# 7 Grafos no dirigidos

Un grafo no dirigido $G = (V, A)$ consta de un conjunto finito de vértices $V$ y de un conjunto de aristas $A$. Se diferencia de un grafo dirigido en que cada arista en $A$ es un par no ordenado de vértices †. Si $(v, w)$ es una arista no dirigida, entonces $(v, w) = (w, v)$. De ahora en adelante se hará referencia a los grafos no dirigidos tan sólo como grafos.

Los grafos se emplean en distintas disciplinas para modelar relaciones simétricas entre objetos. Los objetos se representan por los vértices del grafo, y dos objetos están conectados por una arista si están relacionados entre sí. En este capítulo se presentan varias estructuras de datos que pueden usarse para representar grafos, y los algoritmos para tres problemas comunes que se relacionan con grafos no dirigidos: construcción de árboles abarcadores minimales, componentes biconexos y comparaciones maximales.

## 7.1 Definiciones

Buena parte de la terminología para grafos dirigidos es aplicable también a los no dirigidos. Por ejemplo, los vértices $v$ y $w$ son *adyacentes* si $(v, w)$ es una arista [o, en forma equivalente, si $(w, v)$ lo es]. Se dice que la arista $(v, w)$ es *incidente* sobre los vértices $v$ y $w$.

Un *camino* es una secuencia de vértices $v_1, v_2, \ldots, v_n$ tal que $(v_i, v_{i+1})$ es una arista para $1 \leq i < n$. Un camino es *simple* si todos sus vértices son distintos, con excepción de $v_1$ y $v_n$, que pueden ser el mismo. La longitud del camino es $n - 1$, el número de aristas a lo largo del camino. Se dice que el camino $v_1, v_2, \ldots, v_n$ *conecta* $v_1$ y $v_n$. Un grafo es *conexo* si todos sus pares de vértices están conectados.

Sea $G = (V, A)$ un grafo con conjunto de vértices $V$ y conjunto de aristas $A$. Un *subgrafo* de $G$ es un grafo $G' = (V', A')$ donde

1. $V'$ es un subconjunto de $V$.
2. $A'$ consta de las aristas $(v, w)$ en $A$ tales que $v$ y $w$ están en $V'$.

Si $A'$ consta de todas las aristas $(v, w)$ en $A$, tal que $v$ y $w$ están en $V'$, entonces $G'$ se conoce como un *subgrafo inducido* de G.

---

† A menos que se especifique lo contrario, aquí se supondrá que una arista siempre es un par de vértices distintos.

**Ejemplo 7.1.** En la figura 7.1(a) se observa un grafo $G = (V, A)$ con $V = \{a, b, c, d\}$ y $A = \{(a, b), (a, d), (b, c), (b, d), (c, d)\}$, y en la figura 7.1(b), uno de sus subgrafos inducidos, definido por el conjunto de vértices $\{a, b, c\}$ y todas las aristas de la figura 7.1(a) que no inciden sobre el vértice $d$. □

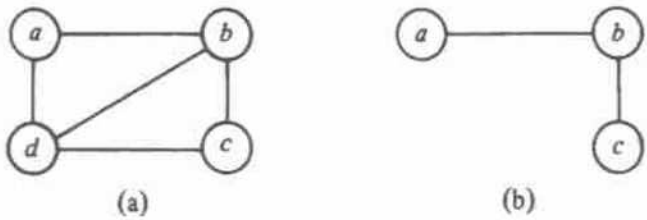



**Fig. 7.1.** Grafo con uno de sus subgrafos.

Un *componente conexo* de un grafo $G$ es un subgrafo conexo inducido maximal, esto es, un subgrafo conexo inducido que por sí mismo no es un subgrafo propio de ningún otro subgrafo conexo de $G$.

**Ejemplo 7.2.** La figura 7.1 es un grafo conexo que tiene sólo un componente conexo, y que es él mismo. La figura 7.2 es un grafo con dos componentes conexos. □

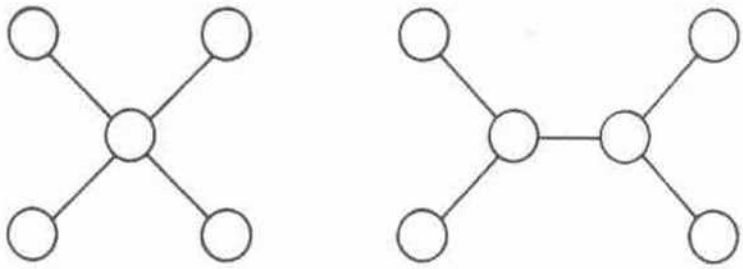


**Fig. 7.2.** Grafo no conexo.

Un *ciclo* (simple) de un grafo es un camino (simple) de longitud mayor o igual a tres, que conecta un vértice consigo mismo. No se consideran ciclos los caminos de la forma $v$ (camino de longitud 0), $v, v$ (camino de longitud 1), o $v, w, v$ (camino de longitud 2). Un grafo es *cíclico* si contiene por lo menos un ciclo. Un grafo conexo acíclico algunas veces se conoce como *árbol libre*. La figura 7.2 muestra un grafo que consta de dos componentes conexos, cada uno de los cuales es un árbol libre. Un árbol libre puede convertirse en ordinario si se elige cualquier vértice deseado como raíz y se orienta cada arista desde ella.

Los árboles libres tienen dos propiedades importantes que se usarán en la siguiente sección.

1. Todo árbol libre con $n \geq 1$ vértices contiene exactamente $n - 1$ aristas.
2. Si se agrega cualquier arista a un árbol libre, resulta un ciclo.

Se puede probar (1) por inducción en $n$, o en forma equivalente, con un argumento basado en el "contraejemplo más pequeño". Supóngase que $G = (V, A)$ es un contraejemplo de (1) con un mínimo de vértices $n$, por ejemplo $n$ no puede valer uno, porque el único árbol libre con un vértice tiene cero aristas, y (1) se satisface. Por tanto, $n$ debe ser mayor que uno.

Ahora se pretende que en el árbol libre exista algún vértice con exactamente una arista incidente. En la demostración, ningún vértice puede tener cero aristas incidentes, o $G$ no sería conexo. Supóngase que todo vértice tiene por lo menos dos aristas incidentes. Después, pártase de algún vértice $v_1$, y sígase cualquier arista desde $v_1$. En cada paso, se abandona un vértice por una arista diferente a la que se utilizó para llegar, formando un camino $v_1, v_2, v_3, \ldots$ .

