8

Problemas de optimización de grafos y algoritmos codiciosos

8.1 Introducción

En este capítulo estudiaremos varios problemas de optimización de grafos que se pueden resolver exactamente empleando algoritmos codiciosos. Es común que en los problemas de optimización el algoritmo tenga que tomar una serie de decisiones cuyo efecto general es reducir al mínimo el costo total, o aumentar al máximo el beneficio total, de algún sistema. El método codicioso consiste en tomar las decisiones sucesivamente, de modo que cada decisión individual sea la mejor de acuerdo con algún criterio limitado "a corto plazo" cuya evaluación no sea demasiado costosa. Una vez tomada una decisión, no se podrá revertir, ni siquiera si más adelante se hace obvio que no fue una buena decisión. Por esta razón, los métodos codiciosos no necesariamente hallan la solución óptima exacta de muchos problemas. No obstante, en el caso de los problemas que estudiaremos en este capítulo, es posible demostrar que la estrategia codiciosa *apropiada* produce soluciones óptimas. En el capítulo 13 veremos problemas con los que estrategias codiciosas muy similares fracasan. En el capítulo 10 veremos otros problemas con los que las estrategias codiciosas fracasan.

Este capítulo presenta un algoritmo ideado por R. C. Prim para hallar un árbol abarcante mínimo en un grafo no dirigido, un algoritmo íntimamente relacionado con el anterior ideado por E. W. Dijkstra para hallar caminos más cortos en grafos dirigidos y no dirigidos, y un segundo algoritmo para hallar un árbol abarcante mínimo, que se debe a J. B. Kruskal. Los tres algoritmos emplean una cola de prioridad para seleccionar la mejor opción actual de entre un conjunto de opciones candidatas.

8.2 Algoritmo de árbol abarcante mínimo de Prim

El primer problema que estudiaremos es el de hallar un árbol abarcante mínimo para un grafo no dirigido, conectado y ponderado. En el caso de los grafos no conectados, la extensión natural del problema consiste en hallar un árbol abarcante mínimo para cada componente conectado. Ya vimos que los componentes conectados se pueden determinar en tiempo lineal (sección 7.4.2).

Los árboles abarcantes mínimos sólo tienen sentido en los grafos no dirigidos cuyas aristas están ponderadas, así que todas las referencias a "grafos" en esta sección son a grafos no dirigidos, y los pesos son siempre pesos de aristas. Recordemos que la notación $G = (V, E, W)$ significa que W es una función que asigna un peso a cada una de las aristas de E. Ésta es meramente la descripción matemática. En la implementación por lo regular no hay tal "función"; el peso de cada arista simplemente se almacena en la estructura de datos para esa arista.

8.2.1 Definición y ejemplos de árboles abarcantes mínimos

Definición 8.1 Árbol abarcante mínimo

Un *árbol abarcante* para un grafo no dirigido conectado $G = (V, E)$ es un subgrafo de G que es un árbol no dirigido y contiene todos los vértices de G. En un grafo ponderado $G = (V, E, W)$, el peso de un subgrafo es la suma de los pesos de las aristas incluidas en ese subgrafo. Un *árbol abarcante mínimo* (*MST*, por sus siglas en inglés) para un grafo ponderado es un árbol abarcante cuyo peso es mínimo. ■

Hay muchas situaciones en las que es preciso hallar árboles abarcantes mínimos. Siempre que se busca la forma más económica de conectar un conjunto de terminales, trátese de ciudades, terminales eléctricas, computadoras o fábricas, empleando por ejemplo carreteras, cables o líneas

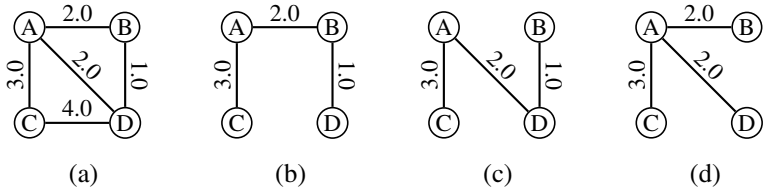


Figura 8.1 Grafo y algunos árboles abarcantes. Dos de ellos son *mínimos*

telefónicas, una solución es un árbol abarcante mínimo para el grafo que tiene una arista por cada posible conexión, ponderada con el costo de dicha conexión. Hallar árboles abarcantes mínimos también es un subproblema importante en diversos algoritmos de ruteo, es decir, algoritmos para hallar caminos eficientes a través de un grafo que visiten todos los vértices (o todas las aristas).

Como muestra el sencillo ejemplo de la figura 8.1, un grafo ponderado podría tener más de un árbol abarcante mínimo. De hecho, el método empleado para transformar un árbol abarcante mínimo en otro en este ejemplo es una ilustración de una propiedad general de los árboles abarcantes mínimos, que veremos en la sección 8.2.3.

8.2.2 Generalidades del algoritmo

Puesto que un árbol no dirigido está conectado y cualquier vértice se puede considerar como la raíz, una estrategia natural para hallar un árbol abarcante mínimo consiste en "cultivarlo" arista por arista a partir de algún vértice inicial. Quizá lo más conveniente sea probar primero nuestros métodos de recorrido estándar, búsqueda primero en profundidad y búsqueda primero en amplitud. Si podemos adaptar uno de estos esqueletos para resolver el problema, tendremos una solución en tiempo lineal, que seguramente es óptima. Recomendamos al lector dedicar cierto tiempo a probar algunas ideas empleando esos métodos de búsqueda, y construir grafos de ejemplo en los que no se pueda hallar el mínimo (véase el ejercicio 8.1).

Una vez convencidos de que un simple recorrido no es apropiado, y en vista de que se trata de un problema de optimización, la siguiente idea natural es probar el *método codicioso*. La idea básica del método codicioso es avanzar escogiendo una acción que incurra en el costo a corto plazo más bajo posible, con la esperanza de que muchos costos a corto plazo pequeños den un costo total también pequeño. (La posible desventaja es que acciones con costo a corto plazo bajo podrían llevar a una situación en la que no sea posible evitar costos altos posteriores.) Tenemos una forma muy natural de minimizar el costo a corto plazo de añadir una arista al árbol que estamos "cultivando": simplemente añadimos una arista que esté unida al árbol por exactamente un extremo y tenga el peso más bajo de todas las aristas que están en ese caso. El algoritmo de Prim adopta este enfoque codicioso.

Ahora que ya tenemos una idea de cómo resolver el problema, debemos hacernos las dos preguntas de siempre. ¿Funciona correctamente? ¿Con qué rapidez se ejecuta? Como ya hemos dicho, una serie de costos a corto plazo pequeños podría llevarnos a una situación poco favorable, así que, aunque tengamos la certeza de haber obtenido un árbol abarcante, falta ver si su peso es el mínimo de entre todos los árboles abarcantes. Además, puesto que necesitamos escoger entre muchas aristas en cada paso, y el conjunto de candidatas cambia después de cada decisión, es preciso pensar en qué estructuras de datos podrían hacer eficientes estas operaciones. Volveremos a estas preguntas después de precisar la idea general.

Lo primero que hace el algoritmo de Prim es seleccionar un vértice inicial arbitrario; luego se "ramifica" desde la parte del árbol que se ha construido hasta el momento escogiendo un nuevo vértice y arista en cada iteración. La nueva arista conecta al nuevo vértice con el árbol anterior. Durante la ejecución del algoritmo, podemos considerar que los vértices están divididos en tres categorías (disjuntas), a saber:

1. vértices *de árbol*: los que están en el árbol que se ha construido hasta ese momento,
2. vértices *de borde*: los que no están en el árbol, pero están adyacentes a algún vértice del árbol,
3. vértices *no vistos*: todos los demás.

El paso clave del algoritmo es la selección de un vértice del borde y una arista incidente. En realidad, puesto que los pesos están en las aristas, la decisión se concentra en la arista, no en el vértice. El algoritmo de Prim siempre escoge la arista de peso más bajo que va de un vértice de árbol a un vértice de borde. La estructura general del algoritmo podría describirse como sigue:

```
primMST(G, n) // BOSQUEJO
    Inicializar todos los vértices como no vistos.
    Seleccionar un vértice arbitrario s para iniciar el árbol; reclasificarlo como de árbol.
    Reclasificar todos los vértices adyacentes a s como de borde.
    Mientras haya vértices de borde:
        Seleccionar una arista con peso mínimo entre un vértice de árbol t y un vértice de
          borde v;
        Reclasificar v como de árbol; añadir la arista tv al árbol;
        Reclasificar todos los vértices no vistos adyacentes a v como de borde.
```

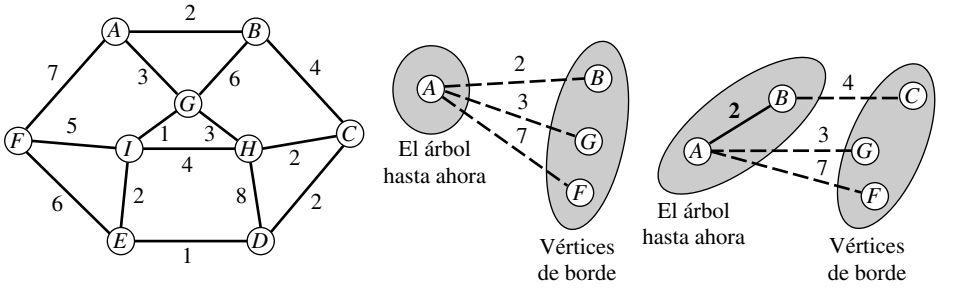
Ejemplo 8.1 Algoritmo de Prim, una iteración

La figura 8.2(a) muestra un grafo ponderado. Supóngase que *A* es el vértice inicial. Los pasos previos al ciclo nos llevan a la figura 8.2(b). En la primera iteración del ciclo, se determina que la arista de peso mínimo que conduce a un vértice de borde es *AB*. Por tanto, añadimos *B* al árbol y los vértices no vistos adyacentes a *B* entran en el borde; esto conduce a la figura 8.2(c). ■

¿Podemos tener la certeza de que esta estrategia produce un árbol abarcante mínimo? ¿Ser codiciosos a corto plazo es una buena estrategia a largo plazo? En este caso sí. En las dos subsecciones que siguen estudiaremos una propiedad general de todos los árboles abarcantes mínimos y la usaremos para demostrar que el árbol construido en cada etapa del algoritmo de Prim es un árbol abarcante mínimo en el subgrafo que ese árbol abarca. Volveremos a las consideraciones de implementación en la sección 8.2.5.

8.2.3 Propiedades de los árboles abarcantes mínimos

La figura 8.1 puso de manifiesto que un grafo ponderado puede tener más de un árbol abarcante mínimo. De hecho, los árboles abarcantes mínimos tienen una propiedad general que nos permite transformar cualquier árbol abarcante mínimo en otro siguiendo ciertos pasos. Examinar dicha propiedad también nos ayudará a familiarizarnos con los árboles no dirigidos en general.



(a) Un grafo ponderado

(b) El árbol y el borde después de escogerse el vértice inicial

(c) Después de seleccionar una arista y vértice: no se muestra *BG* porque *AG* es una mejor opción (tiene menor peso) para llegar a *G*

Figura 8.2 Una iteración del ciclo del algoritmo de Prim: las líneas continuas son aristas de árbol, y las punteadas, aristas a vértices de borde.

Definición 8.2 Propiedad de árbol abarcante mínimo

Sea $G = (V, E, W)$ un grafo ponderado conectado y sea T cualquier árbol abarcante de G. Supóngase que, para cada arista uv de G que *no está* en T, si uv se añade a T se crea un ciclo tal que uv es una arista de peso máximo de ese ciclo. En tal caso, el árbol T tiene la *propiedad de árbol abarcante mínimo* (*propiedad MST*, para abreviar). ■

Primero veremos qué significa la definición. Luego demostraremos que el nombre es apropiado; ¡el mero hecho de llamarla "propiedad de árbol abarcante mínimo" no implica que tenga algo que ver con los árboles abarcantes mínimos!

Ejemplo 8.2 Propiedad de árbol abarcante mínimo

Por definición, una arista no dirigida conecta cualesquier dos vértices de un árbol, y no tiene ciclos. Examinemos la figura 8.1, que muestra un grafo sencillo que llamaremos G y tres árboles abarcantes. Llamemos primero T al árbol de la parte (b). Supóngase que añadimos a T una arista de G que no está en T, formando un nuevo subgrafo G_1 (añadiremos la que pesa 2). Esto crea un ciclo (y sólo un ciclo) en el subgrafo G_1. (¿Por qué?) Ninguna otra arista del ciclo tiene un peso mayor que 2, que es el peso de la nueva arista. Como alternativa, si añadimos la arista que pesa 4, ninguna de las otras aristas del ciclo que se forma con *esa* arista pesará más de 4 (de hecho, ninguna pesa más de 3). No hay más aristas que probar, así que T tiene la propiedad de árbol abarcante mínimo. El árbol de la parte (c) es similar.

Sea ahora T el árbol de la parte (d), y añadamos la arista faltante que pesa 1. En esta ocasión, alguna otra arista del ciclo así formado pesa más de 1. Por tanto, este T *no* tiene la propiedad de árbol abarcante mínimo. Obsérvese que podemos extirpar cualquier arista de este ciclo para formar un nuevo subgrafo G_2, así que G_2 también debe ser un árbol (de hecho, un árbol abarcante). Esto se demuestra en el ejercicio 8.2 y se usa para demostrar el lema y el teorema siguientes. Pues-

to que existe una arista con peso mayor que 1, optamos por extirpar una de esas aristas. Ello implica que G_2 pesa menos que T, así que T no podría ser un árbol abarcante mínimo. ■

Lema 8.1 En un grafo ponderado conectado $G = (V, E, W)$, si T_1 y T_2 son árboles abarcantes que poseen la propiedad MST, tienen el mismo peso total.

