CAPÍTULO

14 Grafos y caminos

Este capítulo estudia los *grafos* y muestra cómo resolver un problema concreto de gran interés, denominado cálculo de caminos mínimos. Éste es un problema fundamental en computación ya que muchas aplicaciones interesantes pueden modelarse mediante un grafo. Ejemplos de cálculo del camino mínimo son: calcular la ruta más rápida en un transporte, o dirigir el correo electrónico a través de una red de computadores. Este capítulo estudia diversas variantes del problema que dependen de cómo interpretemos el significado de «mínimo» y la clase de propiedades que tengan los grafos. Los problemas de búsqueda del camino mínimo son interesantes porque, aunque los algoritmos son sencillos, también son lentos con grafos de gran tamaño a menos que se tenga cuidado con las estructuras de datos elegidas.

En este capítulo veremos:

- Definiciones formales de grafo y de sus componentes.
- Las estructuras de datos utilizadas para representar un grafo.
- Algoritmos para resolver diversas variantes del problema del camino mínimo, dando implementaciones completas en Java.

14.1 Definiciones

Un grafo $G = (V, E)$ está formado por un conjunto de vértices, V, y un conjunto de aristas, E. Cada arista es un par (v, w), donde $v, w \in V$. En ocasiones los vértices se denominan *nodos*, y las aristas *arcos*. Si los pares que definen las aristas están ordenados, se dice que el grafo es *dirigido*. Los grafos dirigidos se denominan en ocasiones *digrafos*. En un digrafo, el vértice w es *adyacente* al vértice v si y sólo si $(v, w) \in E$. Algunas veces las aristas tienen una tercera componente, denominada *peso* o *coste*. En este capítulo todos los grafos que consideramos serán dirigidos.

Un *grafo* está formado por un conjunto de vértices y un conjunto de aristas que conectan los vértices. Si las aristas son pares ordenados, el grafo es *dirigido*.

w es adyacente a v si hay una arista de v a w.

El grafo de la Figura 14.1 tiene los siguientes 7 vértices:

$$V = \{V_0, V_1, V_2, V_3, V_4, V_5, V_6\}$$

y 12 aristas:

$$E = \begin{Bmatrix} (V_0, V_1, 2), (V_0, V_3, 1), (V_1, V_3, 3), (V_1, V_4, 10) \\ (V_3, V_4, 2), (V_3, V_6, 4), (V_3, V_5, 8), (V_3, V_2, 2) \\ (V_2, V_0, 4), (V_2, V_5, 5), (V_4, V_6, 6), (V_6, V_5, 1) \end{Bmatrix}.$$

Los vértices adyacentes a V_3 son: V_2, V_4, V_5, V_6. En este grafo $|V| = 7$ y $|E| = 12$, donde $|S|$ representa el cardinal de S.

Un *camino* es una secuencia de vértices conectados entre sí por aristas.

Un *camino* en un grafo es una secuencia de vértices w_1, w_2, ..., w_N tal que $(w_i, w_{i+1}) \in E$ para $1 \leqslant i < N$. La *longitud* de dicho camino es el número de aristas en el camino, es decir, $N - 1$. Esta longitud se denomina *longitud del camino sin pesos*. La *longitud del camino con pesos* es la suma de los costes de las aristas en el camino. Por ejemplo, V_0, V_3, V_5 es un camino desde el vértice V_0 al V_5. La longitud del camino es dos aristas, y la longitud del camino con pesos es 9 (éste es el camino más corto de V_0 a V_5). Sin embargo, si el coste es importante, el camino con pesos más corto entre estos vértices tiene coste 6 y es V_0, V_3, V_6, V_5. Consideramos en particular caminos de un vértice a él mismo. Cuando un camino tal no contiene aristas, su longitud es 0; ésta es la forma más conveniente de tratar este caso especial. Un *camino simple* es un camino en el que todos los vértices son distintos, excepto en lo que refiere al primero y al último, que sí pueden ser iguales.

Un *ciclo* en un grafo dirigido es un camino que empieza y termina en un mismo vértice y que contiene, al menos, una arista.

Un *ciclo* en un grafo dirigido es un camino de longitud al menos 1 que cumple $w_1 = w_N$; este ciclo es simple si el camino lo es. Un grafo dirigido acíclico, a veces referenciado por su abreviatura, GDA, es un grafo dirigido sin ciclos.

Un ejemplo de la vida real que puede modelarse mediante un grafo es un sistema de aeropuertos. Cada aeropuerto es un vértice. Dos vértices están conectados por una arista si hay un vuelo sin paradas entre los correspondientes aeropuertos. La arista puede tener un coste, representando el tiempo, la distancia o el coste del vuelo. Generalmente, la existencia de una arista (v, w) implicará la existencia de la arista (w, v). Pero es razonable suponer que el coste de estas aristas podría ser diferente, ya que volar en sentido diferente podría llevar más tiempo (dependiendo de los vientos dominantes) o costar más (dependiendo de las tarifas locales). Un problema natural que nos gustaría resolver con cierta celeridad es el de determinar el mejor vuelo entre dos aeropuertos, donde «mejor» podría significar el camino con menor número de aristas, o podría entenderse con respecto a alguna de las medidas de peso (distancia, coste, etc.).

Un segundo ejemplo de la vida real que podemos modelar con un grafo es el envío de mensajes de correo electrónico a través de una red de computadores. Los vértices representarían computadores, las aristas enlaces entre pares de computadores y el coste de cada arista sería el coste de la comunicación (pasos de teléfono por megabyte), coste de retraso (segundos por megabyte), o combinaciones de éstos y otros factores.

En la mayoría de los grafos, hay como mucho una arista entre un vértice v y otro w (esto incluye el caso en el que hay una arista en cada dirección). En consecuencia, $|E| \leqslant |V|^2$. Cuando la mayoría de los vértices están presentes, tenemos $|E| = \Theta(|V|^2)$, y el grafo se denomina *denso* (pues tiene un gran número de aristas).

En la mayoría de las aplicaciones, los grafos son *dispersos* en vez de densos. Por ejemplo, en el modelo de los vuelos, no esperamos que haya vuelos directos entre cada par de aeropuertos. Por el contrario, se dará el caso en el que algunos pocos aeropuertos están muy bien conectados y el resto tiene relativamente pocos vuelos. De la misma forma, en un sistema complejo de transporte de multitudes que involucre trenes y autobuses, desde una estación se pueden alcanzar directamente, lo que estará representado por una arista, sólo unas pocas estaciones. También, en una red de computadores, la mayoría de los computadores están conectadas a unos pocos computadores locales. Por tanto, en la mayoría de los casos, el grafo es relativamente *disperso*, teniéndose $|E| = \Theta(|V|)$, o quizás algo más (no

hay una definición estándar de disperso). Por tanto, es particularmente importante que los algoritmos que desarrollemos sean eficientes para grafos dispersos.

14.1.1 Representación

Una *matriz de adyacencia* representa un grafo utilizando un espacio cuadrático.

La primera cuestión a considerar es cómo se representa un grafo internamente. Supongamos que los vértices se numeran de forma secuencial empezando en 0, como sugiere el grafo de la Figura 14.1. Una forma sencilla de representar un grafo es utilizar una matriz bidimensional. Esta representación se denomina *matriz de adyacencia*. Para cada arista (v, w), `a[v][w]` representa el coste de la arista; las aristas inexistentes se pueden representar mediante un `INFINITO` lógico. La inicialización del grafo parece requerir que la matriz de adyacencia completa se inicialice a `INFINITO` y, después, para cada arista se actualice la entrada apropiada de la matriz. En este marco, la inicialización conllevaría un tiempo $O(|V|^2)$. Aunque es posible evitar el tiempo cuadrático de inicialización (*véase* Ejercicio 14.3), el coste en espacio será aún $O(|V|^2)$, que resulta razonable para grafos densos pero es totalmente inaceptable para grafos dispersos.

Mediante *listas de adyacencia* se representa un grafo utilizando un espacio lineal.

Para grafos dispersos, una representación mejor es la suministrada por las *listas de adyacencia*. Para cada vértice, se mantiene una lista enlazada de todos sus vértices adyacentes. La representación mediante listas de adyacencia del grafo de la Figura 14.1 se muestra en la Figura 14.2. Ya que cada arista aparece en un nodo de una lista, el número de nodos en las listas es exactamente igual al número de aristas. En consecuencia, se utiliza un espacio $O(|E|)$ para almacenar los nodos. Ya que tenemos $|V|$ listas, también se necesita un espacio adicional $O(|V|)$. Si asumimos que todo vértice está en alguna arista, el número de aristas es, al menos, $\lceil |V|/2 \rceil$, por lo que se puede descartar todo término $O(|V|)$ cuando está presente un término $O(|E|)$. Por consiguiente, decimos que el espacio necesario es $O(|E|)$, o lineal con respecto al tamaño del grafo.

Las listas de adyacencia pueden construirse en tiempo lineal a partir de la lista de aristas.

Las listas de adyacencia pueden construirse en tiempo lineal a partir de la lista de aristas. Partimos de todas las listas vacías, y cuando nos encontramos con una arista $(v, w, c_{v,w})$, añadimos a la lista de adyacencia de v una entrada formada por w y el coste $c_{v,w}$. La inserción puede realizarse en cualquier punto de la lista, pero resultaría especialmente conveniente hacerlo al principio de la lista, para así garantizar tiempo constante. De esta forma cada arista puede insertarse en tiempo constante, por lo que la estructura de listas de adyacencia puede construirse en tiempo lineal. Observe que cuando insertamos una arista no comprobamos si ya está presente. Esto no puede hacerse en tiempo constante (si utilizamos una lista enlazada sin más), y la comprobación acabaría con la cota lineal en tiempo de la construcción.

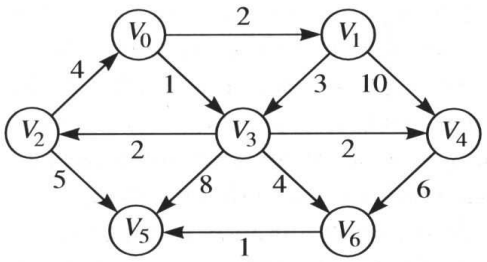

Figura 14.1 Grafo dirigido.

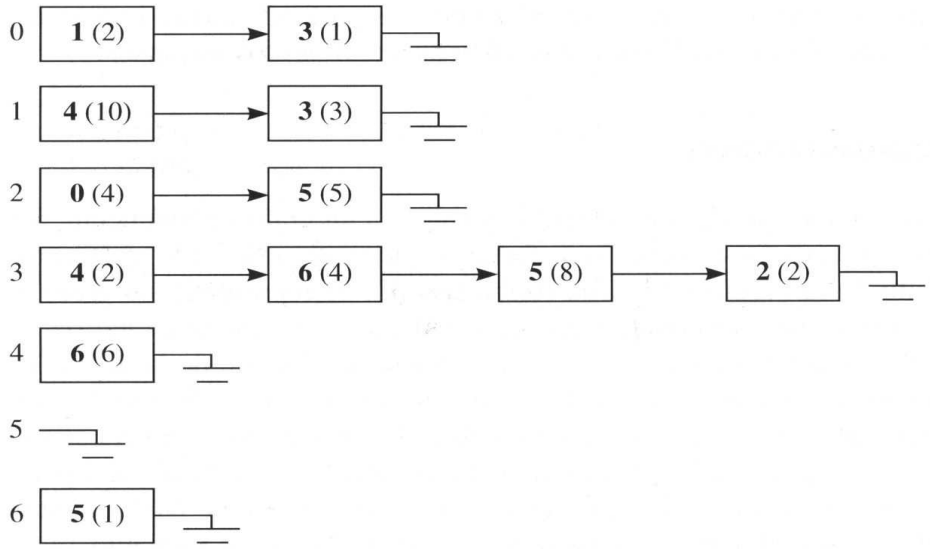

Figura 14.2 Representación con listas de adyacencia del grafo de la Figura 14.1; los nodos en la lista *i* representan los vértices adyacentes a *i* y el coste de la arista correspondiente.

En cualquier caso, esto no tiene gran importancia, pues la mayoría de los algoritmos que funcionan con grafos seguirán funcionando con *multigrafos* en los que hay dos o más aristas de diferentes costes conectando un par de vértices, aunque en algún caso habría que tomar ciertas precauciones al respecto.

Se puede utilizar un diccionario para convertir nombres de vértices en números internos.

En la mayoría de las aplicaciones de la vida real los vértices tienen nombres, en vez de estar denotados por números consecutivos. En tal caso necesitamos proporcionar una forma de transformar dichos nombres en números. La forma más fácil de hacer esto es proporcionar un *diccionario* en el que se asigna a cada nombre de vértice un número interno en el rango de 0 a $|V| - 1$ (el número de vértices se determina al ir ejecutando el programa). Los números internos se van asignando según se va leyendo el grafo. El primer número asignado es el cero. Cada vez que se introduce una arista, comprobamos si se le ha asignado número a cada uno de los dos vértices, viendo si están en el diccionario. Si es así, utilizamos el número interno. En otro caso, asignamos al vértice el siguiente número disponible e insertamos en el diccionario el nombre del vértice junto al número. Con esta transformación, todos los algoritmos sobre grafos utilizarán solamente los números internos.

Ocasionalmente, necesitaremos mostrar el nombre real del vértice y no el número interno, por lo que deberemos guardar, para cada número interno, el correspondiente nombre de vértice. Una forma de hacer esto es guardar una cadena de caracteres para cada vértice. Esta técnica se utilizará más adelante en el capítulo para implementar la clase `Grafo`. Esta clase y los algoritmos de búsqueda de camino mínimo requieren varias estructuras de datos: listas enlazadas, colas, tablas hash y colas de prioridad. Las colas y colas de prioridad se utilizan en varios cálculos de caminos mínimos. Las listas enlazadas y tablas hash se utilizan para representar el grafo. En particular, una tabla hash implementa el diccionario.

