_2

4.

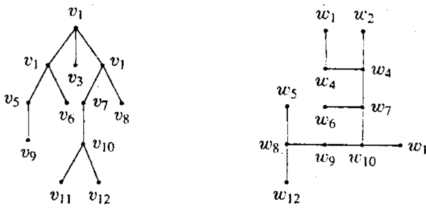


5.



T_1 T_2

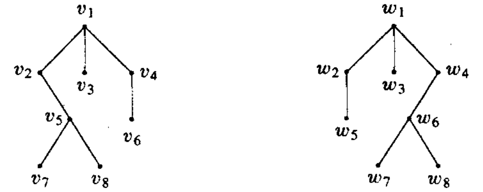
6.



T_1 T_2

En los ejercicios 7-9, determine si cada par de árboles con raíz son isomorfos. Si los árboles son isomorfos, especifique un isomorfismo. Si no son isomorfos, mencione un invariante que tenga uno de los árboles y el otro no. Además, determine si los árboles son isomorfos como árboles libres.

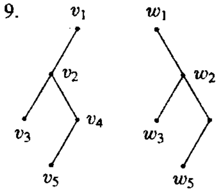
7.



T_1 T_2

8. T_1 y T_2 como en el ejercicio 3

9.

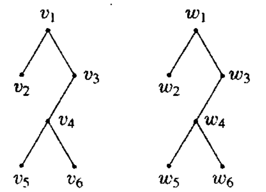


T_1 T_2

En los ejercicios 10-12, determine si cada par de árboles binarios son isomorfos. Si los árboles son isomorfos, especifique un isomorfismo. Si no son isomorfos, mencione un invariante que tenga uno de los árboles y el otro no. Además, determine si los árboles son isomorfos como árboles libres o como árboles con raíz.

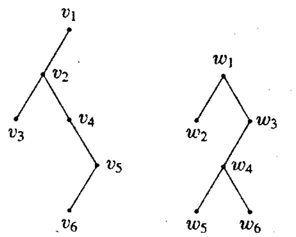
10. T_1 y T_2 como en el ejercicio 9

11.



T_1 T_2

12.



T_1 T_2

13. Trace todos los árboles libres no isomorfos con tres vértices.

14. Trace todos los árboles libres no isomorfos con cuatro vértices.

15. Trace todos los árboles libres no isomorfos con seis vértices.

16. Trace todos los árboles con raíz no isomorfos con tres vértices.

17. Trace todos los árboles con raíz no isomorfos con cinco vértices.

18. Trace todos los árboles binarios no isomorfos con dos vértices.

19. Trace todos los árboles binarios no isomorfos con cuatro vértices.

20. Trace todos los árboles binarios completos no isomorfos con siete vértices. (Un árbol binario completo es un árbol binario en donde cada vértice interno tiene dos hijos.)

21. Trace todos los árboles binarios completos no isomorfos con nueve vértices.
22. Determine una fórmula para el número de árboles binarios completos no isomorfos con n vértices.
23. Determine todos los árboles de expansión (como árboles libres y no como árboles con raíz) para cada una de las gráficas de los ejercicios 7-9, sección 7.3.
24. Utilice inducción para mostrar que cuando se introducen como dato dos árboles binarios isomorfos con k vértices en el algoritmo 7.8.13, el número de comparaciones con *null* es igual a $6k + 2$.
25. Muestre que cuando se introducen como dato los dos árboles binarios de la figura 7.8.15 en el algoritmo 7.8.13, el número de comparaciones con *null* es igual a $6k + 4$.
26. Escriba un algoritmo para generar un árbol binario arbitrario con n vértices.
27. [*Proyecto*] Realice un informe acerca de las fórmulas para el número de árboles libres no isomorfos y para el número de árboles con raíz no isomorfos, con n vértices (véase [Deo]).

†7.9 *ÁRBOLES DE JUEGOS*

Los árboles son útiles en el análisis de juegos como el gato, el ajedrez y las damas, donde los jugadores alternan sus movimientos. En esta sección mostraremos la forma de utilizar los árboles para desarrollar estrategias para ganar juegos. Este tipo de método se utiliza para desarrollar muchos programas de computadora que permiten a los humanos jugar contra las computadoras, o incluso computadoras contra computadoras.

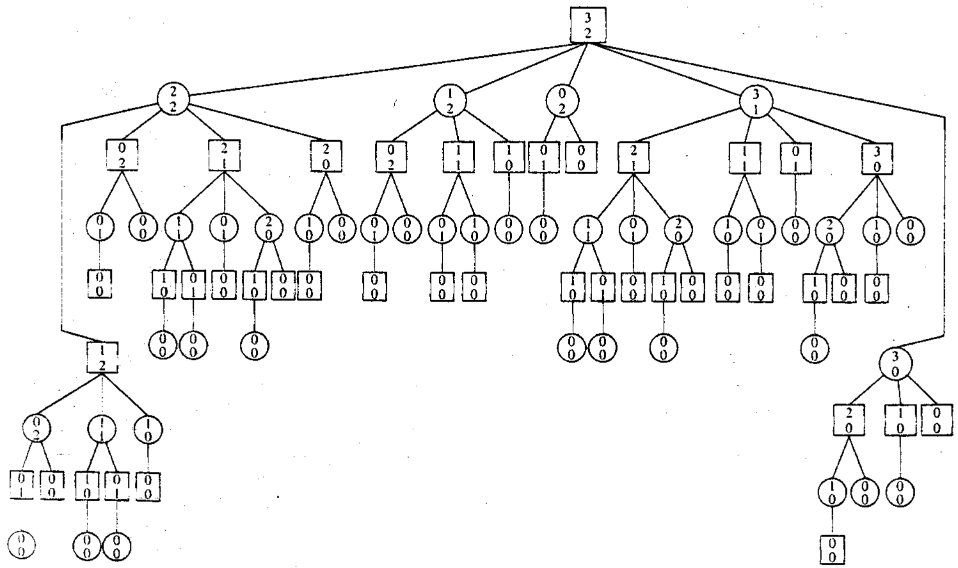


FIGURA 7.9.1 Un árbol de juego para nim. La distribución inicial es dos pilas de tres y dos fichas, respectivamente.

