# 22

# Algoritmos sobre grafos

## Contenido



## Objetivos del capítulo

- los principales problemas que pueden ser representados mediante el uso de grafos y estudiar los algoritmos que los resuelven: Dijkstra, Prim, Kruskal.
- Comparar las implementaciones "*greddy*" y "dinámica" de algunos de estos algoritmos.

## Competencias específicas

- Analizar y resolver problemas de la vida real que pueden ser representados con grafos.

Leonhard Euler (1707 - 1783). Fue un matemático y físico suizo.
Es reconocido como uno de los matemáticos más importantes de su época, realizó descubrimientos muy importantes en diversas áreas como el cálculo o la Teoría de grafos.
Konigsberg, antiguo nombre de la actual Kaliningrado, es una ciudad portuaria de Rusia. Está situada en la desembocadura del río Pregel, que desemboca en el Lago del Vístula que, a su vez, desemboca en el Mar Báltico por el estrecho de Baltiysk.
En la Web de apoyo, encontrará el vínculo a Google Maps para ver los siete puentes de Kaliningrado.

## 22.1 Introducción

La teoría de Grafos fue desarrollada en 1736 por el matemático ruso Leonhard Euler para resolver el problema de los 7 puentes de la ciudad de Königsberg.

El problema consistía en determinar la existencia de algún camino que, partiendo desde un punto de la ciudad, permitiera recorrer los 7 puentes pasando solo una vez por cada uno y regresar al mismo punto de inicio.

Como vemos en la siguiente figura, la ciudad es atravesada por un río y entre costa y costa hay dos islas que, en aquella época, estaban unidas entre sí y, también, entre las dos costas mediante 7 puentes.

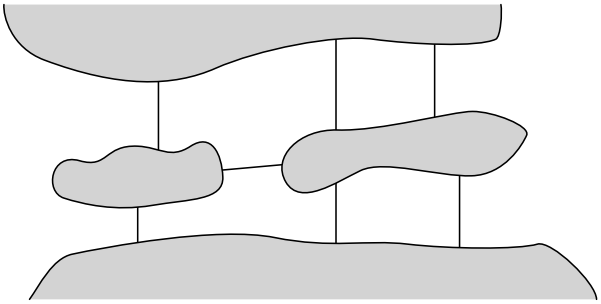


Fig. 22.1 Los 7 puentes de la ciudad de Königsberg.

Para solucionar el problema, Euler utilizó una simplificación del mapa representando a las regiones terrestres con puntos o círculos y a los puentes con líneas.

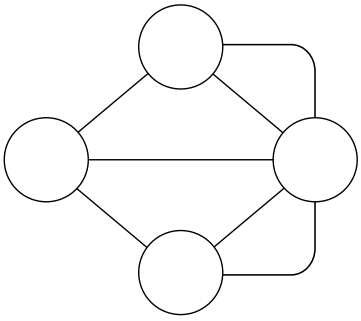


Fig. 22.2 Representación del problema de los 7 puentes mediante un grafo.

En esta abstracción Euler pudo observar que los puntos por los que el recorrido fuese a pasar deberían estar conectados por una cantidad par de líneas ya que, si llegamos a ese punto a través de un puente entonces, necesariamente, para retirarnos tendremos que utilizar un puente diferente.

Dado que en este problema todos los puntos están conectados con una cantidad impar de líneas, la conclusión a la que llegó fue que no existe un camino con las características del que buscamos. Luego, en este caso el problema no tiene solución.

Sin embargo, la abstracción desarrollada por Euler dio origen a la idea de “grafo” que, en la actualizad, tiene aplicación en diversos campos, como ser: física, química, arquitectura, ingeniería e informática.

Con un grafo podemos representar una red de carreteras que unen diferentes ciudades y analizar, por ejemplo, cuál es el camino más corto para llegar desde una ciudad hasta otra.

En este capítulo, además de estudiar conceptos teóricos, analizaremos algunos de los principales problemas que se representan mediante el uso de grafos y desarrollaremos los algoritmos para resolverlos.

## 22.2 Definición de grafo

Llamamos grafo a un conjunto de nodos o vértices que pueden estar o no conectados por líneas a las que denominaremos aristas.

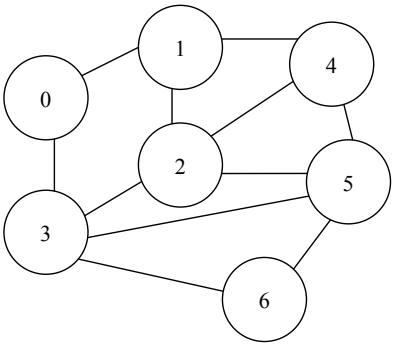



Fig. 22.3 Grafo.

Cada vértice del grafo se identifica con una etiqueta numérica o alfanumérica que lo diferencia de todos los demás. Por ejemplo, en el grafo de la figura anterior, los vértices están etiquetados con valores numéricos consecutivos, comenzando desde cero.

### 22.2.1 Grafos conexos y no conexos

Si, partiendo desde un vértice y desplazándonos a través de las aristas podemos llegar a cualquier otro vértice del grafo decimos que el grafo es "conexo". En cambio, si existe al menos un vértice al que no se puede llegar será porque está desconectado. En este caso, diremos que el grafo es "no conexo".

### 22.2.2 Grafos dirigidos y no dirigidos

Cuando las aristas están representadas con flechas, que indican una determinada dirección, decimos que se trata de un "grafo dirigido" o "dígrafo". En cambio, si las aristas no especifican un sentido en particular tendremos un grafo "no dirigido".

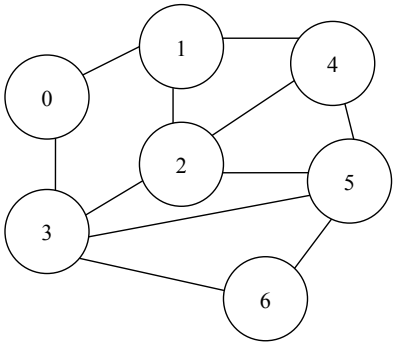



Fig. 22.4 Grafo no dirigido.

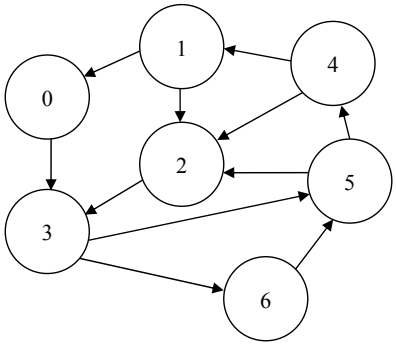



Fig. 22.5 Grafo dirigido.

### 22.2.3 El camino entre dos vértices

Sean *a* y *b* dos vértices de un grafo, diremos que el camino entre *a* y *b* es el trayecto o conjunto de aristas que debemos recorrer para llegar desde *a* hasta *b*. Si el grafo tiene ciclos entonces es probable que exista más de un camino posible para unirlos.