Antes de mostrar el esqueleto de la clase `Grafo`, examinemos la Figura 14.3, que muestra cómo será representado nuestro grafo. Como indica la tabla *Entrada*, el usuario proporciona una lista de aristas, una por línea. Al ir a ejecutar el

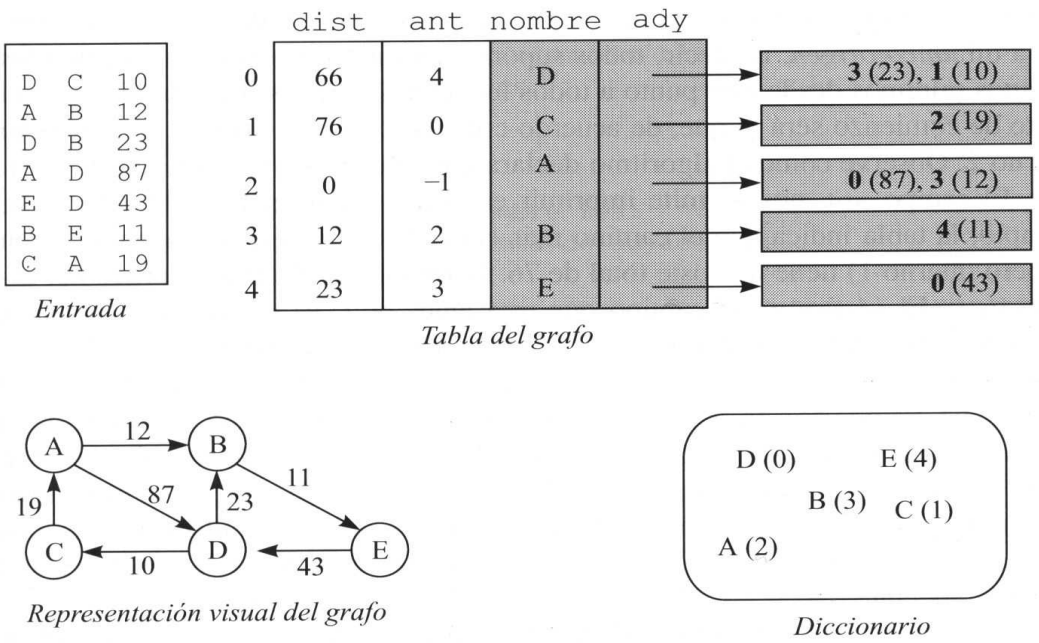

Figura 14.3 Estructuras de datos utilizadas en el cálculo de los caminos mínimos, con un grafo de entrada tomado de un fichero. El camino con pesos más corto desde `A` hasta `C` es (`A`, `B`, `E`, `D`, `C`), cuyo coste es 76.

algoritmo no conocemos los nombres de ningún vértice, ni cuántos vértices hay, ni cuántas aristas. Utilizamos dos estructuras de datos para representar el grafo: un diccionario y una tabla. Mantener un diccionario nos permitirá determinar el número interno de cada vértice. Por ejemplo, ya que `D` es el primer vértice en el fichero de entrada, le será asignado el número 0. `C` es el segundo vértice en el fichero de entrada, por lo que se le asignará el número 1. La otra estructura de datos es una gran tabla que almacena información sobre todos los vértices. Como es evidente a partir de la Figura 14.3, la *Tabla del grafo* mantiene cuatro informaciones por cada vértice:

- `dist`: la longitud del camino más corto (con peso o sin peso, dependiendo del algoritmo) desde el vértice de comienzo hasta este vértice. Este valor es calculado por el algoritmo de búsqueda del camino mínimo.
- `ant`: el vértice anterior en el camino más corto a este vértice.
- `nombre`: el nombre correspondiente a este vértice. Se fija cuando el vértice se coloca en el diccionario y no cambiará nunca. Ninguno de los algoritmos de búsqueda del camino mínimo lo utilizará. Sólo se utiliza al imprimir el camino final.
- `ady`: una lista de vértices adyacentes. Se crea al leer el grafo. Ninguno de los algoritmos de búsqueda del camino mínimo modificará dichas listas.

Para ser más específicos, en la Figura 14.3 los elementos sombreados no serán modificados por ninguno de los cálculos del algoritmo de camino mínimo. Estos elementos representan el grafo de entrada y no cambiarán a menos que cambie el grafo (quizás por la posterior incorporación o eliminación de aristas). Los elementos que no están sombreados serán computados por los algoritmos de búsqueda del camino mínimo. Antes del cálculo, podemos suponer que están sin inicializar.

Todos los algoritmos que estudiaremos corresponden a un *único origen*. Calculan los caminos más cortos desde un punto de origen hasta todos los vértices. El campo `ant` puede utilizarse para construir cada camino mínimo.

Los algoritmos de búsqueda del camino mínimo que estudiaremos corresponden a un *único origen*. Es decir, todos suponen un punto de origen y calculan los caminos mínimos desde este punto a todos los demás vértices. En este ejemplo, el punto de comienzo será `A`, que, de acuerdo con el diccionario, tiene como número interno 2. Observe cómo el algoritmo declara que el camino más corto hasta `A` vale 0. El campo `ant` nos permite imprimir el camino, y no sólo su longitud. Por ejemplo, la tabla indica que el camino más corto desde el vértice origen a `C` (con número interno 1) tiene un coste total de 76. Obviamente, el último vértice del camino es `C`. El vértice en el camino que está antes de `C` es el vértice 0, o sea `D`. Antes de `D` está el vértice 4, que es el `E`. Antes del `E` está el vértice 3, es decir, `B`. Y antes de `B` está el vértice 2, es decir, `A`, que es el origen. Retrocediendo con el campo `ant` vamos construyendo el camino mínimo. Aunque esta traza nos da el camino en orden inverso, es sencillo invertirlo. El resto de esta sección describe cómo se construye la parte sombreada de la tabla del grafo y presenta el método que imprime un camino mínimo, asumiendo que ya se han calculado los campos `dist` y `ant`. Los algoritmos para rellenar los caminos mínimos se discutirán uno a uno más adelante.

Los elementos en la lista de adyacencia consisten en el número interno del vértice adyacente y el coste de la arista. Cada entrada de la tabla del grafo es de tipo `Vertice`.

La Figura 14.4 muestra el elemento básico que se coloca en las listas de adyacencia: un número interno de vértice y el coste de la correspondiente arista. Asumimos que la clase `Grafo` utiliza un valor de tipo `String` para el nombre del vértice y un valor de tipo `int` para el coste de la arista. La clase `Vertice` se muestra en la Figura 14.5. Se proporciona un campo adicional denominado `extra`, que tiene diferentes usos según el algoritmo. Todos los demás elementos siguen la descripción anterior. El único método es el constructor que inicializa la lista enlazada.

Ya estamos preparados para examinar el esqueleto de la clase `Grafo`. Su parte pública se muestra en la Figura 14.6 y su parte privada en la Figura 14.7. En las líneas de la 33 a la 35 se declaran algunas constantes. La línea 35 almacena el valor de `INFINITO` (dividimos por 3 para que `INFINITO + INFINITO` no esté fuera de rango). A continuación vienen los atributos.

`dicVertices` almacena el diccionario. Implementamos el diccionario utilizando una tabla hash que almacena un `ElementoHash` que contiene el nombre de un vértice y su número, pero en la que se busca basándonos en el nombre del vér-

```
1  /**
2   * Esta clase representa el elemento básico
3   * de las listas de adyacencia.
4   */
5  class Arista
6  {
7                              // El primer vértice está implícito
8      public int dest;        // Segundo vértice de la arista
9      public int coste;       // Coste de la arista
10
11     public Arista( int d, int c )
12     {
13         dest=d;
14         coste=c;
15     }
16 }
```

Figura 14.4 El elemento básico almacenado en una lista de adyacencia.

```
1  /**
2   * Esta clase representa el elemento básico
3   * almacenado para un vértice.
4   */
5  class Vertice
6  {
7      String nombre; // El nombre real
8      Lista ady;     // La lista de adyacencia
9
10     int dist;      // Coste (después de ejecutar el algoritmo)
11     int ant;       // Vértice anterior en el camino mínimo
12     int extra;     // Variable extra usada por los algoritmos
13
14     Vertice( String nom )
15     {
16         nombre=nom; // Nombre compartido en tabla hash
17         ady =new ListaEnlazada( ); // Nueva lista
18     }
19 }
```

Figura 14.5 La clase `Vertice` almacena información de cada vértice.

```
1  import EstructurasDatos.*;
2  import Soporte.*; import Soporte.Comparable;
3  import Excepciones.*;
4  import java.util.StringTokenizer;
5  import java.io.*;
6
7  // Clase Grafo: calcula caminos mínimos
8  //
9  // CONSTRUCCIÓN: sin inicializador
10 //
11 // ******************OPERACIONES PÚBLICAS**********************
12 // void insArista( String orig, String dest, int coste )
13 //                        --> Inserta arista adicional
14 // boolean procesarPeticion( BufferedReader in )
15 //                        --> Ejecuta varios alg. camino mínimo
16 // ********************ERRORES*******************************
17 // Se hace alguna comprobación de errores sobre si el grafo
18 // es correcto, los parámetros de procesarPeticion son
19 // vértices, y para estar seguros de que el grafo cumple las
20 // propiedades requeridas por cada algoritmo
21
22 /**
23  * Clase Grafo: calcula caminos mínimos.
24  */
25 public class Grafo
26 {
27     public Grafo( )
28       { /* Figura 14.8 */ }
29     public void insArista( String orig, String dest, int coste )
30       { /* Figura 14.12 */ }
31     public boolean procesarPeticion ( BufferedReader in )
32       { /* Figura 14.16 */ }
```

Figura 14.6 Esqueleto de la clase `Grafo` (parte 1: la parte pública).

```
33      private static final int TAMANYO_TABLA_INI = 50;
34      private static final int VERTICE_NULO      = -1;
35      private static final int INFINITO          = 2147483647 / 3;
36
37      private TablaHash dicVertices; // Almacena números internos
38      private Vertice [ ] tabla;     // El vector tabla
39      private int numVertices;       // Número actual de vértices
40
41      private void duplicarVectorTabla( )
42        { /* muy usual, no mostrado */ }
43      private int insNodo( String nombreVertice )
44        { /* Figura 14.10 */ }
45      private void insAristaInterna( int orig, int dest, int coste )
46        { /* Figura 14.11 */ }
47      private void limpiarDatos( )
48        { /* Figura 14.13 */ }
49
50          // Varios algoritmos de caminos mínimos que
51          // requieren un número interno para empezar
52      private void sinPesos( int nodoOrig )
53        { /* Figura 14.24 */ }
54      private boolean dijkstra( int nodoOrig )
55        { /* Figura 14.29 */ }
56      private boolean negativos( int nodoOrig )
57        { /* Figura 14.31 */ }
58      private boolean aciclico( int nodoOrig )
59        { /* Figura 14.34 */ }
60  }
```

Figura 14.7 Esqueleto de la clase `Grafo` (parte 2: la parte privada).

tice. Esto se discutirá brevemente al estudiar la implementación. La línea 38 es la tabla del grafo, `tabla`. El número de vértices en la tabla se almacena en `numVertices` (línea 39). El resto de la clase proporciona métodos para realizar la inicialización, añadir vértices, imprimir el camino más corto y realizar varios cálculos de caminos mínimos. Cada rutina se discute al estudiar su implementación.

Lo primero es el constructor. La Figura 14.8 muestra que el constructor inicializa el número de vértices a 0 y construye la `tabla`. Se crea un diccionario vacío.

El diccionario se implementa utilizando una tabla hash.

Para implementar el diccionario, declaramos una clase denominada `ElementoHash` que almacenará el nombre del vértice, `nombre`, y su número interno, `rango`. La igualdad y la función hash se realizan solamente a partir del campo `nombre`. Las implementaciones se muestran en la Figura 14.9.

```
1  /**
2   * Constructor.
3   */
4  public Grafo( )
5  {
6      numVertices = 0;
7      tabla       = new Vertice[ TAMANYO_TABLA_INI ];
8      dicVertices = new TablaExploracionCuadratica( );
9  }
```

Figura 14.8 Constructor de la clase `Grafo`.

```
1  /**
2   * Esta clase representa la entrada básica
3   * en el diccionario de vértices.
4   * Implementa el interfaz Hashable proporcionando
5   * las funciones hash y equals.
6   */
7  class ElementoHash implements Hashable
8  {
9      public String nombre;          // El nombre real
10     public int rango;              // El número asignado
11
12     public ElementoHash( )
13       { this( null ); }
14
15     public ElementoHash( String nom )
16       { nombre=nom; }
17
18     public int hash( int tamanyoTabla )
19       { return TablaExploracionCuadratica.hash( nombre, tamanyoTabla ); }
20
21     public boolean equals( Object lder )
22       { return nombre.equals( ((ElementoHash) lder).nombre ); }
23 }
```

Figura 14.9 `ElementoHash` utilizado para implementar el diccionario.

```
1  /**
2   * Si nombreVertice ya es un vértice, devuelve su
3   * número interno. En otro caso, lo añade como nuevo vértice,
4   * y devuelve su nuevo número interno.
5   */
6  private int insNodo( String nombreVertice )
7  {
8      ElementoHash hashV = new ElementoHash( nombreVertice );
9      ElementoHash resultado;
10
11     try
12     {
13         resultado = (ElementoHash) dicVertices.buscar( hashV );
14         return resultado.rango;
15     }
16     catch( ElementoNoEncontrado e )
17     {
18         // Vértice nuevo
19         hashV.rango = numVertices;
20         hashV.nombre = new String( nombreVertice );
21         dicVertices.insertar( hashV );
22
23         if( numVertices == tabla.length )
24             duplicarVectorTabla( );
25         tabla[ numVertices ] = new Vertice( hashV.nombre );
26         return numVertices++;
27     }
28 }
```

Figura 14.10 Rutina `insNodo`, devuelve el número interno de `nombreVertice`.

insNodo devuelve el número interno correspondiente al parámetro nombreVertice. Si nombreVertice no se ha encontrado todavía, se inserta en el diccionario y se inicializa su entrada en el vector tabla.

insNodo, mostrado en la Figura 14.10, devuelve el número interno del vértice, correspondiente al parámetro nombreVertice. El nombre de la rutina refleja el hecho de que si nombreVertice no se ha encontrado todavía (lo que viene reflejado por su ausencia en el diccionario), se le asigna el próximo número interno disponible y se inserta en el diccionario, inicializándose también su entrada en el vector tabla.

El procedimiento empieza con una consulta a la tabla hash. Para hacer esto, debemos crear primero, en la línea 8, un objeto de la clase ElementoHash, hashV, inicializándolo con nombreVertice (utilizando un constructor apropiado).

Una vez construido hashV, podemos realizar una búsqueda en la tabla hash y almacenar el valor devuelto en resultado. Si la búsqueda tiene éxito, podemos devolver el campo rango de resultado, y acabamos. En otro caso, hemos encontrado un vértice nuevo.

En la línea 19, le asignamos a ese nuevo vértice como número interno numVertices, el cual representa el próximo índice disponible en la tabla (porque el índice empieza en 0).

En la línea 21, realizamos una inserción en la tabla hash. Observe cómo creamos un nuevo objeto de la clase String que será referenciado por la tabla hash y por el vector tabla. Se construye la correspondiente entrada Vertice, y se añade a tabla en la línea 25. Si la tabla del grafo tabla se llena, en la línea se duplica su tamaño. La línea 26 completa la rutina devolviendo numVertices (el número interno asignado a nombreVertice) e incrementando numVertices.

Las aristas se añaden mediante inserciones en las listas de adyacencia adecuadas.