† Esta sección se puede omitir sin pérdida de continuidad.

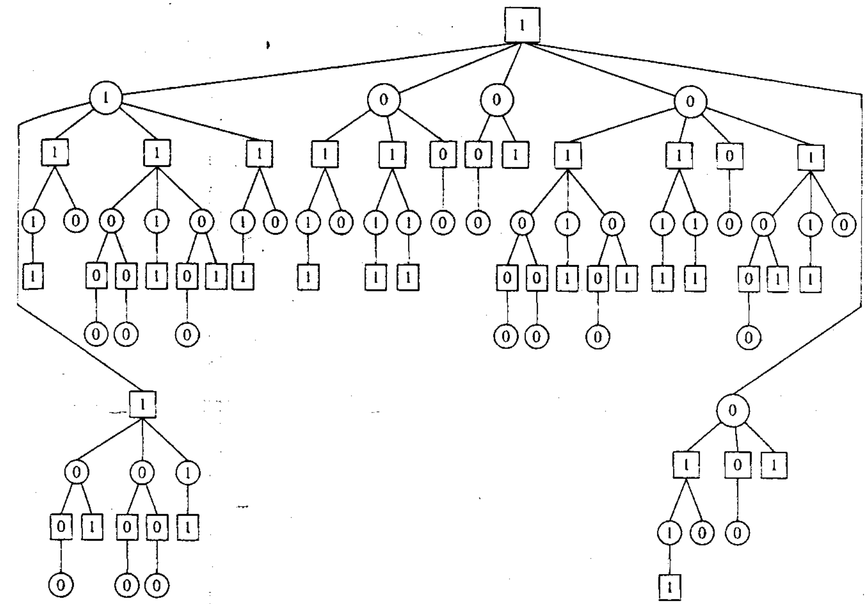


FIGURA 7.9.2 El árbol del juego de la figura 7.9.1 con los valores de todos los vértices.

Como ejemplo del punto de vista general, consideremos una versión del juego de nim. En un principio, existen n pilas, cada una de las cuales tiene un cierto número de fichas idénticas. Los jugadores alternan sus movimientos. Un movimiento consiste en retirar una o más fichas de cualquiera de las pilas. El jugador que quita la última ficha pierde. Como caso particular, considere una distribución inicial con dos pilas: una con tres fichas y la otra con dos. Todas las sucesiones de movimientos posibles se pueden enumerar en un **árbol del juego** (véase la figura 7.9.1). El primer jugador se representa mediante un cuadro y el segundo mediante un círculo. Cada vértice muestra una posición particular del juego. En nuestro juego, la posición inicial aparece como $\binom{3}{2}$. Un camino representa una serie de movimientos. Si una posición aparece dentro de un cuadro, es el movimiento del primer jugador; si una posición aparece dentro de un círculo, es el movimiento del segundo jugador. Un vértice terminal representa el final del juego. En el juego de nim, si el vértice terminal es un círculo, el primer jugador quitó la última ficha y pierde el juego. Si el vértice terminal es un cuadro, pierde el segundo jugador.

El análisis comienza con los vértices terminales. Etiquetamos cada vértice terminal con el valor de la posición para el primer jugador. Si el vértice terminal es un círculo, como ha perdido el primer jugador, esta posición no es importante para el primer jugador y le asignamos el valor 0 (véase la figura 7.9.2). Si el vértice terminal es un cuadro, como ha ganado el primer jugador, esta posición es valiosa para el primer jugador y la etiquetamos con un valor mayor que 0, digamos, 1 (véase la figura 7.9.2). En este momento, todos los vértices terminales tienen asignado un valor.

Ahora, consideremos el problema de asignar valores a los vértices internos. Supongamos, por ejemplo, que tenemos un cuadro interno, tal que todos sus hijos tienen asignado un valor. Por ejemplo, si tenemos la situación que aparece en la figura 7.9.3, el primer jugador (el cuadro) debe pasar a la posición representada por el vértice B, pues esta posición es la más valiosa. En otras palabras, el cuadro se mueve a una posición representada por un hijo con el valor máximo. Asignamos este valor máximo al vértice del cuadro.

Consideremos la situación desde el punto de vista del segundo jugador (círculo). Supongamos que tenemos la situación de la figura 7.9.4. El círculo se debe mover a la posición representada por el vértice C, pues esta posición es la menos valiosa para el cuadro y por tanto la más valiosa para el círculo. En otras palabras, el círculo se mueve a una posición representada por un hijo con el valor mínimo. Asignamos este valor mínimo al vértice del círculo. El proceso mediante el cual el círculo busca el mínimo de sus hijos y el cuadro busca el máximo de sus hijos es el **procedimiento minimax**.

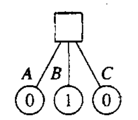


FIGURA 7.9.3 El primer jugador (cuadro) se debe mover a la posición B pues es más valiosa. Este valor máximo (1) se asigna al cuadro.

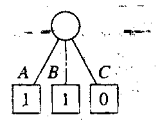


FIGURA 7.9.4 El segundo jugador (círculo) se debe mover a la posición C pues es menos valiosa (para el cuadro). Este valor mínimo (0) se asigna al círculo.