En un grafo dirigido debemos respetar la dirección de las aristas. Así, en el dígrafo de la figura anterior, el camino que une los vértices 5 y 0 es único. Partiendo desde 5 llegamos hasta 0 pasando por los siguientes vértices: *c* = {5, 4, 1, 0}. Este camino también podemos expresarlo como el conjunto de aristas por el que nos tendremos que desplazar: *c* = {(5,4), (4,1), (1,0)}

En el caso del grafo no dirigido que vemos en la parte izquierda de la figura, podemos encontrar varios caminos que nos permiten unir los vértices 5 y 0. Como las aristas no especifican ninguna dirección, entonces, podemos transitarlas en cualquier sentido. Algunos de estos caminos son:

*c*1 = {5, 2, 1, 0}

*c*2 = {5, 2, 3, 0}

*c*3 = {5, 3, 0}

### 22.2.4 Ciclos

Llamamos ciclo a un camino que, sin pasar dos veces por la misma arista, comienza y finaliza en el mismo vértice. Por ejemplo, en el dígrafo de la figura anterior algunos de los ciclos que podemos identificar son:

*c*1 = {0, 3, 6, 5, 4, 1, 0}

*c*2 = {0, 3, 5, 4, 1, 0}

*c*3 = {3, 6, 5, 2, 3}

*c*4 = {5, 4, 1, 2, 3, 5}

Y en el grafo no dirigido ubicado a la izquierda podemos destacar los siguientes ciclos:

*c*1 = {0, 1, 4, 2, 3, 0}

*c*2 = {2, 1, 0, 3, 2}

*c*3 = {6, 5, 4, 2, 1, 0, 3, 6}

### 22.2.5 Árboles

Un árbol es un grafo conexo y sin ciclos.

En la siguiente figura, a la izquierda, vemos el grafo con el que hemos estado trabajando desde el principio del capítulo. Luego, a la derecha, observamos cómo quedaría el grafo si suprimimos, arbitráriamente, algunas de sus aristas para eliminar la existencia de ciclos. Por ejemplo, suprimimos las aristas: (0, 3), (1, 4), (4, 5), (2, 3), (3, 6).

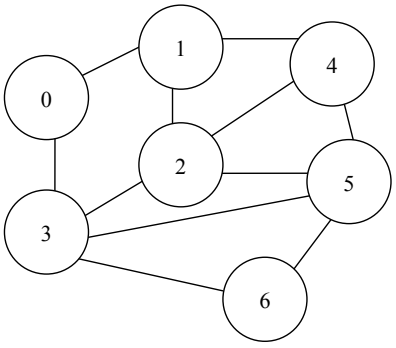



Fig. 22.6 Grafo conexo con ciclos.

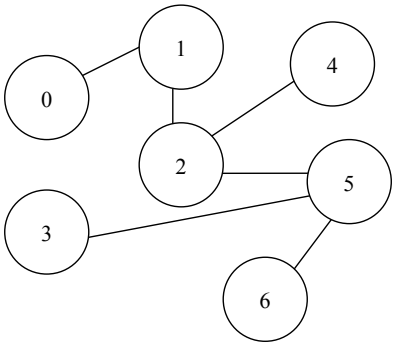



Fig. 22.7 Grafo conexo sin ciclos.

Luego, simplemente reacomodamos los vértices para poder visualizar el grafo con el formato natural de los árboles.

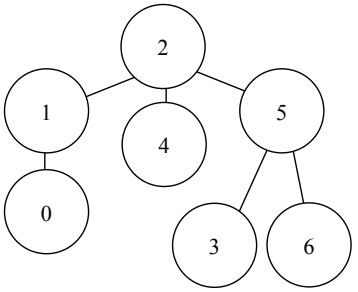



Fig. 22.8 Árbol: grafo conexo sin ciclos.

### 22.2.6 Grafos ponderados

Sobre las aristas de un grafo, dirigido o no, se puede indicar un determinado valor para representar, por ejemplo, un costo, una ponderación o una distancia. En este caso, decimos que el grafo está ponderado.

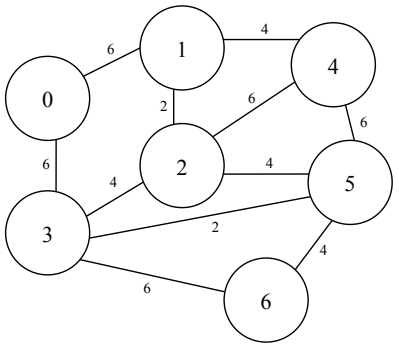



Fig. 22.9 Grafo ponderado no dirigido.

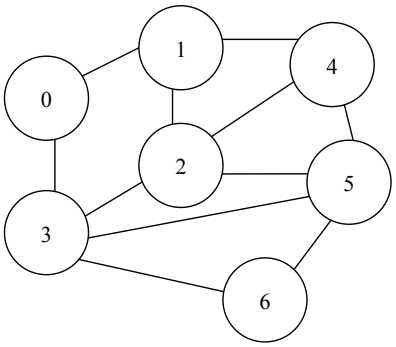
Por ejemplo, si los nodos del grafo representan ciudades entonces los valores de las aristas podrían representar las distancias que existen entre estas. O también podrían representar el costo que implicaría trasladarnos desde una ciudad hacia otra.

### 22.2.7 Vértices adyacentes y matriz de adyacencias

Cuando dos vértices están unidos por una arista decimos que son “vértices adyacentes”. Luego, utilizando una matriz, podemos representar las adyacencias entre los vértices de un grafo dirigido, no dirigido, ponderado o no ponderado.

La matriz de adyacencias es una matriz cuadrada, con tantas filas y columnas como vértices tenga el grafo que representa. Cada celda que se origina en la intersección de una fila i con una columna j representa la relación de adyacencia existente entre los vértices homólogos.

Para representar un grafo no dirigido ni ponderado, utilizaremos una matriz booleana. Las celdas de esta matriz tendrán el valor 1 (o *true*) para indicar que los vértices representados por la intersección fila/columna son adyacentes o 0 (o *false*) para indicar que ese par de vértices no lo son. Veamos:

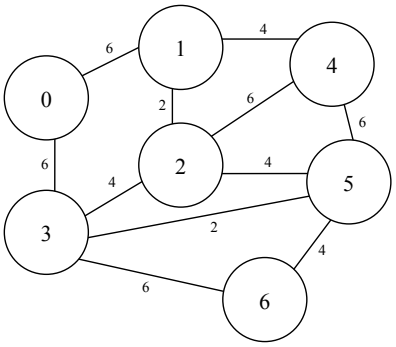



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| **0** | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| **1** | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| **2** | 0 | 1 | 0 | 1 | 1 | 1 | 0 |
| **3** | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| **4** | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| **5** | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| **6** | 0 | 0 | 0 | 1 | 0 | 1 | 0 |

Fig. 22.10 Matriz de adyacencias de un grafo no dirigido ni ponderado.

La matriz de adyacencias de un grafo no dirigido es simétrica ya que, para todo par de vértices *a* y *b*, si se verifica que *a* es adyacente con *b* entonces *b* será adyacente con *a*.

Veamos ahora la matriz de adyacencias de un grafo no dirigido y ponderado.




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| **0** | 0 | 6 | 0 | 6 | 0 | 0 | 0 |
| **1** | 6 | 0 | 2 | 0 | 4 | 0 | 0 |
| **2** | 0 | 2 | 0 | 4 | 6 | 4 | 0 |
| **3** | 6 | 0 | 4 | 0 | 0 | 2 | 6 |
| **4** | 0 | 4 | 6 | 0 | 0 | 6 | 0 |
| **5** | 0 | 0 | 4 | 2 | 6 | 0 | 4 |
| **6** | 0 | 0 | 0 | 6 | 0 | 4 | 0 |

Fig. 22.11 Matriz de adyacencias de un grafo no dirigido y ponderado

En este caso, la relación de adyacencia entre dos vértices *a* y *b* se reflejada cuando en la celda que surge como intersección de la fila *a* y la columna *b* existe un valor mayor que cero, que coincide con el valor de la arista que los une.