La rutina de la Figura 14.11 añade una arista cuyos vértices vienen dados por números internos. Ésta es una rutina sencilla porque lo único que precisa es crear un objeto de la clase Arista con el vértice destino y el coste, e insertarlo en la lista de adyacencias correspondiente al vértice de origen. La rutina de uso público se muestra en la Figura 14.12. insArista inserta una arista cuyos vértices vienen

```
1  /**
2   * Añade una arista dados los números internos de sus vértices.
3   */
4  private void insAristaInterna( int orig, int dest, int coste )
5  {
6      ListaIter p = new ListaEnlazadaIter( tabla[ orig ].ady );
7      try
8        { p.insertar( new Arista( dest, coste ) ); }
9      catch( ElementoNoEncontrado e ) { } // No puede ocurrir
10 }
```

Figura 14.11 Añade al grafo la arista (orig, dest, coste) insertándola dentro de la lista de adyacencia de orig; orig y dest son números internos.

```
1  /**
2   * Añade la arista ( orig, dest, coste ) al grafo.
3   */
4  public void insArista( String orig, String dest, int coste )
5  {
6      insAristaInterna( insNodo( orig ), insNodo( dest ), coste );
7  }
```

Figura 14.12 La misma rutina que insAristaInterna, pero aquí orig y dest son objetos de la clase String.

dados por valores de tipo `String`. Ésta es otra rutina sencilla pues se limita a llamar a `insNodo` para obtener los números internos correspondientes. Tras ello se llama a `insAristaInterna`.

```
1  /**
2   * Inicializa la tabla.
3   */
4  private void limpiarDatos( )
5  {
6      for( int i = 0; i < numVertices; i++ )
7      {
8          tabla[ i ].dist = INFINITO;
9          tabla[ i ].ant = VERTICE_NULO;
10         tabla[ i ].extra = 0;
11     }
12 }
```

Figura 14.13 Rutina para inicializar los campos de la `tabla`, que se usan en los algoritmos de búsqueda de caminos mínimos.

```
1  /**
2   * Imprime recursivamente el camino mínimo a nodoDest
3   * (especificado por su número interno).
4   * imprimirCamino es la rutina guía
5   */
6  private void imprimirCaminoRec( int nodoDest )
7  {
8      if( tabla[ nodoDest ].ant != VERTICE_NULO )
9      {
10         imprimirCaminoRec( tabla[ nodoDest ].ant );
11         System.out.print( " a " );
12     }
13     System.out.print( tabla[ nodoDest ].nombre );
14 }
```

Figura 14.14 Rutina recursiva para imprimir el camino mínimo.

```
1  /**
2   * Rutina guía para tratar casos inalcanzables e
3   * imprimir el coste total. Llama a la rutina recursiva
4   * para imprimir el camino mínimo a nodoDest.
5   */
6  private void imprimirCamino( int nodoDest )
7  {
8      if( tabla[ nodoDest ].dist == INFINITO )
9          System.out.println( tabla[ nodoDest ].nombre +
10                             " es inalcanzable" );
11     else
12     {
13         imprimirCaminoRec( nodoDest );
14         System.out.println( " (el coste es " +
15                             tabla[ nodoDest ].dist + ")" );
16     }
17     System.out.println( );
18 }
```

Figura 14.15 Rutina para imprimir el camino mínimo consultando la tabla.

`limpiarDatos` inicializa los atributos para que los algoritmos puedan empezar. `imprimirCamino` imprime el camino más corto después de que se ha ejecutado el algoritmo.

Los atributos eventualmente calculados por algún algoritmo de búsqueda de caminos mínimos son inicializados por la rutina `limpiarDatos`, mostrada en la Figura 14.13. La siguiente es una rutina para imprimir el camino mínimo una vez su cálculo ha sido realizado. Recuerde que el campo `ant` puede utilizarse para recorrer el camino hacia atrás. Sin embargo, esto nos dará el camino en orden inverso. Esto no es un problema si utilizamos recursión; los vértices en el camino a `dest` son los mismos que aquéllos en el camino al vértice anterior a `dest` (en el camino), seguidos de `dest`. Esta estrategia se traduce directamente en la pequeña rutina recursiva mostrada en la Figura 14.14. La rutina supone que existe el camino en cuestión. `imprimirCamino`, mostrada en la Figura 14.15, realiza primero esta comprobación; si no existe camino, imprime un mensaje. En otro caso, llama a la rutina recursiva e imprime el coste del camino.

```
1  /**
2   * Procesa una petición; devuelve falso si se acaba el fichero.
3   */
4  public boolean procesarPeticion( BufferedReader in )
5  {
6      String nombreOrig, nombreDest;
7      ElementoHash orig = new ElementoHash( );
8      ElementoHash dest = new ElementoHash( );
9
10     try
11     {
12         System.out.println( "Introduzca nodo origen:" );
13         if( ( nombreOrig = in.readLine( ) ) == null )
14           return false;
15         System.out.println( "Introduzca nodo destino:" );
16         if( ( nombreDest = in.readLine( ) ) == null )
17           return false;
18     }
19     catch( IOException e )
20     {
21         System.out.println( "Error: " + e );
22         return false;
23     }
24
25     try
26     {
27         orig.nombre = nombreOrig;
28         orig = (ElementoHash) ( dicVertices.buscar( orig ) );
29         dest.nombre = nombreDest;
30         dest = (ElementoHash) ( dicVertices.buscar( dest ) );
31
32         if( dijkstra( orig.rango ) )
33           imprimirCamino( dest.rango );
34         else
35           System.out.println( "Dijkstra ha fallado" );
36     }
37     catch( ElementoNoEncontrado e )
38       { System.err.println( "Vértice no está en el grafo" ); }
39     return true;
40 }
```

Figura 14.16 Para probar, `procesarPeticion` llama al algoritmo de búsqueda del camino mínimo (con pesos).

La última rutina, a parte de las que calculan los caminos mínimos, es el método `procesarPeticion`. En la Figura 14.16 se muestra una implementación sencilla que pregunta el vértice origen y el vértice destino y ejecuta el algoritmo de búsqueda del camino con peso mínimo (el código en Internet ejecuta también otros algoritmos adicionales).

```
/**
 * Rutina main que: pide el nombre de un fichero que
 * contenga un grafo; construye el grafo a partir del fichero;
 * repetidamente pide dos vértices y ejecuta los algoritmos
 * de caminos mínimos. El fichero de datos es una secuencia
 * de líneas: origen destino coste.
 */
public static void main( String [ ] args )
{
    System.out.println( "Introduzca fichero con el grafo:" );
    BufferedReader in = new BufferedReader( new
                        InputStreamReader( System.in ) );
    FileReader fich;
    String nombreFichero = "";

    try
    {
        nombreFichero = in.readLine( );
        fich = new FileReader( nombreFichero );
    }
    catch( Exception e )
      { System.err.println( e ); return; }

    BufferedReader fichGrafo = new BufferedReader( fich );
    Grafo g = new Grafo( );

        // Lee las aristas y las inserta
    try
    {
        String linea;
        while( ( linea = fichGrafo.readLine( ) ) != null )
        {
            StringTokenizer st = new StringTokenizer( linea );
            try
            {
                if( st.countTokens( ) != 3 )
                    throw new Exception( );
                String orig = st.nextToken( );
                String dest = st.nextToken( );
                int coste = Integer.parseInt( st.nextToken( ) );
                g.insArista( orig, dest, coste );
            }
            catch( Exception e )
              { System.err.println( "Error: " + linea ); }
        }
    }
    catch( Exception e )
      { System.err.println( "Error: " + e ); }
    while( g.procesarPeticion( in ) )
        ;
}
```

Figura 14.17 Ejemplo de rutina `main`.

Una rutina `main` sencilla, mostrada en la Figura 14.17, ilustra el comportamiento de la clase. Pide el nombre de un fichero que contenga un grafo. Repetidamente lee líneas de la entrada, asignando la línea a un objeto de la clase `StringTokenizer`, y analiza la línea. Esta técnica nos permite comprobar que cada línea contiene los tres campos de una arista. Cuando se encuentran los tres campos, se llama a `insArista`. Una vez leído el grafo, se ejecuta el algoritmo de búsqueda del camino con peso mínimo, llamando a `procesarPeticion`.

14.2 Problema del camino mínimo sin pesos

La *longitud del camino sin pesos* mide el número de aristas en el camino.

La *longitud del camino sin pesos* mide el número de aristas. Esta sección estudia el problema de encontrar el camino que tenga menor longitud sin pesos entre dos vértices especificados.

PROBLEMA DEL CAMINO SIN PESOS MÍNIMO CON UN ÚNICO ORIGEN
Encontrar el camino más corto (medido por el número de aristas) desde el vértice O a cualquier otro vértice.

El problema del camino mínimo sin pesos es un caso especial del problema del camino mínimo con pesos (en el que todos los pesos valen 1). Por tanto, debería tener una solución más eficiente que el problema con pesos. Esto en efecto es cierto, aunque los algoritmos para todos los problemas de caminos mínimos son muy similares.

14.2.1 Teoría

Todas las variantes del problema de los caminos mínimos tienen soluciones similares.

Para resolver el problema del camino mínimo sin pesos, utilizamos el grafo de la Figura 14.1, con V_2 como nodo origen O. Por ahora, nos ocuparemos de encontrar la longitud de los caminos mínimos. Más tarde, también llevaremos cuenta del correspondiente camino.

Inmediatamente, podemos decir que el camino más corto desde O a V_2 es un camino de longitud 0. Esta información produce el grafo de la Figura 14.18. Ahora podemos empezar a mirar los vértices que están a distancia 1 de O. Si hacemos esto, vemos que V_0 y V_5 están a una arista de distancia de O. Esto se muestra en la Figura 14.19.

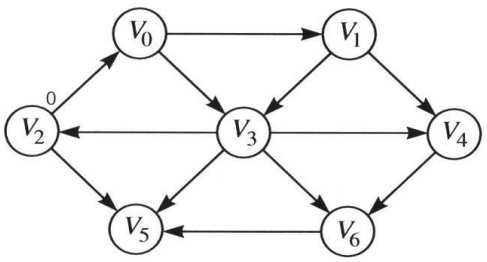

Figura 14.18 El grafo después de marcar el nodo origen como alcanzable con cero aristas.

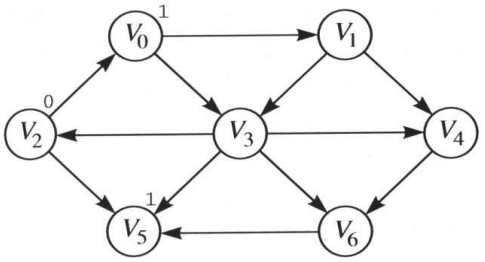

Figura 14.19 Grafo después de encontrar todos los vértices cuyo camino mínimo desde el origen tiene longitud 1.

El siguiente paso es buscar los vértices cuyo camino mínimo desde O sea exactamente 2. Hacemos esto buscando los nodos adyacentes a V_0 y V_5 (los vértices a distancia 1) cuyo camino mínimo no se conozca aún. Esta búsqueda nos dice que el camino más corto a V_1 y V_3 es 2. La Figura 14.20 muestra el progreso realizado hasta el momento.

Finalmente, examinando los vértices adyacentes a los últimamente alcanzados V_1 y V_3, encontramos que V_4 y V_6 tienen un camino mínimo de 3 aristas. Y ya hemos calculado el camino para todos los vértices. La Figura 14.21 muestra el resultado final del algoritmo.

Esta estrategia para buscar un camino se denomina *búsqueda en anchura*. Opera procesando los vértices por niveles: los vértices más cercanos al origen se procesan los primeros, y los más alejados los últimos.

La *búsqueda en anchura* procesa los vértices por niveles: los vértices más cercanos al origen se calculan los primeros.

La Figura 14.22 ilustra un principio fundamental. Si un camino al vértice v tiene coste D_v, y w es adyacente a v, entonces existe un camino a w de coste $D_w = D_v + 1$. Todos los algoritmos de búsqueda de caminos mínimos empiezan con $D_w = \infty$ y van reduciendo este valor cuando se explora un v apropiado. Para

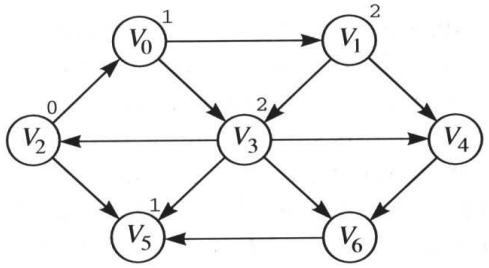

Figura 14.20 Grafo después de encontrar todos los vértices cuyo camino mínimo desde el origen tiene longitud 2.

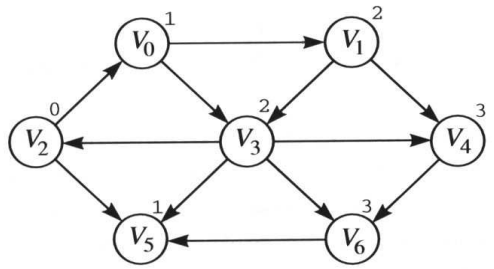

Figura 14.21 Caminos mínimos finales.

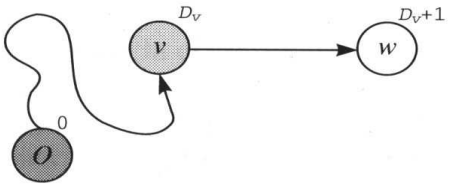

Figura 14.22 Si *w* es adyacente a *v* y hay un camino a *v*, también hay un camino a *w*.

hacer esto eficientemente, los vértices deben explorarse de forma sistemática. Cuando se explora un determinado v, se realizan las actualizaciones de los vértices w adyacentes recorriendo la lista de adyacencia de v.

El *punto de atención* se mueve de vértice a vértice, actualizándose las distancias de los vértices adyacentes.

De la discusión anterior, concluimos el siguiente algoritmo para la resolución del problema de los caminos mínimos sin pesos: sea D_i la longitud del camino más corto desde O hasta i. Sabemos que $D_O = 0$, e inicialmente $D_i = \infty$ para todo $i \neq O$. Mantenemos nuestra *atención* fija en un vértice que inicialmente es O y vamos saltando de vértice en vértice. Si v es el vértice sobre el que nos, entonces, para todo w adyacente a v, hacemos $D_w = D_v + 1$ si $D_w = \infty$. Esto refleja el hecho de que podemos llegar a w siguiendo un camino a v y extendiendo el camino con la arista (v, w). De nuevo esto se ilustra en la Figura 14.22. Ya que nuestra atención va procesando cada vértice en orden de distancias al origen, y cada arista suma exactamente uno a la longitud del camino a w, podemos estar seguros de que la primera vez que D_w deja de ser ∞, tomará el valor del camino mínimo a w. Por la misma razón tenemos que el penúltimo vértice en el camino a w es v, por lo que una línea adicional de código nos permitirá almacenar el camino en cuestión.

Después de haber procesado todos los vértices adyacentes a v, pasamos nuestra atención a otro vértice u (en el que todavía no nos hayamos fijado) tal que $D_u = D_v$. Si ello no es posible, pasamos a un u que satisfaga $D_u = D_v + 1$. Si tampoco es posible, es que hemos acabado. La Figura 14.23 muestra cómo vamos visitando vértices y actualizando las distancias. El nodo con sombra clara en cada estado representa nuestro punto actual de atención. En este dibujo y los que siguen, los estados se muestran consecutivamente de arriba abajo, y de izquierda a derecha.

Todos los vértices adyacentes a *v* se encuentran recorriendo la lista de adyacencia de *v*.

