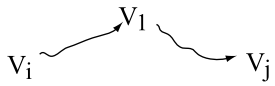
16.2. MATRIZ DE CAMINOS: ALGORITMO DE WARSHALL

Este algoritmo calcula la matriz de caminos `P` (también llamado cierre transitivo) de un grafo `G` de `n` vértices, representado por su matriz de adyacencia `A`. La estrategia que sigue el algoritmo consiste en definir, a nivel lógico, una secuencia de matrices *n-cuadradas* P_0, P_1, P_2, P_3 ... P_n; los elementos de cada una de las matrices $P_k[i,j]$ tienen el valor `0` si no hay camino y `1` si existe un camino del vértice `i` y al `j`. La matriz P_0 es la matriz de adyacencia.

Los elementos P_{ij} de P_1 son igual a `1` si lo son los de la matriz P_0, o bien si se puede formar un camino desde el vértice v_i a v_j, con la ayuda del vértice `1`. En definitiva, $P_1[i,j] = 1$ si lo es $P_0[i,j]$, o bien, hay un camino:



La matriz P_2 se forma a partir de P_1, añadiendo el vértice `2` para poder formar camino entre dos vértices todavía no conectados.

La diferencia entre dos matrices consecutivas P_k y P_{k-1} viene dada por la incorporación del vértice de orden `k`, para estudiar si es posible formar camino del vértice `i` y al `j` con la ayuda del vértice `k`. La descripción de cada matriz es la siguiente:

$P_0[i,j]$ =
- `1` si existe un arco del vértice `i` al `j`.
- `0` en otro caso.

$P_1[i,j]$ =
- `1` si existe un camino simple de v_i a v_j que no pasa por ningún otro vértice, a no ser por el vértice `1`.
- `0` en otro caso.

$P_2[i,j]$ =
- `1` si existe un camino simple de v_i a v_j que no pasa por otro vértice, a no ser por los que están comprendidos entre los vértices `1` y `2`.
- `0` en otro caso.

$P_3[i,j]$ =
- `1` si existe un camino simple de v_i a v_j que no pasa por otro vértice, a no ser por los que están comprendidos entre los vértices `1` y `3`.
- `0` en otro caso.

En cada paso se incorpora un nuevo vértice, el que coincide con el índice de la matriz `P`, a los anteriores para poder formar camino.

$P_k[i,j]$ =
- `1` si existe un camino simple de v_i a v_j que no pasa por otro vértice, a no ser por los que están comprendidos entre los vértices `1` y `k`.
- `0` en otro caso.

$P_n[i,j]$ =
- `1` si existe un camino simple de v_i a v_j que no pasa por otro vértice, a no ser por los que están comprendidos entre los vértices `1` y `n`.
- `0` en otro caso.

Según estas definiciones, P_0 es la matriz de adyacencia del grafo de `n` vértices, la matriz P_n es la matriz de caminos.

El algoritmo de Warshall encuentra una relación recurrente entre los elementos de la matriz P_k y los elementos de la matriz P_{k-1}. Un elemento $P_k[i,j] = 1$ si ocurren uno de estos dos casos:

1. Existe un camino simple de v_i a v_j del cual pueden formar parte los vértices de índice `1` al `k-1` (v_1 a v_{k-1}); por consiguiente, el elemento $P_{k-1}[i,j] = 1$.

2. Existe un camino simple de v_i a v_k y otro camino simple de v_k a v_j de los cuales pueden formar parte los vértices de índice 1 al k-1; por consiguiente, esto equivale:

$(P_{k-1}\ [i,k] = 1)$ y $(P_{k-1}\ [k,j] = 1)$

$v_i \rightarrow \ldots \rightarrow v_j$ (1)

$v_i \rightarrow \ldots \rightarrow v_k \rightarrow \ldots \rightarrow v_j$ (2)

Camino de v_i a v_j

Camino $v_i - v_k - v_j$

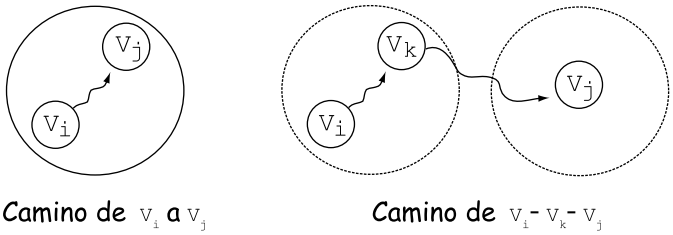


Figura 16.2 Posibilidad de camino de v_i a v_j

En definitiva, Warshall encuentra una relación recurrente entre la matriz P_{k-1} y P_k que permite, a partir de, la matriz de adyacencia P_0, encontrar P_n, matriz de caminos. La relación para encontrar los elementos de P_k puede expresarse como una operación lógica:

