Está claro que es más eficiente rechazar todas las monedas restantes de 100 pesetas (digamos) cuando el valor restante que hay que pagar cae por debajo de ese valor. El uso de la división entera para calcular cuántas monedas de un cierto valor hay que tomar también es más eficiente que actuar por sustracciones sucesivas. Si se adopta cualquiera de estas tácticas, entonces podemos relajar la condición consistente en que el conjunto de monedas debe ser finito.

6.3 GRAFOS: ÁRBOLES DE RECUBRIMIENTO MÍNIMO

Sea $G = \langle N, A \rangle$ un grafo conexo no dirigido en donde N es el conjunto de nodos y A es el conjunto de aristas. Cada arista posee una *longitud* no negativa. El problema consiste en hallar un subconjunto T de las aristas de G tal que utilizando solamente las aristas de T, todos los nodos deben quedar conectados, y además la suma de las longitudes de las aristas de T debe ser tan pequeña como sea posible. Dado que G es conexo, debe existir al menos una solución. Si G tiene aristas de longitud 0, pueden existir varias soluciones cuya longitud total sea la misma, pero que tengan números distintos de aristas. En este caso, dadas dos soluciones de igual longitud total, preferimos la que contenga menos aristas. Incluso con esta condición el problema puede tener varias soluciones diferentes y de igual valor. En lugar de hablar de las longitudes, podemos asociar un *coste* a cada arista. El problema, entonces, consiste en hallar un subconjunto T de las aristas cuyo coste total sea el menor posible. Evidentemente, este cambio de terminología no afecta a la forma en que resolvemos el problema.

Sea $G' = \langle N, T \rangle$ el grafo parcial formado por los nodos de G y las aristas de T, y supongamos que en N hay n nodos. Un grafo conexo con n nodos debe tener al menos $n - 1$ aristas, así que éste es el número mínimo de aristas que puede haber en T. Por otra parte, un grafo con n nodos y más de $n - 1$ aristas contiene al menos un ciclo; véase el Problema 6.7. Por tanto, si G' es conexo y T tiene más de $n - 1$ aristas, se puede eliminar al menos una arista sin desconectar G', siempre y cuando seleccionemos una arista que forme parte de un ciclo. Esto hará o bien que disminuya la longitud total de las aristas de T, o bien que la longitud total quede intacta (si hemos eliminado una arista de longitud 0) a la vez que disminuye el número de aristas que hay en T. En ambos casos, la nueva solución es preferible a la anterior. Por tanto, un conjunto T con n o más aristas no puede ser óptimo. Se sigue que T debe tener exactamente $n - 1$ aristas, y como G' es conexo, tiene que ser un árbol.

El grafo G' se denomina *árbol de recubrimiento*[2] *mínimo* para el grafo G. Este problema tiene muchas aplicaciones. Por ejemplo, supongamos que los nodos de G representan ciudades, y sea el coste de una arista $\{a, b\}$ el coste de tender una línea telefónica desde a hasta b. Entonces un árbol de recubrimiento mínimo de G se corresponde con la red más barata posible para dar servicio a todas las ciudades en cuestión, siempre y cuando sólo se puedan utilizar conexiones directas entre ciudades (en otras palabras, siempre y cuando no se permita establecer centrales telefónicas en el campo, *entre* las ciudades). Relajar esta condición es equivalente a permitir que se añadan nodos adicionales, auxiliares, a G. Esto puede permitirnos obtener soluciones más baratas: véase el problema 6.8.

A primera vista, parece que existen al menos dos líneas de ataque si deseamos encontrar un algoritmo voraz para este problema. Está claro que nuestro conjunto de candidatos debe ser el conjunto A de aristas que están en G. Una táctica posible consiste en comenzar por un conjunto vacío T, y seleccionar en cada etapa la arista más corta que todavía no haya sido seleccionada o rechazada, independientemente de donde se encuentra esta arista en G. Otra línea de ataque implica seleccionar un nodo y construir un árbol a partir de él, seleccionando en cada etapa la arista más corta posible que pueda extender el árbol hasta un nodo adicional. Es poco frecuente, pero ¡para este problema concreto, funcionan las dos aproximaciones! Antes de presentar los algoritmos, mostraremos la forma en que se aplica a este caso el esquema general de los algoritmos voraces, y presentaremos un lema que será de utilidad posteriormente:

◊ Los candidatos, como ya se ha indicado, son las aristas de G.
◊ Un conjunto de aristas es una solución si constituye un árbol de recubrimiento para los nodos de N.
◊ Un conjunto de aristas es factible si no contiene ningún ciclo.
◊ La función de selección que utilizamos varía con el algoritmo.
◊ La función objetivo que hay que minimizar es la longitud total de las aristas de la solución.