El detalle que falta concretar es la estructura de datos subyacente. Hay dos acciones básicas a realizar. Debemos, primero, buscar repetidamente el vértice en el que depositaremos nuestra atención. En segundo lugar, durante el algoritmo necesitamos comprobar todos los vértices w adyacentes a v (el vértice actual). La segunda acción se implementa fácilmente iterando a través de la lista de adyacencia de v. De hecho, ya que cada arista se procesa exactamente una vez, el coste total de todas las iteraciones es $O(|E|)$. La primera acción es más complicada: no podemos limitarnos a explorar la tabla buscando el vértice apropiado, pues cada exploración requeriría un tiempo $O(|V|)$ y necesitamos hacerlo $O(|V|)$ veces. Entonces el coste total sería $O(|V|^2)$, lo cual es inaceptable para grafos dispersos. Afortunadamente, esto no es necesario.

Cuando se rebaja la distancia a un vértice (lo que ocurre una única vez), se coloca éste en una cola de forma que en un futuro podamos fijar la atención en él. El vértice origen se introduce en la cola cuando su distancia se inicializa a 0.

Cuando a un vértice w se le rebaja su D_w desde ∞, se convierte en candidato para fijar nuestra atención en él en algún momento del futuro. Una vez hayamos fijado nuestra atención en todos los vértices que se encuentran a igual distancia del origen, D_v, que el actual, pasaremos a considerar los que se encuentran a distancia $D_v + 1$, entre los que se encuentra w. Por lo que w sólo le queda esperar pacientemente su turno. Claramente, no tiene por qué hacerlo delante de cualquier

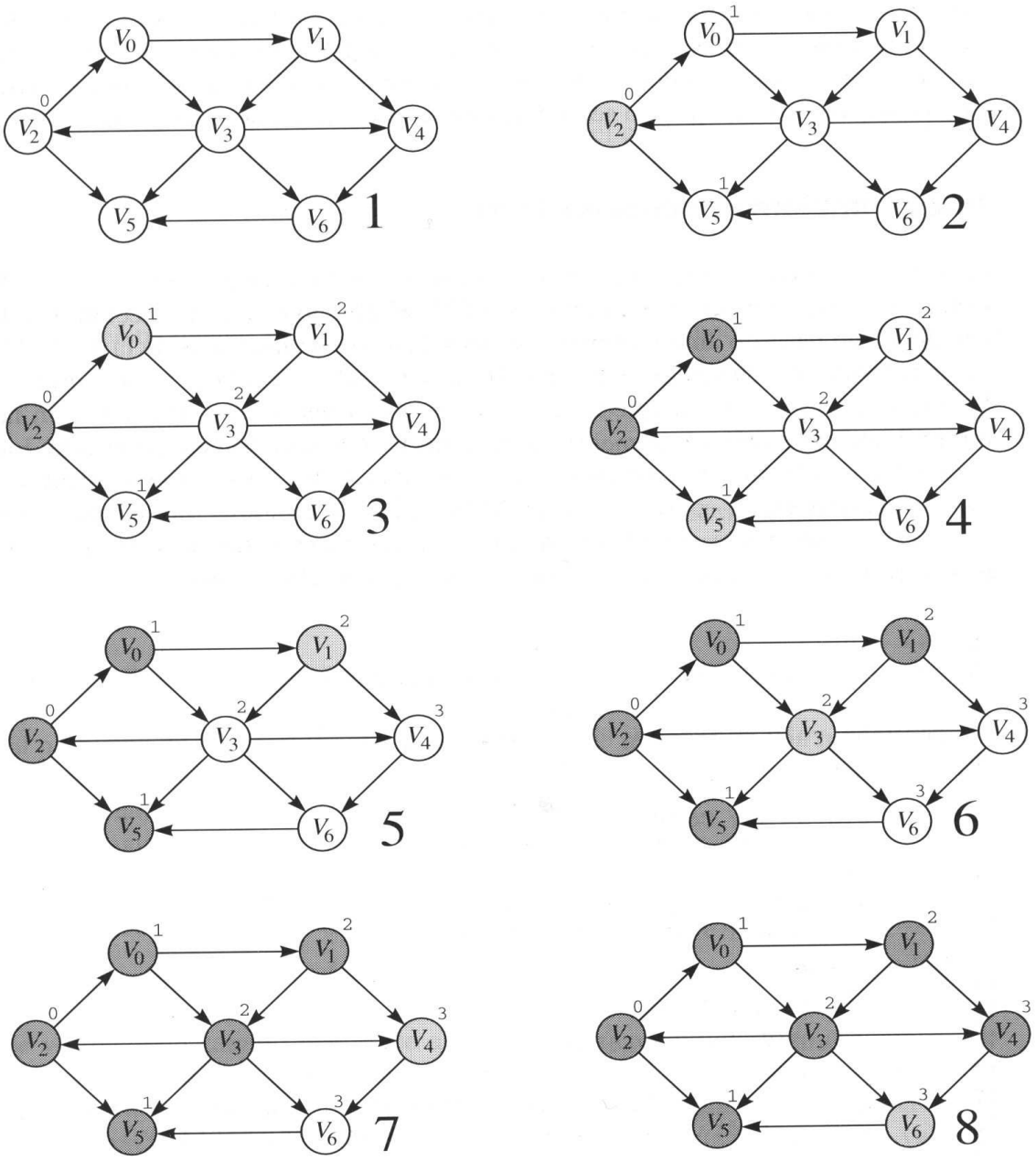

Figura 14.23 Cómo se recorre el grafo en el cálculo de los caminos mínimos sin pesos. Los vértices más oscuros han sido ya totalmente procesados, los vértices más claros todavía no han sido considerados como *v*, y aquel con sombreado intermedio es el vértice *v* actual. Los estados van de arriba abajo y de izquierda a derecha, según indica la numeración.

otro nodo cuya distancia se haya rebajado ya, por lo que puede colocarse al final de una cola de vértices en espera.

Para seleccionar el siguiente vértice *v* en el que fijemos nuestra atención, simplemente tomamos el primer vértice de la cola. Empezamos con una cola vacía, y para que el mecanismo empiece a funcionar colocamos en la cola el vértice origen *O*. Ya que cada vértice se introduce en la cola y se elimina de ella exactamente

una vez, y ya que las operaciones sobre las colas son constantes, el coste total de elegir un vértice *para el algoritmo completo* es $O(|V|)$. Por tanto, el coste de la búsqueda en anchura está dominado por los recorridos de las listas de adyacencias, y en consecuencia es $O(|E|)$, o sea lineal con respecto al tamaño del grafo.

14.2.2 Implementación en Java

La implementación es más sencilla de lo que parece. Sigue al dedillo la descripción del algoritmo.

La implementación del algoritmo de los caminos mínimos sin peso se realiza en el método `sinPeso`, mostrado en la Figura 14.24. El código es una traducción literal del algoritmo descrito anteriormente. La inicialización de las líneas de la 9 a la 11 inicializa todas las distancias a infinito, D_O a 0 y añade a la cola el vértice origen. Mientras que la cola no se quede vacía, hay vértices que visitar. Por tanto, en la línea 17, pasamos a fijar la atención en el vértice v que es el primero de la cola. La línea 19 itera sobre su lista de adyacencia y produce todos los vértices `w` adyacentes a `v`. El test $D_w = \infty$ se realiza en la línea 22. Si es `true`, la actualización $D_w = D_v + 1$ se realiza en la línea 24, junto con la actualización del campo `ant` y la inserción en la cola de `w`, en las líneas 25 y 26, respectivamente.

```
1  /**
2   * Calcula el camino mínimo sin pesos.
3   */
4  private void sinPesos( int nodoOrig )
5  {
6      int v, w;
7      Cola q=new ColaVec( );
8
9      limpiarDatos( );
10     tabla[ nodoOrig ].dist = 0;
11     q.insertar( new Integer( nodoOrig ) );
12
13     try
14     {
15         while( !q.esVacia( ) )
16         {
17             v=( (Integer) q.quitarPrimero( ) ).intValue( );
18             ListaIter p = new ListaEnlazadaIter( tabla[ v ].ady );
19             for( ; p.estaDentro( ); p.avanzar( ) )
20             {
21                 w=( (Arista) p.recuperar( ) ).dest;
22                 if( tabla[ w ].dist == INFINITO )
23                 {
24                     tabla[ w ].dist = tabla[ v ].dist + 1;
25                     tabla[ w ].ant = v;
26                     q.insertar( new Integer( w ) );
27                 }
28             }
29         }
30     }
31     catch( DesbordamientoInferior e ) { } // No puede ocurrir
32 }
```

Figura 14.24 Algoritmo para el problema del camino mínimo sin pesos utilizando una búsqueda en anchura.

14.3 Problema de los caminos mínimos con pesos positivos

La *longitud de un camino con pesos* es la suma del coste de las aristas del camino. Esta sección considera el problema de encontrar el camino más corto con pesos. En el problema de los caminos mínimos con pesos positivos, las aristas tienen costes no negativos. Queremos calcular los caminos mínimos desde un vértice origen al resto de vértices. Como se verá enseguida, la hipótesis de que los costes de las aristas son no negativos es muy importante pues permite utilizar un algoritmo sencillo y relativamente eficiente. El método descrito se conoce como *algoritmo de Dijkstra*. En la siguiente sección estudiaremos un algoritmo más lento que funciona aun con aristas negativas.

La *longitud de un camino con pesos* es la suma del coste de las aristas del camino.

PROBLEMA DE LOS CAMINOS MÍNIMOS CON COSTE POSITIVO Y ORIGEN ÚNICO

Encontrar el camino más corto (medido con su coste total) desde el vértice origen O al resto de vértices. El coste de cada arista es no negativo.

14.3.1 Teoría: algoritmo de Dijkstra

El problema de los caminos mínimos con pesos se resuelve de una manera similar al problema sin pesos. Sin embargo, debido al coste de las aristas, unas pocas cosas cambian.

El *algoritmo de Dijkstra* resuelve el problema del camino mínimo con pesos.

Las siguientes cuestiones deben ser examinadas:

1. ¿Cómo se ajusta D_w?
2. ¿Cómo buscamos el siguiente vértice v en el que centrar nuestra atención?

Empezamos examinando como se altera D_w. Al resolver el problema sin pesos, si $D_w = \infty$, hacíamos $D_w = D_v + 1$ porque rebajábamos el valor de D_w si el vértice v ofrecía un camino más corto a w. La dinámica del algoritmo aseguraba que sólo necesitábamos alterar D_w una vez. Sumábamos 1 a D_w porque la longitud del camino a w es 1 más que la longitud del camino a v. Si aplicamos la misma lógica al caso con pesos, deberíamos hacer $D_w = D_v + c_{v,w}$, cuando este nuevo valor de D_w sea mejor que el original. Sin embargo, ahora no podemos garantizar que D_w se actualice una sola vez. En consecuencia, D_w debería modificarse si su valor actual es mayor que $D_v + c_{v,w}$ (en vez de simplemente compararlo con ∞). Simplemente, el algoritmo decide si de momento es una buena idea pasar por v para ir hasta w. El coste original D_w es el coste sin utilizar v; el coste $D_v + c_{v,w}$ es el camino más barato utilizando los vértices v considerados hasta la fecha.

Utilizamos $D_V + c_{V,W}$ como nueva distancia y para decidir si la distancia debe ser actualizada.

La Figura 14.25 muestra una situación típica. Anteriormente durante la ejecución del algoritmo, se rebajó la distancia a w a 8 cuando fijamos nuestra atención en el vértice u. Sin embargo, cuando consideramos v, necesitamos volver a cambiar la distancia a w, rebajándola a 6, pues tenemos un nuevo camino mínimo provisional. Esto nunca ocurría en el algoritmo sin pesos porque todas las aristas sumaban 1 a la longitud del camino, por lo que $D_u \leqslant D_v$ implicaba $D_u + 1 \leqslant D_v + 1$, y por tanto $D_w \leqslant D_v + 1$. Aquí, aunque $D_u \leqslant D_v$, es todavía posible que el camino a w se mejore al considerar v.

Una simple cola no es ahora apropiada para almacenar los vértices que esperan a que los consideremos.

La distancia para vértices no visitados representa un camino utilizando solamente vértices visitados como nodos intermedios.

La Figura 14.25 muestra otro punto importante. Cuando se rebaja la distancia a w, es siempre porque es adyacente a algún vértice en el que hemos fijado nuestra atención. Por ejemplo, después de visitar v y completado su procesamiento, el valor de D_w será 6 y el último vértice en el camino será un nodo ya visitado. De forma análoga, el vértice anterior a v debe haberse visitado, y así sucesivamente. Por tanto, en cualquier momento, el valor de D_w representa un camino desde O hasta w utilizando como nodos intermedios solamente vértices que han sido ya visitados. Este hecho crucial nos lleva al Teorema 14.1.

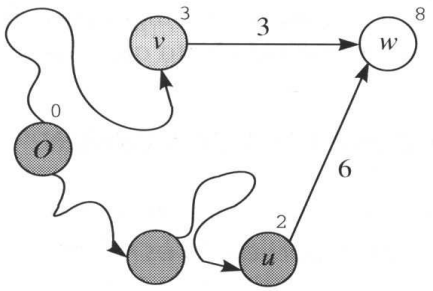

Figura 14.25 Tenemos fijada la atención en el vértice v; w es adyacente; D_w debería reducirse a 6.

Teorema 14.1

Si vamos pasando nuestra atención a un vértice aún no visitado de entre los que minimicen el valor D_i, el algoritmo producirá correctamente los caminos mínimos siempre que no haya aristas con coste negativo.

Demostración

Llamemos «paso» a cada visita a un nodo. Probaremos por inducción que, después de cada paso, los valores de D_i para vértices ya visitados corresponden al camino mínimo, y que los valores de D_i para los demás vértices corresponden al camino más corto hasta ellos que utiliza solamente como nodos intermedios vértices ya visitados. Ya que el primer nodo visitado es el origen, la afirmación es cierta para el primer paso. Supongamos que es correcta para los primeros k estados. Sea v el vértice visitado en el paso $k+1$. Supongamos, con la intención de llegar a una contradicción, que hay un camino de O a v de longitud menor que D_v. Este camino debe pasar por un vértice intermedio que no haya sido todavía visitado. Llamemos u al primer vértice intermedio en el camino que no haya sido visitado. La situación se muestra en la Figura 14.26. El camino hasta u utiliza sólo nodos intermedios visitados, por lo que, por inducción, D_u representa la distancia óptima a v. Es más, $D_u < D_v$, porque u está en el supuesto camino a v. Esto es una contradicción, porque si fuera así habríamos elegido u como siguiente vértice a visitar en lugar de v. La demostración se completa mostrando que el resto de valores D_i para nodos no visitados siguen siendo correctos; esto es evidente, dada la regla de actualización.

La Figura 14.27 muestra los pasos del algoritmo de Dijkstra. Lo que nos queda por hacer es elegir las estructuras de datos apropiadas. Para grafos densos, podemos recorrer la tabla buscando el vértice apropiado. Como en el caso sin pesos,

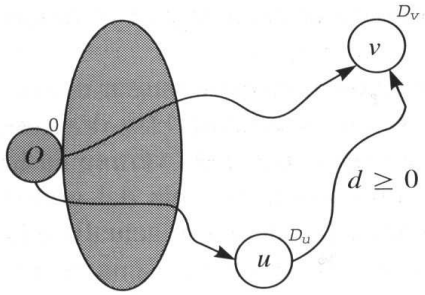

Figura 14.26 Si D_V es mínimo entre todos los vértices no visitados y si todas las aristas tienen coste no negativo, entonces D_V representa el camino mínimo hasta v.