Demostración La demostración es por inducción con k, el número de aristas que están en T_1 y no están en T_2. (Hay asimismo exactamente k aristas en T_2 que no están en T_1.) El caso base es $k = 0$; en este caso, T_1 y T_2 son idénticos, así que tienen el mismo peso.

Para $k > 0$, supóngase que el lema se cumple para los árboles que difieren en j aristas, donde $0 \leq j < k$. Sea uv una arista *de peso mínimo* que está en uno de los árboles T_1 o T_2, pero no en ambos. Supóngase $uv \in T_2$; el caso en que $uv \in T_1$ es simétrico. Consideremos el camino (único) que va de u a v en T_1: $w_0, w_1, \ldots, w_p$, donde $w_0 = u$, $w_p = v$ y $p \geq 2$. Este camino debe contener alguna arista que no está en T_2. (¿Por qué?) Sea $w_i w_{i+1}$ tal arista. Por la propiedad MST de T_1, $w_i w_{i+1}$ no puede tener un peso *mayor* que el peso de uv. Por el hecho de que uv se escogió como arista de peso mínimo de entre todas las aristas que difieren, $w_i w_{i+1}$ no puede pesar *menos* que uv. Por tanto, $W(w_i w_{i+1}) = W(uv)$. Añadimos uv a T_1, con lo que creamos un ciclo, y luego quitamos $w_i w_{i+1}$ con lo que rompemos ese ciclo y dejamos un nuevo árbol abarcante T_1' cuyo peso total es igual al de T_1. Sin embargo, T_1' y T_2 sólo difieren en $k - 1$ aristas, así que por la hipótesis inductiva tienen el mismo peso total. Por consiguiente, T_1 y T_2 tienen el mismo peso total. □

La demostración de este lema también nos muestra el método paso por paso para transformar cualquier árbol abarcante mínimo, T_1, en cualquier otro, T_2. Escogemos una arista de peso mínimo en T_2 que no esté en T_1; llamémosla uv. Examinamos el camino que conduce de u a v en T_1. En algún punto de ese camino habrá una arista con el mismo peso que uv, la cual no está en T_2; digamos que esa arista es xy (véase el ejercicio 8.3). Quitamos xy y añadimos uv. Esto nos acerca un paso más a T_2. Repetimos el paso hasta que los árboles coinciden.

Teorema 8.2 En un grafo ponderado conectado $G = (V, E, W)$, un árbol T es un árbol abarcante mínimo si y sólo si tiene la propiedad MST.

Demostración (Sólo si) Supóngase que T es un árbol abarcante mínimo para G. Supóngase que existe alguna arista uv que no está en T, tal que la adición de uv crea un ciclo en el que alguna *otra* arista xy tiene un peso $W(xy) > W(uv)$. Entonces, la eliminación de xy creará un nuevo árbol abarcante con un peso total menor que el de T, lo que contradice el supuesto de que T tenía peso mínimo.

(Si) Supóngase que T tiene la propiedad MST. Sea $T_{mín}$ *cualquier* árbol abarcante mínimo de G. Por la primera mitad del teorema, $T_{mín}$ tiene la propiedad MST. Por el lema 8.1, T tiene el mismo peso total que $T_{mín}$. □

8.2.4 Corrección del algoritmo MST de Prim

Ahora usaremos la propiedad MST para demostrar que el algoritmo de Prim construye un árbol abarcante mínimo. Esta demostración adopta una forma que se presenta con frecuencia al usar in-

ducción: la afirmación a demostrar por inducción es un poco más detallada que el teorema que nos interesa. Por ello, primero demostramos esa afirmación más detallada como *lema*. Luego el teorema simplemente extrae la parte interesante del lema. En este sentido, el teorema se parece mucho a la "envoltura" de un procedimiento recursivo, que describimos en la sección 3.2.2.

Lema 8.3 Sea $G = (V, E, W)$ un grafo ponderado conectado con $n = |V|$; sea T_k el árbol de k vértices construido por el algoritmo de Prim, para $k = 1, \ldots, n$; y sea G_k el subgrafo de G inducido por los vértices de T_k (es decir, uv es una arista de G_k si es una arista de G y tanto u como v están en T_k). Entonces T_k tiene la propiedad MST en G_k.

Demostración La demostración es por inducción con k. El caso base es $k = 1$. En este caso, G_1 y T_1 contienen el vértice inicial y ninguna arista, así que T_1 tiene la propiedad MST en G_1.

Para $k > 1$, suponemos que T_j tiene la propiedad MST en G_j para $1 \leq j < k$. Suponemos que el k-ésimo vértice que el algoritmo de Prim añadirá al árbol es v, y que las aristas entre v y los vértices de T_{k-1} son $u_1v, \ldots, u_dv$. Para concretar, supóngase que u_1v es la arista de peso más bajo de todas éstas que el algoritmo escoge. Necesitamos verificar que T_k tiene la propiedad MST. Es decir, si xy es cualquier arista de G_k que no está en T_k, necesitamos demostrar que xy tiene peso máximo en el ciclo que se crearía añadiendo xy a T_k. Si $x \neq v$ y $y \neq v$, quiere decir que xy también estaba en G_{k-1} pero no en T_{k-1}, así que por la hipótesis inductiva es máxima en el ciclo que se crea al añadírsele a T_{k-1}. Sin embargo, éste es el mismo ciclo en T_k, así que T_k tiene la propiedad MST en este caso. Falta demostrar que se posee la propiedad cuando xy es una de las aristas $u_2v, \ldots, u_dv$ (puesto que u_1v está en T_k). Si $d < 2$, ya terminamos, así que suponemos que no es el caso.

Podría ser útil consultar la figura 8.3 durante el resto de la demostración. Consideremos el camino de v a u_i en T_k para cualquier i, $1 \leq i \leq d$. Supóngase que alguna arista de este camino pesa *más* que u_iv, que a su vez pesa por lo menos lo mismo que u_1v. (Si no, se satisfaría la propiedad MST.) Específicamente, sea dicho camino $v, w_1, \ldots, w_p$, donde $w_1 = u_1$ y $w_p = u_i$. Entonces $w_1, \ldots, w_p$ es un camino en T_{k-1}. Sea w_aw_{a+1} la *primera* arista de este camino cuyo peso es mayor que $W(u_iv)$ y sea $w_{b-1}w_b$ la *última* arista de este camino cuyo peso es mayor que $W(u_iv)$ (posiblemente, $a + 1 = b$; véase la figura 8.3). Afirmamos que w_a y w_b no pueden existir en T_{k-1} si éste fue construido por el algoritmo de Prim. Supóngase que w_a se añadió al árbol antes que w_b. Entonces todas las aristas del camino que va de w_1 (que es u_1) a w_a se añadirían antes que w_aw_{a+1} y antes que $w_{b-1}w_b$, porque todas tienen menor peso, y u_1v también se habría añadido antes que cualquiera de las dos. Asimismo, si w_b se añadió al árbol antes que w_a, u_iv se habría añadido antes que w_aw_{a+1} y antes que $w_{b-1}w_b$. Sin embargo, ni u_1v ni u_iv están en T_{k-1}, así que ninguna arista del camino $w_1, \ldots, w_p$ pesa más de $W(u_iv)$, y queda establecida la propiedad MST para T_k. □

Teorema 8.4 El algoritmo de Prim produce un árbol abarcante mínimo.

Demostración En la terminología del lema 8.3, $G_n = G$ y T_n es la salida del algoritmo, de lo cual se sigue que T_n tiene la propiedad MST en G. Por el teorema 8.2, T_n es un árbol abarcante mínimo de G. □

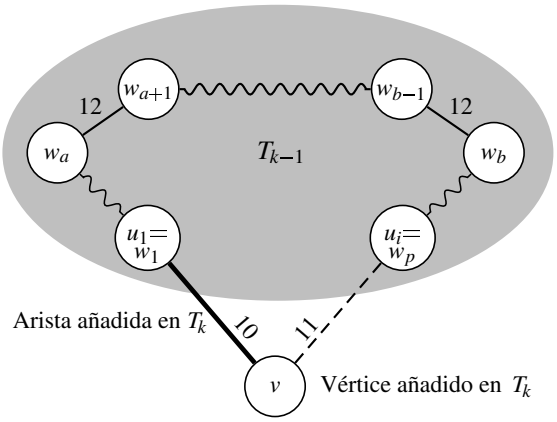


Figura 8.3 Ilustración para el lema 8.3. Los pesos que se dan son ejemplos. Las líneas onduladas son caminos en T_{k-1}. La arista punteada crearía un ciclo, como se muestra. Todas las aristas del camino entre u_1 ($= w_1$) y w_a, y las aristas del camino entre w_b y u_i ($= w_p$) tienen pesos no mayores que $W(u_i v)$, que es 11 en este ejemplo. Posiblemente, $w_b = w_{a+1}$ y $w_{b-1} = w_a$.

8.2.5 Cómo manejar el borde de manera eficiente con una cola de prioridad

Después de cada iteración del ciclo del algoritmo, podría haber nuevos vértices de borde, y el conjunto de aristas de entre las cuales se escogerá la siguiente cambiará. La figura 8.2(c) sugiere que no es necesario considerar todas las aristas entre vértices de árbol y vértices de borde. Después de escogerse *AB*, *BG* se convirtió en una opción, pero se desechó porque *AG* pesa menos y sería una mejor opción para llegar a *G*. Si *BG* pesara menos que *AG*, podría desecharse *AG*. Para cada vértice de borde, sólo necesitamos recordar una de las aristas que llegan a él desde el árbol: la de menor peso. Llamamos *aristas candidatas* a tales aristas.

El TDA de cola de prioridad (sección 2.5.1) tiene precisamente las operaciones que necesitamos para implementar el bosquejo del algoritmo dado en la sección 8.2.2. La operación `insertar` introduce un vértice en la cola de prioridad. La operación `obtenerMin` puede servir para escoger el vértice de borde que se puede conectar al árbol actual incurriendo en un costo mínimo. La operación `borrarMin` saca a ese vértice del borde. La operación `decrementarClave` registra un costo más favorable para la conexión de un vértice de borde cuando se descubre una mejor arista candidata. El costo mínimo conocido de conectar cualquier vértice de borde es el `pesoBorde` de ese vértice. Este valor hace las veces de prioridad del vértice y lo devuelve la función de acceso `obtenerPrioridad`.

Utilizando las operaciones del TDA de cola de prioridad, el algoritmo de alto nivel es el siguiente. Hemos introducido una subrutina `actualizarBorde` para procesar los vértices adyacentes al vértice seleccionado v. La figura 8.4 muestra un ejemplo de la acción del algoritmo.

```
primMST(G, n) // BOSQUEJO
    Inicializar la cola de prioridad cp como cola vacía.
    Seleccionar un vértice arbitrario s para iniciar el árbol;
    Asignar (−1, s, 0) a su arista candidata e invocar insertar(cp, s, 0).
    Mientras cp no esté vacía:
       v = obtenerMin(cp); borrarMin(cp);
       Añadir la arista candidata de v al árbol.
       actualizarBorde(cp, G, v);

actualizarBorde(cp, G, v) // BOSQUEJO
    Para todos los vértices w adyacentes a v, con nuevoPeso = W(v, w):
       Si w es no visto:
          Asignar (v, w, nuevoPeso) a su arista candidata.
          insertar(cp, w, pesoNuevo);
       Si no, si pesoNuevo < pesoBorde de w:
          Cambiar su arista candidata a (v, w, nuevoPeso).
          decrementarClave(cp, w, nuevoPeso);
```

Análisis preliminar

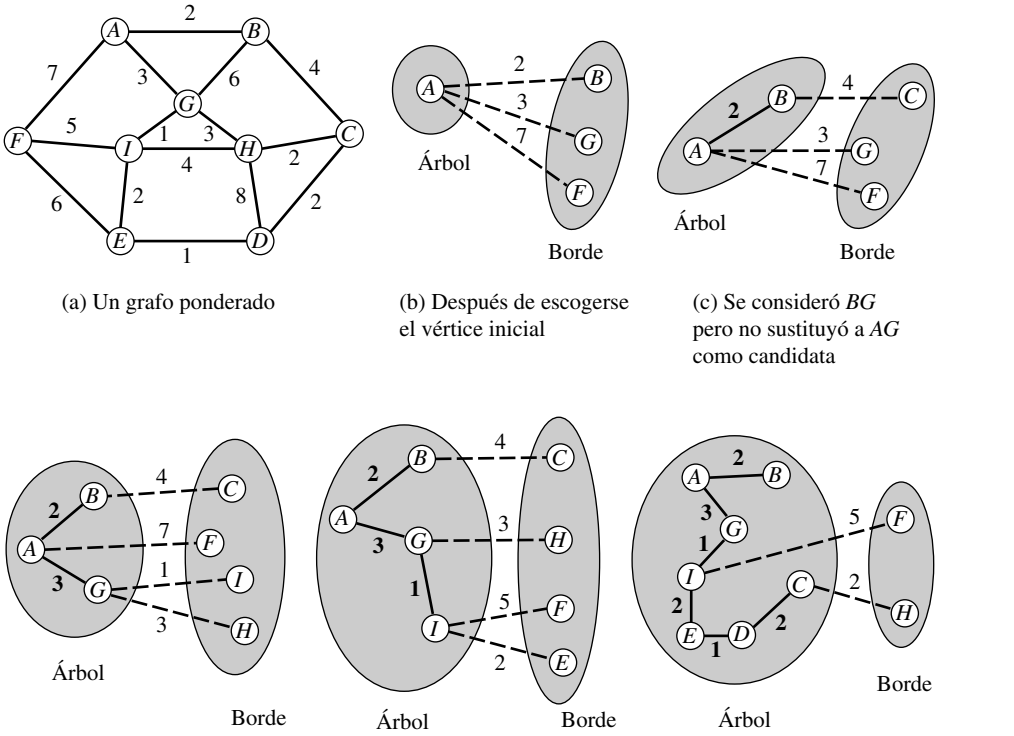
¿Qué podemos decir acerca del tiempo de ejecución de este algoritmo sin saber cómo se implementa el TDA de cola de prioridad? El primer paso sería estimar cuántas veces se efectúa cada operación del TDA. Luego podríamos escribir una expresión en la que los costos de las operaciones del TDA son parámetros. Supóngase, como es nuestra costumbre, que el grafo tiene n vértices y m aristas. Es fácil ver que el algoritmo ejecuta `insertar`, `obtenerMin` y `borrarMin` aproximadamente n veces, y que ejecuta `decrementarClave` cuando más m veces. (Hay $2m$ iteraciones del ciclo **`for`** que ejecuta `decrementarClave` porque cada arista se procesa desde ambas direcciones, pero veremos después que la condición del segundo **`if`** no se satisface más de una vez para cada arista.) Trabajando un poco podremos construir ejemplos en los que prácticamente cada arista dispare un `decrementarClave`. Es razonable suponer que `insertar` es menos costoso que `borrarMin`. Por tanto, tenemos una expresión en la que las T de la derecha denotan el tiempo medio que tarda la operación indicada en el curso de una ejecución del algoritmo:

$$T(n, m) = O\,(n\,T\,(\texttt{obtenerMin}) + n\,T\,(\texttt{borrarMin}) + m\,T\,(\texttt{decrementarClave})). \qquad (8.1)$$

En grafos generales m podría ser mucho mayor que n, por lo que obviamente nos interesa una implementación que se concentre en la eficiencia de `decrementarClave`.