También necesitamos algo más de terminología. Diremos que un conjunto de aristas factible es *prometedor* si se puede extender para producir no sólo una solución, sino una solución óptima para nuestro problema. En particular, el conjunto vacío siempre es prometedor (puesto que siempre existe una solución óptima). Además, si un conjunto de aristas prometedor ya es una solución, la extensión requerida es irrelevante, y esta solución debe ser óptima . Decimos que una arista *sale* de un conjunto de nodos dado si esa arista tiene exactamente un extremo en el conjunto de nodos. Por tanto, una arista puede no salir de un conjunto dado de nodos bien porque ninguno de sus extremos esté en el conjunto, bien porque —y esto es menos evidente— sus dos extremos se encuentren en el conjunto. El lema siguiente es crucial para demostrar la corrección de los próximos algoritmos.

Lema 6.3.1. *Sea $G=\langle N, A \rangle$ un grafo conexo no dirigido en el cual está dada la longitud de todas las aristas. Sea $B \subset N$ un subconjunto estricto de los nodos de G. Sea $T \subseteq A$ un conjunto prometedor de aristas tal que no haya ninguna arista de T que sale de B. Sea v la arista más corta que salga de B (o una de las más cortas si hay empates). Entonces $T \cup \{v\}$ es prometedor.*

Demostración. Sea U un árbol de recubrimiento mínimo de G tal que $T \subseteq U$. Este U tiene que existir puesto que T es prometedor por hipótesis. Si $v \in U$, entonces no hay nada que probar. Si no, cuando añadimos v a U creamos exactamente un ciclo. (Ésta es una propiedad de los árboles: véase la Sección 5.5.) En este ciclo, puesto que v sale de B, existe necesariamente al menos otra arista u que también sale de B, o bien el ciclo no se cerraría; véase la figura 6.1. Si ahora eliminamos u, el ciclo desaparece y obtenemos un nuevo árbol V que abarca G. Sin embargo, la longitud de v, por definición, no es mayor que la longitud de u, y por tanto la longitud total de las aristas de V no sobrepasa la longitud total de las aristas de U. Por tanto, V es también un árbol de recubrimiento mínimo de G y contiene a v. Para completar la demostración, sólo hay que comentar que $T \subseteq V$ porque la arista u que se ha eliminado sale de B, y por tanto no podría haber sido una arista de T.

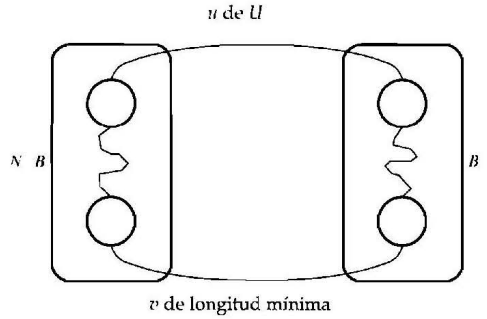

Figura 6.1. Se crea un ciclo si añadimos una arista v a U

6.3.1 Algoritmo de Kruskal

El conjunto de aristas T está vacío inicialmente. A medida que progresa el algoritmo, se van añadiendo aristas a T. Mientras no haya encontrado una solución, el grafo parcial formado por los nodos de G y las aristas de T consta de varios componentes conexos. (Inicialmente, cuando T está vacío, cada nodo de G forma una componente conexa distinta trivial.) Los elementos de T que se incluyen en una componente conexa dada forman un árbol de recubrimiento mínimo para los nodos de esta componente. Al final del algoritmo, sólo queda una componente conexa, así que T es un árbol de recubrimiento mínimo para todos los nodos de G.

Para construir componentes conexas más y más grandes, examinamos las aristas de G por orden creciente de longitudes. Si una arista une a dos nodos de componentes conexas distintas, se lo añadimos a T. Consiguientemente, las dos componentes conexas forman ahora una única componente. En caso contrario, se rechaza la arista: une a dos nodos de la misma componente conexa, y por tanto no se puede añadir a T sin formar un ciclo (porque las aristas de T forman un árbol para cada componente). El algoritmo se detiene cuando sólo queda una componente conexa.

