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
```

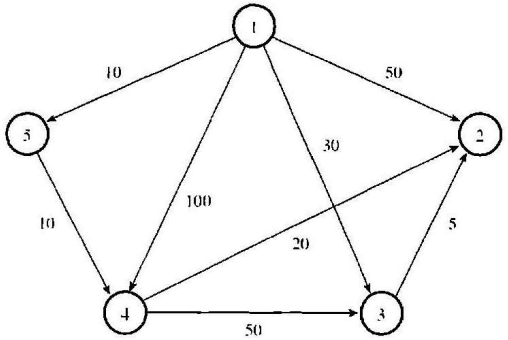

Figura 6.3. Un grafo dirigido

La demostración de que el algoritmo funciona es, una vez más, por inducción matemática.

Teorema 6.4.1 *El algoritmo de Dijkstra halla los caminos más cortos desde un único origen hasta los demás nodos del grafo*

Demostración. Demostraremos por inducción matemática que:

(a) Si un nodo $i \neq 1$ está en S, entonces $D[i]$ da la longitud del camino más corto desde el origen hasta i, y

(b) si un nodo i no está en S, entonces $D[i]$ da la longitud del camino *especial* más corto desde el origen hasta i.

◊ *Base*: inicialmente, sólo el nodo 1, que es el origen, se encuentra en S, así que la situación (a) es cierta sin más demostración. Para los demás nodos, el único camino especial desde el origen es el camino directo, y D recibe valores iniciales en consecuencia. Por tanto la situación (b) también es cierta cuando comienza el algoritmo.

◊ *Hipótesis de inducción*: la hipótesis de inducción es que tanto la situación (a) como la situación (b) son válidas inmediatamente antes de añadir un nodo v a S. Detallamos por separado los pasos de inducción para las situaciones (a) y (b).

◊ *Paso de inducción para la situación (a)*: para todo nodo que ya esté en S antes de añadir v, no cambia nada, así que la situación (a) sigue siendo válida. En cuanto al nodo v, ahora pertenecerá a S. Antes de añadirlo a S, es preciso comprobar que $D[v]$ proporcione la longitud del camino más corto que va desde el origen hasta v. Por hipótesis de inducción, nos da ciertamente la longitud del camino *especial* más corto. Por tanto, hay que verificar que el camino más corto desde el origen hasta v no pase por ninguno de los nodos que no pertenecen a S.

Supongamos lo contrario; esto es, supongamos que cuando se sigue el camino más corto desde el origen hasta v, se encuentran uno o más nodos (sin contar el propio v) que no pertenecen a S. Sea x el primer nodo encontrado con estas características; véase la figura 6.4. Ahora el segmento inicial de esta ruta, hasta llegar a x, es una ruta especial, así que la distancia hasta x es $D[x]$ por la parte (b) de la hipótesis de inducción. Claramente la distancia total hasta v a través de x no es más corta que este valor, porque las longitudes de las aristas son no negativas. Finalmente, $D[x]$ no es menor que $D[v]$, porque el algoritmo ha seleccionado a v antes que a x. Por tanto, la distancia total hasta v a través de x es como mínimo $D[v]$, y el camino a través de x no puede ser más corto que el camino especial que lleva hasta v.

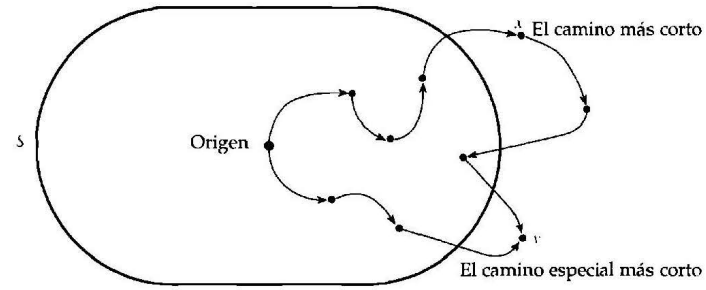

Figura 6.4. El camino más corto hasta v no puede visitar x

Por tanto, hemos verificado que cuando se añade v a S, la parte (a) de la inducción sigue siendo válida.