Aquí nos damos cuenta de la ventaja de diseñar con tipos de datos abstractos. Ya razonamos que nuestro algoritmo es correcto, sea como sea que se implemente el TDA de cola de prioridad, siempre que la implementación cumpla con las especificaciones lógicas del TDA. Ahora estamos en libertad de adecuar la implementación a fin de minimizar, o por lo menos reducir, el miembro derecho de la ecuación (8.1).

Ya vimos que un montón es una implementación eficiente de una cola de prioridad (sección 4.8.1). ¿Cómo le va en la ecuación (8.1)? Esta pregunta es el tema del ejercicio 8.9, donde descubrimos que el peor caso es peor que $\Theta(n^2)$. ¿Podemos mejorarlo?



(a) Un grafo ponderado

(b) Después de escogerse el vértice inicial

(c) Se consideró *BG* pero no sustituyó a *AG* como candidata

(d) Después de seleccionarse *AG* y de actualizarse el borde y las candidatas

(e) *IF* ha sustituido a *AF* como candidata

(f) Después de varias pasadas más: las dos aristas candidatas se colocarán en el árbol

Figura 8.4 Un ejemplo de árbol abarcante mínimo de Prim

Si se quiere mejorar en general el tiempo del montón, es evidente que será preciso considerar implementaciones con las que `decrementarClave` se ejecute en un tiempo menor que $\Theta(\log n)$. Sin embargo, podemos darnos el lujo de hacer que `obtenerMin` y `borrarMin` tarden más de $\Theta(\log n)$ como concesión. ¿Hay alguna implementación en la cual `decrementarClave` tarde $O(1)$ y las otras operaciones no tarden más de $O(n)$?, si así fuera, la ecuación (8.1) daría $\Theta(n^2 + m) = \Theta(n^2)$. Invitamos a los lectores a considerar algunas alternativas antes de continuar.

■ ■ ■

La respuesta es tan simple, que es probable que la pasemos por alto. Basta con almacenar la información en uno o más arreglos, indizados por número de vértice. Es decir, podemos usar un arreglo aparte para cada campo, o juntar los campos en una clase organizadora y tener un arreglo cuyos elementos sean objetos de esa clase. Optaremos por usar arreglos individuales porque ello simplifica un poco la sintaxis. La operación `decrementarClave` es $O(1)$ porque basta con se-

guir el índice al vértice y actualizar dos o tres campos. La operación `obtenerMin` se efectúa examinando las *n* entradas de los arreglos; `borrarMin` puede usar el resultado del `obtenerMin` precedente o examinar otra vez los arreglos si el resultado anterior es obsoleto. Uno de los arreglos deberá ser el indicador de *situación* para indicar si el vértice está o no en el borde; sólo esos vértices serán elegibles como mínimos. Otro arreglo contiene la prioridad del vértice. En nuestro algoritmo, la prioridad del vértice debe corresponder siempre a `pesoBorde`, el peso de la arista candidata para ese vértice. Asimismo, las aristas candidatas se pueden mantener en un arreglo llamado `padre`, como se hizo el las búsquedas primero en amplitud y primero en profundidad. Es decir, `(v, padre[v])` es la arista candidata para *v*. Con esta implementación hemos llegado básicamente al algoritmo de Prim clásico.

Aunque el tema del encapsulamiento del diseño de TDA sugiere que las estructuras de datos de la cola de prioridad estén ocultas, las "desabstraeremos" para que todas las partes del algoritmo tengan acceso simple. No obstante, seguiremos usando las operaciones del TDA para *actualizar* esas estructuras de datos.

El algoritmo de Prim se publicó antes de que se inventaran las colas de prioridad y antes de que los lenguajes de programación apoyaran las estructuras de datos modernas, así que su implementación se basó simplemente en arreglos. Desde entonces, se han realizado investigaciones sustanciales acerca de la eficiencia de las colas de prioridad. Después de presentar la implementación más sencilla del algoritmo de Prim, mostraremos (en la sección 8.2.8) cómo adaptar la estructura de datos de *bosque de apareamiento* (sección 6.7.2) para este algoritmo. Esto puede servirnos como guía para usar otras implementaciones avanzadas de la cola de prioridad, como el *montón de Fibonacci*, que rebasan el alcance de este libro. (Véase Notas y referencias al final del capítulo.)

8.2.6 Implementación

Las principales estructuras de datos para el algoritmo (además de las empleadas para el grafo mismo) son tres arreglos: `situación`, `pesoBorde` y `padre`, indizados por número de vértice. La clasificación de los vértices está dada por el arreglo `situacion` y suponemos que hemos definido constantes con los nombres `novisto`, `deborde` y `dearbol`. Existe una fuerte correlación entre éstos y los colores blanco, gris y negro empleados en la búsqueda primero en amplitud (algoritmo 7.1). A veces, una búsqueda basada en la cola de prioridad en vez de la cola FIFO se denomina *búsqueda de primero el mejor*.

Los tres arreglos principales, `situacion`, `padre` y `pesoBorde` se reúnen en el objeto `cp` por comodidad al pasarlos a las subrutinas. Además, adecuaremos las operaciones `insertar` y `decrementarClave` de modo que registren el padre, no sólo la prioridad, del vértice que se está insertando o actualizando. Cuando se construye inicialmente `cp`, todos los elementos tienen la situación `novisto`.

Cuando invocamos `insertar` para un vértice, sus `padre` y `pesoBorde` adquieren valores y su situación cambia a `deborde`. Cabe señalar que `insertar` se invoca con el vértice inicial para crear el primer vértice de borde; su `padre` no es un vértice real.

Cuando invocamos `borrarMin`, la situación del vértice que actualmente es el mínimo cambia a `dearbol`, con lo que efectivamente lo sacamos de la cola de prioridad. Otro efecto de esto es que se congelan sus campos `pesoBorde` y `padre`.

En el ciclo principal del algoritmo `primMST`, conforme se recupera cada uno de los vértices (v) que en ese momento son el mínimo, su lista de adyacencia se procesa (en la subrutina `actualizarBorde`) para ver si alguna de estas aristas (v, w) ofrecen una conexión de menor costo con

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*
`pesoBorde`	0	2	4			7	3	3	1
`padre`	−1	*A*	*B*			*A*	*A*	*G*	*G*
`situacion`	`dearbol`	`dearbol`	`deborde`	`novisto`	`novisto`	`deborde`	`dearbol`	`deborde`	`deborde`

Figura 8.5 Estructura de datos de árbol abarcante mínimo para la situación de la figura 8.4(d); no se muestran listas de adyacencia. Se supone que los vértices están en orden alfabético dentro de cada lista.

el vértice adyacente *w*. Suponemos que las listas de adyacencia contienen elementos de la clase organizadora `InfoArista` con dos campos, `a` y `peso`, según la descripción de la sección 7.2.3 y la ilustración de la figura 7.11.

La figura 8.5 muestra la estructura de datos en un punto intermedio de la ejecución del algoritmo del ejemplo de la figura 8.4 (específicamente, en el punto ilustrado por la figura 8.4d). Para facilitar su comprensión, mostramos los nombres de los vértices como letras, igual que en la figura 8.4.

Cuando el algoritmo termina, el arreglo `padre` implica las aristas del árbol. Es decir, para cada vértice *v* distinto del vértice inicial (raíz), `(v, padre[v])` es una arista del MST y `pesoBorde[v]` es su peso.

El contador `numCP` lleva la cuenta del número de vértices cuya situación es `deborde`, para que `estaVacia(cp)` se pueda ejecutar en tiempo constante. Si el grafo de entrada no está conectado, `padre[v]` y `pesoBorde[v]` no estarán definidos para los *v* que no estén conectados con el vértice inicial cuando el algoritmo termine. Esta condición nunca deberá presentarse porque una condición previa del algoritmo es que el grafo está conectado.

Algoritmo 8.1 Árbol abarcante mínimo (de Prim)

Entradas: Un arreglo `infoAdya` de listas de adyacencia que representa un grafo no dirigido, conectado y ponderado $G = (V, E, W)$, según se describe en la sección 7.2.3; *n*, el número de vértices; y *s*, el vértice inicial deseado. Todos los arreglos deberán estar definidos para los índices 1, ..., *n*; el elemento número 0 no se usa. Los elementos de `infoAdya` son listas del TDA `ListaAristas` que se describe más adelante.

Salidas: Un árbol abarcante mínimo almacenado en el arreglo `padre` como árbol adentro y el arreglo `pesoBorde` que contiene, para cada vértice *v*, el peso de la arista entre `padre[v]` y *v*. (El padre de la raíz es −1.) El invocador reserva espacio para los arreglos y los pasa como parámetros y el algoritmo los llena.

Comentarios: El arreglo `situacion[1]`, ..., `situacion[n]` denota la situación de búsqueda actual de todos los vértices. Los vértices no descubiertos son `novisto`; los que ya se descubrieron pero todavía no se procesan (en la cola de prioridad) son `deborde`; los que ya se procesaron son `dearbol`. Las listas de adyacencia son del tipo `ListaAristas` y tienen las operaciones estándar del TDA Lista (sección 2.3.2). Los elementos pertenecen a la clase organizadora `InfoArista` que tiene dos campos `a` y `peso`.

```
void primMST(ListaAristas[] infoAdya, int n, int s, int[] padre,
float[] pesoBorde)
   int[] situacion = new int[n+1];
   CPMin cp = crear(n, situacion, padre, pesoBorde);

   insertar(cp, s, −1, 0);
   while (estaVacia(cp) == false)
     int v = obtenerMin(cp);
     borrarMin(cp);
     actualizarBorde(cp, infoAdya[v], v);
   return;

/** Ver si se halla una mejor conexión con cualquier vértice
  * de la lista infoAdyaDeV, y decrementarClave en tal caso.
  * Para una conexión nueva, insertar el vértice. */
void actualizarBorde(CPMin cp, ListaAristas infoAdyaDeV, int v)
   ListaAristas adyaRest;
   adyaRest = infoAdyaDeV;
   while (adyaRest ≠ nil)
     InfoArista infoW = primero(adyaRest);
     int w = infoW.a;
     float nuevoPeso = infoW.peso;
     if (cp.situacion[w] == novisto)
       insertar(cp, w, v, nuevoPeso);
     else if (cp.situacion[w] == deborde)
       if (nuevoPeso < obtenerPrioridad(cp, w))
         decrementarClave(cp, w, v, nuevoPeso);
     adyaRest = resto(adyaRest);
   return;
```

La implementación de cola de prioridad se muestra en las figuras 8.6 a 8.8.

8.2.7 Análisis (tiempo y espacio)

Ahora completaremos el análisis del algoritmo 8.1 ejecutado con $G = (V, E, W)$. Nos habíamos quedado en la ecuación (8.1). El procedimiento principal es `primMST`, que invoca la subrutina `actualizarBorde`, pero ambos invocan las operaciones del TDA `CPMin`. Sean $n = |V|$ y $m = |E|$. El número de operaciones de inicialización (`crear`) es lineal en n. El cuerpo del ciclo **`while`** se ejecuta n veces, porque cada pasada efectúa un `borrarMin`. Necesitamos estimar el tiempo que se requiere para las invocaciones de procedimientos en este ciclo: `estaVacia`, `obtenerMin`, `borrarMin` y `actualizarBorde`.