$P_k\ [i,j] = P_{k-1}[i,j] \vee (P_{k-1}\ [i,k] \wedge P_{k-1}\ [k,j])$

16.2.1. Implementación del algoritmo de Warshall

Se codifica el algoritmo para un grafo G representado por su matriz de adyacencia. El método devuelve la matriz de caminos P.

```
public static int [][] matrizCaminos(GrafoMatriz g) throws Exception
{
   int n = g.numeroDeVertices();
   int [][] P = new int[n][n]; // matriz de caminos
   // Se obtiene la matriz inicial: matriz de adyacencia
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
         P[i][j] = g.adyacente(i,j) ? 1 : 0;

   // se obtienen, virtualmente, a partir de P0, las sucesivas
   // matrices P1, P2, P3 ,..., Pn-1, Pn que es la matriz de caminos

   for (int k = 0; k < n; k++)
      for (int i = 0; i < n; i++)
         for (int j = 0; j < n; j++)
            P[i][j] = Math.min(P[i][j] + P[i][k] * P[k][j], 1);

   return P;
}
```

Nota sobre eficiencia

La complejidad del algoritmo de *Warshall* es cúbica, $O(n^3)$ siendo n el número de vértices. Esta característica hace que el tiempo de ejecución crezca rápidamente para grafos con, relativamente, muchos nodos.

16.3. CAMINOS MÁS CORTOS CON UN SOLO ORIGEN: ALGORITMO DE DIJKSTRA

Uno de los problemas que se plantean con relativa frecuencia es determinar la longitud del camino más corto entre un par de vértices de un grafo. Por ejemplo, determinar la mejor ruta (menor tiempo) para ir desde un lugar a un conjunto de centros de la ciudad. Para resolver este tipo de problemas se considera un grafo dirigido y *valorado*; es decir, cada arco (v_i,v_j) del grafo tiene asociado un coste c_{ij}. La longitud del camino es la suma de los costes de los arcos que forman el camino. Matemáticamente, si $v_1,v_2 \ \ldots \ v_k$ es la secuencia de vértices que forman un camino:

$$\textit{Longitud de camino} = \sum_{i=1}^{k-1} C_{i,j+1}$$

se pretende encontrar, entre todos los caminos, el camino de longitud o coste mínimo.

Otro problema relativo a los caminos entre dos vértices v_1, v_k, es encontrar aquel de menor número de arcos para ir de v_1 a v_k (el camino de longitud mínima en un grafo no valorado). Este problema se resuelve con una *búsqueda en anchura* a partir del vértice de partida (véase el capítulo anterior).

El algoritmo que se describe a continuación encuentra el camino más corto desde un vértice origen al resto de vértices, en un grafo con factor de peso positivo. Recibe el nombre de algoritmo de Dijkstra en honor del científico que lo expuso.

16.3.1. Algoritmo de Dijkstra

Dado un grafo dirigido $G=(V,A)$ valorado y con factores de peso no negativos, el algoritmo de Dijkstra determina el camino más corto desde un vértice al resto de los vértices del grafo. Éste, es un ejemplo típico de algoritmo *ávido (voraz)* que selecciona en cada paso la solución más optima para resolver el problema.

Recordemos que se trabaja con un grafo valorado donde cada arco (v_i,v_j) tiene asociado un coste c_{ij}, de tal forma que si v_0 es el vértice origen, y $v_0,v_1 \ \ldots \ v_k$ es la secuencia de vértices que forman el camino de v_0 a v_k, entonces $\sum_{i=0}^{k-1} C_{i,i+1}$ es la longitud del camino.

El algoritmo *voraz* de Dijkstra considera dos subconjuntos de vértices, F y $V-F$, donde V es el conjunto de todos los vértices. Se define la función `distancia(v)`: coste del camino más corto del origen s a v que pasa solamente por los vértices de F y tal que el primer vértice que se añade a F es el origen s. En cada paso se selecciona un vértice v de $V-F$ cuya *distancia* a F es la menor; el vértice v se *marca* para indicar que ya se conoce el camino mas corto de s a v. Se

siguen *marcando* nuevos vértices de V–F hasta que lo estén todos; en ese momento el algoritmo termina y ya se conoce el camino más corto del vértice origen s al resto de los vértices.

Para realizar esta estrategia *voraz,* el *array* D almacena el camino más corto (coste mínimo) desde el origen s a cada vértice del grafo. Se parte con el vértice origen s en F y los elementos D_i, del *array* D, con el coste o peso de los arcos (s, v_i); si no hay arco de s a i se pone *coste* ∞ . En cada paso se agrega algún vértice v de los que todavía no están en F, es decir, de V–F, que tenga el menor valor D(v). Además, se actualizan los valores D(w) para aquellos vértices w que todavía no están en F, $D(w) = D(v)+c_{v,w}$ cuando este nuevo valor de D(w) sea menor que el anterior. De esta forma se asegura que la incorporación de v implica que hay un camino de s a v cuyo coste mínimo es D(v).