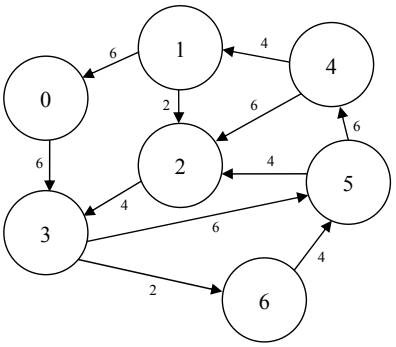
La no adyacencia de *a* con *b* puede representarse con el valor 0 en la celda intersección de la fila *a* con la columna *b* o con el valor "infinito" que, en Java, podemos implementar con: `Integer.MAX_VALUE`, siempre y cuando el tipo de datos de la matriz sea: `int[][]`.

En Java, los *wrappers* de todos los tipos de datos numéricos primitivos proveen una constante que representa el mayor valor que las variables de ese tipo pueden contener. Por ejemplo: `Integer.MAX_VALUE`, `Long.MAX_VALUE`, `Float.MAX_VALUE`, `Double.MAX_VALUE`, etc.

En Java, los *wrappers* de todos los tipos de datos numéricos primitivos proveen una constante que representa el mayor valor que las variables de ese tipo pueden contener. Por ejemplo:
`Integer.MAX_VALUE`,
`Long.MAX_VALUE`,
`Float.MAX_VALUE`,
`Double.MAX_VALUE`, etc.

En un grafo no dirigido, cada arista puede transitarse en cualquier dirección. Por esto, su matriz de adyacencias será simétrica ya que, si una arista conecta al vértice *a* con el vértice *b* entonces la misma arista conectará al vértice *b* con el vértice *a*,

Por último, en la matriz de adyacencias de un grafo dirigido las filas representan el "vértice origen" y las columnas representan al "vértice destino". Luego, si el grafo es ponderado, en la intersección de cada fila/columna se coloca el valor de la arista que conecta al par de vértices homólogos. Si no lo es, simplemente la celda intersección se completa con 1 o 0 o *true* o *false*.




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| **0** | | | | 6 | | | |
| **1** | 6 | | 2 | | | | |
| **2** | | | | 4 | | | |
| **3** | | | | | | 2 | 6 |
| **4** | | 4 | 6 | | | | |
| **5** | | | 4 | | 6 | | |
| **6** | | | | | | 4 | |

Fig. 22.12 Matriz de adyacencias de un grafo dirigido y ponderado.

### 22.2.8 La clase `Grafo`

Para facilitar la comprensión de los algoritmos que estudiaremos más adelante, vamos a desarrollar la clase `Grafo` que nos permitirá representar grafos no dirigidos y ponderados.

También, con el objetivo de no desviar la atención y podernos concentrar en la lógica de estos algoritmos, que no son para nada simples, aceptaremos que los vértices de los grafos que la clase podrá representar deberán ser numéricos, consecutivos y comenzar desde 0.

La clase tendrá una variable de instancia de tipo `int[][]` para guardar la matriz de adyacencias del grafo, que deberá ser provista como argumento en el constructor. La no adyacencia entre dos vértices será representada con “infinito”.

```
package libro.cap22;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;

public class Grafo
{
   // matriz de adyacencias
   private int matriz[][];

   // constructor, recibe la matriz de adyacencias
   public Grafo(int mat[][])
   {
      this.matriz = mat;
   }

   // sigue mas abajo
   // :
```

Agregaremos los métodos triviales `getDistancia` y `size`. El primero retornará la distancia que existe entre dos vértices del grafo. El segundo retornará la dimensión de la matriz de adyacencias que, como ya sabemos, es cuadrada.

```
   // :
   // viene de mas arriba

   // retorna la longitud de la matriz de adyacencias
   public int size()
   {
      return matriz.length;
   }

   // retorna el valor de matriz[a][b]
   public int getDistancia(int a, int b)
   {
      return matriz[a][b];
   }

   // sigue mas abajo
   // :
```

Ahora agregaremos dos constructores alternativos. El primero solo recibirá un valor entero indicando la dimensión de la matriz de adyacencias. Luego instanciará e inicializará la variable de instancia `matriz` asignando valores "infinito" en cada una de sus celdas. El segundo constructor alternativo permitirá construir un grafo con *n* nodos y especificar las adyacencias mediante un conjunto de aristas, cada una de las cuales estará determinada por dos vértices y una distancia. Las aristas las representaremos con instancias de la clase `Arista` cuyo código veremos a continuación.

```
package libro.cap22;

public class Arista
{
   private int n1;
   private int n2;
   private int distancia;

   // constructor
   public Arista(int n1,int n2, int dist)
   {
      this.n1 = n1;
      this.n2 = n2;
      this.distancia = dist;
   }

   // determinar si "yo", que soy una arista, soy igual a otra
   public boolean equals(Object a)
   {
      Arista otra = (Arista)a;

      return (n1==otra.n2 && n2==otra.n1)
          || (n1==otra.n1 && n2==otra.n2);
   }

   public String toString()
   {
      return "Desde: "+n1+", Hasta: "+n2+" Distancia: "+distancia;
   }

   // :
   // settters y getters
   // :
}
```

Además del constructor y el método `toString`, en la clase `Arista` sobrescribimos el método `equals` para determinar si los valores de las variables de instancia coinciden con los de la arista que recibimos como parámetro. El criterio que utilizamos para comparar dos aristas es simple: son iguales si coinciden sus extremos directos u opuestos.

Agreguemos a la clase `Grafo` el código de los constructores alternativos.

```
// :
// viene de mas arriba

// recibe la dimension de la matriz, la instancia
// y le asigna "infinito" a cada celda
public Grafo(int n)
{
   // matriz cuadrada de n filas y n columnas
   matriz = new int[n][n];

   // inicializo la matriz
   for(int i=0;i<n; i++)
   {
      for(int j=0; j<n; j++)
      {
         matriz[i][j] = Integer.MAX_VALUE;
      }
   }
}

// recibe la dimension de la matriz, y un conjunto de aristas
// los nodos no afectados por las aristas no seran adyacentes
public Grafo(int n, Arista[] arr)
{
   // invoco al otro constructor para inicializar la matriz
   this(n);

   for(Arista a:arr)
   {
      addArista(a);
   }
}

// agregar una arista implica asignar la distancia
// en las celdas correspondientes
public void addArista(Arista a)
{
   matriz[a.getN1()][a.getN2()]=a.getDistancia();
   matriz[a.getN2()][a.getN1()]=a.getDistancia();
}

// sigue mas abajo
// :
```

Por último, incluiremos un método llamado `getVecinos` que retornará los vértices adyacentes de un determinado nodo que recibirá como parámetro.

```
   // :
   // viene de mas arriba

   // retorna los nodos adyacentes
   public ArrayList<Integer> getVecinos(int n)
   {
      ArrayList<Integer> a = new ArrayList<Integer>();
      for( int i=0; i<matriz.length; i++ )
      {
         if( matriz[n][i]!=Integer.MAX_VALUE )
         {
            a.add(i);
         }
      }

      return a;
   }
}
```

### 22.2.10 Determinar la existencia de un ciclo

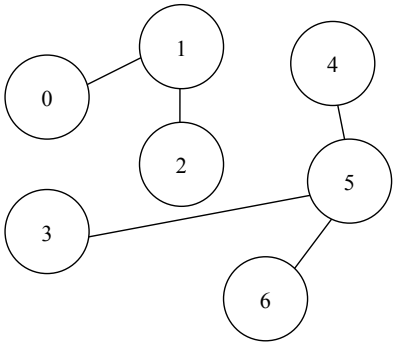
Como introducción a los algoritmos que estudiaremos más adelante, analizaremos un pequeño programa para determinar si la acción de agregar una nueva arista a un grafo hará que este pase a tener ciclos.