Un parámetro de `actualizarBorde` es la lista `vertsAdya`, el cuerpo de su ciclo **`while`** se ejecuta una vez para cada elemento de esta lista (la invocación se omitiría para el último vértice en eliminarse de la cola de prioridad). Durante el curso del algoritmo, `actualizarBorde` procesa la lista de adyacencia de cada vértice una vez, así que el número total de pasadas por el cuerpo del ciclo **`while`** es aproximadamente $2m$. Una pasada por ese ciclo invoca `primero`, `resto` y

```
class CPMin
    // Campos de ejemplar
    int numVertices, numCP;
    int minVertice;
    float oo;
    int[] situacion;
    int[] padre;
    float[] pesoBorde;

    /** Construir cp con n vertices, todos "no vistos". */
    CPMin crear(int n, int[] situacion, int[] padre, float[] pesoBor-
    de)
      CPMin cp = new CPMin();
      cp.padre = padre;
      cp.pesoBorde = pesoBorde;
      cp.situacion = situacion;
      Inicializar situacion[1], ..., situacion[n] con novisto.
      cp.numVertices = n; cp.numCP = 0;
      cp.minVertice = -1;
      cp.oo = Float.POSITIVE_INFINITY;
      return cp;
```

Figura 8.6 Implementación de cola de prioridad para el algoritmo de árbol abarcante mínimo de Prim, parte 1: estructuras de datos y constructor del TDA. Los arreglos tienen un elemento por cada vértice del grafo; el elemento con índice 0 no se usa.

`obtenerPrioridad`, que suponemos están en $\Theta(1)$, pero también invoca `insertar` o `decrementarClave`. Con la implementación de la figura 8.7, `insertar` y `decrementarClave` también están en $\Theta(1)$, pero debemos tener presente que otras implementaciones quizá no lo logren; se trata de una decisión crucial: se podría invocar `decrementarClave` para casi todas las aristas de G, un total de aproximadamente $m - n$ invocaciones. Con la implementación que escogimos, el tiempo total para todas las invocaciones de `actualizarBorde` está en $\Theta(m)$.

Hasta aquí parece que el tiempo de ejecución del algoritmo podría ser lineal en m (G está conectado, así que m no puede ser mucho menor que n, pero podría ser mucho mayor). Sin embargo, `obtenerMin` se invoca aproximadamente n veces desde `primMST` y debe invocar la subrutina `hallarMin` en cada una de esas invocaciones. La subrutina `hallarMin` efectúa una comparación de peso para cada vértice que está "en" la cola de prioridad, a fin de hallar la arista candidata mínima. En el peor caso, no habrá vértices "no vistos" después de la primera invocación de `actualizarBorde`. Entonces el número medio de vértices que requerirán una comparación de peso será de aproximadamente $n/2$, puesto que se borra uno después de cada invocación de `obtenerMin`. (Nos concentramos en las comparaciones de pesos porque son inevitables; alguna otra implementación podría evitar la verificación de `situacion`.) Tenemos un total de (aproximadamente) $n^2/2$ comparaciones, aun si el número de aristas es menor. Una vez más, hacemos hincapié en que el tiempo que tarda `hallarMin` depende de la implementación y es una decisión

```
/** Asentar a nuevoPadre y nuevoP como padre y prioridad de v,
  * respectivamente, y hacer situacion[v] = deborde. */
void insertar(CPMin cp, int v, int nuevoPadre, float nuevoP)
   cp.padre[v] = nuevoPadre;
   cp.pesoBorde[v] = nuevoP;
   cp.situacion[v] = deborde;
   cp.verticeMin = -1;
   cp.numCP ++;
   return

/** Asentar nuevoPadre, nuevoP como padre, prioridad de v. */
void decrementarClave(CPMin cp, int v, int nuevoPadre, float nuevoP)
   cp.padre[v] = nuevoPadre;
   cp.pesoBorde[v] = nuevoP;
   cp.verticeMin = -1;
   return

/** Borrar de cp vértice de borde con peso mínimo. */
void borrarMin(CPMin cp)
   int viejoMin = obtenerMin(cp);

   cp.situacion[viejoMin] = dearbol;
   cp.verticeMin = -1;
   cp.numCP --;
   return
```

Figura 8.7 Implementación de cola de prioridad para el algoritmo de árbol abarcante mínimo de Prim, parte 2: procedimientos de manipulación.

crucial para la eficiencia general de `primMST`. Obsérvese que `borrarMin` también se invoca unas n veces, pero sólo requiere $O(1)$ por invocación; una implementación distinta podría desplazar el trabajo de `obtenerMin` a `borrarMin`. Por lo regular será necesario analizar juntas estas dos invocaciones.

Así pues, el tiempo de ejecución de peor caso, lo mismo que el número de comparaciones efectuadas en el peor caso, están en $\Theta(m + n^2) = \Theta(n^2)$. (Recomendamos al lector investigar formas de reducir el trabajo que se requiere para hallar las candidatas mínimas, pero véanse los ejercicios 8.7 a 8.9.)

La estructura de datos de la figura 8.5 emplea $3n$ celdas además de las de la representación del grafo con listas de adyacencia. Este espacio adicional es mayor que el ocupado por cualquiera de los algoritmos que hemos estudiado hasta ahora, y podría parecer demasiado. Sin embargo, hace posible una implementación del algoritmo eficiente en términos del tiempo. (Sería peor si el espacio extra requerido estuviera en $\Theta(m)$, pues para muchos grafos $\Theta(m) = \Theta(n^2)$.)

```
boolean estaVacia(CPMin cp)
    return (numCP == 0);

float obtenerPrioridad(CPMin cp, int v)
    return cp.pesoBorde[v];

/** Devolver vértice de borde con peso mínimo.
 ** Devolver -1 si no quedan vértices de borde.
  */
int obtenerMin(CPMin cp)
    if (cp.verticeMin == -1)
      hallarMin(cp);
    return cp.verticeMin;

// ¡Esta subrutina hace casi todo el trabajo!
void hallarMin(CPMin cp)
    int v;
    float pesoMin;

    pesoMin = cp.oo;
    for (v = 1; v <= cp.numVertices; v++)
      if (cp.situacion[v] == deborde)
        if (cp.pesoBorde[v] < pesoMin)
          cp.verticeMin = v;
          pesoMin = cp.pesoBorde[v];
      // Continuar ciclo
    return;
```

Figura 8.8 Implementación de cola de prioridad del algoritmo de árbol abarcante mínimo de Prim, parte 3: funciones de acceso y subrutina `hallarMin` de `obtenerMin`.

8.2.8 La interfaz de bosque de apareamiento

La estructura de datos de bosque de apareamiento general y las implementaciones de las operaciones de cola de prioridad se describieron en la sección 6.7.2. Adaptaciones menores permiten utilizarla en el algoritmo de Prim.

En primer lugar, la estructura de bosque de apareamiento general supone que un nodo de árbol contiene campos tanto para el identificador de elemento como para la prioridad. Sin embargo, en este caso el identificador (número de vértice) es suficiente porque podemos acceder a la prioridad de v como `pesoBorde[v]`. Por ello, los pasos de la sección 6.7.2 que dicen "crear `nuevoNodo` ..." deben modificarse para que digan "guardar la prioridad en `pesoBorde[v]` y crear `nuevoNodo` con `id` = v". El arreglo `refx` es un arreglo adicional parecido a `situacion`, el cual mantienen las operaciones del TDA de cola de prioridad. (Con algunos valores artificiales especiales de tipo `Arbol` para representar las situaciones `novisto` y `dearbol`, el arreglo `refx` puede sustituir al arreglo `situacion`, como optimización de espacio.) ¿Qué ganamos al usar el

bosque de apareamiento? Las operaciones `insertar` y `decrementarClave` se siguen ejecutando en tiempo constante. Los posibles ahorros están en la operación `obtenerMin`. En la implementación directa, `obtenerMin` debe examinar todo el arreglo `situacion` y posiblemente una buena parte del arreglo `pesoBorde` en cada operación. El bosque de apareamiento tiene la propiedad de que sólo las raíces de los árboles del bosque son candidatas para el mínimo. Aunque es difícil analizar cuántos árboles podría haber en diversos momentos durante la ejecución del algoritmo, es evidente que lo normal es que cada árbol tenga varios nodos, así que no será necesario examinar todos los vértices.

En general, el orden asintótico de peor caso no se conoce con exactitud. Sin embargo, se ha obtenido un resultado de optimidad parcial para una variante del TDA de bosque de apareamiento, llamada *montones de apareamiento de dos pasadas*. Para la clase de grafos en la que m crece según $\Theta(n^{1+c})$ para alguna constante $c > 0$, el costo amortizado de `obtenerMin` está en $\Theta(\log n)$ y los costos amortizados de `insertar` y `decrementarClave` están en $\Theta(1)$. Estas cotas implican que el algoritmo de Prim con montones de apareamiento de dos pasadas se ejecuta en $\Theta(m + n \log n) = \Theta(m)$ con esta clase de grafos.

Se sabe también que el algoritmo de Prim con montones de Fibonacci se ejecuta en $\Theta(m + n \log n)$ con todos los grafos, lo cual es asintóticamente óptimo. Sin embargo, se ha informado que los factores constantes para los montones de Fibonacci son muy grandes y las operaciones mismas son muy complicadas y difíciles de implementar. Por estas razones, se considera que el montón de apareamiento o el bosque de apareamiento es una alternativa práctica. El tema se trata en Notas y referencias al final del capítulo.

8.2.9 Cota inferior

¿Qué tanto trabajo es indispensable para hallar un árbol abarcante mínimo? Afirmamos que cualquier algoritmo de árbol abarcante mínimo requiere un tiempo en $\Omega(m)$ en el peor caso porque debe examinar, o procesar de alguna manera, todas las aristas del grafo. Para ver esto, sea G un grafo ponderado conectado en el que todas las aristas pesan por lo menos 2, y suponiendo que existiera un algoritmo que no hiciera absolutamente nada con una arista xy de G. Entonces xy no estaría en el árbol T que el algoritmo produce como salida. Cambiemos el peso de xy a 1. Esto no podría alterar la acción del algoritmo porque nunca examinó a xy. Sin embargo, ahora T no tiene la propiedad MST y, por el teorema 8.2, no es un árbol abarcante mínimo. De hecho, si queremos producir un árbol más ligero bastará con añadir xy a T para crear un ciclo, eliminando cualquier otra arista de ese ciclo. Por consiguiente, ningún algoritmo que no examine xy podrá ser correcto.

8.3 Caminos más cortos de origen único

En la sección 7.2 consideramos brevemente el problema de hallar la mejor ruta entre dos ciudades en un mapa de rutas aéreas, como la figura 7.8. Utilizando como criterio el precio de los pasajes de avión, observamos que la mejor forma —es decir, la más barata— de viajar de San Diego a Sacramento era haciendo una escala en Los Ángeles. Éste es un ejemplo, o aplicación, de un problema muy común en grafos ponderados; hallar un camino de peso mínimo entre dos vértices dados.

Da la casualidad que, en el peor caso, no es más fácil hallar un camino de peso mínimo entre un par de nodos s y t dado que hallar caminos de peso mínimo entre s y cualquier vértice al

que se puede llegar desde *s*. Este último problema se denomina *problema de camino más corto de origen único*. Se usa el mismo algoritmo para resolver ambos problemas.

En esta sección consideraremos el problema de hallar el camino de peso mínimo desde un vértice de origen dado hasta cualquier otro vértice de un grafo ponderado dirigido o no dirigido. El peso (longitud, o costo) de un camino es la suma de los pesos de las aristas que incluye. Si el peso se interpreta como distancia, decimos que un camino de peso mínimo es un *camino más corto*, y éste es el nombre que más a menudo se usa. (Lamentablemente, se acostumbra mezclar la terminología de peso, costo y longitud.)

¿Cómo determinamos el camino más corto de SD a SAC en la figura 7.8? De hecho, en ese ejemplo pequeño usamos un método muy poco algorítmico, plagado de supuestos (como el de que las tarifas eran proporcionales a la distancia entre las ciudades y el de que el mapa estaba dibujado aproximadamente a escala). Luego escogimos una ruta que "se veía" corta y calculamos su costo total. Por último, verificamos algunos otros caminos (de forma un tanto desorganizada) y no observamos ninguna mejora, así que declaramos el problema resuelto. Éste difícilmente sería un algoritmo que esperaríamos programar en una computadora. Lo mencionamos para contestar sinceramente la pregunta anterior; la gente por lo regular usa formas muy poco rigurosas de resolver problemas, sobre todo si el conjunto de datos es muy pequeño.

En la práctica, el problema de hallar caminos más cortos en un grafo se presenta en aplicaciones en las que *V* podría contener cientos, miles o incluso millones de vértices. En teoría, un algoritmo podría considerar todos los posibles caminos y comparar sus pesos, pero en la práctica ello podría requerir mucho tiempo, posiblemente siglos. Al tratar de hallar un mejor enfoque, es útil examinar algunas propiedades generales de los caminos más cortos para ver si sugieren una estrategia más eficiente. El algoritmo que presentamos se debe a E. W. Dijkstra y requiere que los pesos de las aristas no sean negativos. En Notas y referencias al final del capítulo se mencionan otros algoritmos que no imponen este requisito.

8.3.1 Propiedades de los caminos más cortos

En general, al tratar de resolver un problema grande, tratamos de dividirlo en problemas más pequeños. ¿Qué podemos decir acerca de los caminos más cortos entre nodos distantes, en términos de caminos más cortos entre nodos menos distantes? ¿Podemos usar algún enfoque del tipo de divide y vencerás? Supóngase que el camino *P* es un camino más corto de *x* a *y* y que *Q* es un camino más corto de *y* a *z*. ¿Implica esto que *P* seguido de *Q* es un camino más corto de *x* a *z*? No es difícil imaginar un ejemplo en el que lo anterior no es cierto. Sin embargo, hay una variación sutil de este tema que *sí* se cumple. La demostración de este lema se deja como ejercicio.