Dado que sólo se tiene un número finito de vértices en $V$, no es posible que todos los vértices en el camino sean diferentes; en un momento dado, se encuentra $v_i = v_j$ para alguna $i < j$. No se puede tener $i = j - 1$, porque no hay ciclos de un vértice a sí mismo, y tampoco $i = j - 2$, ya que se llegaría y se abandonaría el vértice $v_{i+1}$ por la misma arista. Así, $i \leq j - 3$, y se tiene un ciclo $v_i, v_{i+1}, \ldots, v_j = v_i$, con lo que se contradice la hipótesis de que $G$ no tiene vértices con sólo una arista incidente y, por tanto, se concluye que existe tal vértice $v$ con arista $(v, w)$.

Ahora, considérese el grafo $G'$ formado al eliminar el vértice $v$ y la arista $(v, w)$ de $G$. $G'$ no puede contradecir (1), porque si lo hiciera podría ser un contraejemplo más pequeño que $G$. Por tanto, $G'$ tiene $n - 1$ vértices y $n - 2$ aristas. Pero $G$ tiene un vértice y una arista más que $G'$, es decir, tiene $n - 1$ aristas, probando que $G$ satisface realmente (1). Como no hay un contraejemplo más pequeño para (1), se concluye que no existe ese contraejemplo, y (1) es cierto.

Ahora, es posible probar con facilidad la proposición (2) de que la adición de una arista a un árbol libre forma un ciclo. De no ser así, el resultado de agregar la arista a un árbol libre de $n$ vértices sería un grafo con $n$ vértices y $n$ aristas. Este grafo aún sería conexo, y se ha supuesto que agregando la arista quedaría un grafo acíclico. Con esto, se tendría un árbol libre cuyas cantidades de vértices y de aristas no satisfarían la condición (1).

## Métodos de representación

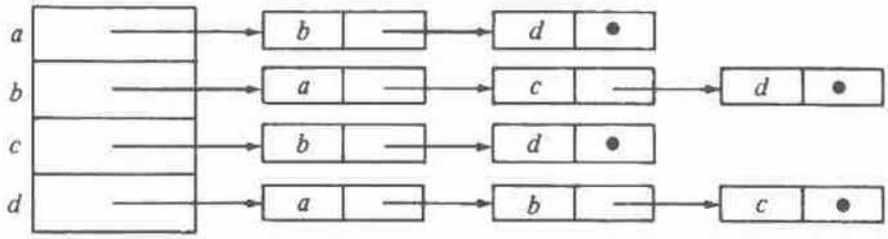
Los métodos de representación de grafos dirigidos se pueden emplear también para representar los no dirigidos. Una arista no dirigida entre $v$ y $w$ se representa simplemente con dos aristas dirigidas, una de $v$ a $w$, y otra de $w$ a $v$.

**Ejemplo 7.3.** Las representaciones con matriz y lista de adyacencia para el grafo de la figura 7.1(a) se muestran en la figura 7.3. □

Es notorio que la matriz de adyacencia para un grafo es simétrica. En la representación con lista de adyacencia, si $(i, j)$ es una arista, el vértice $j$ estará en la lista del vértice $i$ y el vértice $i$ estará en la lista del vértice $j$.

| | a | b | c | d |
|---|---|---|---|---|
| a | 0 | 1 | 0 | 1 |
| b | 1 | 0 | 1 | 1 |
| c | 0 | 1 | 0 | 1 |
| d | 1 | 1 | 1 | 0 |

(a) Matriz de adyacencia




(b) Lista de adyacencia

**Fig. 7.3.** Representaciones.

## 7.2 Arboles abarcadores de costo mínimo

Supóngase que $G = (V, A)$ es un grafo conexo en donde cada arista $(u, v)$ de $A$ tiene un costo asociado $c(u, v)$. Un *árbol abarcador* para $G$ es un árbol libre que conecta todos los vértices de $V$; su *costo* es la suma de los costos de las aristas del árbol. En esta sección se muestra cómo obtener el árbol abarcador de costo mínimo para $G$.

**Ejemplo 7.4.** La figura 7.4 muestra un grafo ponderado y su árbol abarcador de costo mínimo. □

Una aplicación típica de los árboles abarcadores de costo mínimo tiene lugar en el diseño de redes de comunicación. Los vértices del grafo representan ciudades, y las aristas, las posibles líneas de comunicación entre ellas. El costo asociado a una arista representa el costo de seleccionar esa línea para la red. Un árbol abarcador de costo mínimo representa una red que comunica todas las ciudades a un costo mínimal.

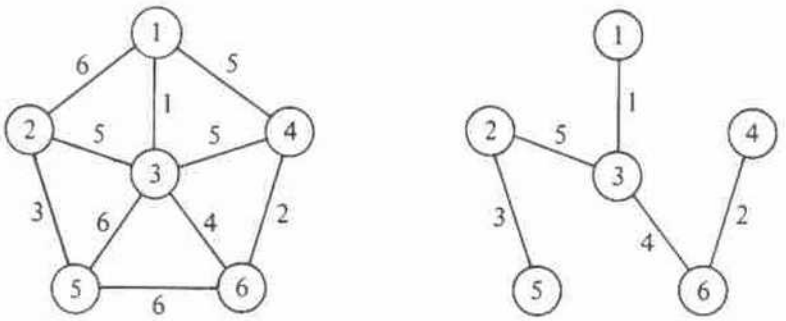



**Fig. 7.4.** Grafo y árbol abarcador.

## La propiedad AAM (Arbol Abarcador de costo Mínimo)

Hay distintas formas de construir un árbol abarcador de costo mínimo Muchos de esos métodos utilizan la siguiente propiedad de los árboles abarcadores de costo mínimo, que se denomina *propiedad AAM*. Sea $G = (V, A)$ un grafo conexo con una función de costo definida en las aristas. Sea $U$ algún subconjunto propio del conjunto de vértices $V$. Si $(u, v)$ es una arista de costo mínimo tal que $u \in U$ y $v \in V-U$, existe un árbol abarcador de costo mínimo que incluye $(u, v)$ entre sus aristas.