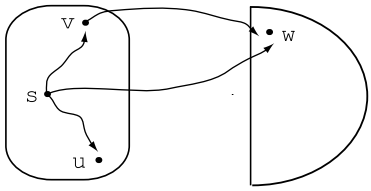


Figura 16.3 Al incorporar v, la distancia de s a w, D[w] puede mejorar

Para reconstruir el camino de *coste mínimo* que lleva de s a cada vértice v del grafo, se almacena, para cada uno de los vértices, el último vértice que hizo el coste mínimo. Entonces, asociado a cada vértice, resultan de interés dos datos: el *coste mínimo* y el último vértice del camino con el que se minimizó el *coste*.

Ejemplo 16.1

Dado el grafo dirigido y con pesos no negativos de la Figura 16.4 se desea calcular el coste mínimo de los caminos desde el vértice 1 a los otros vértices del grafo, aplicando el algoritmo de Dijkstra.

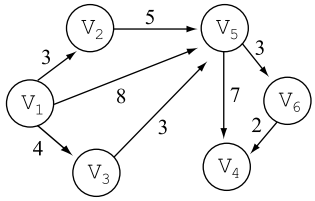


Figura 16.4 Grafo dirigido con factor de peso positivo

La matriz de pesos de los arcos, considerando peso ∞ cuando no existe arco es:

$$C = \begin{vmatrix} \infty & 3 & 4 & \infty & 8 & \infty \\ \infty & \infty & \infty & \infty & 5 & \infty \\ \infty & \infty & \infty & \infty & 3 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 7 & \infty & 3 \\ \infty & \infty & \infty & 2 & \infty & \infty \end{vmatrix}$$

Los sucesivos valores del conjunto `F` (vértices *marcados*), vértice `v` incorporado y el vector distancia, `D`, que se obtienen en cada paso se representan en la siguiente tabla:

Paso	F	v	D[2]	D[3]	D[4]	D[5]	D[6]
Inicial	1		3	4	∞	8	∞
1	1,2	2	3	4	∞	8	∞
2	1,2,3	3	3	4	∞	7	∞
3	1,2,3,5	5	3	4	14	7	10
4	1,2,3,5,6	6	3	4	12	7	10
5	1,2,3,5,6,4	4	3	4	12	7	10

Por ejemplo: el camino mínimo de v_1 a v_6 es 10; la secuencia de vértices que hacen el camino mínimo: v_1 - v_3 - v_5 - v_6 .

16.3.2. Codificación del algoritmo de Dijkstra

La clase `GrafMatPeso` representa un grafo valorado. La matriz de pesos contiene el *coste* de cada arco, o bien `INFINITO` si no hay arco entre dos vértices. Además, el método `pesoArco()` devuelve el peso o coste de un arco (se puede consultar la clase en la página web del libro).

Se escribe la clase `CaminoMinimo` para implementar el algoritmo y operaciones auxiliares. Los *arrays* `ultimo[]` y `D[]` representan la información asociada a cada vértice: el último vértice en el camino y el coste mínimo, siendo el vértice el índice del *array*. El conjunto `F` de vértices se representa mediante un *array* de tipo `boolean`, de tal forma que si el vértice `i` se incluye en `F`, entonces `F[i]` se pone a *true* (verdadero).

```
package Grafo;

public class CaminoMinimo
{
   private int [][] Pesos;
   private int [] ultimo;
   private int [] D;
   private boolean [] F;
   private int s, n;          // vértice origen y número de vértices
   public CaminoMinimo(GrafMatPeso gp, int origen)
   {
      n = gp.numeroDeVertices();
      s = origen;
      Pesos = gp.matPeso;
      ultimo = new int [n];
      D = new int [n];
      F = new boolean [n];
   }
   public void caminoMinimos()
   {
      // valores iniciales
      for (int i = 0; i < n; i++)
```

```
      {
         F[i] = false;
         D[i] = Pesos[s][i];
         ultimo[i] = s;
      }
      F[s] = true; D[s] = 0;
         // Pasos para marcar los n-1 vértices
      for (int i = 1; i < n; i++)
      {
         int v = minimo();  /* selecciona vértice no marcado
                                                    de menor distancia */
         F[v] = true;
         // actualiza distancia de vértices no marcados
         for (int w = 1; w < n; w++)
            if (!F[w])
               if ((D[v] + Pesos[v][w]) < D[w])
               {
                  D[w] = D[v] + Pesos[v][w];
                  ultimo[w] = v;
               }
      }
   }
   private int minimo()
   {
      int mx = GrafMatPeso.INFINITO;
      int v = 1;
      for (int j = 1; j < n; j++)
         if (!F[j] && (D[j]<= mx))
         {
            mx = D[j];
            v = j;
         }
      return  v;
   }

}
```

El tiempo de ejecución del algoritmo es cuadrático, se deduce que tiene una complejidad cuadrática, $O(n^2)$, debido a los dos bucles anidados de orden n (número de vértices). Ahora bien, en el caso de que el número de arcos, a, fuera mucho menor que n^2, el tiempo de ejecución mejora representando el grafo con listas de adyacencia. Entonces puede obtenerse un tiempo $O(a \log n)$.