Al trabajar de abajo hacia arriba, partiendo de los vértices terminales y utilizando el procedimiento minimax, podemos asignar valores a todos los vértices del árbol del juego (véase la figura 7.9.2). Estos números representan el valor del juego, en cualquier posición, para el primer jugador. Observe que la raíz de la figura 7.9.2, la cual representa la posición original, tiene un valor de 1. Esto significa que el primer jugador siempre puede ganar, si utiliza una estrategia óptima. Esta estrategia óptima está contenida en el árbol del juego: El primer jugador siempre se mueve a una posición que maximiza el valor de los hijos. Sin importar lo que haga el segundo jugador, el primer jugador siempre puede pasar a un vértice con el valor 1. Al final, se llega a un vértice terminal con valor 1 si el primer jugador gana.

Muchos juegos interesantes, como el ajedrez, tienen árboles de juego tan grandes que no es factible utilizar una computadora para generar todo el árbol. Sin embargo, el concepto de árbol del juego sigue siendo útil para analizar tales juegos.

Al utilizar un árbol de juego, debemos utilizar una búsqueda a profundidad. Si el árbol del juego es tan grande que no sea factible llegar a un vértice terminal, limitamos el nivel hasta el cual se puede realizar la búsqueda a profundidad. La búsqueda es una **búsqueda de *n* niveles** si limitamos la búsqueda a n niveles debajo del vértice dado. Como los vértices del nivel más bajo podrían no ser vértices terminales, se necesita un método para asignarles un valor. Aquí hay que analizar los detalles específicos de cada juego. Se construye una **función de evaluación** E, la cual asigna a cada posición posible del juego P el valor $E(P)$ de la posición para el primer jugador. Después de asignar valores a los vértices del nivel más bajo mediante la función E, podemos aplicar el procedimiento minimax para generar los valores de los demás vértices. Ilustraremos estos conceptos mediante un ejemplo.

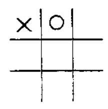
EJEMPLO 7.9.1

Aplicar el procedimiento minimax para determinar el valor de la raíz en el juego de gato, utilizando una búsqueda minimax a profundidad de dos niveles. Utilizar la función de evaluación E, la cual asigna a una posición el valor

$$NX - NO$$

donde NX (respectivamente, NO) es el número de renglones, columnas o diagonales que contienen una X (respectivamente, O) y que X (respectivamente, O) podría completar. Por ejemplo, la posición P de la figura 7.9.5 tiene $NX = 2$, pues X podría completar la columna o la diagonal, y $NO = 1$, pues O sólo puede completar una columna. Por tanto,

$$E(P) = 2 - 1 = 1.$$



P

FIGURA 7.9.5 El valor de la posición P es $E(P) = NX - NO = 2 - 1 = 1$.

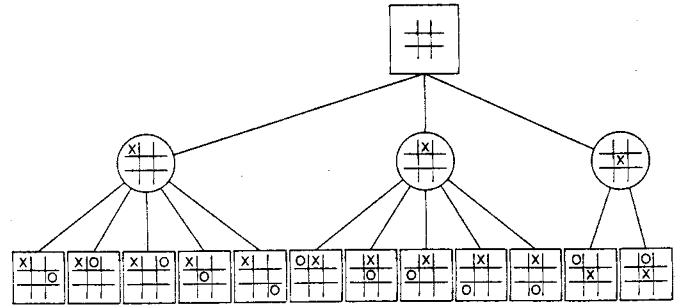


FIGURA 7.9.6 El árbol de juego del gato, hasta el nivel 2, omitiendo las posiciones simétricas.

En la figura 7.9.6 hemos trazado el árbol de juego del gato hasta el nivel 2. Hemos omitido las posiciones simétricas. Primero asignamos a los vértices del nivel 2 los valores dados por E (véase la figura 7.9.7). A continuación, calculamos los valores del círculo minimizando sobre los hijos. Por último, calculamos el valor de la raíz maximizando sobre los hijos. Con este análisis, el primer movimiento del primer jugador debe ser en el cuadro del centro. □

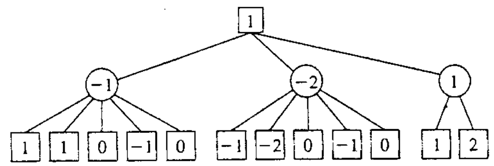


FIGURA 7.9.7 El árbol de juego de la figura 7.9.6 mostrando los valores de todos los vértices.

La evaluación de un árbol de juego, o incluso de una parte de un árbol de juego, puede consumir mucho tiempo, de modo que cualquier técnica que reduzca este esfuerzo es bienvenida. La técnica más general es la **poda alfa-beta**. En general, la poda alfa-beta permite evitar muchos vértices de un árbol de juego, y aun así determinar el valor de un vértice. El valor obtenido es igual al que se obtendría evaluando todos los vértices.

Por ejemplo, consideremos el árbol de juego de la figura 7.9.8. Suponga que queremos evaluar el vértice A mediante una búsqueda a profundidad de dos niveles. Evaluamos los hijos de izquierda a derecha. Comenzamos en la parte inferior izquierda, evaluando los vértices E, F y G. Los valores mostrados se obtienen a partir de una función de evaluación. El vértice B es 2, el mínimo de sus hijos. En este momento, sabemos que el valor x de A debe ser al menos 2, pues el valor de A es el máximo de sus hijos; es decir,

$$x \geq 2. \tag{7.9.1}$$

Esta cota inferior para A es el **valor alfa** de A. Los siguientes vértices por evaluar son H, I y J. Al evaluar I como 1, sabemos que el valor y de C no puede exceder a 1, ya que el valor de C es el mínimo de sus hijos; es decir,

$$y \leq 1. \tag{7.9.2}$$

(7.9.1) y (7.9.2) implican que, sin importar el valor de y, éste no afectará el valor de x; así, ya no tenemos que preocuparnos por el subárbol con raíz en el vértice C. Decimos que ocurre un **corte alfa**. A continuación evaluamos los hijos de D y el propio D. Por último, vemos que el valor de A es 3.