Lema 8.5 (Propiedad de camino más corto) En un grafo ponderado *G*, supóngase que un camino más corto de *x* a *z* consiste en un camino *P* de *x* a *y* seguido de un camino *Q* de *y* a *z*. En tal caso, *P* será un camino más corto de *x* a *y* y *Q* será un camino más corto de *y* a *z*. □

Supóngase que estamos tratando de hallar un camino más corto entre *x* y *z*. Quizá existe una arista directa *xz* que ofrece la ruta más corta. Sin embargo, si el camino más corto incluye dos o más aristas, el lema nos dice que se puede dividir en dos caminos, cada uno con menos aristas que el camino entero, cada uno de los cuales es un camino más corto por derecho propio. Si queremos desarrollar un algoritmo, necesitaremos establecer algún esquema organizado para desglosar caminos.

Ejemplo 8.3 Autobús lleno de turistas

Podemos entender mejor el problema si relacionamos los caminos más cortos desde un vértice de origen s con un proceso físico. Veamos el grafo como un grupo de islas conectadas por puentes de un solo sentido, como en el ejemplo 7.7, sólo que ahora los puentes tienen diferentes longitudes. La longitud del puente que corresponde a la arista uv es $W(uv)$.

Imaginemos un autobús lleno de turistas que deja a sus pasajeros en un vértice de origen s en el tiempo cero, igual que en ese ejemplo. Los turistas se dispersan desde s caminando a velocidad constante, digamos un metro por segundo. Cuando un grupo numeroso de turistas llega a una isla (vértice) nueva, se divide; grupos más pequeños toman cada uno de los puentes (aristas) que salen de esa isla. Es evidente que los primeros turistas en llegar a cualquier isla han seguido un camino más corto. En este ejemplo, "más corto" puede referirse al tiempo o a la distancia.

Consideremos la situación en la que los turistas están llegando por primera vez al vértice z. Supóngase que están recorriendo una arista yz. Entonces, un camino más corto de s a z pasa por y y, por el lema 8.5, consiste en un camino más corto de s a y seguido de la arista yz. ■

Simulación del grupo de turistas

Supóngase ahora que queremos *predecir* el momento en que llegarán los primeros turistas a z, y que tenemos un arreglo, `llegar`, para guardar los tiempos en que los primeros turistas llegan a cada vértice. Sean $y_1, y_2, \ldots, y_k$ los vértices conectados por una arista a z. El camino más corto tiene que usar uno de estos vértices, así que los consideraremos todos. Tan pronto como lleguen turistas a y_i, digamos en el tiempo `llegar`$[y_i]$, podremos predecir que los primeros turistas en llegar a z *no arribarán después de* `llegar`$[y_i] + W(y_iz)$. Por tanto, `llegar[z]` será el mínimo de esas predicciones. Puesto que no permitimos pesos negativos (nada de viajes al pasado para estos turistas), no tenemos que preocuparnos por los y_i desde los cuales los turistas lleguen después de `llegar[z]`.

¿Podemos usar una búsqueda primero en profundidad para organizar este cálculo? Puesto que necesitamos examinar los vértices de los cuales salen aristas *hacia* z, no las aristas que salen de z, nos interesa buscar en G^T, el grafo transpuesto (véase la definición 7.10). La idea general es que la búsqueda desde z iría a cada y_i, y al retroceder de y_i a z calcularíamos `llegar`$[y_i] + W(y_iz)$ y lo compararíamos con el valor previamente almacenado en `llegar[z]`. Cada vez que se obtenga un valor más pequeño, se guardará en `llegar[z]`. Exploraremos esta idea en el ejercicio 8.21, donde se demuestra que funciona con una clase importante de grafos, pero no con todos los grafos.

Otra idea natural para organizar el cálculo es el enfoque codicioso, puesto que hemos observado que podemos calcular `llegar[z]` una vez que conocemos los valores de `llegar`$[y_i]$ que son menores que `llegar[z]`. La heurística codiciosa en este caso consiste en hallar el vértice al cual los turistas llegarán *más pronto*, dado que ya han llegado a ciertos vértices.

8.3.2 Algoritmo de camino más corto de Dijkstra

En esta sección estudiaremos el algoritmo de camino más corto de Dijkstra; es muy similar en su enfoque y tiempos al algoritmo de árbol abarcante mínimo de Prim que vimos en la sección anterior.

Definición 8.3

Sea P un camino no vacío en un grafo ponderado $G = (V, E, W)$ que consta de k aristas xv_1, $v_1v_2, \ldots, v_{k-1}y$ (podría ser $v_1 = y$). El *peso* de P, denotado por $W(P)$, es la suma de los pesos

$W(xv_1)$, $W(v_1v_2)$, …, $W(v_{k-1}y)$. Si $x = y$, se considera que el camino vacío es un camino de x a y. El peso del camino vacío es cero.

Si ningún camino entre x y y pesa menos que $W(P)$, decimos que P es un *camino más corto*, o *camino de peso mínimo*. ■

Escogimos con cuidado las palabras de la definición anterior para dejar abierta la posibilidad de tener pesos negativos. No obstante, en esta sección supondremos que los pesos no son negativos. En tales circunstancias, los caminos más cortos pueden restringirse a los caminos simples.

Problema 8.1 Caminos más cortos de origen único

Se nos da un grafo ponderado $G = (V, E, W)$ y un vértice de origen s. El problema consiste en hallar un camino más corto de s a cada vértice v. ■

Antes de proceder, debemos considerar si necesitamos un algoritmo nuevo. Supóngase que usamos el algoritmo de árbol abarcante mínimo, partiendo de s. ¿El camino a v en el árbol construido por el algoritmo siempre será el camino más corto de s a v? Consideremos el camino de A a C en el árbol abarcante mínimo de la figura 8.4. Ése *no es* un camino más corto; el camino A, B, C es más corto.

El algoritmo de camino más corto de Dijkstra encuentra caminos más cortos de s a los demás vértices en orden de distancia creciente desde s. El algoritmo, al igual que el algoritmo MST de Prim de la sección 8.2, parte de un vértice (s) y se "ramifica" seleccionando ciertas aristas que conducen a nuevos vértices. El árbol construido por este algoritmo se denomina *árbol de caminos más cortos*. (Ponerle ese nombre no hace que sea cierto; hay que demostrar que los caminos del árbol realmente son caminos más cortos.)

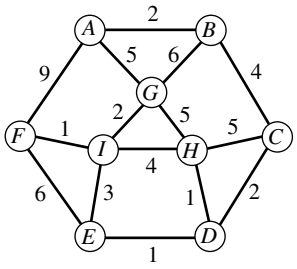
Otra semejanza con el algoritmo MST de Prim es que el algoritmo de Dijkstra es codicioso; siempre escoge una arista al vértice que parece estar "más cerca", aunque en este caso el sentido de "más cerca" es "más cerca de s", no "más cerca del árbol". Una vez más, los vértices se dividen en las tres categorías (disjuntas) siguientes:

1. vértices *de árbol*: las que están en el árbol construido hasta ahora,
2. vértices *de borde*: los que no están en el árbol, pero están adyacentes a algún vértice del árbol,
3. vértices *no vistos*: todos los demás.

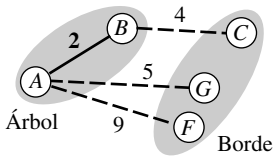
Además, como en el algoritmo de Prim, sólo recordamos una arista candidata (la mejor hallada hasta el momento) por cada vértice de borde. Para cada vértice de borde z hay por lo menos un vértice de árbol v tal que vz es una arista de G. Para cada v semejante hay un camino (único) de s a v en el árbol (podría ser que $s = v$); $d(s, v)$ denota el peso de ese camino. La adición de la arista vz a a este camino da un camino de s a z, y su peso es $d(s, v) + W(vz)$. La arista candidata para z es la arista vz tal que $d(s, v) + W(vz)$ sea mínimo entre todas las opciones de vértice v del árbol que se ha construido hasta ese momento.

Ejemplo 8.4 Crecimiento de un árbol de caminos más cortos

Examinemos el grafo de la figura 8.9(a). Cada arista no dirigida se trata como un par de aristas dirigidas en direcciones opuestas. Supóngase que el vértice de origen es A. Sigamos los pasos del

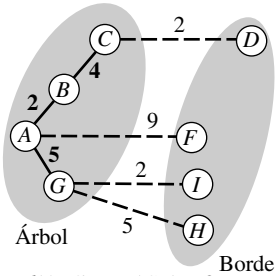


(a) El grafo



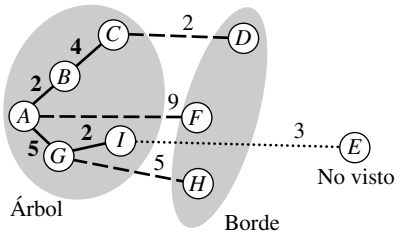
$d(A, B) + W(BC) = 6$
$d(A, A) + W(AG) = 5$
$d(A, A) + W(AF) = 9$
Escoger AG después

(b) Paso intermedio

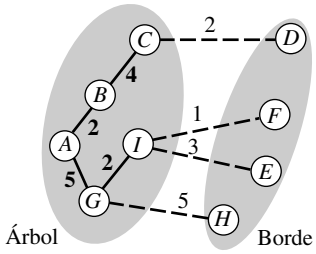


$d(A, C) + W(CD) = 8$
$d(A, A) + W(AF) = 9$
$d(A, G) + W(GI) = 7$
$d(A, G) + W(GH) = 10$
Seleccionar GI después

(c) Paso intermedio: se consideró CH para sustituir a GH como candidata, pero no se escogió



(d) Se seleccionó GI



(e) AF fue sustituida por IF como candidata

Figura 8.9 Ejemplo para el algoritmo de camino más corto de Dijkstra: el problema consiste en hallar el camino más corto de A a H.

"crecimiento" del árbol. En un principio, sólo tiene el vértice A, y $d(A, A) = 0$. Los turistas se bajan del autobús en A en el tiempo 0 y comienzan a caminar por los puentes AB, AG y AF. Al igual que en el ejemplo 7.7, los primeros turistas que llegan a una isla compran todas las gangas. Una vez que un grupo de turistas llegue tarde a una isla, sus integrantes ya no podrán ser los primeros en llegar a ninguna isla futura, porque el grupo anterior se dividió y exploró todos los puentes que salen de la isla. El algoritmo no seguirá la pista a estos grupos.

En la parte (b) de la figura se ha añadido la arista AB porque B es el vértice más cercano al origen A, y $d(A, B) = 2$. Un grupo de turistas llegó a B en el tiempo 2 y se dividió en subgrupos que comienzan a caminar por los puentes BA, BC y BG. Los demás grupos siguen caminando por AG y AF.

Ahora todos los vértices a los que se puede llegar por una arista desde A o desde B están en el borde, a menos que ya estén en el árbol, claro. Para cada vértice de borde, la arista candidata se muestra como línea punteada; obsérvese que BG no es una arista candidata. Sabemos que llegarán turistas a G a más tardar en el tiempo 5 (desde A), así que los que están cruzando el puente BG no van a ser los primeros, por tanto podemos olvidarnos de ellos.

Con base en las aristas de árbol y candidatas, G es el vértice de borde más cercano a A, así que AG será la siguiente arista en añadirse al árbol, y $d(A, G) = 5$. Es decir, llegan turistas a G en el tiempo 5 y se dividen en grupos para explorar GA, GB, GH y GI. El algoritmo sólo sigue la pista a los que se dirigen hacia H (pronóstico de llegada = 10) e I (pronóstico de llegada = 7). Sin embargo, los turistas que están en el puente BC llegan a C en el tiempo 6, así que la siguiente arista añadida será BC, y $d(A, C) = 6$. Salen turistas de C en el tiempo 6 y toman los puentes CB, CD y CH. Podemos olvidarnos de los que tomaron CB porque B ya se visitó, y también de los que tomaron CH porque llegarán a H en el tiempo 11, que es posterior al pronóstico de llegada de los turistas "competidores" que están en GH. Esto nos lleva a la parte (c) de la figura.

Dada la situación de la figura 8.9(c), el siguiente paso consiste en escoger una arista candidata y un vértice de borde. Escogemos como candidata a yz para la cual $d(s, y) + W(yz)$ es mínimo. Éste es el peso del camino que se obtiene juntando yz al camino conocido (y, es de esperar, más corto) de s a y. El vértice z se selecciona entre D, F, I y H, los vértices de borde actuales. En este caso, GI es la arista escogida, y $d(A, I) = 7$. ■

La estructura general del algoritmo de Dijkstra puede describirse así:

```
dijsktraCMCOU(G, n) // BOSQUEJO
    Inicializar todos los vértices como no vistos.
    Iniciar el árbol con el vértice de origen especificado s; reclasificarlo como de árbol;
    definir d(s, s) = 0.
    Reclasificar todos los vértices adyacentes a s como de borde.
    Mientras haya vértices de borde:
        Seleccionar una arista entre un vértice de árbol t y un vértice de borde v tal que
        (d(s, t) + W(tv)) sea mínimo;
        Reclasificar v como de árbol; añadir la arista tv al árbol;
        definir d(s, v) = (d(s, t) + W(tv)).
        Reclasificar todos los vértices no vistos adyacentes a v como de borde.
```

Puesto que la cantidad $d(s, y) + W(yz)$ para una arista candidata yz podría usarse varias veces, se le puede calcular una vez y guardar. Para calcularla de manera eficiente recién que yz se convierte en candidata, también guardamos $d(s, y)$ para cada y del árbol. Así, podríamos usar un arreglo `dist`, a saber:

`dist`$[y] = d(s, y)$ para y en el árbol;

`dist`$[z] = d(s, y) + W(yz)$ para z en el borde, donde yz es la arista candidata a z.