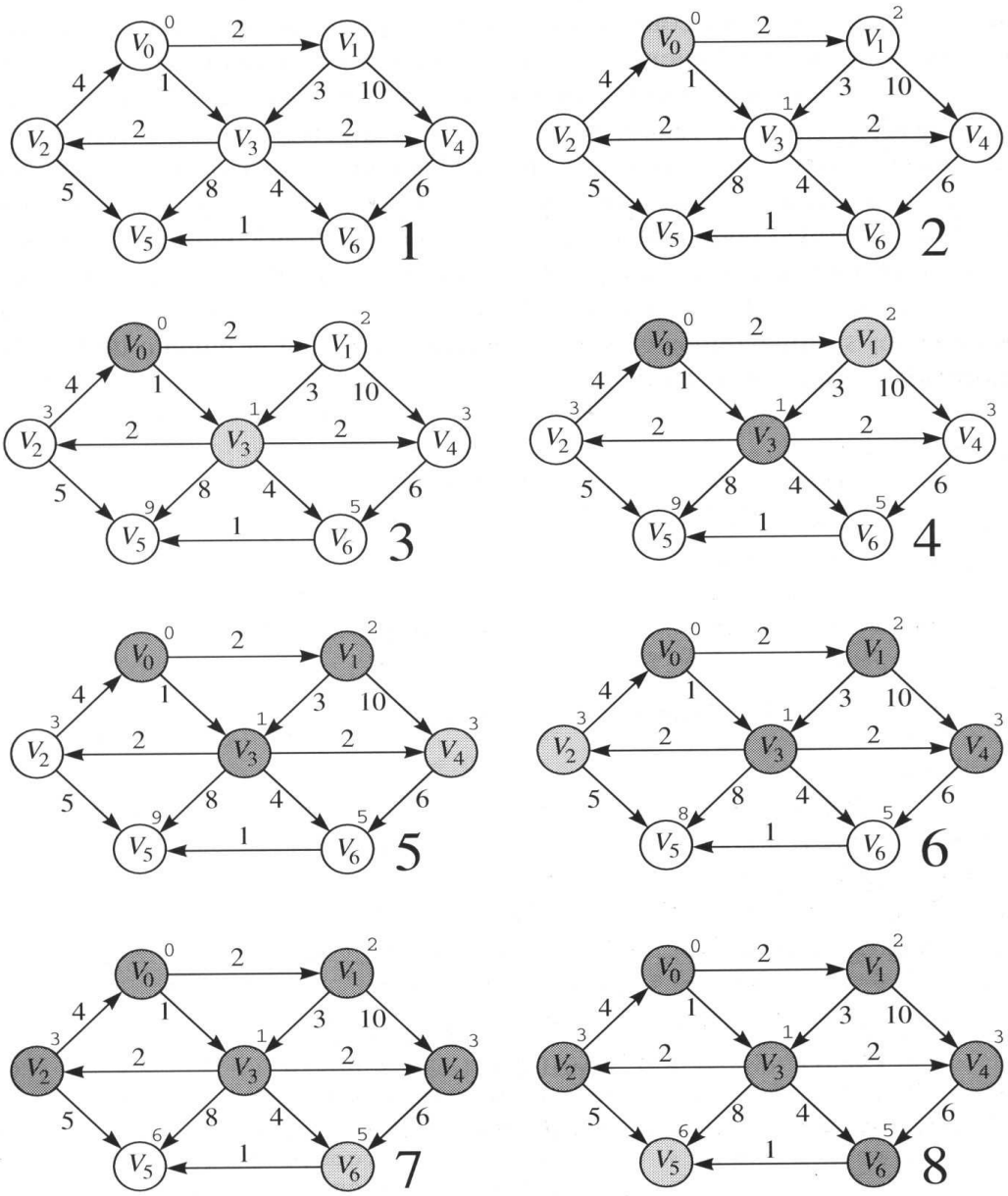

Figura 14.27 Pasos del algoritmo de Dijkstra. Sigue los mismos convenios que en la Figura 14.23.

esto nos llevaría a un tiempo $O(|V|^2)$, que es óptimo para un grafo denso. Pero para un grafo disperso, podemos hacerlo mejor.

La cola de prioridad es una estructura apropiada. El método más sencillo es insertar en la cola de prioridad una nueva entrada, formada por un vértice y una distancia, cada vez que se reduce la distancia de un vértice. Podemos encontrar el vértice a donde movernos eliminando repetidamente el vértice de mínima distancia hasta que salga un vértice no visitado.

Ciertamente, una cola no funcionaría. El hecho de que debamos buscar el vértice v con menor D_v sugiere la utilización de una cola de prioridad. Hay dos formas de utilizar esta cola de prioridad. Una forma es almacenar cada vértice en la cola de prioridad y utilizar la distancia (obtenida consultando la tabla del grafo) como función de ordenación. Cuando alteramos cualquier D_w, debemos actualizar la cola de prioridad restableciendo la propiedad de orden. Esto equivale a una operación `flotar`. Para hacer esto necesitamos ser capaces de encontrar la posición de w en la cola de prioridad. No todas las implementaciones de las colas de prioridad soportan esta operación. En otras palabras, `flotar` no es una operación de la interfaz `ColaPrioridad`. Una implementación que sí soporta esta operación es el montículo de emparejamientos; su utilización en esta aplicación se discute en el Capítulo 22.

En vez de utilizar una cola de prioridad compleja, utilizaremos un método que funciona con cualquier estructura de datos que implemente el interfaz `ColaPrioridad`. Nuestro método consistirá en insertar en la cola de prioridad un objeto formado por w y D_w cuando reducimos D_w. Para elegir el nuevo vértice v a visitar, eliminaremos repetidamente el menor elemento (utilizando las distancias) de la cola de prioridad hasta que salga un elemento no visitado. Ya que el tamaño de la cola de prioridad podría ser tan grande como $|E|$, y hay como mucho $|E|$ inserciones y eliminaciones, el tiempo de ejecución sería $O(|E| \log |E|)$. Ya que $|E| \leqslant |V|^2$ implica $\log |E| \leqslant 2 \log |V|$, obtenemos el mismo coste $O(|E| \log |V|)$ que tendríamos si utilizáramos el primer método (en el cual el tamaño de la cola de prioridad es como mucho $|V|$).

```
1  /**
2   * Objeto almacenado en la cola de prioridad
3   * en el algoritmo de Dijkstra
4   */
5  class Camino implements Comparable
6  {
7      int dest;     // w
8      int coste;    // D(w)
9
10     static Camino infNeg = new Camino( ); // Centinela
11
12     Camino( )
13       { this( 0 ); }
14
15     Camino( int d )
16       { this( d, 0 ); }
17
18     Camino( int d, int c )
19       { dest = d; coste = c; }
20
21     public boolean menorQue( Comparable lder )
22       { return coste < ( (Camino) lder ).coste; }
23
24     public int compara( Comparable lder )
25       { return coste < ( (Camino) lder ).coste ? -1 :
26                coste > ( (Camino) lder ).coste ? 1 : 0; }
27 }
```

Figura 14.28 Elemento básico almacenado en la cola de prioridad.

14.3.2 Implementación en Java

El objeto que se colocará en la cola de prioridad se muestra en la Figura 14.28. Está formado por w y D_w, y una función de comparación definida en base a D_w. La Figura 14.29 muestra la rutina `dijkstra` que calcula los caminos mínimos.

Una vez más, la implementación sigue la descripción casi al pie de la letra.

```
1  /**
2   * Algoritmo de Dijkstra utilizando un montículo binario.
3   * Devuelve false si se detectan aristas negativas.
4   */
5  private boolean dijkstra( int nodoOrig )
6  {
7      int v, w;
8      ColaPrioridad cp=new MonticuloBinario( Camino.infNeg );
9      Camino vrec;
10
11     limpiarDatos( );
12     tabla[ nodoOrig ].dist=0;
13     cp.insertar( new Camino( nodoOrig, 0 ) );
14
15     try
16     {
17         for( int nodosVistos=0; nodosVistos < numVertices;
18                                                 nodosVistos++ )
19         {
20             do
21             {
22                 if( cp.esVacia( ) )
23                     return true;
24                 vrec=(Camino) cp.eliminarMin( );
25             } while( tabla[ vrec.dest ].extra != 0 );
26
27             v=vrec.dest;
28             tabla[ v ].extra=1;
29
30             ListaIter p=new ListaEnlazadaIter( tabla[ v ].ady );
31             for( ; p.estaDentro( ); p.avanzar( ) )
32             {
33                 w=( (Arista) p.recuperar( ) ).dest;
34                 int cvw=( (Arista) p.recuperar( ) ).coste;
35
36                 if( cvw < 0 )
37                     return false;
38
39                 if( tabla[ w ].dist > tabla[ v ].dist+cvw )
40                 {
41                     tabla[ w ].dist=tabla[ v ].dist+cvw;
42                     tabla[ w ].ant=v;
43                     cp.insertar( new Camino( w, tabla[ w ].dist ) );
44                 }
45             }
46         }
47     }
48     catch( DesbordamientoInferior e ) { } // No puede ocurrir
49
50     return true;
51 }
```

Figura 14.29 Algoritmo de búsqueda de caminos mínimos con pesos: algoritmo de Dijkstra.

La línea 8 declara la cola de prioridad `cp`. Recuerde que debemos proporcionar un centinela para que se garantice que sea menor o igual que cualquier objeto insertado. `vrec`, declarado en la línea 9, almacenará el resultado de cada `eliminarMin`. Como en el caso del algoritmo sin pesos, empezamos inicializando todas las distancias a infinito, haciendo $D_O = 0$ y colocando el vértice origen dentro de la estructura de datos.

Cada iteración del bucle `for` más externo empieza en la línea 17 fijando nuestra atención en el vértice v y procesándolo, examinando los vértices adyacentes w. v se elige eliminando repetidamente elementos de la cola de prioridad (en la línea 24) hasta encontrar un vértice aún no procesado. Utilizamos el atributo `extra` para almacenar esta información. Inicialmente `extra` vale 0. Por tanto, cuando el vértice no ha sido procesado, la comparación del bucle `while` fallará en la línea 25. Cuando el vértice se procesa, `extra` se actualiza a 1 (en la línea 28). La cola de prioridad podría quedarse vacía si, por ejemplo, alguno de los vértices es inalcanzable. En este caso, podemos acabar inmediatamente. El bucle de las líneas 31 a 45 es muy similar al caso del algoritmo sin pesos. La diferencia está en que en la línea 34 debemos extraer `cvw` de la lista de adyacencia, comprobar que el coste no es negativo (en otro caso, nuestro algoritmo podría producir respuestas incorrectas) y sumar `cvw`, en vez de 1, en las líneas 39 y 41.

14.4 Problema del camino mínimo con costes negativos

Las aristas negativas hacen que el algoritmo de Dijkstra no funcione correctamente. Es necesario un algoritmo alternativo.

El algoritmo de Dijkstra necesita que los costes de las aristas sean no negativos. Esto es razonable para la mayoría de las aplicaciones, pero en ocasiones es demasiado restrictivo. Esta sección estudia brevemente el caso más general.

PROBLEMA DE LOS CAMINOS MÍNIMOS CON COSTES NEGATIVOS Y ÚNICO ORIGEN

Buscar el camino más corto (medido con su coste total) desde el vértice O al resto de vértices. Los costes de las aristas pueden ser negativos.

14.4.1 Teoría

Un *ciclo de coste negativo* hace que la mayoría de los caminos mínimos, si no todos, estén indefinidos porque podemos dar vueltas al ciclo arbitrariamente para así obtener una longitud arbitrariamente negativa.

La demostración de la corrección del algoritmo de Dijkstra necesita que los costes de las aristas, y por tanto los de los caminos, sean no negativos. De hecho, si el grafo tiene aristas de coste negativo, el algoritmo de Dijkstra no funciona correctamente. El problema se encuentra en que una vez que un vértice v ha sido procesado, puede existir un camino de vuelta a v desde otro vértice u no procesado, con coste bastante negativo. En tal caso, el camino que va desde O hasta v pasando por u es mejor que ir de O a v sin pasar por u. Si esto último ocurriera, estaríamos en dificultades: no sólo el camino a v sería incorrecto, sino que tendríamos que visitar de nuevo v, porque los vértices alcanzables desde v podrían también ser afectados.

Hay otro problema adicional por el que preocuparse. Consideremos el grafo de la Figura 14.30. El camino de V_3 a V_4 tiene coste 2. Sin embargo, existe un camino más corto siguiendo el ciclo V_3, V_4, V_1, V_3, V_4, que tiene coste -3. Y este camino no es aún el más corto, ya que podríamos dar un número arbitrario de vueltas al ciclo. Por lo que el camino más corto entre estos dos puntos está indefinido.

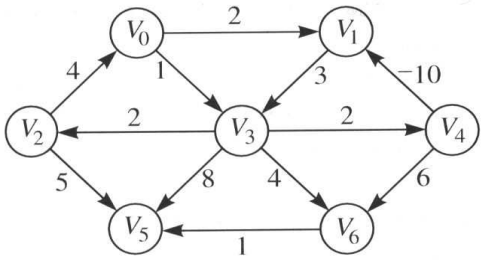

Figura 14.30 Grafo con ciclo de coste negativo.

Este problema no se restringe a nodos en el ciclo. El camino más corto de V_2 a V_5 también está indefinido porque hay una forma de entrar y salir del bucle. Este bucle se denomina *ciclo de coste negativo*; cuando existe uno en un grafo, algunos (probablemente muchos) caminos mínimos no están definidos. Las aristas de coste negativo no son malas por sí mismas; son los ciclos los que lo son. Nuestro algoritmo encontrará los caminos mínimos o hará notar la existencia de ciclos con coste negativo.

Una combinación de los algoritmos con pesos y sin pesos resolverá el problema, pero a costa de un drástico incremento potencial en el tiempo de ejecución. Como hemos sugerido anteriormente, cuando se modifica D_w, debemos (re)visitar este vértice en algún momento del futuro. En consecuencia, utilizamos una cola como hicimos en el algoritmo sin pesos, pero usando $D_v + c_{v,w}$ como medida de la distancia (como en el algoritmo de Dijkstra).

Cuando se rebaja la distancia a un vértice, éste debe colocarse en una cola. Esto puede pasar repetidas veces para un mismo vértice. El tiempo de ejecución puede ser elevado, especialmente cuando hay ciclos de coste negativo.

Cuando visitamos el vértice v por i-ésima vez, el valor de D_v es la longitud del camino más corto formado por, a lo sumo, i aristas. Una demostración de esto se deja como ejercicio (Ejercicio 14.9). En consecuencia, si no hay ciclos de coste negativo, un vértice puede salir de la cola como mucho $|V|$ veces y el algoritmo consume un tiempo $O(|E|\,|V|)$. Es más, si un vértice sale de la cola más de $|V|$ veces, habremos detectado un ciclo de coste negativo.