Recuperación de los caminos

Los sucesivos vértices que conforman el camino mínimo se obtienen *volviendo hacia atrás*; es decir, desde el *último* que hizo que el camino fuera mínimo, al *último del último* y así sucesivamente hasta llegar al vértice origen del camino. Las llamadas recursivas del método (de la clase `CaminoMinimo`) que se escribe a continuación permiten, fácilmente, volver atrás y reconstruir el camino.

```
public void recuperaCamino(int v)
{
   int anterior = ultimo[v];
   if (v != s)
   {
      recuperaCamino(anterior); // vuelve al último del último
```

```
      System.out.print(" -> V" + v);
   }
   else
      System.out.print("V" + s);
   }
```

16.4. TODOS LOS CAMINOS MÍNIMOS: ALGORITMO DE FLOYD

En algunas aplicaciones resulta interesante determinar el camino mínimo entre todos los pares de vértices de un grafo dirigido y valorado. Considerando como vértice origen cada uno de los vértices del grafo, el algoritmo de Dijkstra resuelve el problema. Existe otra alternativa, más elegante y más directa, propuesta por Floyd que recibe el nombre de algoritmo de Floyd.

El grafo, de nuevo, está representado por la matriz de pesos, de tal forma que todo arco (v_i,v_j) tiene asociado un peso c_{ij}; si no existe arco, $c_{ij} = \infty$. Además, cada elemento de la diagonal, c_{ii}, se hace igual a 0. El algoritmo de Floyd determina una nueva matriz, D, de n x n elementos tal que cada elemento, D_{ij}, contiene el coste del camino mínimo de v_i a v_j .

El algoritmo tiene una estructura similar al algoritmo de Warshall para encontrar la matriz de caminos. Se generan consecutivamente las matrices $D_1, D_2, \ldots, D_k, \ldots, D_n$ a partir de la matriz D_o que es la matriz de pesos. En cada paso se incorpora un nuevo vértice y se estudia si con ese vértice se puede mejorar los caminos para ser más cortos. El significado de cada matriz es:

D_0[i,j] = c_{ij} *coste* (peso) del arco del vértice i al vértice j

∞ si no hay arco.

D_1[i,j] = minimo(D_0[i,j],D_0[i,1]+D_0[1,j]).

Menor de los costes entre el anterior camino mínimo de i a j y la suma de costes de caminos de i a 1 y de 1 a j.

D_2[i,j] = minimo(D_1[i,j],D_1[i,2]+D_1[2,j]).

Menor de los costes entre el anterior camino mínimo de i a j y la suma de los costes de caminos de i a 2 y de 2 a j.

En cada paso se incorpora un nuevo vértice para determinar si hace el camino menor entre un par de vértices.

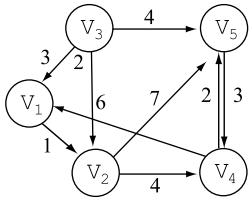
D_k[i,j] = minimo(D_{k-1}[i,j],D_{k-1}[i,k]+D_{k-1}[k,j]).

Menor de los costes entre el anterior camino mínimo de i a j y la suma de los costes de caminos de i a k y de k a j.

De forma recurrente, se añade en cada paso un nuevo vértice para determinar si se consigue un nuevo camino mínimo, hasta llegar al último vértice y obtener la matriz D_n, que es la matriz de caminos mínimos del grafo.

Ejemplo 16.2

La Figura 16.5 muestra un grafo dirigido con factor de peso y la correspondiente matriz de pesos. Aplicar el algoritmo de Floyd para obtener, en los sucesivos pasos, la matriz de caminos mínimos.



a)

$$C = \begin{vmatrix} 0 & 1 & \infty & \infty & \infty \\ \infty & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & \infty & 4 \\ 6 & \infty & \infty & 0 & 2 \\ \infty & \infty & \infty & 3 & 0 \end{vmatrix}$$

b)

Figura 16.5 *a)* Grafo dirigido valorado de 5 vértices; *b)* matriz de pesos del grafo

El grafo consta de cinco vértices; por ello se forman cinco matrices: D_1 , D_2 , D_3, D_4, D_5 que es la matriz de caminos mínimos. La matriz D_0 es la matriz de pesos C.

$$D_2 = \begin{vmatrix} 0 & 1 & \infty & \infty & \infty \\ \infty & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & \infty & 4 \\ 6 & 7 & \infty & 0 & 2 \\ \infty & \infty & \infty & 3 & 0 \end{vmatrix}$$

Al incorporar el vértice 1 ha cambiado $D_1(4,2)$ ya que $C(4,1) + C(1,2) < C(4,2)$.

$$D_2 = \begin{vmatrix} 0 & 1 & \infty & 5 & 8 \\ \infty & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & 6 & 4 \\ 6 & 7 & \infty & 0 & 2 \\ \infty & \infty & \infty & 3 & 0 \end{vmatrix}$$