La demostración de que todo árbol abarcador de costo mínimo satisface la propiedad AAM no es muy difícil. Supóngase, por el contrario, que no existe el árbol abarcador de costo mínimo para $G$ que incluye $(u, v)$. Sea $T$ cualquier árbol abarcador de costo mínimo para $G$. Agregar $(u, v)$ a $T$ debe formar un ciclo, ya que $T$ es un árbol libre y, por tanto, satisface la propiedad (2) de los árboles libres. Este ciclo incluye la arista $(u, v)$. Así, debe haber otra arista $(u', v')$ en $T$ tal que $u' \in U$ y $v' \in V-U$, como se ilustra en la figura 7.5. Si no, no habría forma de que el ciclo fuera de $u$ a $v$ sin pasar por segunda vez por la arista $(u, v)$.

Al eliminar la arista $(u', v')$ se rompe el ciclo y se obtiene un árbol abarcador $T'$ cuyo costo en realidad no es mayor que el costo de $T$, ya que, por suposición, $c(u, v) \leq c(u', v')$. Así $T'$ contradice la suposición de que no hay un árbol abarcador de costo mínimo que incluya $(u, v)$.

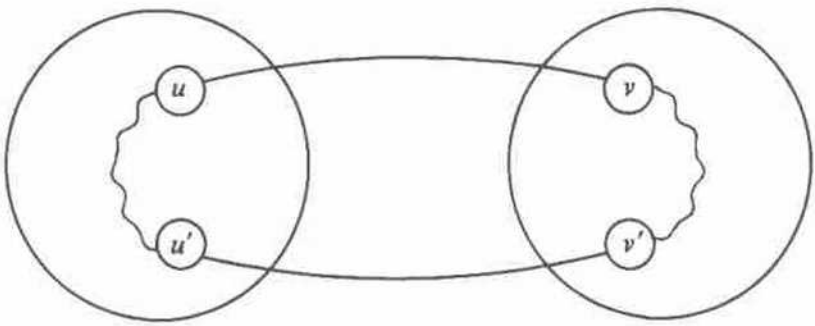



**Fig. 7.5.** Ciclo resultante.

## Algoritmo de Prim

Existen dos técnicas populares que explotan la propiedad AAM para construir un árbol abarcador de costo mínimo a partir de un grafo ponderado $G = (V, A)$; una de ellas se conoce como algoritmo de Prim. Supóngase que $V = \{1, 2, ..., n\}$. El algoritmo de Prim comienza cuando se asigna a un conjunto $U$ un valor inicial $\{1\}$, en el cual «crece» un árbol abarcador, arista por arista. En cada paso localiza la arista más corta $(u, v)$ que conecta $U$ y $V - U$, y después agrega $v$, el vértice en $V - U$, a $U$. Este paso se repite hasta que $U = V$. El algoritmo se resume en la figura 7.6, y la secuencia de aristas agregadas a $T$ para el grafo de la figura 7.4(a) se muestra en la figura 7.7.

```
procedure Prim (G: grafo; var T: conjunto de aristas);
    {Prim construye un árbol abarcador de costo mínimo T para G}
    var
        U: conjunto de vértices;
        u, v: vértice;
    begin
        T := Ø;
        U := {1};
        while U ≠ V do begin
            sea (u, v) una arista de costo mínimo tal que u está en U y
                v en V-U;
            T := T ∪ {(u, v)};
            U := U ∪ {v}
        end
    end; { Prim }
```

**Fig. 7.6.** Esbozo del algoritmo de Prim.

Una forma sencilla de encontrar la arista de menor costo entre $U$ y $V - U$ en cada paso es por medio de dos arreglos; uno, *MAS_CERCANO*[$i$], da el vértice en $U$ que esté más cercano a $i$ en $V - U$. El otro, *MENOR_COSTO*[$i$], da el costo de la arista ($i$, *MAS_CERCANO*[$i$]).

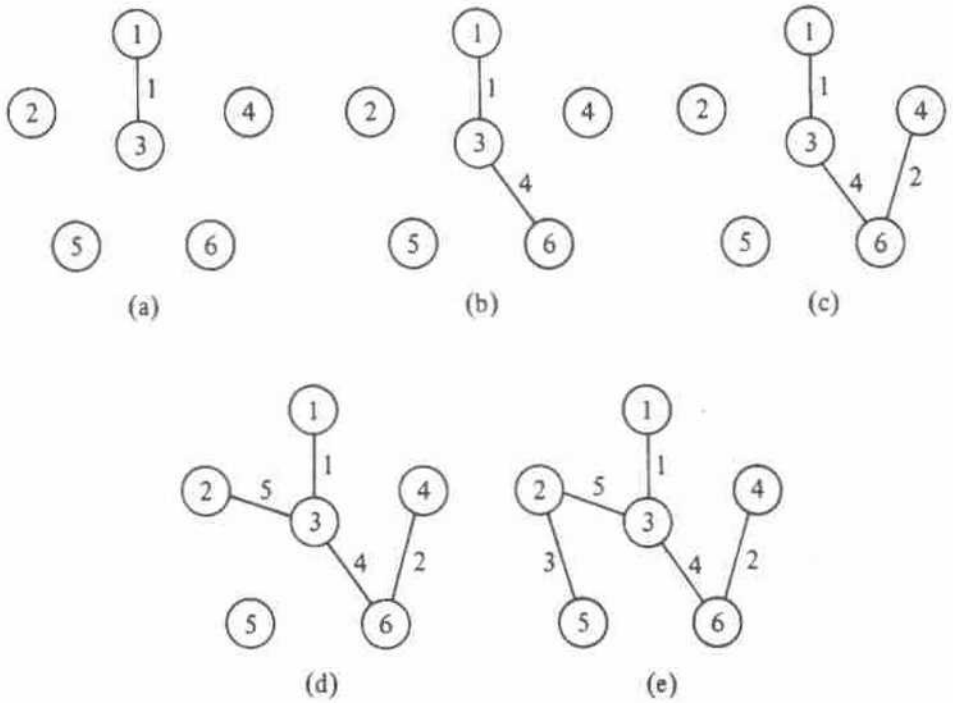



**Fig. 7.7.** Secuencias de aristas añadidas por el algoritmo de Prim.

En cada paso se revisa *MENOR_COSTO* para encontrar algún vértice, como $k$, en $V - U$ que esté más cercano a $U$. Se imprime la arista ($k$, *MAS_CERCANO*[$k$]). Entonces se actualizan los arreglos *MENOR_COSTO* y *MAS_CERCANO*, teniendo en cuenta el hecho de que $k$ ha sido agregada a $U$. En la figura 7.8 se da una versión en Pascal de este algoritmo. Se supone que $C$ es un arreglo de $n \times n$ tal que $C[i, j]$ es el costo de la arista $(i, j)$. Si la arista $(i, j)$ no existe, se supone que $C[i, j]$ tiene un valor grande apropiado.