#### Enunciado

Dado un grafo *G* de *n* vértices y un conjunto de aristas, se pide determinar si, al agregar una arista adicional esta posibilitará o no la existencia de un ciclo. La cantidad de nodos, las aristas iniciales y la arista que debemos analizar se ingresarán por consola.

#### Análisis

Observemos el grafo de la siguiente figura:



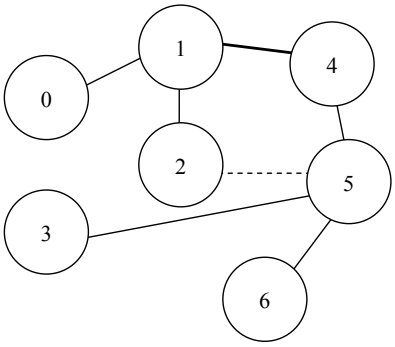

Como podemos ver, el grafo es no conexo ya que los vértices 0, 1, y 2 están desconectados de los vértices 3, 4, 5 y 6.

Lo representaremos con la clase `Grafo`.

```
// 7 nodos etiquetados entre 0 y 6
Grafo g = new Grafo(7);
g.addArista(new Arista(0,1));
g.addArista(new Arista(1,2));
g.addArista(new Arista(3,5));
g.addArista(new Arista(4,5));
g.addArista(new Arista(5,6));
```

Fácilmente, podemos observar que si agregamos una arista conectando los vértices 3 y 6 se formará un ciclo. Lo mismo sucedería si agregásemos una arista que conecte los vértices 0 con 2, 3 con 4 o 4 con 6. Cualquier otra arista que agreguemos unirá los dos pequeños árboles, pero no formará ningún ciclo.

Agreguemos, por ejemplo, una conexión entre los vértices 4 y 1:




Ahora, cualquier arista que vayamos a agregar indefectiblemente formará un ciclo. La línea punteada muestra cómo quedaría el grafo si además agregásemos la arista (2,5).

El algoritmo para determinar si esta arista formará un ciclo debe emprender un recorrido a través de los nodos del grafo y, comenzando desde alguno de los extremos de la arista, debe controlar que en el camino no pasemos por el otro extremo.

Por ejemplo, si partiendo desde 2 podemos llegar a 5 será porque estos nodos ya estaban conectados entonces la arista formará una nueva conexión: un ciclo. Como el grafo no está dirigido, podemos iniciar el recorrido por cualquiera de los extremos de la arista. Por esto, arbitrariamente comenzaremos el análisis partiendo desde el nodo 2.

La estrategia consiste en procesar el vértice 2 (nodo inicial) y verificar si entre sus vecinos se encuentra 5 (nodo final). Veamos:

Nodo en proceso: 2 | *vecinos*(2) = {1}

Como 5 (el nodo final de la arista) no es vecino de 2, repetiremos la operación procesando a cada uno de sus vecinos.

Nodo en proceso: 1 | *vecinos*(1) = {0, 2, 4}

Como 5 tampoco está entre los vecinos de 1 repetiremos el proceso con cada uno de sus vecinos comenzando, arbitrariamente, por el primero: 0.

Nodo en proceso: 0 | *vecinos*(0) = {1}

Evidentemente, la estrategia nos condujo a un *loop* del cual no podremos salir ya que el vecino de 0 es 1 y el primero de los vecinos de 1 es 0. La misma situación se dará entre los vértices 2 y 1 ya que 1 es vecino de 2 y 2 es vecino de 1.

La solución a este problema será "recordar" los nodos que ya hemos procesado para no volverlos a considerar. Siendo así, repetiremos el análisis evaluando, nuevamente, a los vecinos de 2, que es el vértice inicial de la arista en cuestión.

Nodo en proceso: 2 | *vecinos*(2) = {1} | Procesados = {2}

Ya procesamos al nodo 2. Como 5 no está entre sus vecinos, repetiremos la operación procesando a cada uno de ellos:

Nodo en proceso: 1 | *vecinos*(1) = {0, ~~2~~, 4} | Procesados = {2,1}

Ignoramos a 2 porque "recordamos" que este nodo ya fue procesado. Como 5 no está entre los vecinos de 1, nuevamente repetimos la operación con cada uno de ellos.

Nodo en proceso: 0 | *vecinos*(0) = {~~1~~} | Procesados = {0,2,1}

El único vecino de 0 es 1 pero como ya lo hemos procesado lo descartamos y continuamos con el nodo 4 que, como vecino de 1, estaba pendiente de ser procesado.

Nodo en proceso: 4 | *vecinos*(4) = {~~1~~,5} | Procesados = {0,2,1,4}

Como 1 ya fue procesado lo descartamos. Luego, el único vecino de 4 es 5.

Podemos ver que partiendo desde 2, vértice inicial de la arista que estamos analizando, pudimos llegar hasta 5, el vértice final. Esto nos permite determinar que, de agregarla al grafo, estaremos formando un ciclo.

Para implementar el algoritmo utilizaremos un *arraylist* y una cola. El *arraylist* nos permitirá "recordar" los nodos que ya hemos procesado. La cola nos permitirá encolar los vecinos de cada uno de los nodos a medida que los vayamos procesando.

Veamos el programa principal:

```
package libro.cap22.ciclos;

import libro.cap22.Arista;
import libro.cap22.Grafo;
import java.util.Scanner;

public class VerificaCiclos
{
   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);

      // ingresa x consola los valores del grafo
      Grafo g = ingresaGrafo(scanner);

      // ingresa x consola los valores de la arista
      Arista a = ingresaAristaAVerificar(scanner);

      if( hayCiclo(g,a) )
      {
         System.out.println("La arista forma un ciclo");
      }
      else
      {
         System.out.println("La arista no forma ciclo");
      }
   }

   // sigue mas abajo
   // :
```