Al incorporar el vértice 2 ha habido varios cambios; así $D_1(1,4)$: $D_1(1,2) + D_1(2,4) < D_1(1,4)$. También cambian $D_1(1,5)$, $D_1(2,5)$ y $D_1(3,4)$.

$$D_3 = \begin{vmatrix} 0 & 1 & \infty & 5 & 8 \\ \infty & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & 6 & 4 \\ 6 & 7 & \infty & 0 & 2 \\ \infty & \infty & \infty & 3 & 0 \end{vmatrix}$$

Al incorporar el vértice 3 no hay cambios; al vértice 3 no llega ningún arco.

$$D_4 = \begin{vmatrix} 0 & 1 & \infty & 5 & 7 \\ 10 & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & 6 & 4 \\ 6 & 7 & \infty & 0 & 2 \\ 9 & 10 & \infty & 3 & 0 \end{vmatrix}$$

Al incorporar el vértice 4 cambian los elementos: $D_3(1,5)$, $D_3(2,1)$, $D_3(5,1)$ y $D_3(5,2)$.

$$D_5 = \begin{vmatrix} 0 & 1 & \infty & 5 & 7 \\ 10 & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & 6 & 4 \\ 6 & 7 & \infty & 0 & 2 \\ 9 & 10 & \infty & 3 & 0 \end{vmatrix}$$

La incorporación del vértice 5 no genera cambios.

La matriz D_5 es la matriz de caminos mínimos. Dado cualquier par de vértices, se puede conocer la longitud del camino mas corto que hay entre ellos.

A tener en cuenta

Tanto el algoritmo de Dijkstra como el de Floyd permiten obtener los caminos mínimos, pero sólo se pueden aplicar en grafos valorados con *factor de peso* positivo, que son los mas frecuentes.

16.4.1. Codificación del algoritmo de Floyd

Se define la clase `TodoCaminoMinimo` para implentar el algoritmo al que se añade un pequeño cambio: guardar, en la matriz `traza`, el índice del último vértice que ha hecho que el camino sea mínimo desde el vértice v_i al v_j, para cada par de vértices del grafo.

```
package Grafo;

public class TodoCaminoMinimo
{
   private int [][] pesos;
   private int [][] traza;
   private int [][] d;
   private int n;           // número de vértices
   public TodoCaminoMinimo(GrafMatPeso gp)
   {
      n = gp.numeroDeVertices();
      pesos = gp.matPeso;
      d = new int [n][n];
      traza = new int [n][n];
   }
   public void todosCaminosMinimo()
   {
      // matriz inicial es la de pesos.
      for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
      {
         d[i][j] = pesos[i][j];
         traza[i][j] = -1; // indica que camino mas corto es el arco
      }
         //   Camino   mínimo   de   un   vértice   a   si   mismo:   0
      for (int i = 0; i < n; i++)
         d[i][i] = 0;
      for (int k = 0; k < n; k++)
         for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
               if ((d[i][k] + d[k][j]) < d[i][j])  // nuevo mínimo
               {
                  d[i][j] = d[i][k] + d[k][j];
                  traza[i][j]= k;
               }
   }
}
```

Eficiencia del algoritmo

El tiempo de ejecución del algoritmo crece rápidamente para grafos de relativamente muchos vértices, la complejidad es $O(n^3)$.

Para obtener la sucesión de vértices que forman el camino mínimo entre dos vértices se escribe el método `public void recuperaCamino(int vi, int vj)`, de igual forma que el método escrito en la clase `CaminoMinimo`, con la diferencia que ahora se utiliza la matriz `traza` en lugar de `ultimo`.

16.5. ÁRBOL DE EXPANSIÓN DE COSTE MÍNIMO

Los grafos no dirigidos se emplean para modelar relaciones simétricas entre *entes*, los vértices del grafo. Cualquier arista `(v,w)` de un grafo no dirigido está formada por un par no ordenado de vértices. Como consecuencia directa, la representación de un grafo no dirigido da lugar a matrices simétricas.

Una propiedad, que normalmente interesa conocer, de un grafo no dirigido es si para todo par de vértices existe un camino que los une; en definitiva, si el grafo es *conexo*. A los grafos conexos también se les denomina *Red Conectada*. El problema del *árbol de expansión de coste mínimo* consiste en buscar un árbol que *abarque* todos los vértices del grafo, con suma de pesos de aristas mínimo. Los árboles de expansión se aplican en el diseño de redes de comunicación.

Un *árbol*, en una red, es un subconjunto `G'` del grafo `G` que es conectado y sin ciclos. Los árboles tienen dos propiedades importantes:

1. Todo árbol de `n` vértices contiene exactamente `n-1` aristas.
2. Si se añade una arista a un árbol, se obtiene un ciclo.

> ***Definición***
>
> *Árbol de expansión de coste mínimo*: es un subconjunto del grafo que abarca todos los vértices que están conectados cuyas aristas tienen una suma de pesos mínima.