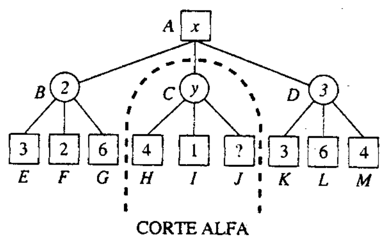


FIGURA 7.9.8 Evaluación del vértice A mediante una búsqueda a profundidad de dos niveles con un corte alfa. Un corte alfa aparece en el vértice C al evaluar el vértice I, pues el valor de I (1) es menor o igual que la estimación de la cota inferior (2) del vértice A.

En resumen, un corte alfa aparece en un vértice de cuadro v cuando un nieto w de v tiene un valor menor o igual al valor alfa de v. El subárbol cuya raíz es el padre de w se puede eliminar (podar). Esta eliminación no afectará el valor de v. Un valor alfa de un vértice v es sólo una cota inferior para el valor de v. El valor alfa de un vértice v depende del estado actual de la búsqueda y cambia al continuar ésta.

De manera análoga, un **corte beta** ocurre en un vértice de círculo v cuando un nieto w de v tiene un valor mayor o igual al valor beta de v. El subárbol cuya raíz es el padre de w se puede podar. Esta eliminación no afectará el valor de v. Un **valor beta** para un vértice v es sólo una cota superior para el valor de v. El valor beta de un vértice depende del estado actual de la búsqueda y cambia al continuar ésta.

EJEMPLO 7.9.2

Evaluar la raíz del árbol de la figura 7.9.9 utilizando la búsqueda a profundidad con una poda alfa-beta. Suponga que los hijos se evalúan de izquierda a derecha. Para cada vértice cuyo valor sea calculado, escribir el valor en el vértice. Colocar un signo de verificación en la raíz de cada subárbol podado. El valor de cada vértice terminal se escribe bajo el vértice.

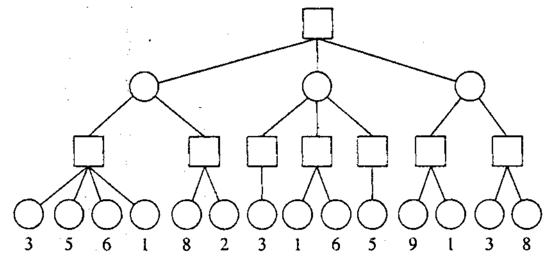


FIGURA 7.9.9 El árbol de juego para el ejemplo 7.9.2.

Primero evaluamos los vértices A, B, C y D (véase la figura 7.9.10). A continuación, vemos que el valor de E es 6. Esto produce un valor beta de 6 para F. A continuación evaluamos el vértice G. Como su valor es 8 y 8 es mayor que el valor beta de F, obtenemos un corte beta y podamos el subárbol con raíz H. El valor de F es 6. Esto produce un valor alfa de 6 para I. A continuación, evaluamos los vértices J y K. Como el valor 3 de K es menor que el valor alfa 6 de I, ocurre un corte alfa y podemos podar el subárbol con raíz L. A continuación evaluamos M, N, O, P, Q, R y S. No se puede podar más. Por último, determinamos que la raíz I tiene el valor 8. □

Se ha mostrado (véase [Pearl]) que para los árboles de juegos tales que cada padre tiene n hijos y donde se han ordenado los vértices terminales de manera aleatoria, para una cantidad dada de tiempo, el procedimiento alfa-beta permite una búsqueda a profundidad $\frac{4}{3}$ mayor que el procedimiento minimax puro, el cual evalúa cada vértice. Pearl también demostró que para tales árboles de juegos, el procedimiento alfa-beta es óptimo.

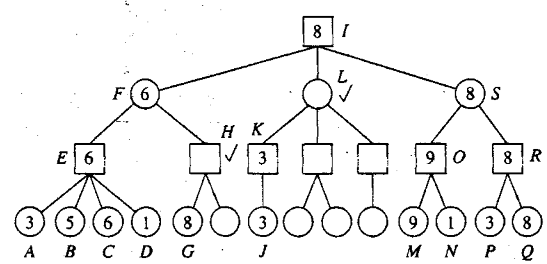


FIGURA 7.9.10 Evaluación de la raíz del árbol de juego de la figura 7.9.9 utilizando una búsqueda a profundidad con poda alfa-beta. Los vértices con un signo de verificación son las raíces de los subárboles podados. Los valores de los vértices evaluados se escriben dentro de los vértices.

Se han combinado otras técnicas con la poda alfa-beta para facilitar la búsqueda en un árbol de juego. Una idea consiste en ordenar los hijos de los vértices por evaluar de modo que se examinen primero los movimientos más promisorios (véanse los ejercicios 23-26). Otra idea consiste en permitir una búsqueda a profundidad variable, en la cual pueda retroceder la búsqueda al llegar a una posición no promisoria, medida mediante alguna función.