Al igual que en el algoritmo de Prim, una vez que se seleccionan un vértice y la candidata correspondiente, se hace necesario actualizar la información de algunos vértices de borde y hasta entonces no vistos en la estructura de datos.

Ejemplo 8.5 Actualización de información de distancia

En la figura 8.9(d) se acaban de seleccionar el vértice I y la arista GI. La arista candidata para F era AF (con `dist[F]` $= 9$), pero ahora es preciso sustituir AF por IF porque IF ofrece un camino más corto a F. También es preciso recalcular `dist[F]`. Por otra parte, IH no ofrece un camino más corto a H porque actualmente `dist[H]` $= 10$, así que no se volverá a considerar esta arista. El vértice E, que no se había visto, ahora está en el borde porque está adyacente a I, que ahora está en el árbol. La arista IE se convierte en candidata. Estos cambios dan pie a la figura 8.9(e). Es necesario calcular los valores de `dist` para los nuevos vértices de borde. ■

¿Funciona este método? El paso crucial es la selección del siguiente vértice de borde y la arista candidata. Para una candidata yz arbitraria, $d(s, y) + W(yz)$ no es necesariamente la distancia más corta de s a z, porque es posible que los caminos más cortos a z no pasen por y. (En la figura 8.9, por ejemplo, el camino más corto a H no pasa por G, aunque GH es una candidata en las partes c, d y e.) Afirmamos que, si $d(s, y)$ es la distancia más corta para cada vértice de árbol y, y yz se escoge de modo que $d(s, y) + W(yz)$ sea mínimo entre todas las candidatas, yz sí ofrece un camino más corto. Demostraremos esa afirmación en el teorema siguiente.

Teorema 8.6 Sea $G = (V, E, W)$ un grafo ponderado con pesos no negativos. Sea V' un subconjunto de V y sea s un miembro de V'. Supóngase que $d(s, y)$ es la distancia más corta de s a y en G, para cada $y \in V'$. Si se escoge la arista yz de modo que $d(s, y) + W(yz)$ entre todas las aristas que tienen un vértice y en V' y un vértice z en $V - V'$, entonces el camino que consiste en un camino más corto de s a y seguido de la arista yz es el camino más corto de s a z.

Demostración Véase la figura 8.10. Supóngase que se escoge $e = yz$ como se indica, y sea s, $x_1, \ldots, x_r, y$ un camino más corto de s a y (podría ser que $y = s$). Sea $P = s, x_1, \ldots, x_r, y, z$. $W(P) = d(s, y) + W(yz)$. Sea $s, z_1, \ldots, z_a, \ldots, z$ cualquier camino más corto de s a z; llamémoslo P'. Se escoge el vértice z_a de modo que sea el primer vértice de P' que no está en V' (podría ser que $z_a = z$). Debemos demostrar que $W(P) \leq W(P')$. (En el algoritmo, $z_{a-1}z_a$ sería una arista candidata; si $a = 1$, $z_0 = s$.) Por la e escogida,

$$W(P) = d(s, y) + W(e) \leq d(s, z_{a-1}) + W(z_{a-1}za). \tag{8.2}$$

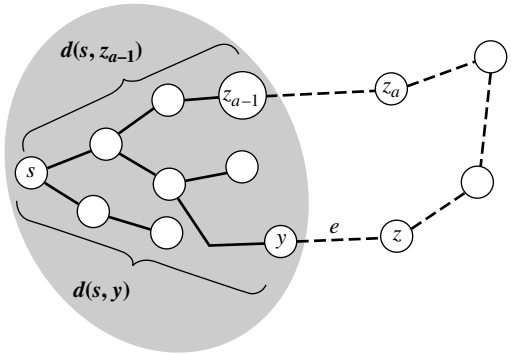


Figura 8.10 Para la demostración del teorema 8.6

Por el lema 8.5, $s, z_1, \ldots, z_{a-1}$ es un camino más corto de s a z_{a-1}, así que el peso de este camino es $d(s, z_{a-1})$. Puesto que $s, z_1, \ldots, z_{a-1}, z_a$ forma parte del camino P' y cualesquier aristas restantes deben tener peso no negativo,

$$d(s, z_{a-1}) + W(z_{a-1}z_a) \leq W(P'). \tag{8.3}$$

Combinando las ecuaciones (8.2) y (8.3), $W(P) \leq W(P')$. □

Teorema 8.7 Dado un grafo ponderado dirigido G con pesos no negativos y un vértice de origen s, el algoritmo de Dijkstra calcula la distancia más corta (peso de un camino de peso mínimo) de s a cada uno de los vértices de G a los que se puede llegar desde s.

Demostración La demostración es por inducción con el orden en que se añaden vértices al árbol de caminos más cortos. Los detalles se dejan para el ejercicio 8.16. □

8.3.3 Implementación

El algoritmo de camino más corto puede usar exactamente el mismo TDA de cola de prioridad que el algoritmo de Prim; véanse las figuras 8.6 a 8.8. Cuando el algoritmo termina, el arreglo `padre` implica las aristas del árbol de caminos más cortos. Es decir, para cada vértice v distinto del vértice de origen, (`v, padre[v]`) es una arista del árbol de caminos más cortos y `pesoBorde[v]` es la distancia de s a v. Si no es posible llegar a todos los vértices desde el origen dado, s, `padre[v]` y `pesoBorde[v]` no estarán definidos para los v a los que no se pueda llegar desde s. Es fácil ajustar el algoritmo para asignar a esos elementos valores especiales, como $n + 1$ e ∞.

Algoritmo 8.2 Caminos más cortos de origen único (de Dijkstra)

Entradas: Un arreglo `infoAdya` de listas de adyacencia que representan un grafo ponderado, dirigido o no dirigido, $G = (V, E, W)$, según la descripción de la sección 7.2.3; n, el número de vértices; y s, el vértice inicial deseado. Todos los arreglos deben estar definidos para los índices $1, \ldots, n$; el elemento con índice 0 no se usa. Los elementos de `infoAdya` son listas del TDA `ListaAristas`, que se describe más adelante.

Salidas: Un árbol de caminos más cortos, almacenado en el arreglo `padre` como árbol adentro, y el arreglo `pesoBorde` que contiene, para cada vértice v, la distancia más corta de s a v. (El padre de la raíz es -1.) El invocador reserva espacio para los arreglo y los pasa como parámetros, y el algoritmo se encarga de llenarlos.

Comentarios: El arreglo `situacion[1]`, ..., `situacion[n]` denota la situación de búsqueda actual de todos los vértices. Los vértices no descubiertos son `novisto`; los que ya se descubrieron pero aún no se procesan (en la cola de prioridad) son `deborde`; los que ya se procesaron son `dearbol`. Las listas de adyacencia son del tipo `ListaAristas` y tienen las operaciones estándar del TDA Lista (sección 2.3.2). Los elementos pertenecen a la clase organizadora `InfoArista` y tienen dos campos, `a` y `peso`.

```
void caminosMasCortos(ListaAristas[] infoAdya, int n, int s, int[]
   padre, float[] pesoBorde)
   int[] situacion = new int[n+1];
   CPMin cp = crear(n, situacion, padre, pesoBorde);

   insertar(cp, s, -1, 0);
   while (estaVacia(cp) == false)
     int v = obtenerMin(cp);
     borrarMin(cp);
     actualizarBorde(cp, infoAdya[v], v);
   return;

/** Ver si se encuentra una mejor conexión con cualquier vértice de
  * la lista infoAdyaDeV, y decrementarClave en tal caso.
  * Si la conexión es nueva, insertar el vértice. */
void actualizarBorde(CPMin cp, ListaAristas infoAdyaDeV, int v)
   float miDist = cp.pesoBorde[v];
   ListaAristas adyaRest;
   adyaRest = infoAdyaDeV;
   while (adyaRest ≠ nil)
     InfoArista infoW = primero(adyaRest);
     int w = infoW.a;
     float nuevaDist = miDist + infoW.peso;
     if (cp.situacion[w] == novisto)
       insertar(cp, w, v, nuevaDist);
     else if (cp.situacion[w] == deborde)
       if (nuevaDist < obtenerPrioridad(cp, w))
         decrementarClave(cp, w, v, nuevaDist);
     adyaRest = resto(adyaRest);
   return;
```

Análisis

El análisis efectuado en la sección 8.2.7 del algoritmo de árbol abarcante mínimo de Prim, algoritmo 8.1, también es válido para el algoritmo de caminos más cortos de Dijkstra, algoritmo 8.2, sin que sea necesario modificarlo. El algoritmo de Dijkstra también se ejecuta en un tiempo $\Theta(n^2)$ en el peor caso. También son válidas la cota inferior de $\Omega(m)$ y las necesidades de espacio de $\Theta(n)$.

Si se espera no poder llegar a un número apreciable de vértices, podría ser más eficiente incluir, como paso de procesamiento previo, una prueba de "asequibilidad", eliminar los vértices a los que no se puede llegar y renumerar los vértices restantes como 1, ..., n_r. El costo total estaría en $\Theta(m + n_r^2)$, en vez de $\Theta(n^2)$.

Es posible usar el bosque de apareamiento (sección 6.7.2), el montón de apareamiento o el montón de Fibonacci para implementar la cola de prioridad en el algoritmo de Dijkstra, de forma análoga a la que se describió en la sección 8.2.8 para el algoritmo de Prim. Las cotas asintóticas son las mismas: el uso de un montón de Fibonacci da el orden asintótico óptimo de $\Theta(m + n \log n)$, pero presenta dificultades prácticas.

8.4 Algoritmo de árbol abarcante mínimo de Kruskal

Sea $G = (V, E, W)$ un grafo ponderado no dirigido. En la sección 8.2 estudiamos el algoritmo de Prim para hallar un árbol abarcante mínimo para G (con la condición de que G estuviera conectado). El algoritmo iniciaba en un vértice arbitrario y se ramificaba desde él escogiendo "codiciosamente" aristas de peso bajo. En cualquier momento, las aristas escogidas formaban un árbol. Aquí examinaremos un algoritmo que usa una estrategia más codiciosa aún. En toda esta sección, los grafos son grafos no dirigidos.

8.4.1 El algoritmo

El bosquejo general del algoritmo de Kruskal es el siguiente. En cada paso se escoge la arista restante de peso más bajo de cualquier punto del grafo, aunque se desecha cualquier arista que formaría un ciclo con las que ya se escogieron. En cualquier momento, las aristas escogidas hasta entonces forman un bosque, pero no necesariamente un árbol. El algoritmo termina cuando se han procesado todas las aristas.

```
kruskalMST(G, n) // BOSQUEJO
    R = E;   // R es las aristas restantes.
    F = ∅;   // F es las aristas de bosque.
    while (R no está vacío)
        Quitar la arista más ligera (más corta), vw, de R;
        if (vw no forma un ciclo en F)
            Añadir vw a F;
    return F;
```

Antes de siquiera pensar en cómo implementar esta idea, debemos preguntarnos si funciona. Puesto que el grafo podría no estar conectado, necesitamos primero una definición.

Definición 8.4 Colección de árboles abarcantes

Sea $G = (V, E, W)$ un grafo ponderado no dirigido. Una *colección de árboles abarcantes* para G es un conjunto de árboles, uno por cada componente conectado de G, tal que cada árbol es un árbol abarcante para su componente conectado. Una *colección de árboles abarcantes mínima* es una colección de árboles abarcantes cuyas aristas tienen un peso total mínimo, es decir, una colección de árboles abarcantes mínimos. ■

En primer lugar, ¿cada uno de los vértices de G está representado en algún árbol? (Podría haber varios árboles si el grafo no está conectado.) Sea v un vértice arbitrario de G. Si por lo menos una arista incide en v, la primer arista que se saque de R y que incida en v se incluirá en F. Por otra parte, si v es un vértice aislado (sin aristas incidentes), *no* estará representado en F y se le tendrá que considerar por separado para no pasarlo por alto.

La siguiente pregunta es si el algoritmo crea o no una colección de árboles abarcantes, suponiendo que G no tiene vértices aislados. Es decir, ¿hay exactamente un árbol en F para cada uno de los componentes conectados de G? El lema que sigue nos ayuda a entender esta pregunta. La demostración es fácil y se deja como ejercicio.

Lema 8.8 Sea F un bosque; es decir, cualquier grafo acíclico no dirigido. Sea $e = vw$ una arista que no está en F. Existe un ciclo que consta de e y aristas de F si y sólo si v y w están en el mismo componente conectado de F. □

Supóngase ahora que algún componente conectado de G corresponde a dos o más árboles en el bosque F que el algoritmo de Kruskal calcula. Deberá haber alguna arista en G que conecte dos de esos árboles, llamémosla vw; es decir, v y w están en diferentes componentes conectados de F. Por tanto, cuando el algoritmo procesó vw, debió haber formado un ciclo en el bosque en ese momento (llamémosle F') porque vw no se añadió a F'. Por el lema 8.8, v y w estaban en el mismo componente conectado de F en ese momento. Pero entonces es imposible que v y w estén en diferentes componentes conectados de F cuando el algoritmo termina. Por tanto, F sólo puede contener un árbol por cada componente conectado de G.

Habiendo determinado que el algoritmo calcula *alguna* colección de árboles abarcantes, la última pregunta en materia de corrección es si los árboles tienen peso mínimo o no. Esta pregunta se contesta con el teorema siguiente, cuya demostración se deja como ejercicio.