Para ilustrar la forma en que funciona este algoritmo, considérese el gráfico de la figura 6.2. Por orden creciente de longitud, las aristas son: {1,2}, {2,3}, {4,5}, {6,7}, {1,4}, {2,5}, {4,7}, {3,5}, {2,4}, {3,6}, {5,7} y {5,6}.

Paso	Arista considerada	Componentes conexas
Iniciación	—	{1} {2} {3} {4} {5} {6} {7}
1	{1, 2}	{1, 2} {3} {4} {5} {6} {7}
2	{2, 3}	{1, 2, 3} {4} {5} {6} {7}
3	{4, 5}	{1, 2, 3} {4, 5} {6} {7}
4	{6, 7}	{1, 2, 3} {4, 5} {6, 7}
5	{1, 4}	{1, 2, 3, 4, 5} {6, 7}
6	{2,5}	rechazado
7	{4, 7}	{1, 2, 3, 4, 5, 6, 7}

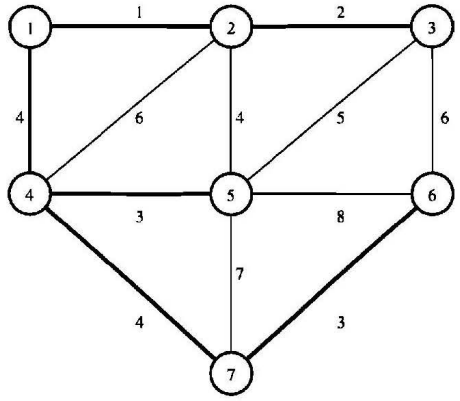

Figura 6.2 Un grafo y su árbol de recubrimiento mínimo

Cuando se detiene el algoritmo, T contiene las aristas seleccionadas {1,2}, {2,3}, {4,5}, {6,7}, {1,4} y {4,7}. El árbol de recubrimiento mínimo se muestra mediante trazos gruesos en la figura 6.2; la longitud total es 17.

Teorema 6.3.2 *El algoritmo de Kruskal halla un árbol de recubrimiento mínimo*

Demostración. La demostración se hace por inducción matemática sobre el número de aristas que hay en el conjunto T. Mostraremos que si T es prometedor entonces sigue siendo prometedor en cualquier fase del algoritmo cuando se le añade una arista adicional. Cuando se detiene el algoritmo, T da una solución de nuestro problema; puesto que también es prometedora, la solución es óptima.

◊ *Base*: el conjunto vacío es prometedor porque G es conexo, y por tanto tiene que existir una solución.

◊ *Paso de inducción*: supongamos que T es prometedor inmediatamente antes de que el algoritmo añada una nueva arista $e = \{u, v\}$. Las aristas de T dividen a los nodos de G en dos o más componentes conexas; el nodo u se encuentra en una de estas componentes, y v está en otra componente distinta. Sea B el conjunto de nodos de esa componente que contiene a u. Ahora:
- el conjunto B es un subconjunto estricto de los nodos de G (puesto que no incluye a v, por ejemplo);
- T es un conjunto prometedor de aristas tal que ninguna arista de T sale de B (porque una arista de T tiene o bien ambas aristas en B, o bien ningúna arista en B, así que por definición no sale de B); y
- e es una de las aristas más cortas que salen de B (porque todas las aristas estrictamente más cortas ya se han examinado, y o bien se han añadido a T, o bien se han rechazado porque tenían los dos extremos en la misma componente conexa.

Entonces se cumplen las condiciones del Lema 6.3.1, y concluimos que el conjunto $T \cup \{e\}$ también es prometedor.

Esto completa la demostración por inducción matemática de que el conjunto T es prometedor en todas las fases del algoritmo, y por tanto cuando se detiene el algoritmo, T no da meramente una solución de nuestro problema, sino una solución óptima.

Para implementar el algoritmo, tenemos que manejar un cierto número de conjuntos, a saber, los nodos de cada componente conexa. Es preciso efectuar rápidamente dos operaciones: *buscar*(x), que nos dice en qué componente conexa se encuentra el nodo x, y *fusionar*(A, B), para fusionar dos componentes conexas. Por tanto, utilizamos estructuras de partición; véase la Sección 5.9. Para este algoritmo, es preferible representar el grafo como un vector de aristas con sus longitudes asociadas, y no como una matriz de distancias; véase el problema 6.9. El algoritmo es el siguiente:


```
función Kruskal(G = ⟨N, A⟩: grafo; longitud: A → ℝ⁺): conjunto de aristas
    {Iniciación}
    Ordenar A por longitudes crecientes
    n ← el número de nodos que hay en N
    T ← Ø {contendrá las aristas del árbol de recubrimiento mínimo}
    Iniciar n conjuntos, cada uno de los cuales contiene un elemento distinto de N
    {bucle voraz}
    repetir
        e ← {u, v} ← arista más corta, aun no considerada
        compu ← buscar(u)
        compv ← buscar(v)
        si compu ≠ compv entonces
            fusionar(compu, compv)
            T ← T ∪ {e}
    hasta que T contenga n - 1 aristas
    devolver T
```