Algunos programas para juegos han tenido algo de éxito. Los mejores programas de backgammon y damas juegan a un nivel comparable al de los mejores jugadores humanos. Se escribió una parte de la historia en 1996 cuando el programa de ajedrez de IBM, Deep Blue, ganó el primer juego de un encuentro a seis partidas contra Garry Kasparov, con lo cual se convirtió en el primer programa de ajedrez en vencer a un campeón mundial bajo las restricciones normales de tiempo. (Los programas de computadora habían derrotado anteriormente a los mejores jugadores humanos, Kasparov incluido, en juegos realizados a mayor velocidad.) Kasparov prevaleció finalmente, ganando tres juegos y empatando en otros dos.

Ejercicios

1. Trace el árbol de juego completo para una versión del juego de nim en la cual la posición inicial conste de una pila de seis fichas y un turno consista en quitar una, dos o tres fichas. Asigne valores a todos los vértices, de modo que el árbol resultante sea análogo a la figura 7.9.2. Suponga que el último jugador que quite una ficha pierde. Jugando con una estrategia óptima, ¿quién ganará siempre: el primer jugador o el segundo? Describa una estrategia óptima para el ganador.
2. Trace el árbol de juego completo para el juego de nim en la cual la posición inicial conste de dos pilas con tres fichas cada una. Omita las posiciones simétricas. Suponga que el último jugador que quite una ficha pierde. Asigne valores a todos los vértices, de modo que el árbol resultante sea análogo a la figura 7.9.2. Jugando con una estrategia óptima, ¿quién ganará siempre: el primer jugador o el segundo? Describa una estrategia óptima para el ganador.
3. Trace el árbol de juego completo para el juego de nim en la cual la posición inicial conste de dos pilas, una con tres fichas y la otra con dos. Suponga que el último jugador que quite una ficha gana. Asigne valores a todos los vértices, de modo que el árbol resultante sea análogo a la figura 7.9.2. Jugando con una estrategia óptima, ¿quién ganará siempre: el primer jugador o el segundo? Describa una estrategia óptima para el ganador.
4. Trace el árbol de juego completo para el juego de nim en la cual la posición inicial conste de dos pilas con tres fichas cada una. Omita las posiciones simétricas. Suponga que el último jugador que quite una ficha gana. Asigne valores a todos los vértices, de modo que el árbol resultante sea análogo a la figura 7.9.2. Jugando con una estrategia óptima, ¿quién ganará siempre: el primer jugador o el segundo? Describa una estrategia óptima para el ganador.
5. Trace el árbol de juego completo para la versión del juego de nim descrita en el ejercicio 1. Suponga que el último jugador que quite una ficha gana. Asigne valores a todos los vértices, de modo que el árbol resultante sea análogo a la figura 7.9.2. Jugando con una estrategia óptima, ¿quién ganará siempre: el primer jugador o el segundo? Describa una estrategia óptima para el ganador.
6. Dé un ejemplo de un árbol de juego completo (posiblemente hipotético) en el cual un vértice terminal sea 1 si gana el primer jugador y 0 si pierde el primer jugador, con las siguientes propiedades: Existen más ceros que unos entre los vértices terminales, pero el primer jugador siempre puede ganar jugando una estrategia óptima.

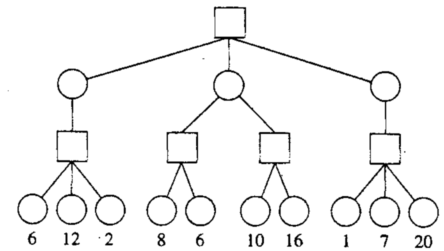
Los ejercicios 7 y 8 se refieren a nim y nim′. Nim es el juego que utiliza n pilas de fichas, descrito en esta sección, en la cual el último jugador que realice un movimiento pierde. Nim′ es el juego que utiliza n pilas de fichas, descrito en esta sección, excepto que el último jugador que realice un movimiento gana. Fijemos n pilas, cada una con un número fijo de fichas. Suponemos que al menos una pila tiene al menos dos fichas.

★ 7. Muestre que el primer jugador siempre puede ganar en el juego de nim si y sólo si el primer jugador siempre puede ganar el juego de nim′.

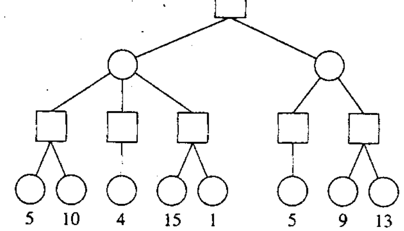
☆ 8. Dada una estrategia ganadora para un jugador particular para el juego de nim, describa una estrategia ganadora para este jugador, para nim′.

Evalúe cada vértice de cada árbol de juego. Se proporcionan los valores de los vértices terminales.

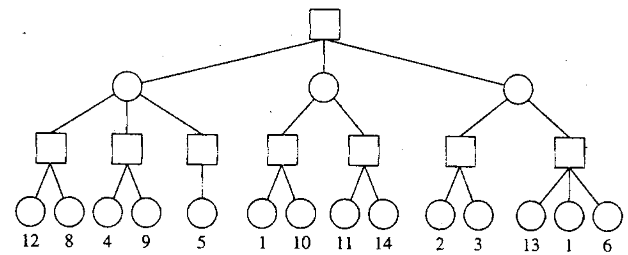
9.



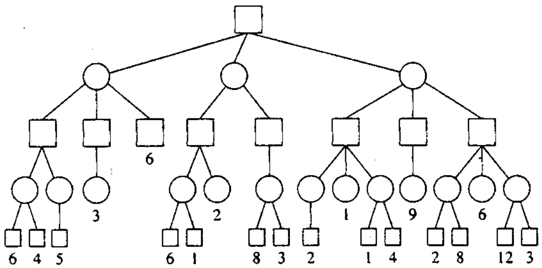
10.