Buscar un árbol de expansión de un grafo, en una red, es una forma de averiguar si está conectado. Todos los vértices del grafo tienen que estar en el árbol de expansión para que sea un grafo conectado. La Figura 16.6 muestra un grafo no dirigido y su árbol de expansión; este grafo es una *red conectada*.

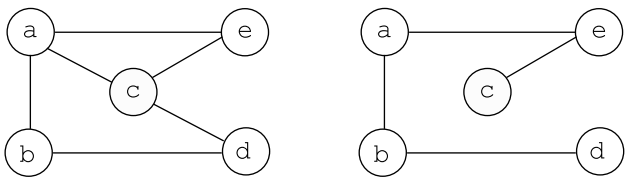


Figura 16.6 Grafo no dirigido y su árbol de expansión

El planteamiento del problema es el siguiente: dado un grafo no dirigido ponderado y conexo, encontrar el subconjunto del grafo compuesto por todos los vértices, con conexión entre cada par de vértices, sin ciclos y que cumpla que la suma de los pesos de las aristas sea mínimo. Estas operaciones encuentran el *árbol de expansión de coste mínimo*.

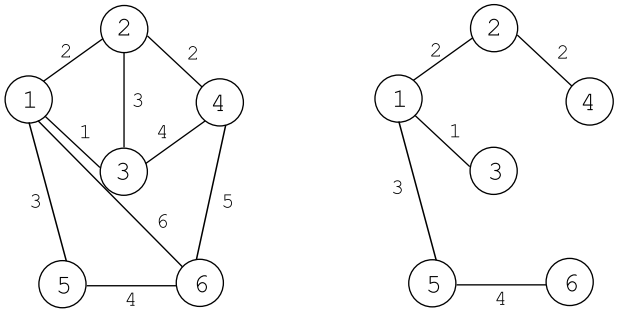


Figura 16.7 Grafo valorado y su árbol de expansión mínimo

16.5.1. Algoritmo de Prim

El algoritmo de Prim encuentra el árbol de expansión mínimo de un grafo no dirigido. Realiza sucesivos pasos, siguiendo la metodología clásica de los algoritmos *voraces:* hacer en cada paso *lo mejor que se pueda hacer*. En este problema, *lo mejor* consiste en incorporar al árbol una nueva arista del grafo de menor longitud.

Se parte de un grafo `G = (V,A)` no dirigido conectado, una red, donde `c(i,j)` es el *peso* o *coste* asociado al arco (v_i, v_j). Para describir el algoritmo, se suponen los vértices numerados de `1` a `n`. El conjunto `W` contiene los vértices que ya han pasado a formar parte del *árbol de expansión*.

El algoritmo arranca asignando un vértice inicial al conjunto `W`; por ejemplo el vértice `1`: `W = {1}`. A partir del vértice inicial, el *árbol de expansión* crece, añadiendo a `W`, en cada pasada otro vértice `z` todavía no incluido en `W`, de tal forma que si `u` es un vértice cualquiera de `W`, la arista `(u,z)` es la *más corta*, la de *menor coste*. El proceso termina cuando todos los vértices del grafo están en `W`, y por consiguiente, el *árbol de expansión* con todos los vértices está formado; además es mínimo porque en cada pasada se ha añadido la menor arista.

La Figura 16.8 muestra un grafo conectado, formado por `10` vértices. Inicialmente se incorpora al conjunto `W` el vértice `1`, `W = {1}`. De todas las aristas que forma parte el vértice `1`, la de menor coste es `(1,2)`, por esta razón se añade a `W` el vértice `2`, `W = {1, 2}`. La siguiente arista mínima, formada por cualquier vértice de `W` y los vértices no incorporados es `(2,4)`; entonces se añade a `W` el vértice `4`, `W = {1, 2, 4}`. La siguiente pasada añade el vértice `5`, al ser la arista `(4,5)` la mínima. Una nueva pasada incorpora el vértice `8` ya que la arista `(5,8)` es la de menor coste, `W = {1, 2, 4, 5, 8}`. A continuación se incorpora el vértice `7` ya que la arista `(5,7)` es la de menor coste, `W = {1, 2, 4, 5, 8, 7}`; le siguen los vértices `9`, `10`, `3` y termina el algoritmo con el vértice `6`, `W = {1, 2, 4, 5, 8, 7, 9, 10, 3, 6}`. La Figura 16.9 muestra el árbol de expansión de coste mínimo que se ha formado.

> ***Observación***
>
> En el sucesivos pasos del algoritmo de Prim, los vértices de `W` forman una componente conexa sin ciclos ya que las aristas elegidas tienen un vértice en `W` y el otro en los restantes vértices, `V–W`.

Otra forma de expresar el algoritmo de Prim: partiendo de un vértice inicial `u` se toma la arista menor `(u,v)` que no forme un ciclo, y de forma iterativa se toman nuevas aristas de menor peso `(z,w)`, sin dar lugar a ciclos y formando, en todo momento, una componente conexa.