Veamos ahora el método `hayCiclo` donde verificaremos si la arista `a` forma un ciclo entre los vértices del grafo `g`.

```
   // :
   // viene de mas arriba

   private static boolean hayCiclo(Grafo g, Arista a)
   {
      // arraylist para recordar los nodos procesados
      ArrayList<Integer> procesados = new ArrayList<Integer>();

      // cola de nodos pendientes de proceso
      Queue<Integer> q = new LinkedList<Integer>();
```

```
        // extremos de la arista
        int nInicio = a.getN1();
        int nFin = a.getN2();

        // encolamos el extremo inicial
        q.add(nInicio);

        while( !q.isEmpty() )
        {
            // desencolo y proceso un nodo
            int n = q.poll();

            // llegamos al extremo final?
            if( n==nFin )
            {
                return true; // HAY CICLO
            }

            // agrego el nodo a la lista de nodos procesados...
            procesados.add(n);

            // pido sus vecinos...
            ArrayList<Integer> vecinos = g.getVecinos(n);

            // encolo a sus vecinos, salvo los que ya fueron procesados
            for(int v:vecinos)
            {
                if( !procesados.contains(v) )
                {
                    q.add(v);
                }
            }
        }

        // si llegamos hasta aqui sera porque la arista no forma ciclo
        return false;
    }

    // sigue mas abajo
    // :
```

Por último, solo para completar el programa, veamos los métodos que permiten el ingreso de datos por consola.

```
    // :
    // viene de mas arriba

    private static Grafo ingresaGrafo(Scanner scanner)
    {
        System.out.println("--Ingreso del grafo --");
        System.out.print("Cantidad de vertices: ");
        int n = scanner.nextInt();

        Grafo g = new Grafo(n);
```

```
        System.out.print("Cantidad de aristas: ");
        int a = scanner.nextInt();

        System.out.println("- Ingrese "+a+" aristas -");
        for( int i=0; i<a; i++ )
        {
           g.addArista( ingresaArista(scanner) );
        }

        return g;
     }

     private static Arista ingresaAristaAVerificar(Scanner scanner)
     {
        System.out.println("-- Ingrese la arista a verificar --");
        return ingresaArista(scanner);
     }

     private static Arista ingresaArista(Scanner scanner)
     {
        System.out.print("Extremo inicial: ");
        int n1 = scanner.nextInt();

        System.out.print("Extremo final: ");
        int n2 = scanner.nextInt();

        System.out.print("Distancia: ");
        int d = scanner.nextInt();

        Arista a = new Arista(n1, n2, d);
        System.out.println("Arista ingresada: "+a);

        return a;
     }
}
```

### 22.2.11 Los nodos “vecinos”

Como vimos en el ejemplo anterior, el concepto de “nodo vecino” es un poco más amplio que el concepto de “nodo adyacente” ya que, generalmente, no estaremos interesados en procesar un mismo nodo más de una vez. Por esto, extenderemos el concepto de “vecinos” de un vértice de la siguiente manera:

Sea *s* un vértice de un grafo, entonces llamaremos *vecinos*(*s*) a todos sus nodos adyacentes que aún no hayan sido procesados.

Para aplicar este concepto al método `getVecinos` de la clase `Grafo` tendremos que agregar una variable de instancia de tipo `ArrayList` que nos permita “recordar” cuales nodos han sido procesados. También agregaremos los métodos `setProcesado` para que el usuario pueda “marcar” como procesado a un vértice determinado, `isProcesado` que, dado un vértice, indicará si el mismo previamente fue o no marcado como “procesado” y `resetProcesados` que permitirá “olvidar” todos los nodos marcados.

```
// :
public class Grafo
{
   private int matriz[][];
   private ArrayList<Integer> procesados = new ArrayList<Integer>();

   // :

   public void setProcesado(int n)
   {
      procesados.add(n);
   }

   public boolean isProcesado(int n)
   {
      return procesados.contains(n);
   }

   public void resetProcesados()
   {
      procesados.clear();
   }

   // :
}
```

Ahora sí, modifiquemos el método `getVecinos` para retornar solo aquellos nodos adyacentes que previamente no hayan sido marcados como “procesados”.

```
   // :

   // retorna los adyacentes que no han sido procesados
   public ArrayList<Integer> getVecinos(int n)
   {
      ArrayList<Integer> a = new ArrayList<Integer>();
      for( int i=0; i<matriz.length; i++ )
      {
         if(matriz[n][i]!= Integer.MAX_VALUE)
         {
            // lo agrego solo si no fue procesado
            if(!procesados.contains(i))
            {
               a.add(i);
            }
         }
      }

      return a;
   }

   // :
```

Ahora agregaremos el método `hayCiclo` en la clase `Grafo`. Notemos que antes de “tocar” el *arraylist* `procesados` resguardamos su estado inicial y lo restauramos antes

de finalizar el algoritmo.

```
// :
public boolean hayCiclo(int nInicio, int nFin)
{
   // resguardo el estado actual de los nodos procesados
   ArrayList<Integer> bkpProcesados = (ArrayList<Integer>) procesados.clone();
   resetProcesados();

   // encolo b y veo si llego hasta a
   Queue<Integer> q = new LinkedList<Integer>();
   q.add(nInicio);

   while( !q.isEmpty() )
   {
      int n = q.poll();

      if( n==nFin )
      {
         return true;
      }

      setProcesado(n);

      // obtengo los vecinos
      ArrayList<Integer> vecinos = getVecinos(n);

      for(int x: vecinos)
      {
         q.add(x);
      }
   }

  // restauro el estado de nodos procesados que el grafo tenia al inicio
   procesados = bkpProcesados;

   return false;
}

// :
```

Al resguardar el conjunto de nodos procesados nos aseguramos de no alterar el estado que el grafo tenía antes de ejecutar el método `hayCiclo`.

Observemos bien la siguiente línea:

```
ArrayList<Integer> bkpProcesados = (ArrayList<Integer>) procesados.clone();
```

Si hubiéramos asignado directamente `procesados` a `bkpProcesados`, este objeto sería una referencia al primero. Luego, no podríamos restaurar el *arraylist* original ya que durante todo el proceso solo existiría un único *arraylist* y una referencia a este.

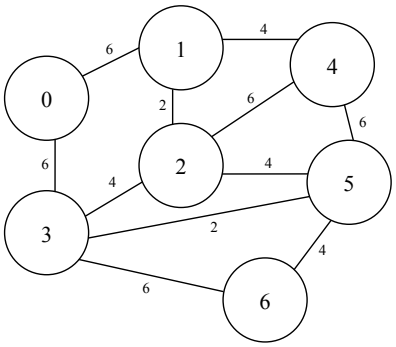
El método `clone` crea una nueva instancia con los mismos valores que el objeto original. Luego de aplicarlo existirán dos *arraylists*, cada uno de estos referenciado por los objetos `procesados` y `bkpProcesados` respectivamente.

## 22.3 El problema de los caminos mínimos

Dado un grafo ponderado y uno de sus vértices, al que llamaremos "vértice de origen", se requiere hallar las menores distancias existentes entre este vértice y todos los demás.

Es decir, sea *s* el vértice de origen e *i* cualquier otro nodo del grafo entonces la menor distancia entre *s* e *i* será la menor de las sumatoria de los valores de las aristas que componen cada uno de los caminos que conectan a *s* con *i*.

Repasemos nuestro grafo ponderado.




Considerando como vértice de origen al nodo "0" entonces, para llegar desde 0 hasta 5 podemos tomar varios caminos:

*c1* = {0, 1, 4, 5}, distancia = 16
*c2* = {0, 1, 2, 5}, distancia = 12
*c3* = {0, 1, 4, 2, 5}, distancia = 20
*c4* = {0, 1, 2, 4, 5}, distancia = 20
*c5* = {0, 3, 2, 1, 4, 5}, distancia = 20
*c6* = {0, 3, 2, 4, 5}, distancia = 22
*c7* = {0, 3, 2, 5}, distancia = 14
*c8* = {0, 3, 5}, distancia = **8**
*c9* = {0, 3, 6, 5}, distancia = 16

Como vemos, de todos los caminos que conectan al nodo 0 con el nodo 5, el menos costoso o el más corto es: *c8* = {0, 3, 5}.