Si encuentra otro vértice $k$ para el árbol abarcador, se hace que *MENOR_COSTO*[$k$] sea *infinito*, un valor muy grande, de modo que este vértice ya no se considera en los recorridos siguientes para incluirlo en $U$. El valor *infinito* es mayor que el costo de cualquier arista o que el costo asociado a una arista no existente.

La complejidad de tiempo del algoritmo de Prim es $O(n^2)$, ya que se efectúan $n - 1$ iteraciones del ciclo de las líneas (4) a (16) y cada iteración del ciclo lleva un tiempo $O(n)$, debido a los ciclos más internos de las líneas (7) a (10) y (13) a (16). Conforme $n$ crece, el rendimiento de este algoritmo puede dejar de ser satisfactorio. Ahora se presenta otro algoritmo, debido a Kruskal, para encontrar árboles abarcadores de costo mínimo cuyo rendimiento puede ser como máximo $O(a \log a)$, donde $a$ es el número de aristas del grafo dado. Si $a$ es mucho menor que $n^2$, el algoritmo de Kruskal es superior, pero si es cercano a $n^2$, se debe optar por el algoritmo de Prim.

```
     procedure Prim (C: array[1..n, 1..n] of real );
         { Prim imprime las aristas de un árbol abarcador de costo mínimo
           para un grafo con vértices {1, 2, ..., n} y matriz de costo C
           definida en las aristas }
         var
             MENOR_COSTO: array[1..n] of real;
             MAS_CERCANO: array[1..n] of integer;
             i, j, k, mín: integer;
             { i y j son índices. Durante una revisión del arreglo MENOR_COS-
               TO, k es el índice del vértice más cercano encontrado hasta
               ese punto, y mín = MENOR_COSTO[k] }

         begin
(1)          for i := 2 to n do begin
                 { asigna valor inicial al conjunto U sólo con el vértice 1 }
(2)              MENOR_COSTO[i] := C[1, i];
(3)              MAS_CERCANO[i] := 1
             end;
(4)          for i := 2 to n do begin
                 { encuentra el vértice k fuera de U más cercano a algún vértice
                   en U }
(5)              mín := MENOR_COSTO[2];
(6)              k := 2;
(7)              for j := 3 to n do
(8)                  if MENOR_COSTO[j] < mín then begin
```

```
(9)                    mín := MENOR_COSTO[j];
(10)                   k := j
                   end;
(11)               writeln(k, MAS_CERCANO[k]); { imprime la arista }
(12)               MENOR_COSTO[k] := infinito; { se añade k a U }
(13)               for j := 2 to n do { ajusta los costos de U }
(14)                   if (C[k, j] < MENOR_COSTO[j]) and
                          (MENOR_COSTO[j] < infinito) then begin
(15)                       MENOR_COSTO[j] := C[k, j];
(16)                       MAS_CERCANO[j] := k
                   end
               end
           end; { Prim }
```

**Fig. 7.8.** Algoritmo de Prim.

## Algoritmo de Kruskal

Supóngase de nuevo que se tiene un grafo conexo $G = (V, A)$, con $V = \{1, 2, ..., n\}$ y una función de costo $c$ definida en las aristas de $A$. Otra forma de construir un árbol abarcador de costo mínimo para $G$ es empezar con un grafo $T = (V, \emptyset)$ constituido sólo por los vértices de $G$ y sin aristas. Por tanto, cada vértice es un componente conexo por sí mismo. Conforme el algoritmo avanza, habrá siempre una colección de componentes conexos, y para cada componente se seleccionarán las aristas que formen un árbol abarcador.

Para construir componentes cada vez mayores, se examinan las aristas a partir de $A$, en orden creciente de acuerdo con el costo. Si la arista conecta dos vértices que se encuentran en dos componentes conexos distintos, entonces se agrega la arista $T$. Se descartará la arista si conecta dos vértices contenidos en el mismo componente, ya que puede provocar un ciclo si se la añadiera al árbol abarcador para ese componente conexo. Cuando todos los vértices están en un solo componente, $T$ es un árbol abarcador de costo mínimo para $G$.

**Ejemplo 7.5.** Considérese el grafo ponderado de la figura 7.4(a). La secuencia de aristas agregadas a $T$ se muestra en la figura 7.9. Las aristas de costo 1, 2, 3 y 4 se consideran primero, y todas son aceptadas, ya que ninguna de ellas causa un ciclo. Las aristas (1, 4) y (3, 4) de costo 5 no pueden aceptarse, porque conectan vértices que están dentro del mismo componente en la figura 7.9(d) y, por tanto, pueden completar un ciclo. Sin embargo, la arista restante de costo 5, o sea (2, 3), no crea ciclos. Una vez que se acepta, el proceso termina. □

Es posible aplicar este algoritmo mediante los conjuntos y sus operaciones asociadas de los capítulos 4 y 5. Primero se necesita un conjunto formado por las aristas de $A$. Al conjunto se le aplica en forma repetida el operador *SUPRIME_MIN* para seleccionar aristas en orden creciente de acuerdo con el costo. El conjunto de aristas, por tanto, forma una cola de prioridad, y entonces un árbol parcialmente ordenado es la estructura de datos más apropiada para usar aquí.

También se requiere mantener un conjunto de componentes conexos $C$. Las operaciones que se le aplican son:

1. COMBINA($A$, $B$, $C$), para combinar los componentes $A$ y $B$ en $C$ y llamar al resultado $A$ o $B$ en forma arbitraria †.
2. ENCUENTRA($v$, $C$), para devolver el nombre del componente de $C$, del cual el vértice $v$ es miembro. Esta operación se usará para determinar si los dos vértices de una arista se encuentran en dos componentes distintos o en el mismo.
3. INICIAL($A$, $v$, $C$), para que $A$ sea el nombre de un componente que pertenece a $C$, y que inicialmente contiene sólo el vértice $v$.