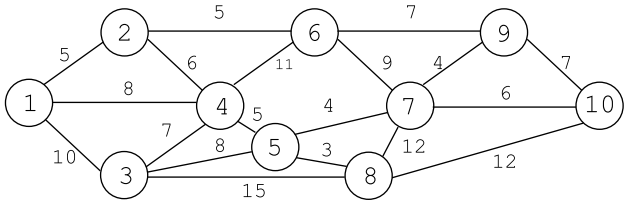


Figura 16.8 Grafo valorado conexo, representa una red telefónica

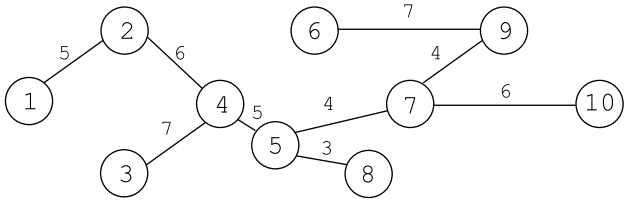


Figura 16.9 Árbol de expansión del grafo de la Figura 16.8

El algoritmo puede expresarse de la siguiente forma:

```
arbolExpansion Prim(G, T)
    G : grafo
    T : conjunto de arcos del árbol de coste mínimo
  var locales
    W: < conjunto de vértices >;
    u,w: vértices;
 inicio
    T <- {}
    V <- {1..n}
    u <- 1
    W <- {u}
    mientras W<> V hacer
      <Encontrar v de V-W tal que (u,v) sea mínimo>
      W <- W+{v}
      T <- T+{(u,v)}
    fin _ mientras
fin _ arbolExpansion
```

16.5.2. Codificación del algoritmo de Prim

La clase `ArbolExpansionMinimo` implementa los diversos algoritmos que calculan el árbol de expansión. El constructor inicializa la matriz de *costes* y el número de vértices.

Para resolver el problema de encontrar, en cada pasada, la arista de menor *peso* que une un vértice de `W` con otro de `V-W` se utilizan dos *arrays:*

- `masCerca`, tal que `masCerca[i]` contiene el vértice de `W` de menor *coste* respecto el vértice `i` de `V-W`.
- `coste`, tal que `coste[i]` contiene el peso de la arista `(i, masCerca[i])`.

En cada pasada se revisa `coste` con el fin de encontrar el vértice `z` de `V-W` que es el más cercano a `W`. A continuación se actualizan `masCerca` y `coste` teniendo en cuenta que `z` ha sido añadido a `W`.

```
package Grafo;

public class ArbolExpansionMinimo
{
   private int [][] Pesos;
   private int n; // vértice origen y número de vértices

   public ArbolExpansionMinimo(GrafMatPeso gp) // constructor
   {
      n = gp.numeroDeVertices();
      Pesos = gp.matPeso;
   }

   public int arbolExpansionPrim()    // implementación del algoritmo
   {
      int longMin, menor;
      int z;
      int [] coste = new int [n];
      int [] masCerca = new int [n];
      boolean [] W = new boolean [n];

      for (int i = 0; i < n; i++)
         W[i] = false;    // conjunto vacío
      longMin = 0;
      W[0] = true; //se parte del vértice 0
      // inicialmente, coste[i] es la arista (0,i)

      for (int i = 1; i < n; i++)
      {
         coste[i] = Pesos[0][i];
         masCerca[i] = 0;
      }
      for (int i = 1; i < n; i++)
      {  // busca vértice z de V-W mas cercano,
         // de menor longitud de arista, a algún vértice de W
         menor = coste[1];
         z = 1;
         for (int j = 2; j < n; j++)
            if (coste[j] < menor)
            {
               menor = coste[j];
               z = j;
            }
         longMin += menor;
         // se escribe el arco incorporado al árbol de expansión
         System.out.println("V" + masCerca[z] + " -> " + "V" + z);
         W[z] = true; // vértice z se añade al conjunto W
         coste[z] = GrafMatPeso.INFINITO;
         // debido a la incorporación de z,
         // se ajusta coste[] para el resto de vértices
         for (int j = 1; j < n; j++)
            if ((Pesos[z][j] < coste[j]) && !W[j])
            {
               coste[j] = Pesos[z][j];
```

```
                masCerca[j] = z;
            }
        }
        return longMin;
    }
}
```

16.5.3. Algoritmo de Kruscal

Kruskal propone otra estrategia para encontrar el árbol de expansión de coste mínimo. El árbol se empieza a construir con todos los vértices del grafo `G` pero sin aristas; se puede afirmar que cada vértice es una componente conexa en sí misma. El algoritmo construye componentes conexas cada vez mayores examinando las aristas del grafo en orden creciente del *peso*. Si la arista conecta dos vértices que se encuentran en dos componentes conexas distintas, entonces se añade la arista al árbol de expansión `T`. En el proceso, se descartan las aristas que conectan dos vértices pertenecientes a la misma componente, ya que darían lugar a un ciclo si se añaden al árbol de expansión ya que están en la misma componente. Cuando todos los vértices están en un solo componente, `T`, éste es el árbol de expansión de coste mínimo del grafo `G`.