Edsger Wybe Dijkstra (1930 - 2002). Estudió física teórica en la Universidad de Leiden.
La solución del problema del camino más corto fue una de sus contribuciones a la informática, también conocido como el algoritmo de Dijkstra.
En la Web de apoyo, encontrará el vínculo a la página Web personal de Edsger Dijkstra.

### 22.3.1 Algoritmo de Dijkstra

El algoritmo de Dijkstra ofrece una solución al problema de hallar los caminos mínimos proveyendo las menores distancias existentes entre un vértice y todos los demás.

Como analizaremos dos implementaciones diferentes desarrollaremos una *interface* que nos permita desacoplar el programa y la implementación.

La *interface* Dijkstra definirá un único método tal como lo vemos a continuación.

```
package libro.cap22.dijkstra;

import java.util.Hashtable;
import libro.cap22.Grafo;

public interface Dijkstra
{
   public Hashtable<Integer, Integer> procesar(Grafo g, int s);
}
```

El método `procesar` recibe como parámetros el grafo, el vértice de origen `s` y retorna un conjunto solución montado sobre una *hashtable* tal que sus *keys* sean los vértices del grafo y los *values* sean las distancias mínimas que existen entre `s` y cada uno de estos vértices.

### 22.3.2 Dijkstra, enfoque greddy

Antes de comenzar analizaremos algunos casos concretos sobre nuestro grafo modelo para lo cual vamos a considerar como vértice de origen al nodo 4.

**Algoritmo de Dijkstra por greddy**

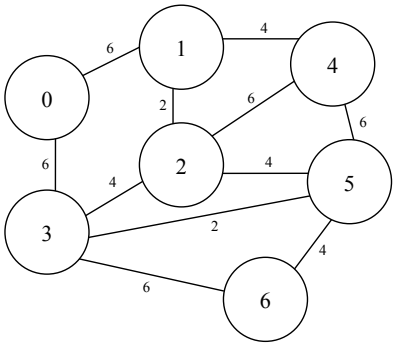



Fig. 22.13 Grafo ponderado.

Siendo así, queremos obtener las menores distancias existentes entre el nodo 4 y cada uno de los otros nodos del grafo.

Comenzaremos analizando sus vecinos, que son: {1, 2, 5}. Las distancias que existen entre 4 y cada uno de estos nodos son, respectivamente: 4, 6 y 6. Por el momento consideraremos que estas distancias son mínimas y, mientras no puedan ser mejoradas, formarán parte del conjunto solución. Luego procesaremos cada uno de los vecinos de 4 comenzando, arbitrariamente, por 5.

Los vecinos de 5, obviando a 4 que ya fue procesado, son: {2, 3, 6} y sus distancias son, respectivamente, 4, 2 y 4. Sin embargo, para llegar a 5 desde 4 "pagamos" un costo inicial de 6. Por esto, momentáneamente, las mínimas distancias que existen entre 4 y cada uno de los vecinos de 5 son 6+4, 6+2 y 6+4.

Sean *a* y *b* dos nodos adyacentes de un grafo, llamaremos *d*(*a*,*b*) a la distancia directa entre *a* y *b*, que coincide con el valor expresado en la arista que los une.

Veamos: pasando por 5, la distancia entre 4 y 3 es 8 ya que *d*(4,5)=6+*d*(5,3)=2. En algún momento procesaremos el vértice 2, vecino de 4, desde donde también podemos llegar a 3. Sin embargo, para llegar a 3 desde el nodo de origen 4, pasando por 2, la distancia total es 10 y se obtiene sumando: *d*(4,2)=6+*d*(2,3)=4. Este valor es peor que el costo de llegar a 3 a través de 5, razón por la cual lo descartaremos.

El algoritmo de Dijkstra consiste en procesar cada uno de los vecinos de *s*, el vértice de origen, luego los vecinos de los vecinos de *s* y así sucesivamente hasta llegar a procesar todos los nodos del grafo.

Como los nodos deben procesarse solo una vez, utilizaremos la funcionalidad provista por la clase `Grafo`, desarrollada más arriba, cuyo método `getVecinos` retorna todos los nodos adyacentes a un vértice obviando aquellos que previamente hayan sido marcados como "procesados" mediante el método `setProcesado`.

Veamos cómo crear una instancia de `Grafo` en función de su matriz de adyacencias. Recordemos que la matriz debe contener la distancia que existe entre dos nodos adyacentes o el valor `Integer.MAX_VALUE` para aquellos nodos que no lo son.

```
public static void main(String[] args)
{
   int i = Integer.MAX_VALUE;
   int[][] mAdy= { {i, 6, i, 6, i, i, i}
                  ,{6, i, 2, i, 4, i, i}
                  ,{i, 2, i, 4, 6, 4, i}
                  ,{6, i, 4, i, i, 2, 6}
                  ,{i, 4, 6, i, i, 6, i}
                  ,{i, i, 4, 2, 6, i, 4}
                  ,{i, i, i, 6, i, 4, i} };

   // instancio el grafo a partir de su matriz de adyacencias
   Grafo g = new Grafo(mAdy);

   // :
}
```

Esta implementación del algoritmo de Dijkstra utilizará dos *hashtables*: una para componer el conjunto solución (`distMin`) y otra para mantener las distancias acumuladas (`distAcum`). Las *keys* de estas tablas serán las etiquetas de todos los nodos del grafo y los valores iniciales deben ser "infinito" para el conjunto solución ya que, en definitiva, se trata de hallar las distancias mínimas y 0 para las distancias acumuladas que no son otra cosa que acumuladores.

Trabajaremos con una cola donde, inicialmente, encolaremos al vértice de origen. Luego iteraremos mientras que la cola no esté vacía y en cada una de estas iteraciones desencolaremos un nodo, lo procesaremos y encolaremos a todos sus vecinos.

**Nota:** recordemos que para simplificar la lógica del algoritmo hemos aceptado la restricción de que los vértices del grafo solo podrán estar etiquetados con valores numéricos, consecutivos y comenzando desde cero.

```
package libro.cap22.dijkstra.imple;

import java.util.ArrayList;
import java.util.Hashtable;
import java.util.LinkedList;
import java.util.Queue;

import libro.cap22.Grafo;
import libro.cap22.dijkstra.Dijkstra;

public class DijkstraImpleGreddy implements Dijkstra
{
   public Hashtable<Integer, Integer> procesar(Grafo g, int s)
   {
      // asigno "infinito" a cada posicion de distMin
      Hashtable<Integer, Integer> distMin = new Hashtable<Integer, Integer>();
      inicializar(distMin, g.size(), Integer.MAX_VALUE);

      // asigno 0 a cada posicion de distAcum
      Hashtable<Integer, Integer> distAcum = new Hashtable<Integer, Integer>();
      inicializar(distAcum, g.size(), 0);

      Queue<Integer> q = new LinkedList<Integer>();
      q.add(s);

      // continua mas abajo
      // :
```