Podemos evaluar el tiempo de ejecución del algoritmo en la forma siguiente. En un grafo con n nodos y a aristas, el número de operaciones está en:

◊ $\Theta(a \log a)$ para ordenar las aristas, lo cual es equivalente a $\Theta(a \log n)$ porque $n - 1 \leq a \leq n(n-1)/2$
◊ $\Theta(n)$ para iniciar los n conjuntos disjuntos
◊ $\Theta(2a\ \alpha\ (2a, n))$ para todas las operaciones *buscar* y *fusionar*, en donde α es la función de crecimiento lento de la Sección 5.9 (esto se sigue de los resultados de la Sección 5.9, puesto que hay como máximo $2a$ operaciones *buscar* y $n - 1$ operaciones *fusionar* en un universo de n elementos), y
◊ $O(a)$ en el caso peor, para las operaciones restantes.

Concluimos que el tiempo total para el algoritmo está en $\Theta(a \log n)$ porque $\Theta(\alpha(2a, n)) \subset \Theta(\log n)$. Aunque esto no cambia el análisis en el caso peor, es preferible mantener las aristas en un montículo invertido (véase la Sección 7.5): de esta manera la arista más corta se encuentra en la raíz del montículo. Esto permite efectuar la inicialización en un tiempo que está en $\Theta(a)$, aun cuando cada búsqueda de un mínimo en el bucle **repetir** requiere ahora un tiempo que está en $\Theta(\log a) = \Theta(\log n)$. Esto resulta especialmente ventajoso si se encuentra el árbol de recubrimiento mínimo en un momento en el que quede por probar un número considerable de aristas. En tales casos, el algoritmo original desperdicia el tiempo ordenando estas aristas inútiles.

6.3.2 Algoritmo de Prim

En el algoritmo de Kruskal, la función de selección escoge las aristas por orden creciente de longitud, sin preocuparse demasiado por su conexión con las aristas seleccionadas anteriormente, salvo que se tiene cuidado para no formar nunca un ciclo. El resultado es un bosque de árboles que crece al azar, hasta que finalmente todas las componentes del bosque se fusionan en un único árbol. En el algoritmo de Prim, por

otra parte, el árbol de recubrimiento mínimo crece de forma natural, comenzando por una raíz arbitraria. En cada fase se añade una nueva rama al árbol ya construido; el algoritmo se detiene cuando se han alcanzado todos los nodos.

Sea B un conjunto de nodos, y sea T un conjunto de aristas. Inicialmente, B contiene un único nodo arbitrario, y T está vacío. En cada paso, el algoritmo de Prim busca la arista más corta posible $\{u, v\}$ tal que $u \in B$ y $v \in N \backslash B$. Entonces añade v a B y $\{u, v\}$ a T. De esta manera las aristas de T forman en todo momento un árbol de recubrimiento mínimo para los nodos de B. Continuamos mientras $B \neq N$. Lo que sigue es un enunciado informal del algoritmo:

```
función Prim(G = ⟨N, A⟩: grafo; longitud A→ℝ⁺): conjunto de aristas.
  {Iniciación}
  T ← ∅
  B ← {un miembro arbitrario de N}
  mientras B ≠ N hacer
    buscar e = {u, v} de longitud mínima tal que
        u ∈ B y v ∈ N \ B
    T ← T ∪ {e}
    B ← B ∪ {v}
  devolver T
```

Para ilustrar este algoritmo, considérese una vez más el grafo de la figura 6.2. Arbitrariamente, seleccionamos el nodo uno como nodo inicial. Ahora el algoritmo podría progresar en la forma siguiente:

Paso	$\{u, v\}$	B
Inicialización	—	{1}
1	{1, 2}	{1, 2}
2	{2, 3}	{1, 2, 3}
3	{1, 4}	{1, 2, 3, 4}
4	{4, 5}	{1, 2, 3, 4, 5}
5	{4, 7}	{1, 2, 3, 4, 5, 7}
6	{7, 6}	{1, 2, 3, 4, 5, 6, 7}

Cuando se detiene el algoritmo, T contiene las aristas seleccionadas {1, 2}, {2, 3}, {1,4}, {4, 5}, {4, 7} y {7, 6}. La demostración de que el algoritmo funciona es parecida a la demostración del algoritmo de Kruskal.