Teorema 8.9 Sea $G = (V, E, W)$ un grafo ponderado no dirigido. Sea $F \subseteq E$. Si F está contenido en una colección de árboles abarcantes mínima para G, y si e es una arista de peso mínimo en $E - F$ tal que $F \cup \{e\}$ no tiene ciclos, entonces $F \cup \{e\}$ está contenido en una colección de árboles abarcantes mínima para G. □

El algoritmo inicia con $F = \varnothing$ y agrega aristas a F hasta que se han procesado todas las aristas. El teorema garantiza que F siempre está contenido dentro de alguna colección de árboles abarcantes mínima, y ya nos dimos cuenta de que el valor final de F es una colección de árboles abarcantes para G, con la excepción de los árboles triviales que consisten en nodos aislados sin aristas.

Ya estamos en condiciones de considerar métodos de implementación. Para acceder a las aristas en orden de peso creciente, usamos una cola de prioridad minimizante (sección 2.5.1), como un montón (sección 4.8.1). Las aristas de F se pueden almacenar en una lista, pila u otra estructura de datos conveniente.

Un problema que es preciso resolver es cómo determinar si una arista formará un ciclo con otras que ya están en F. El lema 8.8 proporciona el criterio: si v y w están en el mismo componente conectado de F, entonces (y sólo en este caso) la adición de la arista vw a F creará un ciclo. Por ello, nos conviene ir recordando los componentes conectados de F conforme se construye. En particular, dados dos vértices v y w, queremos poder determinar de manera eficiente si están o no

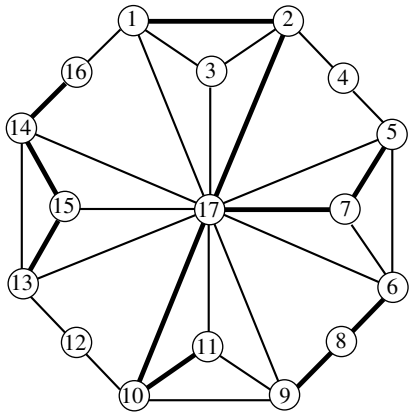


Figura 8.11 Las aristas gruesas están en el subgrafo F. Las clases de equivalencia son {1, 2, 5, 7, 10, 11, 17}, {6, 8, 9}, {13, 14, 15, 16}, {3}, {4} y {12}.

en el mismo componente conectado de F. Podemos aplicar la metodología de relaciones de equivalencia dinámicas que desarrollamos en la sección 6.6.

Definimos una relación, "$\equiv$", entre los vértices de un subgrafo F como $v \equiv w$ si y sólo si v y w están en el mismo componente conectado de F. Es fácil verificar que $\equiv$ es una relación de equivalencia. (Véase un ejemplo en la figura 8.11.) Así pues, por el lema 8.8, el algoritmo de Kruskal escoge una arista vw si y sólo si $v \not\equiv w$. En un principio, todos los vértices de G están en la relación $\equiv$ como clases de equivalencia individuales, y F es un grafo que consiste en todos los vértices de G, pero ninguna arista (con esto también damos cuenta de los vértices aislados). Cada vez que se escoge una arista, el subgrafo F y la relación de equivalencia $\equiv$ cambian; cada arista nueva hace que dos componentes conectados, o dos clases de equivalencia, se fusionen en uno solo.

El mantenimiento y consulta de la relación $\equiv$ se efectúan mediante el TDA Unión-Hallar. Recordemos que `hallar`(v) devuelve el identificador único de la clase de equivalencia del vértice v, y que si s y t son identificadores de clases de equivalencia distintas, `union`(s, t) las fusiona.

Algoritmo 8.3 Árbol abarcante mínimo (de Kruskal)

Entradas: $G = (V, E, W)$, un grafo ponderado, con $|V| = n$, $|E| = m$.

Salidas: F, un subconjunto de E que forma un árbol abarcante mínimo para G, o una colección de árboles abarcantes mínima si G no está conectado.

Comentarios: La estructura `conjuntos` definida en el algoritmo corresponde a la relación de equivalencia $\equiv$ de la explicación. Omitimos los calificadores de nombre de clase en las operaciones del TDA Unión-Hallar y del TDA de cola de prioridad para hacerlas más comprensibles.

```
kruskalMST(G, n, F) // BOSQUEJO
    int cuenta;
    Construir una cola de prioridad minimizante, cp, de aristas de G, en la que las prioridades son los pesos.
    Inicializar una estructura Unión-Hallar, conjuntos, en la que cada vértice de G está en su propio conjunto.
    F = ∅;
    while (estaVacia(cp) == false)
      aristaVW = obtenerMin(cp);
      borrarMin(cp);
      int conjuntoV = hallar(conjuntos, aristaVW.de);
      int conjuntoW = hallar(conjuntos, aristaVW.a);
      if (conjuntoV ≠ conjuntoW)
        Añadir aristaVW a F;
        union(conjuntos, conjuntoV, conjuntoW);
    return;
```

8.4.2 Análisis

La cola de prioridad de aristas se puede implementar de manera eficiente con un montón, pues en este algoritmo no se usa la operación `decrementarClave`. El montón se puede construir en tiempo $\Theta(m)$. La eliminación de todas las aristas requiere un tiempo $\Theta(m \log m)$ en el peor caso, pero podría ser $\Theta(n \log m)$, que equivale a $\Theta(n \log n)$ si sólo es preciso procesar $O(n)$ aristas más ligeras para construir la colección de árboles abarcantes.

Como optimización adicional, si sabemos que el número de componentes conectados de G es `ncc`, sabremos que el número de aristas en la colección de árboles abarcantes es $n - \texttt{ncc}$, y el algoritmo podrá terminar tan pronto como haya añadido esa cantidad de aristas a F, que ahora es la colección de árboles abarcantes mínima, sin procesar las aristas restantes. La determinación del número de componentes conectados se puede efectuar en tiempo lineal. Esto haría posible aprovechar el caso favorable que mencionamos en el párrafo anterior.

En cuanto a las operaciones de Unión-Hallar, `hallar` se podría invocar aproximadamente $2m$ veces, mientras que `union` se invoca cuando más $n - 1$ veces. Así pues, el número total de operaciones Unión-Hallar efectuadas está acotado por $(2m + n)$. Supóngase $m \geq n$, que es lo normal, para simplificar las expresiones. Con la implementación de unión ponderada y compresión de caminos de la sección 6.6, el tiempo total de estas operaciones está en $O(m \lg^*(m))$, donde $\lg^*$ es la función de crecimiento muy lento de la definición 6.9.

Así pues, el tiempo de ejecución de peor caso del algoritmo MST de Kruskal está en $\Theta(m \log m)$. El algoritmo de Prim, algoritmo 8.1, está en $\Theta(n^2)$ en el peor caso. Cuál sea el mejor dependerá de los tamaños relativos de n y m. En el caso de grafos densos, el algoritmo de Prim es mejor. Con grafos ralos, el de Kruskal es más rápido que el algoritmo 8.1. No obstante, considérese la alternativa del ejercicio 8.9 y el hecho de que el algoritmo de Prim puede usar las estructuras de datos que vimos en la sección 8.2.8.

Si las aristas de G ya estuvieran ordenadas, se podría usar una cola de prioridad trivial y cada arista se podría borrar en tiempo $O(1)$, en cuyo caso el algoritmo de Kruskal se ejecutaría en tiempo $O(m \lg^*(m))$, que es muy bueno.

Ejercicios

Sección 8.2 Algoritmo de árbol abarcante mínimo de Prim

8.1 Dé un grafo no dirigido, ponderado, conectado y un vértice de inicio tales que ni el árbol de búsqueda primero en profundidad ni el árbol de búsqueda primero en amplitud sea un MST, sin importar cómo estén ordenadas las listas de adyacencia.

8.2 Sea T cualquier árbol abarcante de un grafo no dirigido G. Suponga que uv es cualquier arista de G que no está en T. Las demostraciones siguientes son fáciles si se usan las definiciones de árbol no dirigido, árbol abarcante y ciclo.

a. Sea G_1 el subgrafo que resulta de añadir uv a T. Demuestre que G_1 tiene un ciclo en el que participa uv, digamos $(w_1, w_2, \ldots, w_p, w_1)$, donde $p \geq 3$, $u = w_1$ y $v = w_p$.

b. Suponga que una arista cualquiera, $w_i w_{i+1}$ se elimina del ciclo creado en la parte (a), creando un subgrafo G_2 (que depende de i). Demuestre que G_2 es un árbol abarcante para G.

8.3 Suponga que T_1 y T_2 son árboles abarcantes mínimos distintos para el grafo G. Sea uv la arista más ligera que está en T_2 pero no está en T_1. Sea *xy cualquier* arista que está en T_1 pero no está en T_2. Demuestre que $W(xy) \geq W(uv)$.

8.4 Demuestre que si todos los pesos de las aristas de un grafo conectado no dirigido son distintos, sólo existe un árbol abarcante mínimo.

8.5

a. Describa una familia de grafos conectados, ponderados, no dirigidos G_n, para $n \geq 1$, tal que G_n tiene n vértices y el tiempo de ejecución del algoritmo MST de Prim (algoritmo 8.1) para G_n es lineal en n.

b. Describa una familia de grafos conectados, ponderados, no dirigidos G_n tal que G_n tiene n vértices y el algoritmo MST de Prim no efectúa comparaciones de pesos cuando G_n es la entrada. Una comparación de un peso con ∞ (`cp.oo`) no cuenta para este fin (porque el procedimiento `hallarMin` puede verificar si `verticeMin` es -1 para evitar dicha comparación). (El algoritmo requerirá un tiempo por lo menos proporcional a n porque debe hallar un árbol abarcante mínimo.)

8.6 Ejecute el algoritmo de árbol abarcante mínimo de Prim manualmente con el grafo de la figura 8.4(a), mostrando cómo evolucionan las estructuras de datos. Indique claramente cuáles aristas pasan a formar parte del árbol abarcante mínimo y en qué orden lo hacen.

a. Inicie en el vértice G.

b. Inicie en el vértice H.

c. Inicie en el vértice I.

8.7 Sea $G = (V, E, W)$ donde $V = \{v_1, v_2, \ldots, v_n\}$, $E = \{v_1 v_i \mid i = 2, \ldots, n\}$, y para $i = 2, \ldots, n$, $W(v_1 v_i) = 1$. Con este G como entrada y v_1 como vértice inicial, ¿cuántas comparaciones de pe-

sos de aristas efectuará el algoritmo MST de Prim, en total, para hallar aristas candidatas mínimas? (Resolver este problema podría sugerirle que guardar información acerca del ordenamiento de los pesos de las aristas candidatas podría reducir el número de comparaciones. Los dos ejercicios siguientes sugieren que quizá no sea fácil.)

8.8

a. ¿Cuántas comparaciones de pesos de aristas efectuará el algoritmo MST, en total, si la entrada es un grafo no dirigido completo con n vértices y v_1 es la arista inicial?

b. Suponga que los vértices son $v_1, \ldots, v_n$, y $W(v_i v_j) = n + 1 - i$ para $1 \leq i < j \leq n$. ¿Cuántas de las aristas son candidatas en algún momento durante la ejecución del algoritmo?

★ **8.9** Considere el almacenamiento de aristas candidatas en un montón minimizante (un montón en el que cada nodo es más pequeño que sus hijos, véase la sección 4.8.1). En este ejercicio evaluaremos el algoritmo MST de Prim bajo este supuesto, para grafos en general y para ciertas clases restringidas de grafos. Usaremos $|V| = n$ y $|E| = m$.

a. Determine el orden asintótico del número de comparaciones de pesos de aristas que efectuará el algoritmo MST de Prim, con base en la ecuación (8.1) en el peor caso. No olvide considerar el trabajo necesario para la operación `decrementarClave` cuando una arista candidata es sustituida por otra.

b. Una *familia de grafos de grado acotado* es cualquier familia para la cual existe una constante k tal que ningún vértice de cualquier grafo de la familia tiene grado mayor que k. Determine el orden asintótico, en función de n, del número de comparaciones de pesos de aristas que efectuaría el algoritmo MST de Prim con una familia de grado acotado.

★ **c.** Un *grafo plano* es un grafo conectado que se puede dibujar en un plano sin que haya cruces de aristas. Para esta clase, el teorema de Euler dice que $|V| - |E| + |F| = 2$, donde $|F|$ es el número de caras (regiones rodeadas por aristas, más una región desde las aristas externas hasta el infinito) que se forman al dibujar el grafo. Por ejemplo, si el grafo es un triángulo simple, tiene dos caras: una adentro del triángulo y otra fuera. La cara exterior se extiende hasta el infinito en todas direcciones. Determine el orden asintótico, en función de n, del número de comparaciones de pesos de aristas que efectuaría el algoritmo MST de Prim con un grafo plano. *Sugerencia:* Observe que todas las aristas están en dos caras.

8.10 El uso de la cola de prioridad se puede simplificar si se introducen inicialmente todos los vértices con `pesoBorde` = "∞" y `situacion` = `deborde`. El valor "∞" sólo tiene que ser mayor que el costo de cualquier arista que esté en el grafo; no tiene que representar realmente "infinito". Entonces `decrementarClave` reducirá el costo de un vértice por debajo de "∞" cuando se halle la primera conexión del vértice y no se necesitará `insertar`. Muestre las modificaciones del algoritmo 8.1 y las operaciones de cola de prioridad que implementan esta estrategia. ¿Mejora el orden asintótico?

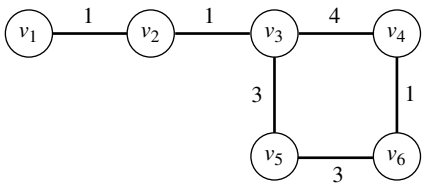


Figura 8.12 Grafo para el ejercicio 8.13

8.11 Supóngase que queremos usar el algoritmo de Prim con un grafo no dirigido ponderado del cual no se sabe si está conectado. Muestre cómo modificar el algoritmo de Prim para hallar una colección de árboles abarcantes mínima (definición 8.4) sin hallar primero los componentes conectados. Trate de no elevar el orden asintótico del algoritmo.