Estas son las operaciones del TDA COMBINA_ENCUENTRA llamado CONJUNTO_CE, estudiado en la sección 5.5. En la figura 7.10 se muestra un esbozo de un programa llamado *Kruskal* para encontrar árboles abarcadores de costo mínimo con estas operaciones.

Se pueden emplear las técnicas desarrolladas en la sección 5.5 para implantar las operaciones utilizadas en este programa. El tiempo de ejecución de este programa depende de dos factores. Si hay $a$ aristas, lleva un tiempo $O(a \log a)$ insertar las aristas en la cola de prioridad ††. En cada iteración del ciclo **while**, la obtención de la

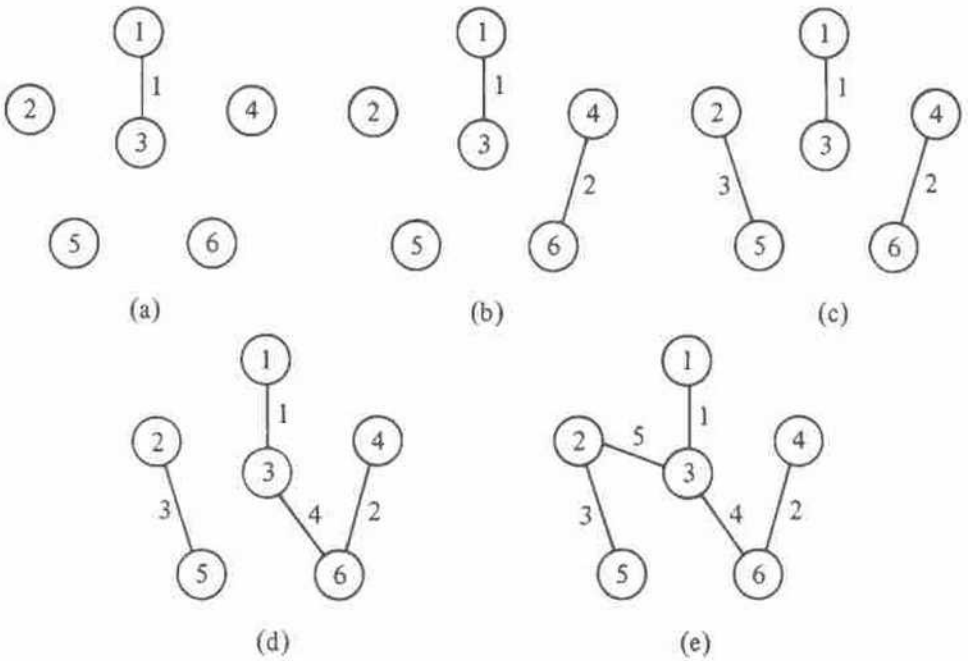



**Fig. 7.9.** Secuencia de aristas añadidas por el algoritmo de Kruskal.

† Obsérvese que COMBINA y ENCUENTRA son ligeramente distintas a las definiciones de la sección 5.5, ya que $C$ es un parámetro que indica dónde se pueden encontrar $A$ y $B$.

†† Se puede asignar valor inicial a un árbol parcialmente ordenado de $a$ elementos en un tiempo $O(a)$, si se hace todo de una vez. Esta técnica se analiza en la sección 8.4, aunque tal vez se debería utilizar aquí, ya que si se examinan menos de $a$ aristas antes de encontrar el árbol abarcador de costo mínimo, se puede ahorrar un tiempo significativo.

arista de menor costo en *aristas* lleva un tiempo $O(\log a)$. Así, las operaciones en la cola de prioridad llevan un tiempo $O(a \log a)$ en el peor caso. El tiempo total requerido para realizar las operaciones COMBINA y ENCUENTRA depende del método usado para implantar el CONJUNTO_CE. Como se muestra en la sección 5.5, hay métodos $O(a \log a)$ y $O(a\alpha(a))$. En cualquiera de los casos, el algoritmo de Kruskal se puede implantar para que se ejecute en tiempo $O(a \log a)$.

## 7.3 Recorridos

En un gran número de problemas con grafos, es necesario visitar sistemáticamente los vértices del grafo. Las búsquedas en profundidad y en amplitud, temas de esta sección, son dos técnicas importantes para hacerlo. Ambas técnicas pueden usarse para determinar de manera eficiente todos los vértices que están conectados a un vértice dado.

```
procedure Kruskal ( V: CONJUNTO de vértices;
                    A: CONJUNTO de aristas;
                    var T: CONJUNTO de aristas );
    var
        comp_n: integer; { número actual de componentes }
        aristas: COLA_DE_PRIORIDAD; { el conjunto de aristas }
        componentes: CONJUNTO_CE; { el conjunto V agrupado en
            un conjunto de componentes COMBINA_ENCUENTRA }
        u, v: vértice;
        a: arista;
        comp_siguiente: integer; { nombre para el nuevo componente }
        comp_u, comp_v; { nombres de componentes }

    begin
        ANULA(T);
        ANULA(aristas);
        comp_siguiente := 0;
        comp_n := número de miembros de V;
        for v en V do begin { asigna valor inicial a un componente
            para que contenga un vértice de V }
            comp_siguiente := comp_siguiente + 1;
            INICIAL(comp_siguiente, v, componentes)
        end;
        for a en A do { asigna valor inicial a la cola de prioridad de aristas }
            INSERTA(a, aristas);
        while comp_n > 1 do begin { considera la siguiente arista }
            a := SUPRIME_MIN(aristas)
            sea a = (u, v);
            comp_u := ENCUENTRA(u, componentes);
            comp_v := ENCUENTRA(v, componentes);
```

```
            if comp_u <> comp_v then begin
                { a conecta dos componentes diferentes }
                COMBINA(comp_u, comp_v, componentes);
                comp_n := comp_n-1;
                INSERTA(a, T)
            end
        end
    end; { Kruskal }
```

**Fig. 7.10.** Algoritmo de Kruskal.

## Búsqueda en profundidad

Recuérdese de la sección 6.5 el algoritmo *bpf* para búsquedas en grafos dirigidos. El mismo algoritmo puede emplearse para búsqueda en grafos no dirigidos, puesto que la arista no dirigida $(v, w)$ puede considerarse como el par de aristas dirigidas $v \rightarrow w$ y $w \rightarrow v$.