◊ *Paso de inducción para la situación* (b): considérese ahora un nodo w, distinto de v, que no se encuentre en S. Cuando v se añade a S, hay dos posibilidades para el camino especial más corto desde el origen hasta w: o bien no cambia, o bien ahora pasa a través de v (y posiblemente, también a través de otros nodos de S). En el segundo caso, sea x el último nodo de S visitado antes de llegar a w. La longitud de este camino es $D[x] + L[x, w]$. Parece a primera vista que para calcular el nuevo valor de $D[w]$ debiéramos comparar el valor anterior de $D[w]$ con los valores de $D[x] + L[x, w]$ para todo nodo x de S (incluyendo a v). Sin embargo, para todos los nodos x de S salvo v, esta comparación se ha hecho cuando se añadió x a S, y $D[x]$ no ha variado desde entonces. Por tanto, el nuevo valor de $D[w]$ se puede calcular sencillamente comparando el valor anterior con $D[v] + L[v, w]$.

Puesto que el algoritmo hace esto explícitamente, asegura que la parte (b) de la inducción siga siendo cierta también cuando se añade a S un nuevo nodo v.

Para completar la demostración de que el algoritmo funciona, obsérvese que cuando se detenga el algoritmo, todos los nodos menos uno estarán en S (aun cuando el conjunto S no se construye explícitamente). En ese momento queda claro que el camino más corto desde el origen hasta el nodo restante es un camino especial.

Análisis del algoritmo. Supongamos que el algoritmo de Dijkstra se aplica a un grafo que posee n nodos y a aristas. Utilizando la representación sugerida hasta el momento, este caso se da en la forma de una matriz $L[1..n, 1..n]$. La iniciación requiere un tiempo que está en $O(n)$. En una implementación directa, la selección de v dentro del bucle **(repetir)** requiere examinar todos los elementos de C, así que examinaremos $n-1, n-2,..., 2$ valores de D en las sucesivas iteraciones, dando un tiempo total que está en $\Theta(n^2)$. El bucle **para (desde)** interno realiza $n-2, n-3,\ldots,$ 1 iteraciones dando también un tiempo total que está en $\Theta(n^2)$. El tiempo requerido por esta versión del algoritmo está, por tanto, en $\Theta(n^2)$.

Si $a << n^2$, podríamos tener la esperanza de evitar examinar las muchas entradas que contienen ∞ en la matriz L. Teniendo esto en cuenta, sería preferible representar el grafo mediante una matriz de n listas, que diera para cada nodo su distancia directa a los nodos adyacentes (como el tipo *listagraf* de la Sección 5.4). Esto nos permite ahorrar tiempo en el bucle **para** interno, porque sólo hay que considerar aquellos nodos w que sean adyacentes a v; ahora bien, ¿cómo podemos evitar que se necesite un tiempo en $\Omega(n^2)$ para determinar sucesivamente los $n-2$ valores que va tomando v?

La respuesta consiste en utilizar un montículo invertido que contiene un nodo para cada elemento v de C, ordenado por el valor de $D[v]$. De esta manera, el elemento v de C que minimiza $D[v]$ siempre se encontrará en la raíz. La iniciación del montículo requiere un tiempo que está en $\Theta(n)$. La instrucción «$C \leftarrow C \setminus \{v\}$» consiste en eliminar la raíz del montículo, lo cual requiere un tiempo que se encuentra en $O(\log n)$. En cuanto al bucle **para** interno, ahora consiste en estudiar, para cada elemento w de C adyacente a v, si $D[v] + L[v, w]$ es menor que $D[w]$. En tal caso, debemos modificar $D[w]$ y filtrar a w dentro del montículo, lo cual requiere una vez más un tiempo que está en $O(\log n)$. Esto no sucede más que una vez para cada arista del grafo.