14.4.2 Implementación en Java

La implementación del algoritmo de búsqueda de caminos mínimos con pesos negativos se presenta en la Figura 14.31. Hemos hecho un pequeño cambio con respecto a la descripción del algoritmo: no añadimos un vértice a la cola cuando ya está en ella. Para hacer esto, utilizamos el campo `extra`. Cuando se añade un vértice a la cola, incrementamos `extra` (en la línea 40). Cuando se quita de la cola, lo incrementamos de nuevo (en la línea 26). Por tanto, `extra` es impar si el vértice está en la cola, y `extra/2` nos dice cuántas veces ha salido de la cola (y esto explica la pregunta de la línea 26). Cuando se cambia la distancia a algún w, pero ya está en la cola (porque `extra` es impar), no lo añadimos de nuevo. Sin embargo, le sumamos 2 para indicar que podría haber salido y entrado de la cola (esto puede acelerar algo el algoritmo en presencia de ciclos negativos). Esto se hace en las líneas 40 y 43. El resto del algoritmo utiliza código que ya se ha visto en el algoritmo sin pesos (Figura 14.24) y en el algoritmo de Dijkstra (Figura 14.29).

La parte astuta de la implementación es la manipulación de la variable `extra`. Intentamos evitar que un mismo vértice aparezca dos veces simultáneamente en la cola.

```
1  /**
2   * Ejecuta el algoritmo de caminos mínimos.
3   * Se permiten aristas con coste negativo.
4   * Devuelve false si se detectan ciclos negativos.
5   */
6  private boolean negativos( int nodoOrig )
7  {
8      int v, w;
9      Cola q = new ColaVec( );
10     int cvw;
11
12     limpiarDatos( );
13     tabla[ nodoOrig ].dist = 0;
14     q.insertar( new Integer( nodoOrig ) );
15     tabla[ nodoOrig ].extra++;
16
17     // Incrementa extra cuando el vértice v entra o sale
18     // de la cola. Si el vértice esta a punto de entrar, pero ya
19     // está en la cola, cuenta como si entrara y saliera
20     try
21     {
22         while( !q.esVacia( ) )
23         {
24             v=( (Integer) q.quitarPrimero( ) ).intValue( );
25
26             if( tabla[ v ].extra++ > 2 * numVertices )
27                 return false; // existe un ciclo
28
29             ListaIter p = new ListaEnlazadaIter( tabla[ v ].ady );
30             for( ; p.estaDentro( ); p.avanzar( ) )
31             {
32                 w = ( (Arista) p.recuperar( ) ).dest;
33                 cvw = ( (Arista) p.recuperar( ) ).coste;
34                 if( tabla[ w ].dist > tabla[ v ].dist + cvw )
35                 {
36                     tabla[ w ].dist = tabla[ v ].dist + cvw;
37                     tabla[ w ].ant = v;
38
39                     // Insertar sólo si no está en la cola
40                     if( tabla[ w ].extra++ % 2 == 0 )
41                         q.insertar( new Integer( w ) );
42                     else // cuenta como una eliminación fantasma
43                         tabla[ w ].extra++;
44                 }
45             }
46         }
47     }
48     catch( DesbordamientoInferior e ) { } // No puede ocurrir
49     return true;
50 }
```

Figura 14.31 Algoritmo de caminos mínimos con costes negativos: se permiten aristas negativas.

14.5 Problemas de caminos en grafos acíclicos

Una clase importante de grafos son aquellos que no contienen ciclos. El problema del camino mínimo es más sencillo si el grafo no contiene ciclos. Por ejemplo, no debemos preocuparnos de la posible existencia de ciclos negativos, ya que no hay ciclos en absoluto. En consecuencia, estudiaremos el siguiente problema:

Un *grafo dirigido acíclico* es un grafo dirigido que no contiene ciclos. Éstos son una clase importante de grafos.

PROBLEMA DE LOS CAMINOS MÍNIMOS CON PESOS DESDE UN ÚNICO NODO SOBRE GRAFOS ACÍCLICOS

Encontrar el camino más corto (medido con su coste total) en un grafo acíclico desde el vértice O hasta cualquier otro vértice. No hay restricciones sobre el coste de las aristas.

Pero antes de abordar el problema de los caminos mínimos en este caso general, estudiaremos un problema relacionado: la ordenación topológica.

14.5.1 Ordenación topológica

Un *orden topológico* ordena los vértices de un grafo dirigido acíclico de tal forma que si hay un camino de u a v, entonces v aparece después de u en la ordenación. Por ejemplo, se puede utilizar un grafo para representar los prerrequisitos entre cursos de una universidad. Una arista (v, w) indica que el curso v debe ser completado antes de que pueda afrontarse el curso w. Un orden topológico de los cursos es cualquier secuencia que no viole los prerrequisitos. Una ordenación *topológica* encuentra todos los ordenes topológicos posibles sobre un grafo acíclico dado.

Una *ordenación topológica* ordena los vértices de un grafo dirigido acíclico de tal forma que si hay un camino de u a v, entonces v aparece *después* de u en la ordenación. Un grafo con ciclos no puede ordenarse topológicamente.

Es evidente que no existirá ningún orden topológico si el grafo contiene ciclos, ya que para cualquier par de vértices v y w en el ciclo, hay un camino de v a w y de w a v. Por tanto, cualquier ordenación de v y w estaría en contradicción con alguno de los caminos. Un grafo puede tener varios ordenes topológicos, y en la mayoría de los casos, cuando busquemos uno de ellos cualquiera nos valdrá.

Un algoritmo sencillo para realizar la ordenación topológica consiste en encontrar primero un vértice v que no tenga aristas de entrada. Se imprime este vértice y se borra (lógicamente) del grafo junto con sus aristas. Tras ello, se sigue aplicando la misma estrategia al resto del grafo. Para formalizar esto, definimos como *grado de entrada* de un vértice v el número de aristas (u, v).

El *grado de entrada* de un vértice es el número de aristas que llegan a él. Se puede realizar una ordenación topológica en tiempo lineal eliminando repetidamente los vértices que no tengan aristas entrantes.

Calculamos los grados de entrada de todos los vértices del grafo. En la práctica, «eliminar lógicamente» significa que reducimos en 1 el grado de entrada de cada vértice adyacente a v. La Figura 14.32 muestra la aplicación del algoritmo a un grafo acíclico. El grado de entrada se calcula para cada vértice. V_2 tiene grado 0; por lo que es el primero en la ordenación. Si hubiera varios vértices con grado 0, podríamos elegir cualquiera de ellos. Cuando se eliminan del grafo V_2 y sus aristas, los grados de V_0, V_3 y V_5 se decrementan en 1. Ahora V_0 tiene grado 0, por lo que es el siguiente en la ordenación topológica, y se reducen los grados de V_1 y V_3. El algoritmo continúa, y el resto de vértices se examinan en el orden V_1, V_3, V_4, V_6 y V_5. Al reiterar, no borramos físicamente las aristas del grafo; la eliminación (virtual) de aristas sólo facilitará la visualización de cómo decrecen los grados de entrada.

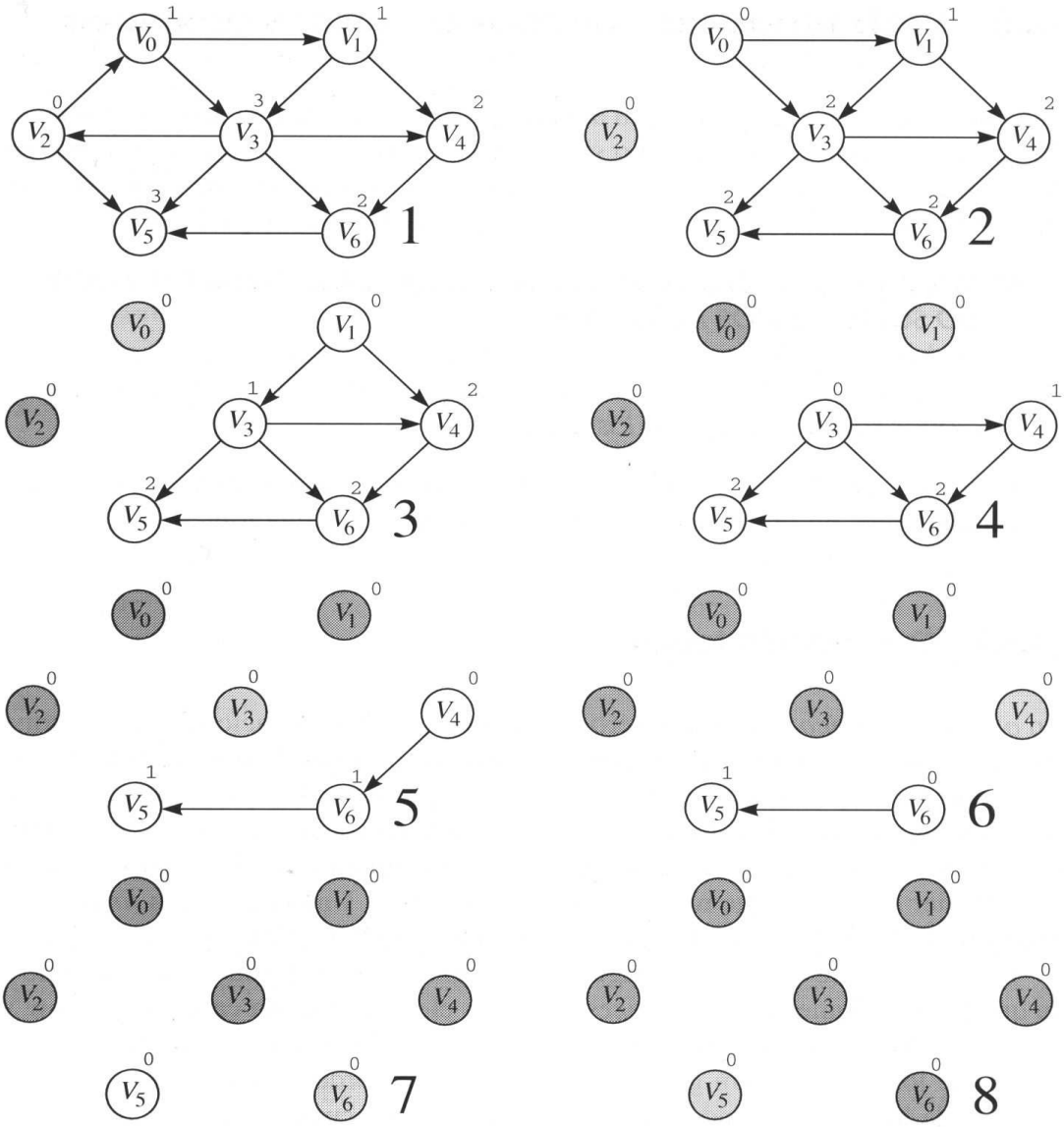

Figura 14.32 Ordenación topológica. Sigue el mismo convenio que la Figura 14.23.

El algoritmo produce la respuesta correcta y detecta si el grafo no es acíclico.

Dos cuestiones importantes a considerar son la corrección y la eficiencia. Claramente, cualquier orden producido por el algoritmo es un orden topológico. La cuestión es si todo grafo acíclico tiene una ordenación topológica, y si es así, si nuestro algoritmo garantiza que encuentra una. La respuesta es afirmativa en ambos casos.

Si en cualquier momento hay vértices aún no visitados, pero ninguno tiene grado de entrada cero, es porque existe un ciclo. Para ver esto, tomemos cualquier vértice A_0. Ya que A_0 tiene una arista de entrada, podemos tomar como A_1 cualquier vértice conectado a A_0. Pero como A_1 tiene también una arista de entrada, podemos tomar como A_2 un vértice conectado a A_1. Repetimos el proceso N veces, donde N es el número de vértices sin visitar en el grafo. Entre A_0, A_1, ..., A_N debe haber dos vértices idénticos (ya que hay N vértices, pero $N + 1$ elementos). Un recorrido hacia atrás entre estos A_i y A_j idénticos, exhibe un ciclo.

El tiempo de ejecución es lineal si se utiliza una cola.

El algoritmo puede implementarse en tiempo lineal manteniendo en una cola todos los vértices no procesados de grado de entrada cero. Inicialmente, todos los

vértices de grado de entrada cero se colocan en la cola. Para encontrar el siguiente vértice en el orden topológico, simplemente tomamos el primero de la cola, eliminándolo de ella. Cuando se reduce a cero el grado de un vértice, se introduce en la cola. Si la cola se queda vacía antes de haber procesado todos los vértices, es porque el grafo contiene un ciclo. El tiempo de ejecución es lineal, por la misma razón argüida en el algoritmo de caminos mínimos sin pesos.

14.5.2 Teoría del algoritmo de caminos mínimos con un grafo acíclico

En un grafo acíclico, iremos visitando sus vértices en orden topológico.

Una aplicación importante de la ordenación topológica es su uso para resolver el problema de los caminos mínimos sobre grafos acíclicos. La idea es la siguiente: ir visitando los vértices según un orden topológico.

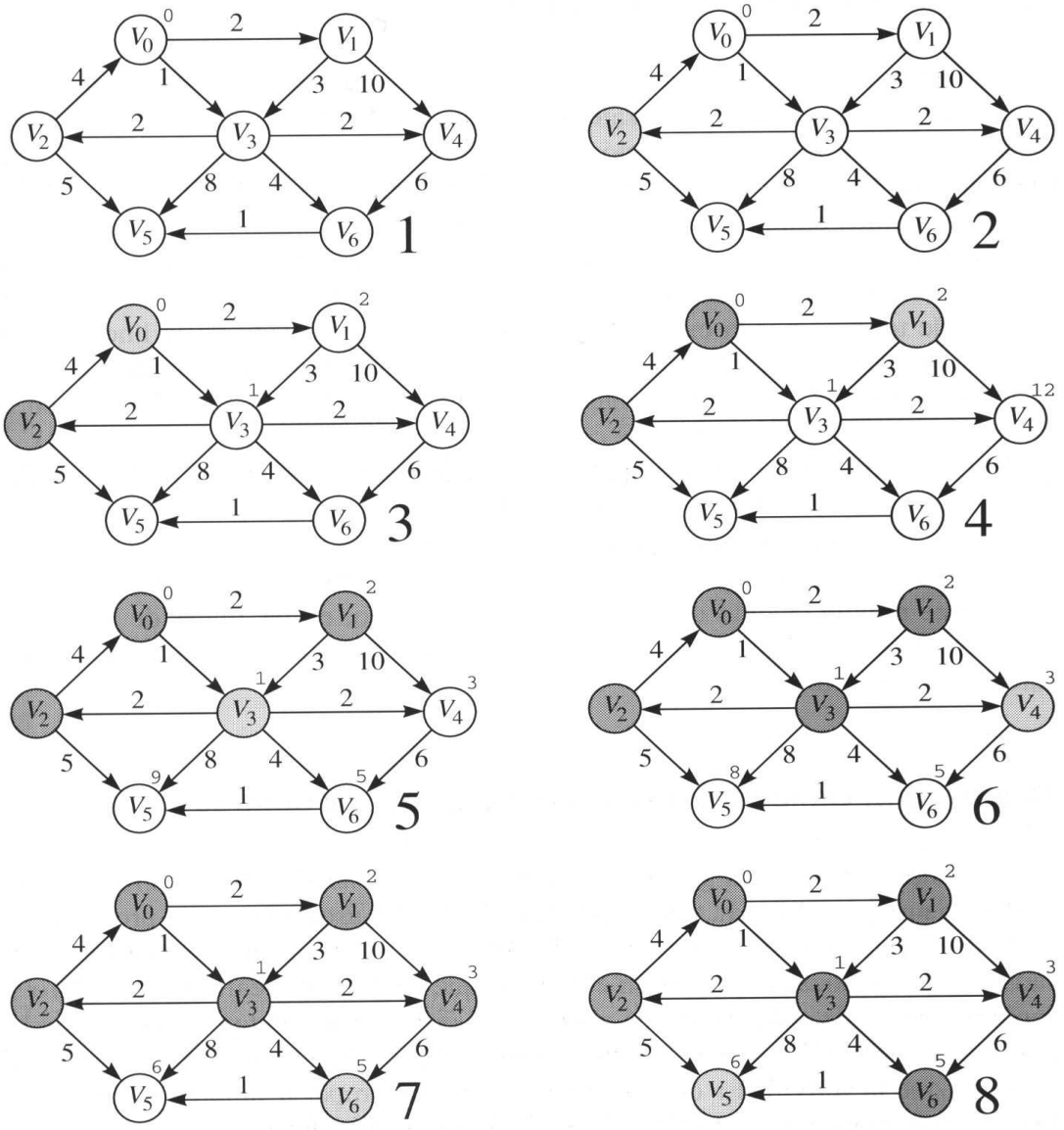

Figura 14.33 Pasos del algoritmo de caminos mínimos sobre un grafo acíclico. Sigue el mismo convenio que la Figura 14.23.