Teorema 6.3.3 *El algoritmo de Prim halla un árbol de recubrimiento mínimo*

Demostración. La demostración es por inducción matemática sobre el número de aristas que hay en el conjunto T. Demostraremos que si T es prometedor en alguna fase del algoritmo,

entonces sigue siendo prometedor al añadir una arista adicional. Cuando se detiene el algoritmo, T da una solución para nuestro problema; puesto que también es prometedora, la solución es óptima.

◊ *Base*: el conjunto vacío es prometedor.

◊ *Paso de inducción*: suponga que T es prometedor inmediatamente antes de que el algoritmo añada una nueva arista $e = \{u, v\}$. Ahora B es un subconjunto estricto de N (porque el algoritmo se detiene cuando $B = N$), T es un conjunto de aristas prometedor por hipótesis de inducción, y e es por definición una de las aristas más cortas que salen de B. Entonces las condiciones del lema 6.3.1 se cumplen, y $T \cup \{e\}$ también es prometedor.

Esto completa la demostración por inducción matemática de que el conjunto T es prometedor en todas las fases del algoritmo. Por tanto, cuando se detiene el algoritmo, T ofrece una solución óptima de nuestro problema.

Para obtener una implementación sencilla en una computadora, supongamos que los nodos de G están numerados de 1 a n, así que $N = \{1, 2, \ldots, n\}$. Supongamos también que hay una matriz simétrica L que da la longitud de todas las aristas, con $L[i, j] = \infty$ si no existe la arista correspondiente. Utilizaremos dos matrices. Para todo nodo $i \in N \setminus B$, *más próximo*$[i]$ proporciona el nodo de B que está más próximo a i, y *distmin*$[i]$ da la distancia desde i hasta este nodo más próximo. Para un nodo $i \in B$, hacemos *distmin*$[i] = -1$. (De esta manera, podemos saber si un nodo está o no en B.) El conjunto B, que recibe el valor inicial arbitrario {1}, no se representa explícitamente; *más próximo*[1] y *distmin*[1] nunca se utilizan. Véase el algoritmo:

```
función Prim(L[1..n, 1..n]): conjunto de aristas
  {Inicialización: sólo el nodo 1 se encuentra en B}
  T ← ∅ {contendrá las aristas del árbol de recubrimiento mínimo}
  para i ← 2 hasta n hacer
    más próximo[i] ← 1
    distmin[i] ← L[i, 1]
  {bucle voraz}
  repetir n − 1 veces
    mín ← ∞
    para j ← 2 hasta n hacer
      si 0 ≤ distmin[j] < mín entonces mín ← distmin[j]
                                       k ← j
    T ← T ∪ {{más próximo[k], k}}
    distmin[k] ← −1 {se añade k a B}
    para j ← 2 hasta n hacer
      si L[j, k] < distmin[j] entonces distmin[k] ← L[j, k]
                                       más próximo[j] ← k
    devolver T
```

El bucle principal del algoritmo se ejecuta n - 1 veces; en cada iteración, el bucle **para** anidado requiere un tiempo que está en $\Theta(n)$. Por tanto, el algoritmo de Prim requiere un tiempo que está en $\Theta(n^2)$.

Se vio anteriormente que el algoritmo de Kruskal requiere un tiempo que está en $\Theta(a \log n)$, en donde a es el número de aristas que hay en el grafo. Para un grafo denso, a tiende a $n(n-1)/2$. En este caso, el algoritmo de Kruskal requiere un tiempo que se encuentra en $\Theta(n^2 \log n)$, y el algoritmo de Prim puede ser mejor. Para un grafo disperso, a tiende a n. En este caso, el algoritmo de Kruskal requiere un tiempo que está en $\Theta(n \log n)$, y el algoritmo de Prim, tal como se ha presentado aquí, es probablemente menos eficiente. Sin embargo, el algoritmo de Prim, al igual que el de Kruskal, se puede implementar utilizando montículos. En este caso —una vez más, como el algoritmo de Kruskal— requiere un tiempo que está en $\Theta(a \log n)$. Existen otros algoritmos más eficientes que el de Prim o el de Kruskal; véase la Sección 6.8.