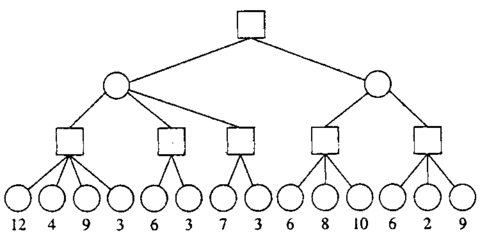
11.



12.



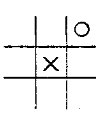
13.



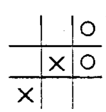
14. Evalúe la raíz de cada uno de los árboles de los ejercicios 9-13 mediante una búsqueda a profundidad con poda alfa-beta. Suponga que los hijos se evalúan de izquierda a derecha. Para cada vértice cuyo valor sea calculado, escriba el valor en el vértice. Coloque un signo de verificación en la raíz de cada subárbol podado. El valor de cada vértice terminal se escribe bajo el vértice.

En los ejercicios 15-18, determine el valor de la posición del gato mediante la función de evaluación del ejemplo 7.9.1.

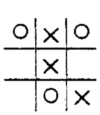
15.



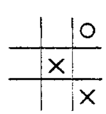
16.



17.



18.



19. Suponga que el primer jugador se mueve al cuadro central en el gato. Trace un árbol de juego de dos niveles, de modo que la raíz tenga una X en el cuadro central. Omita las posiciones simétricas. Evalúe todos los vértices mediante la función de evaluación del ejemplo 7.9.1. ¿Adónde se moverá O?

☆ 20. ¿Haría un juego perfecto de gato un programa de búsqueda de dos niveles basado en la función de evaluación E del ejemplo 7.9.1? En caso contrario, ¿podría modificar E de modo que un programa de búsqueda de dos niveles haga un juego perfecto de gato?

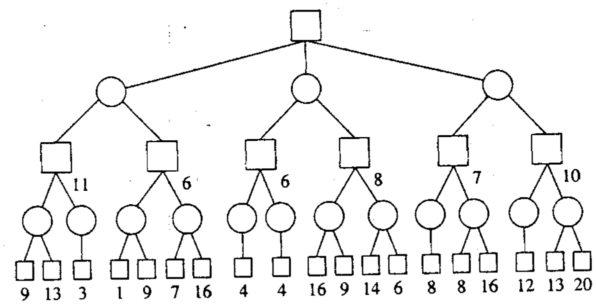
21. Escriba un algoritmo que evalúe los vértices de un árbol de juego hasta el nivel n mediante una búsqueda a profundidad. Suponga la existencia de una función de evaluación E.

★ **22.** Escriba un algoritmo que evalúe la raíz de un árbol de juego mediante una búsqueda a profundidad de n niveles con poda alfa-beta. Suponga la existencia de una función de evaluación E.

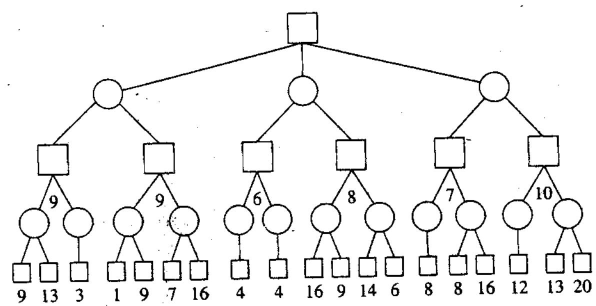
El siguiente método conduce con frecuencia a un mayor número de podas que el procedimiento minimax alfa-beta puro. Primero se realiza una búsqueda de dos niveles, evaluando los hijos de izquierda a derecha. En este momento, todos los hijos de la raíz tienen valores. A continuación, se ordenan los hijos de la raíz, con los movimientos más promisorios a la izquierda. Luego se utiliza una búsqueda a profundidad de n niveles con poda alfa-beta. Se evalúan los hijos de izquierda a derecha.

Realice este procedimiento para $n = 4$ para cada árbol de juego de los ejercicios 23-25. Coloque un signo de verificación en la raíz de cada subárbol podado. El valor de cada vértice, dado por la función de evaluación, está dado debajo del vértice.

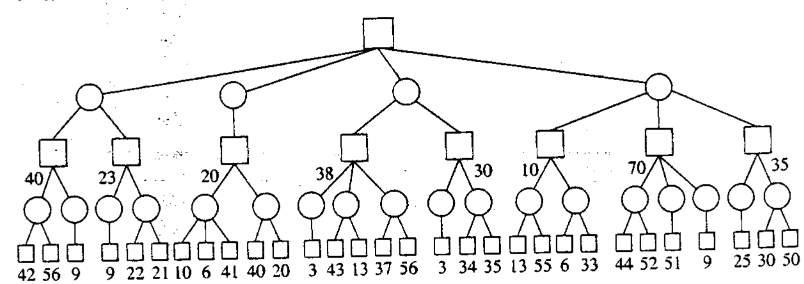
23.



24.

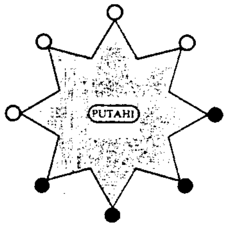


25.



26. Escriba un algoritmo para realizar el procedimiento descrito antes del ejercicio 23.

Mu Torere es un juego de dos personas jugado por los maoríes (véase [Bell]). El tablero es una estrella de ocho puntas (véase la siguiente figura) con un área circular en el centro conocida como *putahi*. El primer jugador tiene cuatro fichas negras y el segundo jugador tie-



ne cuatro fichas blancas. La posición inicial aparece en la figura. Un jugador que no pueda realizar un movimiento pierde. Los jugadores alternan sus movimientos. A lo más una ficha puede ocupar un punto de la estrella o del putahi. Un movimiento es:

(a) Un movimiento a un punto adyacente
(b) Un movimiento desde el putahi a un punto
(c) Un movimiento desde el punto al putahi, siempre que uno o ambos puntos adyacentes contengan las piezas del adversario

☆ 27. Desarrolle una función de evaluación para Mu Torere.