Esta idea funciona porque cuando visitamos el vértice v, estamos seguros de que D_v no será reducido nunca más, ya que por definición de orden topológico, no hay aristas que lleguen a v desde nodos no visitados. La Figura 14.33 muestra los estados del algoritmo de caminos mínimos, utilizando una ordenación topológica para guiar la visita de los vértices. Observe que la secuencia de vértices no es la misma que en el algoritmo de Dijkstra. Observe también que los vértices visitados antes de llegar al vértice origen, son inalcanzables desde el origen, por lo que no tienen influencia en las distancias hasta ningún vértice.

El resultado es un algoritmo lineal en tiempo, aun en presencia de aristas negativas.

Ya que no necesitamos una cola de prioridad, y sólo necesitamos incorporar la ordenación topológica al cálculo del camino mínimo, tenemos que el algoritmo se ejecutará en tiempo lineal y funcionará correctamente en presencia de aristas negativas.

14.5.3 Implementación en Java

La implementación combina el cálculo de un orden topológico y el cálculo de caminos mínimos. La información sobre el grado de entrada se almacena en el campo `extra`.

La implementación del algoritmo de caminos mínimos para grafos acíclicos se muestra en la Figura 14.34. Utilizamos una cola para realizar una ordenación topológica, y mantenemos la información sobre el grado de entrada en el campo `extra`. Las líneas de la 13 a la 19 calculan los grados de entrada, y en las líneas de la 21 a la 23, colocamos en la cola cada vértice con grado 0.

Entonces, en la línea 27 se retira repetidamente un vértice de la cola. Observe que si la cola se vacía, el bucle `for` terminará por la comprobación de la línea 25. Si el bucle termina por la existencia de un ciclo, será reflejado en la línea 50. En otro caso, el bucle en la línea 30 recorre la lista de adyacencia, obteniéndose el valor de w en la línea 32. Inmediatamente se decrementa el grado de entrada de w en la línea 33, y, si ha llegado a cero, se coloca este vértice en la cola (en la línea 34).

Los nodos que aparecen antes del origen O en el orden topológico son inalcanzables.

Recuerde que si el vértice actual v aparece antes de O en la ordenación topológica, entonces v es inalcanzable desde O. En consecuencia, seguirá cumpliéndose $D_v = \infty$ y por tanto, no puede aparecer en ningún camino a los vértices w alcanzables. Realizamos la preceptiva comprobación en la línea 36, y si se da el caso, no se hace ningún calculo de distancias. En otro caso, en las líneas de la 39 a la 44, utilizamos el mismo cálculo que en el algoritmo de Dijkstra para actualizar D_w cuando ello es oportuno.

14.5.4 Una aplicación: análisis de caminos críticos

El *análisis de caminos críticos* se utiliza para planificar tareas asociadas a un proyecto.

Un uso importante de los grafos acíclicos es el *análisis de caminos críticos*, una forma de análisis utilizada para planificar las tareas asociadas a un proyecto. El grafo de la Figura 14.35 sirve como ejemplo. Cada vértice representa una actividad que debe realizarse, junto con el tiempo que se tarda en realizarla. El grafo se denomina grafo de actividades, en el cual los vértices representan actividades y las aristas representan relaciones de precedencia. La existencia de una arista (v, w) significa que la actividad v debe realizarse antes de empezar la tarea w. Evidentemente, esto implica que el grafo debe ser acíclico. Asumiremos que las actividades que no dependen (directa o indirectamente) entre sí pueden ser realizadas en paralelo por diferentes servidores.

```
1  // Algoritmo lineal, funciona sólo con grafos acíclicos
2  private boolean aciclico( int nodoOrig )
3  {
4      int v, w, iteraciones = 0;
5      Cola q = new ColaVec( );
6
7      limpiarDatos( );
8      tabla[ nodoOrig ].dist = 0;
9
10     try
11     {
12         // Calcula los grados de entrada
13         for( v = 0; v < numVertices; v++ )
14         {
15             ListaIter p = new ListaEnlazadaIter( tabla[ v ].ady );
16             for( ; p.estaDentro( ); p.avanzar( ) )
17                 tabla[ ( (Arista) p.recuperar( ) ).dest ].
18                                                     extra++;
19         }
20         // Insertar en la cola vértices de grado cero
21         for( v = 0; v < numVertices; v++ )
22             if( tabla[ v ].extra == 0 )
23                 q.insertar( new Integer( v ) );
24
25         for( iteraciones = 0; !q.esVacia( ); iteraciones++ )
26         {
27             v=( (Integer) q.quitarPrimero( ) ).intValue( );
28
29             ListaIter p = new ListaEnlazadaIter( tabla[ v ].ady );
30             for( ; p.estaDentro( ); p.avanzar( ) )
31             {
32                 w = ( (Arista) p.recuperar( ) ).dest;
33                 if( --tabla[ w ].extra == 0 )
34                     q.insertar( new Integer( w ) );
35
36                 if( tabla[ v ].dist == INFINITO )
37                     continue;
38
39                 int cvw = ( (Arista) p.recuperar( ) ).coste;
40                 if( tabla[ w ].dist > tabla[ v ].dist + cvw )
41                 {
42                     tabla[ w ].dist = tabla[ v ].dist + cvw;
43                     tabla[ w ].ant = v;
44                 }
45             }
46         }
47     }
48     catch( DesbordamientoInferior e ) { } // No puede ocurrir
49
50     return iteraciones == numVertices;
51 }
```

Figura 14.34 Algoritmo de caminos mínimos para grafos acíclicos.

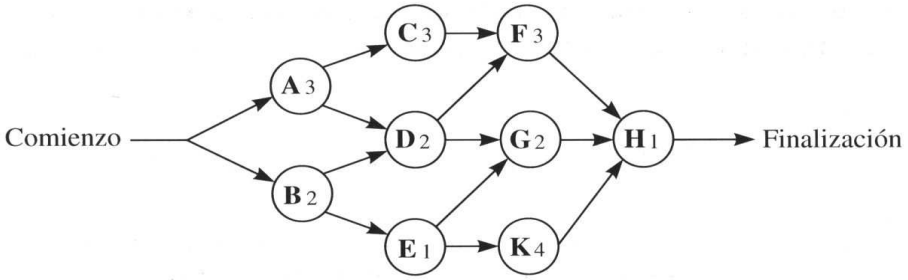

Figura 14.35 Grafo de actividades.

Un *grafo de actividades* representa las actividades como vértices y las relaciones de precedencia como aristas.

Este tipo de grafo puede utilizarse para modelar el desarrollo de proyectos, en el cual son de interés varias cuestiones importantes. En primer lugar, ¿cuál es el menor tiempo de terminación del proyecto? El grafo muestra que se necesitan diez unidades a través del camino *A*, *C*, *F*, *H*. Otra cuestión importante es ¿qué actividades se pueden retrasar, y por cuánto tiempo, sin afectar al tiempo mínimo de terminación? Por ejemplo, retrasar cualquiera de las tareas *A*, *C*, *F* o *H* haría que el tiempo de terminación rebasara las diez unidades. Por otro lado, la actividad *B* es menos crítica y puede retrasarse hasta dos unidades de tiempo sin afectar al tiempo de terminación final.

Un *grafo de eventos* está formado por vértices evento que corresponden a la terminación de una actividad y de todas sus actividades dependientes.

Para realizar estos cálculos, convertimos el grafo de actividades en un *grafo de eventos*, en el cual cada evento corresponde a la terminación de una actividad y de todas sus actividades dependientes. Las actividades correspondientes a los eventos alcanzables desde un nodo *v* en el grafo de eventos no pueden comenzar a ejecutarse hasta después de haber acabado la actividad correspondiente al evento *v*. Este grafo puede construirse automáticamente o a mano (a partir del grafo de actividades). Quizás sea necesario insertar aristas o vértices falsos para evitar introducir dependencias falsas (o faltas erróneas de dependencias). El grafo de eventos correspondiente al grafo de actividades de la Figura 14.35 se muestra en la Figura 14.36. Obsérvese la presencia de tales aristas y vértices falsos.

Las aristas muestran qué actividad debe ser completada para avanzar de un vértice al siguiente. El tiempo de finalización menor es el camino más largo.

Para encontrar el tiempo de terminación mínimo del proyecto, sólo necesitamos calcular la longitud del camino *más largo* desde el primer evento al último evento. Para grafos generales, el problema del camino máximo generalmente no tiene sentido, por la posibilidad de *ciclos de coste positivo*, que son los equivalentes a los ciclos de costes negativos en los problemas de los caminos mínimos. Si hay ciclos de coste positivo, podemos preguntar por el camino simple más largo. Sin embargo, no se conoce ninguna solución satisfactoria para este problema. Afortunadamente, el grafo de eventos es acíclico; por lo que no tenemos que preocuparnos de la presencia de ciclos. Es fácil adaptar el algoritmo de caminos mínimos para

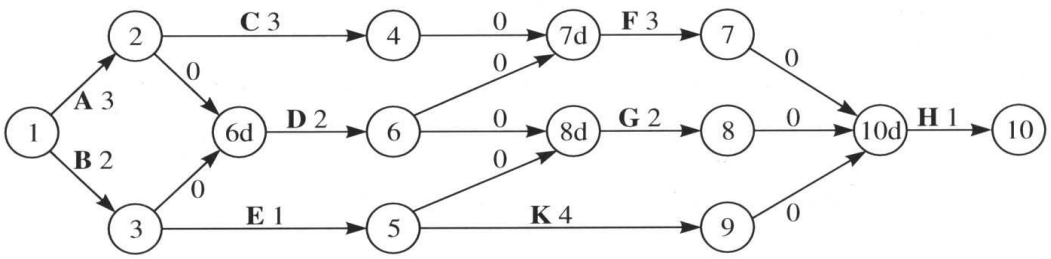

Figura 14.36 Grafo de eventos.

calcular el tiempo de terminación menor para todos los nodos del grafo. Si denotamos el tiempo de terminación mínimo para el nodo i por EC_i, entonces las reglas aplicables son

$$EC_1 = 0 \quad \text{y} \quad EC_w = \text{Max}_{(v,w) \in E}(EC_v + c_{v,w}).$$

El tiempo mayor en el que una tarea puede finalizar sin retrasar todo el proyecto también puede calcularse fácilmente.

La Figura 14.37 muestra el tiempo de terminación mínimo para cada vértice en nuestro ejemplo de grafo de eventos. También podemos calcular el tiempo máximo, LC_i, en el que cada evento puede terminar sin afectar el tiempo de terminación final. Las fórmulas en este caso son

$$LC_N = EC_N \quad \text{y} \quad LC_v = \text{Min}_{(v,w) \in E}(LC_w - c_{v,w}).$$

Estos valores pueden calcularse en tiempo lineal manteniendo, para cada vértice, una lista de los vértices adyacentes y precedentes. Los tiempos de terminación mínimos se calculan siguiendo un orden topológico, y el tiempo de terminación máximo se calcula siguiendo el orden topológico inverso. Los tiempos de terminación máximos se muestran en la Figura 14.38.

El *tiempo de espera* para cada arista en el grafo de eventos es la cantidad de tiempo que puede retrasarse la terminación de la actividad correspondiente sin retrasar el tiempo de terminación total. Es fácil ver que

El *tiempo de espera* es la cantidad de tiempo que una actividad puede retrasarse sin retrasar la terminación total.

$$Espera_{(v,w)} = LC_w - EC_v - c_{v,w}.$$

La Figura 14.39 muestra la espera para cada actividad en el grafo de eventos. Para cada nodo, el número de encima es el mínimo tiempo de terminación y el número de abajo es el tiempo de terminación máximo.

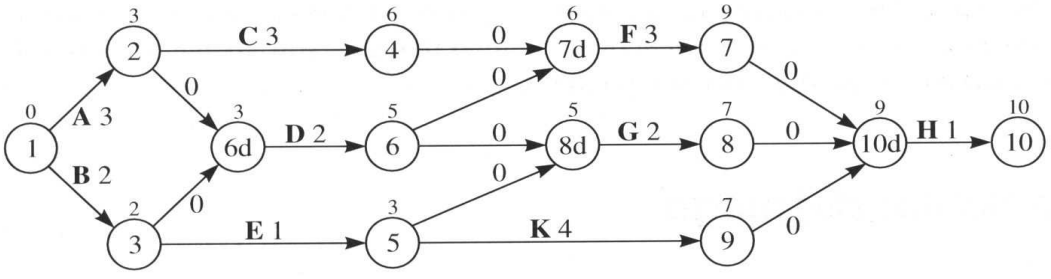

Figura 14.37 Tiempos de terminación mínimos.

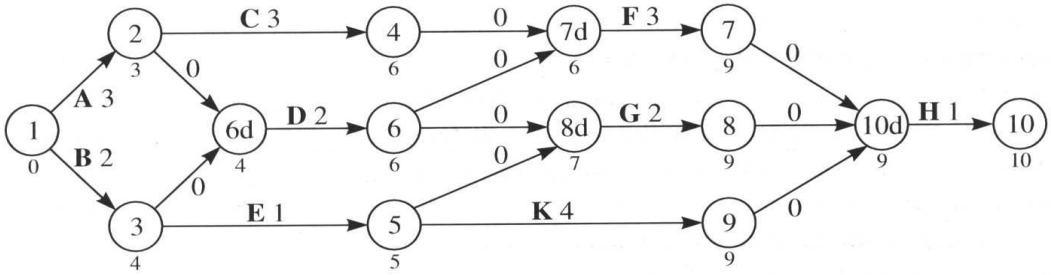

Figura 14.38 Tiempos de terminación máximos.