Luego, dentro del `while` tomamos un elemento de la cola y lo procesamos. El proceso consiste en marcarlo como "procesado" y analizar la distancia que existe entre este y cada uno de sus vecinos.

```
        // :
        // viene de mas arriba

        while(!q.isEmpty())
        {
           // tomo el primero
           int n = q.poll();

           // lo marco como visitado
           g.setProcesado(n);

           int acum = distAcum.get(n);

           // continua mas abajo
           // :
```

Al inicio, la distancia acumulada de cada nodo será cero. Si sumamos la distancia acumulada del nodo `n`, que acabamos de desencolar, con la distancia existente entre este y cada uno de sus vecinos podremos determinar si esta suma es menor a la distancia que, hasta ahora, considerábamos mínima. Si así fuese entonces actualizaremos `distMin`. Luego ingresamos a un `for` para comparar la distancia que existe entre `n` y cada uno de sus vecinos.

```
           // :
           // viene de mas arriba

           // obtengo los vecinos de n
           // pido sus vecinos
           ArrayList<Integer> vecinos = g.getVecinos(n);

           for(int i = 0; i<vecinos.size(); i++)
           {
              // tomo el i-esimo vecino y su distancia
              int t = vecinos.get(i);
              int dist = g.getDistancia(n, t) + acum;
              int min = distMin.get(t);

              if( dist<min )
              {
                 distMin.put(t, dist);
                 distAcum.put(t, dist);
              }

              if(!q.contains(t))
              {
                 q.add(t);
              }
           }
        }

        return distMin;
     }

     // continua mas abajo
     // :
```

Dentro del `for` tomamos el *i-ésimo* vecino de `n` y calculamos la distancia que existe entre estos nodos sumando la distancia directa más la distancia que acumula `n`. Luego comparamos esta suma con el valor que, por el momento, consideramos como la distancia mínima. Si la nueva distancia mejora a la anterior entonces la guardamos.

Solo queda pendiente el código del método `inicializar` que utilizamos para asignar valores iniciales a las *hashtables*.

```
   // :
   // viene de mas arriba

   private void inicializar(Hashtable<Integer, Integer> tdist, int size, int val)
   {
      for(int i=0; i<size;i++)
      {
         tdist.put(i,val);
      }
   }
}
```

Veamos ahora un programa que reciba por línea de comandos al vértice de origen y muestre por pantalla las menores distancias que existen entre este vértice y todos los demás.

```
package libro.cap22.dijkstra;
import java.util.Hashtable;

public class TestDijkstra
{
   public static void main(String[] args)
   {
      int i = Integer.MAX_VALUE;

      int[][] mAdy= { {i, 6, i, 6, i, i, i}
                     ,{6, i, 2, i, 4, i, i}
                     ,{i, 2, i, 4, 6, 4, i}
                     ,{6, i, 4, i, i, 2, 6}
                     ,{i, 4, 6, i, i, 6, i}
                     ,{i, i, 4, 2, 6, i, 4}
                     ,{i, i, i, 6, i, 4, i} };

      // instancio el grafo a partir de su matriz de adyacencias
      Grafo g = new Grafo(mAdy);

      // defino el nodo de origen
      int origen = Integer.parseInt(Integer.parseInt(args[0]);

      Dijkstra d = new DijkstraImpleGreddy();
      Hashtable<Integer, Integer> minimos = d.procesar(g, origen);

      // muestro los resultados
      System.out.println(minimos);
   }
}
```

Esta implementación del algoritmo coincide con un enfoque "voraz" o *greddy*. Veamos:

- El conjunto de candidatos: los vértices del grafo, excepto el nodo "origen".
- El método de selección: será quién determine el menor costo desde el origen.
- La función de factibilidad determinará si, luego de seleccionar un nodo, conviene utilizarlo como "puente" para mejorar el costo entre el origen y cada uno de sus vecinos.

La complejidad del algoritmo es cuadrática y está determinada por los dos ciclos anidados: un `for` dentro del `while`.

Algoritmo de Dijkstra por dinámica

### 22.3.3 Dijkstra, implementación por programación dinámica

La programación dinámica ofrece una técnica simple y eficiente para los problemas que, además de cumplir con el principio de óptimalidad, puedan ser planteados de manera recursiva. Veamos entonces si podemos plantear el algoritmo de Dijkstra como un problema de programación dinámica.

Existe el principio de óptimalidad ya que, si entre dos vértices *a* y *c*, encontramos al vértice *b* entonces los caminos parciales que van de *a* a *b* y de *b* a *c* serán mínimos.

Para formular un planteamiento recursivo del problema, pensaremos en un *arraylist* `solu` (solución) tal que sus elementos coincidirán con las distancias mínimas que existen entre los nodos del grafo, representados en las posiciones homólogas del *array*, y el nodo de origen. Esto es: en la *i-ésima* posición del *arraylist* tendremos la mínima distancia existente entre el nodo *i* y el nodo de origen. Esta tabla será inicializada con las distancias directas entre el nodo de origen y cada uno de sus nodos adyacentes o "infinito" para los nodos que no son adyacentes al origen.

Para simplificar, en el siguiente análisis nos referiremos a este *arraylist* como *S*, y aceptaremos que *S*(*j*) representa la distancia existente entre el nodo de origen y el nodo *j*.

También, con el objetivo de reducir la notación diremos que *M* es la matriz de adyacencias y aceptaremos que *M*(*i*,*j*) representa la distancia directa existente entre los nodos *i* y *j* o "infinito" si estos nodos no son adyacentes.

Luego, considerando un nodo *j* tal que *j*>=0 y *j*<*n*, debemos asignar en la *j-ésima* posición de *S* el resultado de la siguiente expresión:

*min*( *S*(*j*), *S*(*k*)+*M*(*k*,*j*) ), siendo *k*>=0 y *k*<*n*, y *n* la cantidad de nodos del grafo.

Veamos el algoritmo:

```
package libro.cap22.dijkstra.imple;

import java.util.Hashtable;
import libro.cap22.Grafo;
import libro.cap22.dijkstra.Dijkstra;

public class DijkstraImpleDinamica implements Dijkstra
{
   public Hashtable<Integer, Integer> procesar(Grafo g, int s)
   {
      // tabla con la solucion del problema
      Hashtable<Integer, Integer> solu = new Hashtable<Integer, Integer>();
```

```
        // inicializo la solucion de la siguiente manera:
        // * las distancias a los nodos adyacentes o
        // * INFINITO para los nodos no adyacentes

        for(int i = 0; i<g.size(); i++)
        {
           int d = g.getDistancia(s, i);
           solu.put(i, d);
        }

        // seteo como "procesado" al nodo de origen
        g.setProcesado(s);

        // comparo todos contra todos
        for( int i = 0; i<g.size()-1; i++ )
        {
           // retorna la posicion del menor valor contenido en la tabla
           int posMin = menor(g, solu);

           // lo marco como visitado
           g.setProcesado(posMin);

           for( int j=0; j<g.size(); j++ )
           {
              if( !g.isProcesado(j) )
              {
                 // veo si puedo mejorar las distancias
                 int x = sumar(solu.get(posMin), g.getDistancia(posMin, j));
                 solu.put(j, Math.min(solu.get(j), x));
              }
           }
        }

        return solu;
     }

     // sigue mas abajo
     // :
```

Probablemente, el lector se haya sorprendido al ver que utilicé el método `sumar` para obtener la suma entre los valores `solu.get(posMin)` y `g.getDist(posMin, j)`.