☆ 28. Combine la función de evaluación del ejercicio 27 con una búsqueda de dos niveles del árbol de juego para obtener un algoritmo para jugar Mu Torere. Evalúe la capacidad de juego de este algoritmo.

☆ 29. ¿Podría ganar siempre el primer jugador en Mu Torere?

☆ 30. ¿Podría empatar siempre el primer jugador en Mu Torere?

31. [*Proyecto*] Según [Nilsson], el árbol de juego completo para el ajedrez tiene más de 10^{100} vértices. Realice un informe acerca de la ~~forma en que~~ se obtuvo ~~esta~~ estimación.

☆ 32. [*Proyecto*] Desarrolle una función de evaluación para Kalah. (Véanse los detalles en [Ainslie].)

☆ 33. Desarrolle un algoritmo para jugar Kalah con base en la función de evaluación del ejercicio 32. Evalúe la capacidad de juego de este algoritmo.

NOTAS

Se recomienda la siguiente bibliografía para el tema de los árboles: [Berge; Bondy; Deo; Even, 1979; Gibbons; Harary; Knuth, 1973, Vol. 1; Liu, 1985; y Ore].

Véase [Date] para el uso de los árboles en las bases de datos jerárquicas.

[Cormen] tiene información adicional acerca de los códigos de Huffman y una demostración de que el algoritmo 7.1.8 construye un árbol de Huffman óptimo.

[Golomb, 1965] describe el retroceso y contiene varios ejemplos y aplicaciones.

Los algoritmos para los árboles de expansión mínimos y su implantación aparecen en [Tarjan].

[Baase] analiza el tiempo mínimo para el ordenamiento, así como cotas inferiores para otros problemas.

Los algoritmos de ordenamiento clásicos se estudian ampliamente en [Knuth, 1973, Vol. 3]. Véanse [Akl; Baase; Leighton; Lester; Lewis; y Quinn] para el ordenamiento mediante máquinas paralelas.

Algunas buenas referencias para los árboles de juegos son [Nievergelt; Nilsson; y Slagle]. En [Frey], el procedimiento minimax se aplica a un juego sencillo. Se analizan y comparan varios métodos para acelerar la búsqueda en el árbol de juego. Se proporcionan programas de computadora. [Berlekamp] contiene una teoría general de los juegos, así como el análisis de muchos juegos particulares.

CONCEPTOS BÁSICOS DEL CAPÍTULO

Sección 7.1

Árbol libre
Árbol con raíz
Nivel de un vértice en un árbol con raíz
Altura de un árbol con raíz
Árbol de definición jerárquica
Código de Huffman

Sección 7.2

Padre
Ancestro
Hijo
Descendiente
Hermano
Vértice terminal
Vértice interno
Subárbol
Gráfica acíclica
Caracterizaciones alternativas de los árboles (Teorema 7.2.3)

Sección 7.3

Árbol de expansión
Una gráfica tiene un árbol de expansión si y sólo si es conexa
Búsqueda a lo ancho
Búsqueda a profundidad
Retroceso

Sección 7.4

Árbol de expansión mínimo
Algoritmo de Prim para determinar un árbol de expansión mínimo
Algoritmo codicioso

Sección 7.5

Árbol binario
Hijo izquierdo en un árbol binario
Hijo derecho en un árbol binario
Árbol binario completo
Si T es un árbol binario completo con i vértices internos, entonces T tiene $i + 1$ vértices terminales y $2i + 1$ vértices en total.
Si un árbol binario de altura h tiene t vértices terminales, entonces $\lg t \leq h$.
Árbol de búsqueda binaria
Algoritmo para construir un árbol de búsqueda binaria

Sección 7.6

Recorrido en preorden
Recorrido en entreorden
Recorrido en posorden
Forma prefija de una expresión (notación polaca)
Forma entrefija de una expresión
Forma posfija de una expresión (notación polaca inversa)
Representación de una expresión mediante un árbol

Sección 7.7

Árbol de decisión
La altura de un árbol de decisión que representa a un algoritmo es proporcional al tiempo del algoritmo en el peor de los casos.
Cualquier algoritmo de ordenamiento necesita al menos $\Omega(n \lg n)$ comparaciones para ordenar n elementos en el peor de los casos.

Sección 7.8

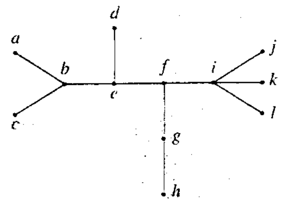
Árboles libres isomorfos
Árboles con raíz isomorfos
Árboles binarios isomorfos
El número de Catalan $C(2n, n)/(n + 1)$ es igual al número de árboles binarios con n vértices no isomorfos.
Algoritmo de tiempo lineal (algoritmo 7.8.13) para verificar si dos árboles binarios son isomorfos

Sección 7.9

Árbol de juego
Procedimiento minimax
Búsqueda de n niveles
Función de evaluación
Poda alfa-beta
Valor alfa
Corte alfa
Valor beta
Corte beta

AUTOEVALUACIÓN DEL CAPÍTULO

Sección 7.1



1. Trace el árbol libre anexo como un árbol con raíz c.

2. Determine el nivel de cada vértice en el árbol anexo con raíz en c.

3. Determine la altura del árbol anexo con raíz en c.

4. Construya un código de Huffman óptimo para el conjunto de letras de la tabla.

Letra	*Frecuencia*
A	5
B	8
C	5
D	12
E	20
F	10

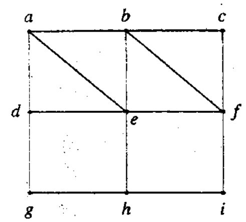
Sección 7.2

5. Trace el árbol libre del ejercicio 1 como un árbol con raíz f. Determine
   (a) El padre de a.
   (b) Los hijos de b.
   (c) Los vértices terminales.
   (d) El subárbol con raíz en e.