De hecho, los bosques abarcadores en profundidad, construidos para grafos no dirigidos, son más simples que para los dirigidos. Primero, se debe observar que cada árbol del bosque es un componente conexo del grafo, y que si el grafo fuera conexo, tendría sólo un árbol en su bosque. Segundo, para grafos dirigidos se identifican cuatro clases de arcos: de árbol, de avance, de retroceso y cruzado. Para grafos no dirigidos sólo hay dos clases: aristas de árbol y de retroceso.

Dado que en grafos no dirigidos no existe distinción entre las aristas de retroceso y las de avance, se denominarán arcos *de retroceso*. En un grafo no dirigido no existen las aristas cruzadas, esto es, aristas $(v, w)$ donde $v$ no es antecesor ni descendiente de $w$ en el árbol abarcador. Supóngase que las hubiera; entonces, sea $v$ un vértice alcanzado antes que $w$ en la búsqueda. La llamada a *bpf*$(v)$ no puede terminar hasta haber buscado $w$, así que $w$ se introduce en el árbol como descendiente de $v$. De modo semejante, si *bpf*$(w)$ se llama antes que *bpf*$(v)$, $v$ se convierte en descendiente de $w$.

Como resultado, durante una búsqueda en profundidad en un grafo no dirigido $G$, todas las aristas pueden ser,

1. *aristas de árbol*, aquellas aristas $(v, w)$ tales que *bpf*$(v)$ llama directamente a *bpf*$(w)$ o viceversa, o bien
2. *aristas de retroceso*, aquellas aristas $(v, w)$ tales que ni *bpf*$(v)$ ni *bpf*$(w)$ se llaman directamente, pero una llamó indirectamente a la otra (es decir, *bpf*$(w)$ llama a *bpf*$(x)$, que llama a *bpf*$(v)$, de modo que $w$ es antecesor de $v$).

**Ejemplo 7.6.** Considérese el grafo conexo $G$ de la figura 7.11(a). Un árbol abarcador en profundidad $T$ resultante de una búsqueda en profundidad de $G$ se muestra en la figura 7.11(b). Se supuso que la búsqueda empezó en el vértice $a$, y se adoptó la convención de mostrar las aristas del árbol con líneas de trazo continuo y las aristas de retroceso con líneas de puntos. El árbol se dibujó con la raíz en la parte

superior, y los hijos de cada vértice, en el orden de izquierda a derecha en que fueron visitados por el procedimiento *bpf*.

Para seguir unos cuantos pasos de la búsqueda, el procedimiento *bpf*(*a*) llama a *bpf*(*b*) y añade la arista (*a*, *b*) a *T*, ya que *b* no ha sido visitado. En *b*, *bpf* llama a *bpf*(*d*) y agrega la arista (*b*, *d*) a *T*. En *d*, *bpf* llama a *bpf*(*e*) y añade la arista (*d*, *e*) a *T*. En *e*, los vértices *a*, *b* y *d* ya están marcados como visitados, de modo que *bpf*(*e*) regresa sin incorporar ninguna arista a *T*. En *d*, *bpf* encuentra los vértices *a* y *b* marcados como visitados, así que *bpf*(*d*) regresa también sin agregar más aristas a *T*. En *b*, *bpf* encuentra los vértices adyacentes restantes *a* y *e* marcados como visitados, así que *bpf*(*b*) regresa. La búsqueda continúa después con *c*, *f* y *g*. □

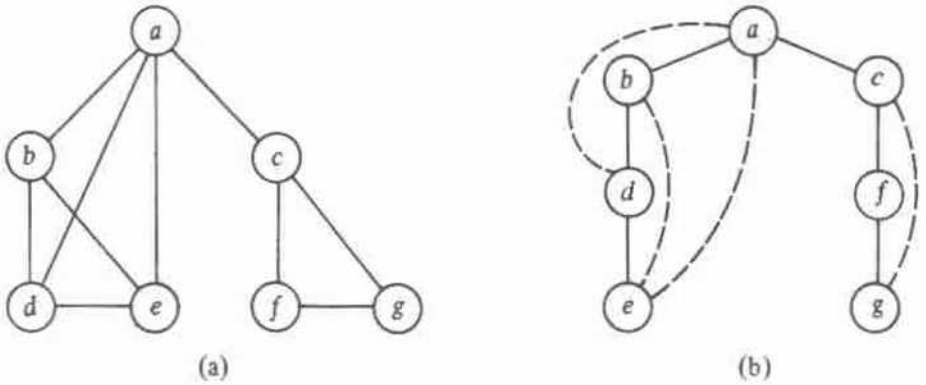



**Fig. 7.11.** Un grafo y su búsqueda en profundidad.

## Búsqueda en amplitud

Otra forma sistemática de visitar los vértices se conoce como *búsqueda en amplitud*. Este enfoque se denomina «en amplitud» porque desde cada vértice *v* que se visita se busca en forma tan amplia como sea posible, visitando todos los vértices adyacentes a *v*. Esta estrategia de búsqueda también se puede aplicar a grafos dirigidos.

Igual que en la búsqueda en profundidad, al realizar una búsqueda en amplitud se puede construir un bosque abarcador. En este caso, se considera la arista (*x*, *y*) como una arista de árbol si el vértice *y* es el que se visitó primero partiendo del vértice *x* del ciclo interno del procedimiento de búsqueda *bea* de la figura 7.12.

Resulta que para la búsqueda en amplitud en un grafo no dirigido, toda arista que no es de árbol es una arista cruzada; esto es, conecta dos vértices ninguno de los cuales es antecesor del otro.

El algoritmo de búsqueda en amplitud de la figura 7.12 inserta las aristas de árbol en un conjunto *T*, que se supone inicialmente vacío. Se presume que cada entrada en el arreglo *marca* tiene asignado el valor inicial *no_visitado*; la figura 7.12 opera en un componente conexo. Si el grafo no es conexo, *bea* debe llamarse desde un vértice de cada componente. Obsérvese que en una búsqueda en amplitud se debe

marcar cada vértice como visitado antes de meterlo en la cola, y así evitar que se coloque en la cola más de una vez.