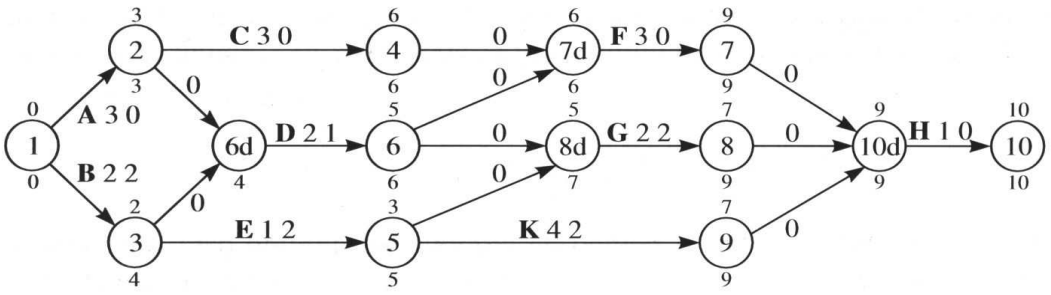

Figura 14.39 Tiempos de terminación mínimos, máximos y de espera (información adicional en cada arista).

Las actividades con tiempo de espera cero son críticas y no pueden ser retrasadas. Un camino con aristas de tiempo de espera cero se denomina *camino crítico.*

Algunas actividades tienen tiempo de espera cero. Éstas son actividades críticas que deben necesariamente acabar a su hora. Hay al menos un camino formado completamente con aristas de tiempo de espera cero; tal camino se denomina *camino crítico.*

Resumen

Este capítulo ha mostrado cómo se pueden utilizar grafos para modelar muchos problemas de la vida real y en particular cómo calcular el camino mínimo bajo gran variedad de circunstancias. Muchos de los grafos que aparecen en la práctica son típicamente muy dispersos, por lo que es importante elegir estructuras de datos apropiadas para implementarlos.

Para grafos sin pesos, los caminos mínimos pueden calcularse en tiempo lineal utilizando una búsqueda en anchura. Para grafos con pesos positivos, se necesita algo más de tiempo, utilizando el algoritmo de Dijkstra y una cola de prioridad eficiente. Para grafos con pesos negativos, el problema se hace aún más complicado. Finalmente, para grafos acíclicos, el tiempo de ejecución vuelve a ser lineal, con la ayuda de una ordenación topológica.

Elementos del juego

algoritmo de Dijkstra Un algoritmo que resuelve el problema de los caminos mínimos con peso.

análisis de caminos críticos Una forma de análisis utilizada en la planificación de tareas asociada a un proyecto.

búsqueda en anchura Un procedimiento de búsqueda que procesa los vértices por niveles. Aquéllos más cercanos al origen se evalúan primero. Nos movemos de vértice a vértice actualizando las distancias de los vértices adyacentes.

camino Secuencia de vértices conectados por aristas.

camino simple Un camino cuyos vértices son todos distintos, excepto el primero y el último que pueden ser iguales.

ciclo En un grafo dirigido, un camino que empieza y termina en el mismo vértice y que contiene al menos una arista.

ciclo de coste negativo Un ciclo cuyo coste es menor que cero. Hace que la mayoría de los caminos mínimos, si no todos, estén indefinidos porque podemos dar vueltas arbitrariamente por el ciclo y obtener caminos tan pequeños como queramos.

ciclo de coste positivo En el problema del camino más largo, el equivalente a los ciclos de coste negativo para el problema del camino más corto.

coste (peso) de una arista La tercera componente de una arista, que mide el coste de atravesarla.

grado de entrada Número de aristas entrantes a un vértice.

grafo Está formado por un conjunto de vértices y un conjunto de aristas que conectan vértices.

grafo de actividades Un grafo con actividades como vértices y relaciones de precedencia como aristas.

grafo de eventos Un grafo que tiene como vértices eventos que corresponden a la terminación de una actividad y de todas sus actividades dependientes. Las aristas muestran qué actividad debe completarse para avanzar de un vértice al siguiente. El tiempo de terminación mínimo es la longitud del camino más largo en dicho grafo.

grafo dirigido Un grafo cuyas aristas son pares ordenados de vértices.

grafo dirigido acíclico (GDA) Un tipo de grafo dirigido que no contiene ciclos.

grafos densos y dispersos Un grafo denso tiene un gran número de aristas (generalmente cuadrático). Los grafos típicos no son densos, sino dispersos.

lista de adyacencia Un vector de listas utilizado para representar un grafo. Utiliza un espacio lineal.

longitud de un camino El número de aristas de un camino.

longitud de un camino con peso La suma del coste de las aristas del camino.

longitud de un camino sin peso El número de aristas de un camino.

matriz de adyacencia Una representación de un grafo mediante una matriz, que utiliza un espacio cuadrático.

origen único Un algoritmo que calcula los caminos mínimos desde un punto de origen a todos los vértices del grafo.

tiempo de espera Cantidad de tiempo que puede retrasarse una actividad, sin afectar al tiempo total de terminación.

vértice adyacente w es adyacente a v si hay una arista de v a w.

Errores comunes

1. Un error común es no comprobar si el grafo de entrada satisface las condiciones requeridas por el algoritmo utilizado (por ejemplo tener todas sus aristas positivas o ser acíclico).
2. En la clase `Camino`, la función de comparación compara solamente el campo `coste`. Si el campo `dest` se utiliza para la comparación, el algoritmo puede parecer que funciona para grafos pequeños, pero para grafos grandes es sin duda incorrecto y dará respuestas subóptimas. Sin embargo, nunca producirá un camino que no exista, por lo que este error es difícil de detectar.
3. El algoritmo del camino mínimo sobre grafos con costes negativos debe contener una comprobación para detectar ciclos negativos; en otro caso, ciclará indefinidamente.

En Internet

Todos los algoritmos de este capítulo se encuentran disponibles en la red, en un único fichero, en el directorio **Chapter14**. El nombre del fichero es el siguiente:

Graph.java Traducido por `Grafo.java`, contiene la clase `Grafo` con la rutina `main`.

Ejercicios

Cuestiones breves

14.1. Encuentre el camino más corto sin pesos desde V_3 al resto de vértices en el grafo de la Figura 14.1.

14.2. Encuentre el camino más corto con pesos desde V_2 al resto de vértices en el grafo de la Figura 14.1.

Problemas teóricos

14.3. Muestre cómo evitar el coste cuadrático de la inicialización de las matrices de adyacencia, manteniendo un tiempo de acceso constante a cada arista.

14.4. Explique cómo modificar el algoritmo de caminos mínimos sin pesos, de tal forma que cuando haya más de un camino mínimo (en términos del número de aristas) el empate se deshaga en favor del de menor peso total.

14.5. Explique cómo modificar el algoritmo de Dijkstra para que cuente el número de caminos mínimos diferentes desde v hasta w.

14.6. Explique cómo modificar el algoritmo de Dijkstra de tal forma que cuando haya más de un camino mínimo desde v hasta w, se elija el camino con menor número de aristas.

14.7. Dé un ejemplo de grafo con aristas negativas, pero sin ciclos negativos, en el que el algoritmo de Dijkstra dé una respuesta errónea.

14.8. Considere el siguiente algoritmo para resolver el problema del camino mínimo con aristas negativas: sume una constante c a cada coste de arista, eliminando los costes negativos; calcule los caminos mínimos en este grafo; traslade el resultado al grafo original. ¿Qué es incorrecto en este algoritmo?

14.9. Demuestre la corrección del algoritmo del camino mínimo con aristas negativas. Para hacerlo, muestre que cuando visitamos el vértice v por i-ésima vez, el valor de D_v es la longitud del camino más corto formado, a lo sumo, por i aristas.

14.10. Dé un algoritmo de tiempo lineal para buscar el camino de mayor peso en un grafo acíclico. ¿Se puede extender su algoritmo a grafos con ciclos?

14.11. Muestre que si los costes de las aristas son 0 o 1 exclusivamente, el algoritmo de Dijkstra puede implementarse en tiempo lineal utilizando una cola doble (Sección 15.4).

Problemas prácticos

14.12. Este capítulo reivindica que en la implementación de los algoritmos sobre grafos que trabajan con grandes datos de entrada, la elección de estructuras de datos adecuadas es crucial para asegurar un rendimiento razonable. Para cada uno de los ejemplos siguientes, en los que se utiliza una estructura de datos pobre o un algoritmo malo, proporcione un análisis en notación O del resultado y compárelo con el rendimiento de los algoritmos y estructuras de datos utilizados en el texto. Incorpore un sólo cambio cada vez. Debería ejecutar sus tests con grafos razonablemente grandes y aleatoriamente dispersos. Haga lo siguiente:

a) Cuando se lee una arista, comprobe si ya está en el grafo.
b) Implemente el diccionario utilizando un recorrido secuencial de la tabla de vértices.
c) Implemente la cola utilizando el algoritmo del Ejercicio 6.13 (esto debería afectar al algoritmo de caminos mínimos sin pesos).
d) En el algoritmo de caminos mínimos sin pesos, implemente la búsqueda del vértice de menor coste con un recorrido secuencial de la tabla de vértices.
e) Implemente la cola de prioridad utilizando el algoritmo del Ejercicio 6.16 (esto debería afectar al algoritmo de caminos mínimos con pesos).
f) Implemente la cola de prioridad utilizando el algoritmo del Ejercicio 6.17 (esto debería afectar al algoritmo de caminos mínimos con pesos).
g) En el algoritmo de caminos mínimos con pesos, implemente la búsqueda del vértice de menor coste como un recorrido secuencial de la tabla de vértices.
h) En el algoritmo de caminos mínimos sobre grafos acíclicos, implemente la búsqueda del vértice con grado de entrada cero con un recorrido secuencial de la tabla de vértices.
i) Implemente cada uno de los algoritmos utilizando una matriz de adyacencia en vez de listas de adyacencia.

Prácticas de programación

14.13. Un grafo dirigido es fuertemente conexo si hay un camino desde cada vértice a cualquier otro. Haga lo siguiente:

a) Elija cualquier vértice O. Muestre que si el grafo es fuertemente conexo, un algoritmo de caminos mínimos declarará que todos los vértices son alcanzables desde O.
b) Muestre que si un grafo es fuertemente conexo, si la dirección de todas sus aristas se invierte y se ejecuta un algoritmo de caminos mínimos desde O, todos los vértices resultarán alcanzables.
c) Muestre que las comprobaciones en (a) y (b) son suficientes para decidir si un grafo es fuertemente conexo (es decir, todo grafo que pase ambos tests será fuertemente conexo).

d) Escriba un programa que compruebe si un grafo es fuertemente conexo. ¿Cuál es su tiempo de ejecución?

Explique cómo puede resolverse cada uno de los siguientes problemas aplicando un algoritmo de caminos mínimos. Diseñe un mecanismo para representar la entrada y escriba un programa que resuelva el problema.

14.14. La entrada es una lista de puntuaciones obtenidas en una liga de un deporte (donde no hay empates). Si todos los equipos han ganado al menos una vez y han perdido al menos una vez, podemos generalmente «probar», por un argumento transitivo tonto, que cualquier equipo es mejor que cualquier otro. Por ejemplo, suponga que en una liga de seis equipos donde todo el mundo juega 3 partidos, tenemos los siguientes resultados: *A* vence a *B* y a *C*; *B* vence a *C* y a *F*; *C* vence a *D*; *D* vence a *E*; *E* vence a *A*; *F* vence a *D* y a *E*. Entonces podemos probar que *A* es mejor que *F* porque *A* venció a *B* que a su vez venció a *F*. De igual forma, podemos probar que *F* es mejor que *A* porque *F* venció a *E* y *E* venció a *A*. Dada una lista de resultados y dos equipos *X* e *Y*, busque una demostración (si existe) de que *X* es mejor que *Y* o indique que tal demostración no existe.

14.15. Una palabra puede transformarse en otra sustituyendo un carácter. Suponga que tenemos un diccionario con palabras de 4 letras. Dé un algoritmo que determine si una palabra *A* puede transformarse en otra palabra *B*, mediante una serie de sustituciones de un carácter, y si es así, que muestre la correspondiente serie de palabras. Por ejemplo, `mesa` se convierte en `poto` con la secuencia `mesa`, `meta`, `seta`, `seto`, `peto`, `poto`.

14.16. La entrada es una colección de divisas y sus tasas de cambio. ¿Hay una secuencia de intercambios que produzca dinero mágicamente? Por ejemplo, si las divisas son *X*, *Y* y *Z*, y las tasas de cambio son 1 *X* es igual a 2 *Y*, 1 *Y* es igual a 2 *Z* y 1 *X* es igual a 3 *Z*, entonces 300 *Z* comprarán 100 *X*, que a su vez comprarán 200 *Y*, que a su vez comprarán 400 *Z*. Por lo que se obtendría un beneficio mágico del 33 por ciento.

14.17. Un estudiante necesita aprobar cierto número de cursos para graduarse, y estos cursos tienen prerrequisitos que deben cumplirse. Supongamos que todos los cursos se ofrecen todos los semestres y que un estudiante puede matricularse en un número indeterminado de cursos. Dada una lista de cursos y sus prerrequisitos, calcule una planificación que requiera el mínimo número de semestres.

14.18. El objetivo del *juego de Kevin Bacon* consiste en conectar al actor de una película con Kevin Bacon por papeles en una misma película. El mínimo número de conexiones es el *número de Bacon* de ese actor. Por ejemplo, Tom Hanks tiene un número de Bacon 1, porque trabajó en *Apolo* 13 con Kevin Bacon. Sally Field tiene un número de Bacon 2, porque trabajó en *Forrest Gump* con Tom Hanks, que trabajó en *Apolo* 13 con Kevin Bacon. Casi todo actor conocido tiene un número de Bacon 1 o 2. Supongamos que tiene una lista extensa de actores y películas, con papeles, y haga lo siguiente:

a) Explique cómo encontrar el número de Bacon de un actor.
b) Explique cómo encontrar el actor con mayor número de Bacon.
c) Explique cómo buscar el menor número de conexiones entre dos actores arbitrarios.

Bibliografía

Las listas de adyacencia para representar grafos fueron utilizadas por primera vez en [3]. El algoritmo de Dijkstra para el problema de los caminos mínimos fue descrito originalmente en [2]. El algoritmo para aristas con coste negativo se ha tomado de [1]. Un test más eficiente para la terminación se describe en [6], donde también se muestra cómo las estructuras de datos juegan un papel importante en una amplia variedad de algoritmos sobre grafos. La ordenación topológica está tomada de [4]. Una gran cantidad de aplicaciones en la vida real de algoritmos sobre grafos puede encontrarse en [5], junto con referencias a otros textos.

1. R. E. Bellman, «On a Routing Problem», *Quarterly of Applied Mathematics* **16** (1958), 87-90.
2. E. W. Dijkstra, «A Note on Two Problems in Connexion with Graphs, *Numerische Mathematik* **1** (1959), 269-271.
3. J. E. Hopcroft y R. E. Tarjan, «Algorithm 447: Efficient Algorithms for Graph Manipulation», *Communications of the ACM* **16** (1973), 372-378.
4. A. B. Kahn, «Topological Sorting of Large Networks», *Communications of the ACM* **5** (1962), 558-562.
5. D. E. Knuth, *The Stanford GraphBase*, Addison-Wesley, Reading, Mass. (1993).
6. R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, Penn. (1985).