Responda verdadero o falso en los ejercicios 6-8 y explique su respuesta.

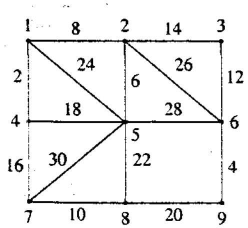
6. Si T es un árbol con seis vértices, entonces T debe tener cinco aristas.
7. Si T es un árbol con raíz y seis vértices, entonces la altura de T es a lo más 5.
8. Una gráfica acíclica con ocho vértices tiene siete aristas.

Sección 7.3



9. Utilice la búsqueda a lo ancho (algoritmo 7.3.6) con el orden de los vértices *eachgbdfi* para determinar un árbol de expansión de la gráfica anexa.
10. Utilice la búsqueda a profundidad (algoritmo 7.3.7) con el orden de los vértices *eachgbdfi* para determinar un árbol de expansión de la gráfica anexa.
11. Utilice la búsqueda a lo ancho (algoritmo 7.3.6) con el orden de los vértices *fdehagbci* para determinar un árbol de expansión de la gráfica anexa.
12. Utilice la búsqueda a profundidad (algoritmo 7.3.7) con el orden de los vértices *fdehagbci* para determinar un árbol de expansión de la gráfica anexa.

Sección 7.4



13. Determine un árbol de expansión mínimo para la gráfica anexa.
14. ¿En qué orden suma las aristas el algoritmo de Prim para la gráfica anexa, si el vértice inicial es 1?
15. ¿En qué orden suma las aristas el algoritmo de Prim para la gráfica adyacente, si el vértice inicial es 6?
16. Dé un ejemplo del uso del método codicioso que no conduzca a un algoritmo óptimo.

Sección 7.5

17. Trace un árbol binario con exactamente dos hijos izquierdos y un hijo derecho.
18. Un árbol binario completo tiene 15 vértices internos. ¿Cuántos vértices terminales tiene?

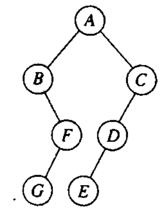
19. Coloque las palabras

WORD PROCESSING PRODUCES CLEAN MANUSCRIPTS
BUT NOT NECESSARILY CLEAR PROSE,

en el orden en que aparecen, en un árbol de búsqueda binaria.

20. Explique la forma en que buscaría MORE en el árbol de búsqueda binaria del ejercicio 19.

Sección 7.6



Los ejercicios 21-23 se refieren al árbol binario anexo.

21. Indique el orden de procesamiento de los vértices al utilizar el recorrido en preorden.
22. Indique el orden de procesamiento de los vértices al utilizar el recorrido en entreorden.
23. Indique el orden de procesamiento de los vértices al utilizar el recorrido en posorden.
24. Represente la expresión prefija $- * E/BD - CA$ como un árbol binario. Además, escriba la forma posfija y la forma entrefija con todos los paréntesis de la expresión.

Sección 7.7

25. Seis monedas tienen apariencia idéntica, pero una de ellas es más pesada o más ligera que las demás, las cuales pesan lo mismo. Demuestre que se necesitan al menos tres pesadas en el peor de los casos para identificar a la moneda mala y determinar si es más pesada o más ligera que las otras utilizando sólo una balanza de platillos.
26. Trace un árbol de decisión que proporcione un algoritmo para resolver el juego de monedas del ejercicio 25 a lo más en tres pesadas en el peor de los casos.
27. El profesor E. Sabic afirma haber descubierto un algoritmo que utiliza a lo más $100n$ comparaciones en el peor de los casos para ordenar n elementos, para toda $n \geq 1$. El algoritmo del profesor compara parejas de elementos y, con base en el resultado de la comparación, modifica la lista original. Proporcione un argumento que muestre que el profesor debe estar equivocado.
28. El algoritmo de *ordenamiento por inserción binaria* ordena un arreglo de tamaño n como sigue. Si $n = 1, 2$ o 3, el algoritmo utiliza un ordenamiento óptimo. Si $n > 3$, el algoritmo ordena $s_1, \ldots, s_n$ de la siguiente manera. Primero se ordenan de manera recursiva $s_1, \ldots, s_{n-1}$. Luego se utiliza la búsqueda binaria para determinar la posición correcta de s_n, después de lo cual se inserta s_n en su posición correcta. Determine el número de comparaciones utilizadas por el ordenamiento por inserción binaria en el peor de los casos, para $n = 4, 5, 6$. ¿Existe algún algoritmo que utilice menos comparaciones para $n = 4, 5, 6$?

Sección 7.8

Responda verdadero o falso en los ejercicios 29 y 30 y explique sus respuestas.

29. Si T_1 y T_2 son isomorfos como árboles con raíz, entonces T_1 y T_2 son isomorfos como árboles libres.
30. Si T_1 y T_2 son isomorfos como árboles libres, entonces T_1 y T_2 son isomorfos como árboles con raíz.
31. Determine si los siguientes árboles libres son isomorfos. Si son isomorfos, proporcione un isomorfismo. Si los árboles no son isomorfos, proporcione un invariante no compartido por los árboles.