**Ejemplo 7.7.** El árbol abarcador en amplitud del grafo $G$ de la figura 7.11(a) se muestra en la figura 7.13. Se supone que la búsqueda empieza en el vértice $a$. Como antes, las aristas de árbol se muestran con líneas de trazo continuo y las otras con líneas de puntos. También se ha dibujado la raíz del árbol en la parte superior, y los hijos, de izquierda a derecha, de acuerdo con el orden en que fueron visitados. □

```
procedure bea (v);
    { bea visita todos los vértices conectados a v usando búsqueda en
      amplitud }
    var
        C: COLA de vértice;
        x, y: vértice;
    begin
        marca[v] := visitado;
        PONE_EN_COLA(v, C);
        while not VACIA(C) do begin
            x := FRENTE(C);
            QUITA_DE_COLA (C);
            for cada vértice y adyacente a x do
                if marca[y] = no_visitado then begin
                    marca[y] := visitado;
                    PONE_EN_COLA(y, C);
                    INSERTA((x, y), T)
                end
        end
    end; { bea }
```

**Fig. 7.12.** Búsqueda en amplitud.

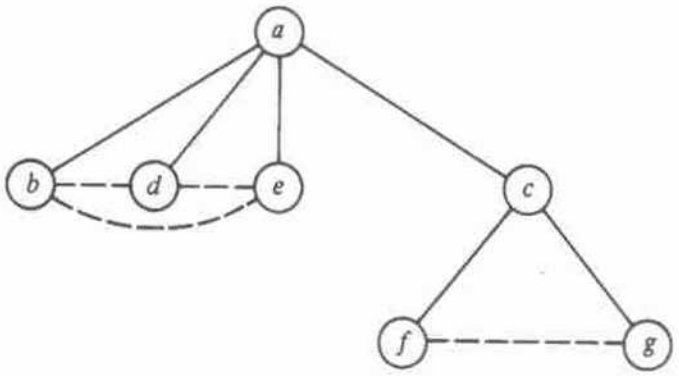



**Fig. 7.13.** Búsqueda en amplitud para $G$.

La complejidad de tiempo de la búsqueda en amplitud es la misma que para la búsqueda en profundidad. Cada vértice visitado se coloca en la cola una vez, así que el ciclo **while** se ejecuta una sola vez para cada vértice. Cada arista $(x, y)$ se examina dos veces, desde $x$ y desde $y$. Así, si el grafo tiene $n$ vértices y $a$ aristas, el tiempo de ejecución de *bea* es $O(\text{máx}(n, a))$ si se utiliza una representación con lista de adyacencia para las aristas. Dado que es típico que $a \geq n$, en general se hará referencia al tiempo de ejecución de la búsqueda en amplitud con $O(a)$, como ocurrió para la búsqueda en profundidad.

Las búsquedas en profundidad y en amplitud se pueden usar como marcos de trabajo, alrededor de los cuales se diseñan eficientes algoritmos para grafos. Por ejemplo, se puede emplear cualquiera de los dos métodos para encontrar los componentes conexos de un grafo, ya que aquéllos son los árboles de los dos bosques abarcadores.

Se puede verificar la existencia de ciclos por medio de la búsqueda en amplitud en un tiempo $O(n)$, donde $n$ es el número de vértices, independientemente del número de aristas. Como se vio en la sección 7.1, cualquier grafo con $n$ vértices y $n$ o más aristas debe tener un ciclo. Sin embargo, un grafo puede tener $n - 1$ o menos aristas y de todos modos tener un ciclo, si tiene dos o más componentes conexos. Una forma segura de encontrar los ciclos es construir un bosque abarcador en amplitud. Así, toda arista cruzada $(v, w)$ debe completar un ciclo simple con las aristas de árbol que conducen a $v$ y $w$ desde su antecesor común más cercano, como se muestra en la figura 7.14.

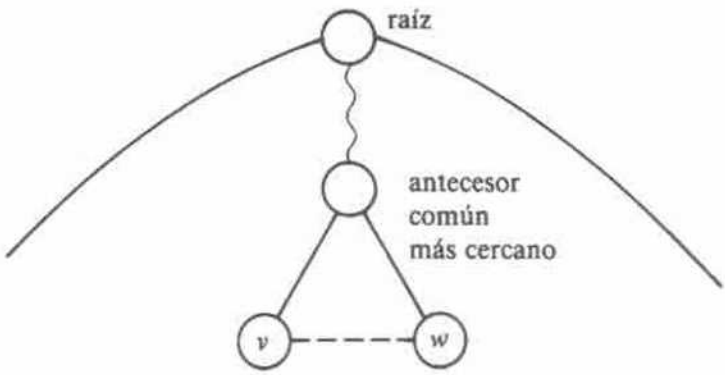



**Fig. 7.14.** Ciclo encontrado por la búsqueda en amplitud.

## 7.4 Puntos de articulación y componentes biconexos

Un *punto de articulación* de un grafo es un vértice $v$ tal que cuando se elimina junto con todas las aristas incidentes sobre él, se divide un componente conexo en dos o más partes. Por ejemplo, los puntos de articulación de la figura 7.11(a) son $a$ y $c$. Si se elimina $a$, el grafo, que es un componente conexo, se divide en dos triángulos: $\{b, d, e\}$ y $\{c, f, g\}$; si se elimina $c$, se divide en $\{a, b, d, e\}$ y $\{f, g\}$. Sin embargo, al eli-

minar cualquier otro vértice del grafo de la figura 7.11(a), el componente conexo no se dividirá. A un grafo sin puntos de articulación se le llama *biconexo*. La búsqueda en profundidad es muy útil para encontrar los componentes biconexos de un grafo.

El problema de encontrar los puntos de articulación es el más simple de muchos problemas importantes relacionados con la conectividad de grafos. Como ejemplo de las aplicaciones de los algoritmos de conectividad, se puede presentar una red de comunicaciones como un grafo en el que los vértices son lugares que hay que mantener comunicados entre sí. Un grafo tiene *conectividad k* si la eliminación de $k - 1$ vértices cualesquiera no lo desconecta. Por ejemplo, un grafo tiene conectividad dos o más si, y sólo si, no tiene puntos de articulación, es decir, si, y sólo si, es biconexo. Cuanto mayor sea su conectividad, tanto más fácil será que sobreviva al fallo de alguno de sus vértices, sea por fallo de las unidades de procesamiento colocadas en los vértices o por motivos externos.