Sección 8.3 Caminos más cortos de origen único

8.12 Dé un grafo dirigido ponderado y un vértice de origen tal que ni el árbol de búsqueda primero en profundidad ni el árbol de búsqueda primero en amplitud sea un árbol de caminos más cortos, sin importar cómo estén ordenadas las listas de adyacencia.

8.13 Para el grafo de la figura 8.12, indique cuáles aristas estarían en el árbol abarcante mínimo construido por el algoritmo MST de Prim (algoritmo 8.1) y cuáles estarían en el árbol construido por el algoritmo de caminos más cortos de Dijkstra (algoritmo 8.2) empleando v_1 como origen.

8.14 ¿El algoritmo de caminos más cortos de Dijkstra (algoritmo 8.2) funciona correctamente si los pesos pueden ser negativos? Justifique su respuesta con un argumento o un contraejemplo.

8.15 He aquí las listas de adyacencia (con los pesos de las aristas entre paréntesis) para un grafo dirigido. Como ayuda, el grafo se muestra también en la figura 8.13.

A: *B*(4, 0), *F*(2.0)
B: *A*(1.0), *C*(3.0), *D*(4.0)
C: *A*(6.0), *B*(3.0), *D*(7.0)
D: *A*(6.0), *E*(2.0)
E: *D*(5.0)
F: *D*(2.0), *E*(3.0)

a. Este grafo dirigido tiene *tres* caminos más cortos de *C* a *E* (es decir, todos tienen el *mismo* peso total). Hállelos. (Enumere la sucesión de vértices de cada camino.)

b. ¿Cuál de estos caminos es el que hallaría el algoritmo de caminos más cortos de Dijkstra con $s = C$? (Dé una explicación convincente o muestre los pasos principales del algoritmo.)

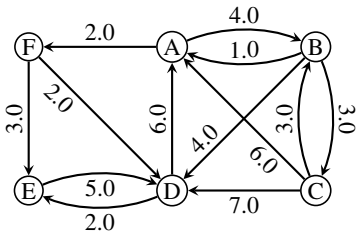


Figura 8.13 Digrafo para el ejercicio 8.15

c. Ejecute el algoritmo de caminos más cortos de Dijkstra manualmente con este grafo, mostrando cómo evolucionan las estructuras de datos, con $s = A$. Indique claramente cuáles aristas pasan a formar parte del árbol de caminos más cortos y en qué orden lo hacen.

d. Repita la parte (c) con $s = B$.

e. Repita la parte (c) con $s = F$.

8.16 Complete la demostración del teorema 8.7.

8.17 Explique cómo hallar un camino más corto real entre s y un vértice dado z utilizando el arreglo `padre` que el algoritmo de caminos más cortos de Dijkstra llena.

⋆ **8.18** Sea $G = (V, E)$ un grafo, y sean s y z vértices distintos. Como sugiere el ejercicio 8.15, puede haber más de un camino más corto entre s y z. Explique cómo modificaría el algoritmo de caminos más cortos de Dijkstra para determinar cuántos caminos más cortos distintos hay entre s y z.

8.19 Considere el problema de hallar sólo la distancia, pero no un camino más corto, desde s hasta un vértice dado z en un grafo ponderado. Bosqueje una versión modificada del algoritmo de caminos más cortos de Dijkstra para hacer esto tratando de eliminar la mayor cantidad de trabajo y de consumo de espacio extra posible. Indique cómo modificaría, si acaso, la estructura de datos empleada por el algoritmo, e indique qué trabajo o espacio eliminaría.

8.20 Algunos algoritmos para grafos se escriben bajo el supuesto de que la entrada siempre es un grafo completo (en el que una arista tiene peso ∞ o 0 para indicar su ausencia del grafo para el cual el usuario realmente quiere resolver el problema). Tales algoritmos suelen ser más cortos y "aseados" porque hay menos casos que considerar. En los algoritmos de las secciones 8.2 y 8.3, por ejemplo, no habría vértices no vistos porque todos los vértices estarían adyacentes a vértices del árbol construido hasta ese momento.

a. Con el objetivo de simplificar lo más posible, reescriba el algoritmo de caminos más cortos de Dijkstra bajo el supuesto de que $G = (V, E, W)$ es un grafo completo y que $W(uv)$ podría ser ∞. Describa cualesquier modificaciones que haría a las estructuras de datos empleadas.

b. Compare su algoritmo y estructuras de datos con las del texto, utilizando el criterio de sencillez, tiempo (peor caso y otros casos) y consumo de espacio (en el caso de grafos con muchas aristas de peso ∞ y grafos con pocas de esas aristas).

★ **8.21** Considere este enfoque general para calcular caminos más cortos desde el vértice s en un grafo ponderado $G = (V, E, W)$. Se forma G^T, el grafo transpuesto (véase la definición 7.10). Se define un arreglo llamado `llegar` con índices $1, \ldots, n$ y todos sus elementos inicializados con ∞. Se efectúa una búsqueda primero en profundidad completa de G^T y se calculan valores para `llegar[v]` según este esquema:

1. `llegar[s] = 0`;
2. Al retroceder de w a v, se calcula `llegar[w]` + $W(wv)$ y se compara con el valor previamente almacenado en `llegar[v]`. Si se halla un valor menor, se guarda como `llegar[v]`.

La intención es que `llegar[v]` sea la distancia de camino más corto entre s y v cuando termine la búsqueda primero en profundidad.

a. Complete el bosquejo anterior insertando enunciados en el esqueleto de búsqueda primero en profundidad para grafos dirigidos, algoritmo 7.3.

b. ¿El algoritmo halla caminos más cortos en todos los casos? Demuestre que lo hace o halle un contraejemplo.

c. ¿Con qué clase (muy conocida) de grafos el algoritmo halla caminos más cortos en todos los casos? Demuestre su respuesta. *Sugerencia:* ¿Qué restricción del grafo permitiría efectuar con éxito la demostración del inciso (b)?

Sección 8.4 Algoritmo de árbol abarcante mínimo de Kruskal

8.22 Demuestre el lema 8.8.

8.23 Demuestre el teorema 8.9. *Sugerencia:* Use la propiedad MST (definición 8.2).

8.24 Halle el árbol abarcante mínimo que el algoritmo de Kruskal (algoritmo 8.3) produciría para el grafo de la figura 8.14, suponiendo que las aristas están ordenadas como se muestra.

Problemas adicionales

8.25 En este ejercicio el lector desarrollará un esqueleto de *búsqueda de primero el mejor*, análogo al esqueleto de búsqueda primero en amplitud del capítulo 7.

a. Considere si usará la estrategia del ejercicio 8.20 (nada de inserciones, crear la cola de prioridad con todos los elementos presentes y pesos infinitos en caso necesario) o el TDA `CPMin` dado en las figuras 8.6 a 8.8 como base para su esqueleto. ¿Cuál ofrece más generalidad? ¿Cuál es probable que sea más eficiente? ¿Son más o menos iguales los dos?

b. Escriba el esqueleto con la estrategia elegida.

c. Muestre cómo modificar su esqueleto (insertando unos cuantos enunciados en ciertos puntos) para producir el algoritmo de Prim, y luego para producir el algoritmo de Dijkstra.

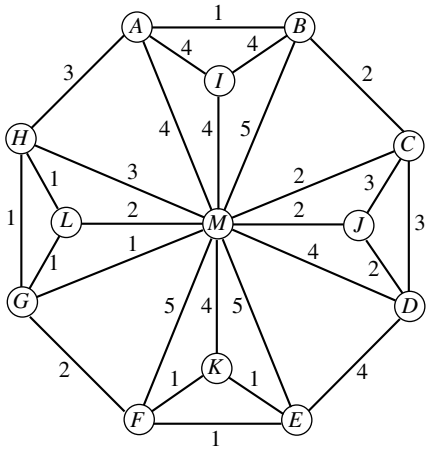


Figura 8.14 Aristas ordenadas: *AB, EF, EK, FK, GH, GL, GM, HL, BC, CM, DJ, FG, JM, LM, AH, CD, CJ, HM, AI, AM, BI, DE, DM, IM, KM, BM, EM, FM.*

d. Comente la interfaz de bosques de apareamiento con su esqueleto. ¿La interfaz es independiente de las aplicaciones con las que podría usarse su esqueleto o tendría que modificarse dependiendo de la aplicación?

8.26 Suponga que quiere hallar un camino más corto de s a w en un grafo G en el que la longitud de un camino es simplemente el número de aristas del camino (por ejemplo, para planear un viaje en avión con el mínimo de escalas). ¿Cuál de los algoritmos o estrategias de recorrido de este capítulo o del capítulo 7 *podría* usar? ¿Cuál *usaría*, y por qué?

8.27 Suponga que necesita determinar si un grafo grande está conectado o no. El grafo tiene n vértices, $V = \{1, \ldots, n\}$ y m aristas, donde m es mucho mayor que n. La entrada consistirá en los enteros n y m y una sucesión de aristas (pares de vértices). No hay suficiente espacio para almacenar todo el grafo; se dispone de cn unidades de espacio, donde c es una constante pequeña, pero no se puede usar espacio proporcional a m. Así pues, es posible procesar cada arista cuando se lee, pero no guardar las aristas ni volverlas a leer. Describa un algoritmo para resolver el problema. ¿Cuánto tiempo tarda su algoritmo en el peor caso?

Programas

Cada uno de los siguientes programas de tarea requiere un procedimiento que lee una descripción de un grafo con aristas ponderadas y establece las listas de adyacencia. Basta una modificación menor del procedimiento de carga de grafos del capítulo 7. Suponga que la entrada contiene el número de vértices seguido de una sucesión de líneas, cada una de las cuales contiene un par de no-

dos que representan una arista y un tercer número que representa su peso. Escriba este procedimiento de modo que, con cambios pequeños, se pueda usar con cualquiera de los problemas.

Se deben escoger datos de prueba que permitan probar todos los aspectos del programa. Incluya algunos de los ejemplos del texto.

1. Algoritmo de árbol abarcante mínimo de Prim, algoritmo 8.1. El programa debe completar el algoritmo, construyendo una estructura de datos para registrar el árbol abarcante mínimo hallado. La salida debe provenir de un procedimiento aparte y debe incluir el grafo, el conjunto de aristas del árbol, junto con sus pesos, y el peso total del árbol.
2. Algoritmo de árbol abarcante mínimo de Kruskal, algoritmo 8.3. El programa debe completar el algoritmo, construyendo una estructura de datos para registrar el árbol abarcante mínimo hallado. La salida debe provenir de un procedimiento aparte y debe incluir el grafo, el conjunto de aristas del árbol, sus pesos y el peso total del árbol.
3. Algoritmo de camino más corto de Dijkstra, algoritmo 8.2. El programa debe completar el algoritmo, construyendo una estructura de datos para registrar los caminos más cortos hallados. La salida debe provenir de un procedimiento aparte y debe incluir el grafo (o digrafo), el vértice de origen, cada uno de los vértices a los que se puede llegar desde el origen, junto con las aristas del camino más corto hallado a ese vértice, sus pesos y el peso total del camino.
4. Después de escribir el programa 1 o el programa 3, modifíquelo para implementar la cola de prioridad con bosques de apareamiento. Efectúe pruebas de tiempo con algunos grafos grandes y compare los tiempos antes y después de las modificaciones.

Notas y referencias

El primer algoritmo de árbol abarcante mínimo se debe a Prim (1957). El algoritmo de camino más corto de origen único se debe a Dijkstra (1959), pero ese artículo no trata la implementación. Dijkstra (1959) también describe un algoritmo de árbol abarcante mínimo parecido al de Prim. La terminología para clasificar los vértices en las secciones 8.2 y 8.3 (por ejemplo, *vértice de borde*) se tomó de Sedgewick (1988). En Notas y referencias del capítulo 6 se mencionan alternativas para implementar colas de prioridad, como bosques de apareamiento, montones de apareamiento y montones de Fibonacci. La cota superior para los algoritmos de Prim y Dijkstra, utilizando montones de apareamiento, con grafos para los que $m = \Theta(n^{1+c})$, se tomó de Fredman (1999).

En algunas aplicaciones es necesario hallar un árbol abarcante con peso mínimo entre los que satisfacen otros criterios requeridos por el problema, así que es útil tener un algoritmo que genere árboles abarcantes en orden según su peso para poder determinar si cada uno satisface los otros criterios. Gabow (1977) presenta algoritmos que hacen esto.

La estrategia de Kruskal para hallar árboles abarcantes mínimos se tomó de Kruskal (1956). La implementación empleando programas de equivalencia al parecer era folklore; se menciona en Hopcroft y Ullman (1973), quienes informan que M. D. McIlroy y R. Morris llevaron a cabo tal implementación. Gran parte del material de esta sección, junto con aplicaciones adicionales y extensiones, aparece en Aho, Hopcroft y Ullman (1974). El lector puede hallar otros algoritmos de caminos más cortos, incluidos algunos que no requieren pesos de arista no negativos, en Cormen, Leiserson y Rivest (1990).

En la sección 7.2 presentamos varias preguntas que podrían hacerse acerca de los grafos y digrafos. Una que no contestamos en este libro es: ¿Qué cantidad de un producto puede fluir de un vértice a otro dadas capacidades de las aristas? Se trata del problema de flujo por red; tiene muy diversas soluciones y aplicaciones. Los lectores interesados pueden consultar Even (1979), Ford y Fulkerson (1962), Tarjan (1983), Wilf (1986) y Cormen, Leiserson y Rivest (1990).