Para resumir, tenemos que eliminar la raíz del montículo exactamente $n-2$ veces, y tenemos que flotar un máximo de a nodos, lo cual da un tiempo total que está en $\Theta((a+n)\log n)$. Si el grafo es conexo, $a \geq n-1$, y el tiempo se encuentra en $\Theta(a \log n)$. La implementación sencilla es preferible, por tanto, si el grafo es denso, mientras que resulta preferible utilizar un montículo si el grafo es disperso. Si $a \in \Theta(n^2/\log n)$, entonces la elección de la representación puede depender de la implementación concreta utilizada. El Problema 6.16 sugiere una forma de acelerar el algoritmo, empleando un montículo k-ario con un valor de k bien seleccionado; se conocen otros algoritmos aún más rápidos; véanse el Problema 6.17 y la Sección 6.8.

6.5 EL PROBLEMA DE LA MOCHILA (1)

Este problema surge de distintas maneras. En este capítulo examinaremos la versión más sencilla; en la Sección 8.4 se presentará una variante más difícil.

Nos dan n objetos y una mochila. Para $i = 1, 2,..., n$, el objeto i tiene un peso positivo w_i y un valor positivo v_i. La mochila puede llevar un peso que no sobrepase W. Nuestro objetivo es llenar la mochila de tal manera que se maximice el valor de los objetos transportados, respetando la limitación de capacidad impuesta. En esta primera versión del problema, suponemos que se pueden romper los objetos en trozos más pequeños, de manera que podamos decidir llevar solamente una fracción x_i del objeto i, con $0 \leq x_i \leq 1$. (Si no se nos permite romper los objetos, el problema es mucho más difícil.) En este caso, el objeto i contribuye en $x_i w_i$ al peso total de la mochila, y en $x_i v_i$ al valor de la carga. En símbolos, el problema se puede enunciar en la forma siguiente:

$$\text{maximizar} \sum_{i=1}^{n} x_i v_i \text{ con la restricción } \sum_{i=1}^{n} x_i w_i \leq W$$

donde $v_i > 0$, $w_i > 0$ y $0 \leq x_i \leq 1$ para $1 \leq i \leq n$. Aquí las condiciones que afectan a v_i y a w_i son restricciones sobre el problema; las de x_i son restricciones sobre la solución. Utilizaremos un algoritmo voraz para resolver el problema. En términos de nuestro esquema general, los candidatos son los diferentes objetos, y la solución es un vector $(x_1,...,x_n)$ que nos dice qué fracción de cada objeto hay que incluir. Una solución será factible cuando se respetan las restricciones indicadas anteriormente, y la función objetivo es el valor total de los objetos que están en la mochila. Queda por ver qué debemos tomar como función de selección.

Si $\Sigma_{i=1}^{n} w_i \leq W$, está claro que lo óptimo es meter todos los objetos dentro de la mochila. Por tanto, podemos asumir que en los casos interesantes del problema, es $\Sigma_{i=1}^{n} w_i > W$. También está claro que una solución óptima debe llenar exactamente la mochila, porque en caso contrario se podría añadir una fracción de alguno de los objetos restantes e incrementar el valor de la carga. Por tanto, en una solución óptima $\Sigma_{i=1}^{n} x_i w_i = W$. Puesto que esperamos encontrar un algoritmo voraz que funcione, nuestra estrategia general consistirá en seleccionar cada objeto por turno en algún orden adecuado, poner la mayor fracción posible del objeto seleccionado en la mochila, y detenernos cuando la mochila esté llena. Véase el algoritmo:

```
función mochila(w[1..n], v[1..n], W): matriz [1..n]
  {Inicialización}
  para i = 1 hasta n hacer x[i] ← 0
  peso ← 0
  {bucle voraz}
  mientras peso < N hacer
    i ← el mejor objeto restante {ver más abajo}
    si peso + w[i] ≤ W entonces   x[i] ← 1
                                  peso ← peso + w[i]
                          sino    x[i] ← (W – peso) / w[i]
                                  peso ← W
  devolver x
```