6.4 GRAFOS: CAMINOS MÍNIMOS

Considere ahora un grafo dirigido $G = \langle N, A \rangle$ en donde N es el conjunto de nodos de G, y A es el conjunto de aristas dirigidas. Cada arista posee una *longitud* no negativa. Se toma uno de los nodos como nodo *origen*. El problema consiste en determinar la longitud del camino mínimo que va desde el origen hasta cada uno de todos los demás nodos del grafo. Tal como sucedía en la Sección 6.3, podríamos hablar igualmente bien acerca del *coste* de una arista, en lugar de mencionar su longitud, y se podría plantear el problema consistente en determinar la ruta más barata desde el origen hasta cada uno de todos los demás nodos.

Este problema se puede resolver mediante un algoritmo voraz que recibe frecuentemente el nombre de *algoritmo de Dijkstra*. El algoritmo utiliza dos conjuntos de nodos, S y C. En todo momento, el conjunto S contiene aquellos nodos que ya han sido seleccionados; como veremos, la distancia mínima desde el origen ya es conocida para todos los nodos de S. El conjunto C contiene todos los demás nodos, cuya distancia mínima desde el origen todavía no es conocida, y que son candidatos a ser seleccionados en alguna etapa posterior. Por tanto, tenemos la propiedad invariante $N = S \cup C$. En un primer momento, S contiene nada más el origen en sí; cuando se detiene el algoritmo, S contiene todos los nodos del grafo y nuestro problema está resuelto. En cada paso seleccionamos aquel nodo de C cuya distancia al origen sea mínima, y se lo añadimos a S.

Diremos que un camino desde el origen hasta algún otro nodo es *especial* si todos los nodos intermedios a lo largo del camino pertenecen a S. En cada fase del algoritmo, hay una matriz D que contiene la longitud del camino especial más corta que va hasta cada nodo del grafo. En el momento en que se desea añadir un nuevo nodo v a S, el camino especial mas corto hasta v es también el más corto de los caminos posibles hasta v (esto se demostrará más adelante). Cuando se detiene el algoritmo, todos los nodos del grafo se encuentran en S, y por tanto todos los caminos desde el origen hasta algún otro nodo son especiales. Consiguientemente, los valores que hay en D dan la solución del problema de caminos mínimos.

Por sencillez, suponemos una vez más que los nodos de G están numerados desde 1 hasta n, así que $N = \{1, 2, \ldots, n\}$. Podemos suponer sin pérdida de generalidad que el nodo uno es el nodo origen. Supongamos también que la matriz L da la longitud de todas las aristas dirigidas: $L[i, j] \geq 0$ si la arista $(i, j) \in A$, y $L[i, j] = \infty$ en caso contrario. Véase a continuación el algoritmo:

```
función Dijkstra(L[1..n, 1..n]): matriz [2..n]
  matriz D[2..n]
  {Iniciación}
  C ← {2, 3,..., n} {S = N \ C sólo existe implícitamente}
  para i ← 2 hasta n hacer D[i] ← L[1, i]
  {bucle voraz}
  repetir n - 2 veces
    v ← algún elemento de C que minimiza D[v]
    C ← C \ {v}  {e implícitamente S ← S ∪ {v}}
    para cada w ∈ C hacer
      D[w] ← mín(D[w], D[v] + L[v, w])
  devolver D
```

El algoritmo procede en la forma siguiente para el grafo de la figura 6.3.

Paso	v	C	D
Inicialización	—	{2, 3, 4, 5}	[50, 30, 100, 10]
1	5	{2, 3, 4}	[50, 30, 20, 10]
2	4	{2, 3}	[40, 30, 20, 10]
3	3	{2}	[35, 30, 20, 10]

Claramente, D no cambiaría si hiciéramos una iteración más para eliminar el último elemento de C. Ésta es la razón por la cual el bucle principal se repite solamente $n - 2$ veces.

Para determinar no solamente la longitud de los caminos mínimos sino también por dónde pasan, se añade una segunda matriz $P[2..n]$, en donde $P[v]$ contiene el número del nodo que precede a v dentro del camino más corto. Para hallar el camino más corto, se siguen los punteros P hacia atrás, desde el destino hacia el origen. Las modificaciones necesarias para el algoritmo son sencillas:

Se inicia $P[i]$ con el valor 1 para $i = 2, 3, \ldots, n$
Se sustituye el contenido del bucle interno **para** por

```
si D[w] > D[v] + L[v, w] entonces  D[w] ← D[v] + L[v, w]
                                   P[w] ← v
```