El algoritmo de Kruskal asegura que el árbol no tiene ciclos, ya que para añadir una arista, sus vértices deben estar en dos componentes distintas; además es de coste mínimo, ya que examina las aristas en orden creciente de sus pesos.

La Figura 16.7 muestra un grafo y, a continuación, se obtiene su árbol de expansión, aplicando este algoritmo. En primer lugar se obtiene la lista de aristas en orden creciente de sus pesos:

```
{(1,3), (1,2), (2,4), (1,5), (2,3), (3,4), (5,6), (4,5), (1,6)}
```

La primera arista que se toma es `(1,3)`, como muestra la Figura 16.10a; a continuación, las aristas `(1,2)`, `(2,4)` y `(1,5)`, en las que se cumple que los vértices forman parte de dos componentes conexas diferentes, como se observa en la Figura 16.10b. La siguiente arista, `(2,3)` como sus vértices están en la misma componente, por tanto se descarta; la arista `(3,4)` por el mismo motivo se descarta. Por último, la arista `(5,6)`, que tiene sus vértices en dos componentes diferentes, pasa a formar parte del árbol, y el algoritmo termina ya que se han alcanzado todos los vértices.

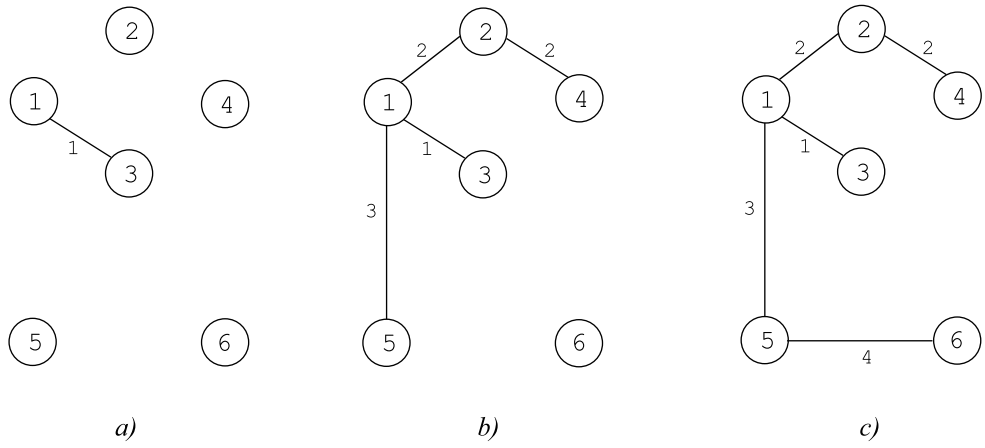


Figura 16.10 Formación del árbol de expansión del grafo de la Figura 16.7

RESUMEN

Si `G` es un *grafo dirigido sin ciclos*, entonces una ordenación topológica de `G` es una lista secuencial de los vértices de `G`, tal que para todos los vértices `v, w ∈ G` si existe una arista desde `v` a `w` entonces `v` precede a `w` en el listado secuencial. El término *acíclico* se utiliza con frecuencia para designar un grafo que no tiene ciclos.

La ordenación topológica es una forma de recorrer un grafo, pero sólo aplicable a grafos dirigidos acíclicos, que cumple la propiedad de que un vértice sólo se visita si ya han sido visitados todos sus predecesores. En un orden topológico, cada vértice debe aparecer antes que todos sus vértices sucesores en el *grafo dirigido*.

Una de las operaciones clave en un grafo es la búsqueda de *caminos mínimos*, es decir, caminos de menor longitud. La longitud de un camino se define como la suma de los *costes* de las aristas que lo componen. El *algoritmo de Dijkstra* determina el coste del camino mínimo desde un vértice al resto de vértices; es un algoritmo *voraz* que se aplica a un grafo valorado cuyas aristas tienen un *coste positivo*.

El *algoritmo de Warshall* es importante porque para todo par de vértices del grafo indica si hay un camino entre ellos. Con un estructura similar al anterior, el *algoritmo de Floyd* se utiliza para encontrar los caminos mínimos entre todos los pares de vértices de un grafo; también da la posibilidad de obtener la traza (secuencia de vértices) de esos caminos.

Un *árbol de expansión* es un árbol que contiene todos los vértices de una red. En un árbol no hay ciclos; es una forma de averiguar si la red está conectada. Uno de los problemas más comunes que se plantean sobre las redes es encontrar aquel *árbol de coste mínimo*. El *algoritmo de Prim* y el *algoritmo de Kruskal* resuelven eficientemente el problema de encontrar el árbol de expansión mínimo.