Ahora se presenta un algoritmo simple de búsqueda en profundidad para encontrar todos los puntos de articulación de un grafo, y probar por medio de su ausencia, si el grafo es biconexo.

1. Realizar una búsqueda en profundidad del grafo, calculando *número_bp*[*v*] para todo vértice *v*, como se analizó en la sección 6.5. En esencia, *número_bp* ordena los vértices como en un recorrido en orden previo del árbol abarcador en profundidad.

2. Para cada vértice *v*, obtener *bajo*[*v*], que es el *número_bp* más pequeño de *v* o de cualquier otro vértice *w* accesible desde *v*, siguiendo cero o más aristas de árbol hasta un descendiente *x* de *v* (*x* puede ser *v*) y después seguir una arista de retroceso (*x*, *w*). Se calcula *bajo*[*v*] para todos los vértices *v*, visitándolos en un recorrido en orden posterior. Cuando se procesa *v*, se ha calculado *bajo*[*y*] para todo hijo *y* de *v*. Se toma *bajo*[*v*] como el mínimo de

   a) *número_bp*[*v*],

   b) *número_bp*[*z*] para cualquier vértice *z* para el cual haya una arista de retroceso (*v*, *z*), y

   c) *bajo*[*y*] para cualquier hijo *y* de *v*.

3. Ahora se encuentran los puntos de articulación como sigue.

   a) La raíz es un punto de articulación si, y sólo si, tiene dos o más hijos. Puesto que no hay aristas cruzadas, la eliminación de la raíz debe desconectar los subárboles cuyas raíces se encuentren en sus hijos, como *a* desconecta {*b*, *d*, *e*} de {*c*, *f*, *g*} en la figura 7.11(b).

   b) Un vértice *v* distinto de la raíz es un punto de articulación si, y sólo si, hay un hijo *w* de *v* tal que *bajo*[*w*] $\geq$ *número_bp*[*v*]. En este caso, *v* desconecta *w* y sus descendientes del resto del grafo. A la inversa, si *bajo*[*w*] $<$ *número_bp*[*v*], debe haber un camino para descender desde *w* en el árbol y regresar hasta un antecesor propio de *v* (el vértice cuyo *número_bp* es *bajo*[*w*]) y, por tanto, la eliminación de *v* no desconecta *w* ni sus descendientes del resto del grafo.

**Ejemplo 7.8.** *número_bp* y *bajo* se calculan para el grafo de la figura 7.11(*a*) en la figura 7.15. Como ejemplo de la obtención de *bajo*, el recorrido en orden posterior visita *e* primero. En *e*, hay aristas de regreso (*e*, *a*) y (*e*, *b*), así que *bajo*[*e*] se iguala a mín(*número_bp*[*e*], *número_bp*[*a*], *número_bp*[*b*]) = 1. Después se visita *d*, y *bajo*[*d*] se hace igual al mínimo de *número_bp*[*d*], *bajo*[*e*] y *número_bp*[*a*]. El segundo de éstos surge porque *e* es un hijo de *d*, y el tercero, por la existencia de la arista de retroceso (*d*, *a*).

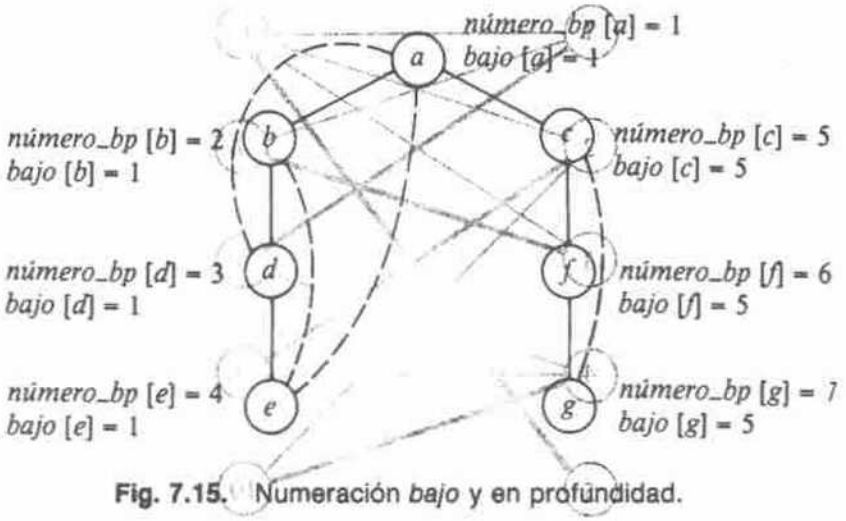



**Fig. 7.15.** Numeración *bajo* y en profundidad.

Después de obtener *bajo*, se considera cada vértice. La raíz *a* es un punto de articulación porque tiene dos hijos. El vértice *c* es un punto de articulación porque tiene un hijo *f* con *bajo*[*f*] ≥ *número_bp*[*c*]. Los otros vértices no son puntos de articulación. □

El tiempo que consume el algoritmo anterior en un grafo de *a* aristas y $n \leq a$ vértices es $O(a)$. Es recomendable comprobar que el tiempo empleado en cada una de las tres fases puede atribuirse al vértice visitado o a una arista que parta de ese vértice, y sólo se le puede atribuir una cantidad constante de tiempo a cualquier arista o vértice en cualquier paso. Así, el tiempo total es $O(n+a)$, el cual es $O(a)$ en el supuesto de que $n \leq a$.

## 7.5 Pareamiento de grafos

En esta sección se bosquejará un algoritmo para resolver «problemas de pareamiento» en grafos. Un ejemplo simple de problema de pareamiento ocurre cuando se tiene un conjunto de profesores para distribuir en un conjunto de cursos. Cada profesor es competente para impartir ciertos cursos, pero no otros. Se desea asignar un curso al profesor adecuado, pero sin asignar dos profesores al mismo curso. Para ciertas distribuciones de profesores y cursos, es imposible asignar un curso a cada profesor; en tal situación, es deseable asignar tantos profesores como sea posible.