Pues bien, analicemos la siguiente línea:

```
int x = sumar(solu.get(posMin), g.getDist(posMin, j));
```

Como `g.getDist(posMin, j)` podría llegar a ser `Integer.MAX_VALUE` entonces este es el mayor valor que una variable de tipo `int` puede contener. Si a esto le sumamos cualquier otro valor estaremos desbordando la capacidad de la variable `x` y el resultado será impredecible.

El método `sumar`, cuyo código veremos a continuación, suma ambos valores manteniendo el concepto de "infinito". Es decir: infinito es un valor "tan grande" que al sumarle cualquier otro valor continuará siendo infinito. Decimos que es absorbente para la suma.

```
   // :
   // viene de mas arriba

   private int sumar(int a, int b)
   {
      if( a==Integer.MAX_VALUE || b==Integer.MAX_VALUE )
      {
         return Integer.MAX_VALUE;
      }
      else
      {
         return a+b;
      }
   }

   // sigue mas abajo
   // :
```

Por último, veremos el método `menor` que retorna la posición dentro del *arraylist* que, temporalmente, representa la menor distancia entre el nodo de origen y el nodo homólogo a dicha posición. Recordemos que, en el *arraylist*, las posiciones representan vértices y los contenidos representan distancias.

```
   // :
   // viene de mas arriba

   private int menor(Grafo g, Hashtable<Integer,Integer> solu)
   {
      int min = Integer.MAX_VALUE;
      int pos = 0;

      for(int i = 0; i<g.size(); i++)
      {
         if(!g.isVisitado(i) && solu.get(i)<min)
         {
            min = solu.get(i);
            pos = i;
         }
      }

      return pos;
   }
}
```

La complejidad del algoritmo implementado por programación dinámica sigue siendo cuadrática ya que se mantienen los dos ciclos anidados. Sin embargo, el código es más corto y legible respecto de la implementación anterior.

Veamos un programa donde el usuario selecciona cuál implementación del algoritmo de Dijkstra desea utilizar para hallar las distancias mínimas entre un vértice y todos los demás.

```
package libro.cap22.dijkstra.test;

import java.util.Hashtable;
import java.util.Scanner;

import libro.cap22.Grafo;
import libro.cap22.dijkstra.Dijkstra;

public class Test
{
   public static void main(String[] args) throws Exception
   {
      Scanner scanner = new Scanner(System.in);

      int i = Integer.MAX_VALUE;
      int[][] mAdy= { {i, 6, i, 6, i, i, i}
                     ,{6, i, 2, i, 4, i, i}
                     ,{i, 2, i, 4, 6, 4, i}
                     ,{6, i, 4, i, i, 2, 6}
                     ,{i, 4, 6, i, i, 6, i}
                     ,{i, i, 4, 2, 6, i, 4}
                     ,{i, i, i, 6, i, 4, i} };

      // instancio el grafo a partir de su matriz de adyacencias
      Grafo g = new Grafo(mAdy);

      System.out.print("Ingrese el vertice de origen: ");
      int origen = scanner.nextInt();

      System.out.println("Ingrese implementacion de Dijkstra: ");
      String sClassImple = scanner.next();

      // instancio dinamicamente la implementacion ingresada por el usuario
      Dijkstra d = (Dijkstra)Class.forName(sClassImple).newInstance();

      // invoco el metodo procesar
      Hashtable<Integer, Integer> minimos = d.procesar(g, origen);

      // muestro los resultados
      System.out.println(minimos);
   }
}
```

Si queremos que el programa ejecute la implementación dinámica para calcular las distancias mínimas hacia el vértice de origen 6 ingresaremos lo siguiente:

```
Ingrese el vertice de origen: 6
Ingrese implementacion de Dijkstra:
libro.cap22.dijkstra.imple.DijkstraImpleDinamica
{6=2147483647, 5=4, 4=10, 3=6, 2=8, 1=10, 0=12}
```

Y si queremos utilizar la implementación *greddy*, debemos ejecutarlo e ingresar:

```
Ingrese el vertice de origen: 6
Ingrese implementacion de Dijkstra:
libro.cap22.dijkstra.imple.DijkstraImpleGreddy
{6=2147483647, 5=4, 4=10, 3=6, 2=8, 1=10, 0=12}
```

En este ejemplo utilizamos la clase `Class` para instanciar, dinámicamente, la implementación de Dijkstra ingresada por el usuario. Observemos la siguiente línea:

```
// instancio dinamicamente la implementacion ingresada por el usuario
Dijkstra d = (Dijkstra)Class.forName(sClassImple).newInstance();
```

El método `Class.forName` instancia dinámicamente un objeto de la clase indicada en la cadena `sClassImple`, ingresada por el usuario. Este es el punto de entrada al mundo de la programación por introspección que, en Java, se encuentra detrás de la API de *reflection* ubicada en el paquete `java.lang.reflect`.
Si bien, el tema excede el alcance de este libro, el mismo es de suma importancia ya que permite llevar al extremo los conceptos de abstracción.
La programación por introspección es la base para el desarrollo de herramientas genéricas y *frameworks* como Hibernate, Spring, Struts, ZK, etc.

## 22.4 Árbol de cubrimiento mínimo (MST)

Dado un grafo *G*, llamaremos árbol generador mínimo, árbol de cubrimiento mínimo o, simplemente, MST (*Minimum Spanning Tree*) de *G* a un árbol *A* que tiene los mismos vértices que *G*, pero solo un subconjunto de sus aristas. Dado que *A* es un árbol, será conexo y acíclico y solo conservará aquellas aristas de *G* que permitan conectar directa o indirectamente a todos sus nodos con el menor costo posible.
Es decir que el costo total requerido para pasar por todos los vértices del árbol será mínimo.

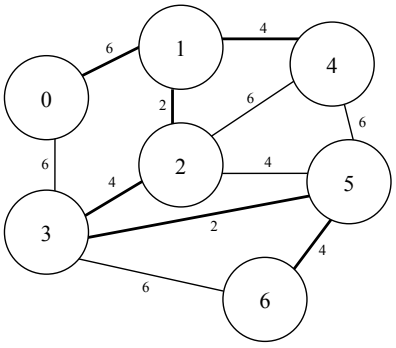



Fig. 22.14 Grafo ponderado.

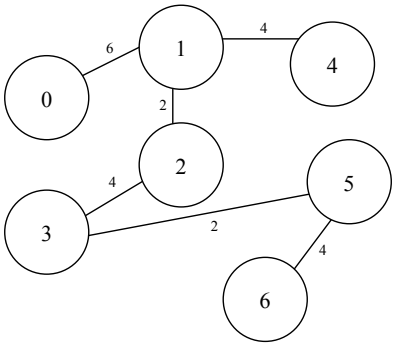



Fig. 22.15 Árbol de cubrimiento mínimo.

Robert C. Prim, nació en 1921 en Estados Unidos, es matemático e ingeniero en sistemas.
En los laboratorios Bell, trabajó como director de investigación matemática. Allí desarrolló el conocido Algoritmo de Prim. Con su compañero, Joseph Kruskal, desarrolló dos algoritmos diferentes para encontrar los árboles abarcadores mínimos en un grafo ponderado.

### 22.4.1 El algoritmo de Prim

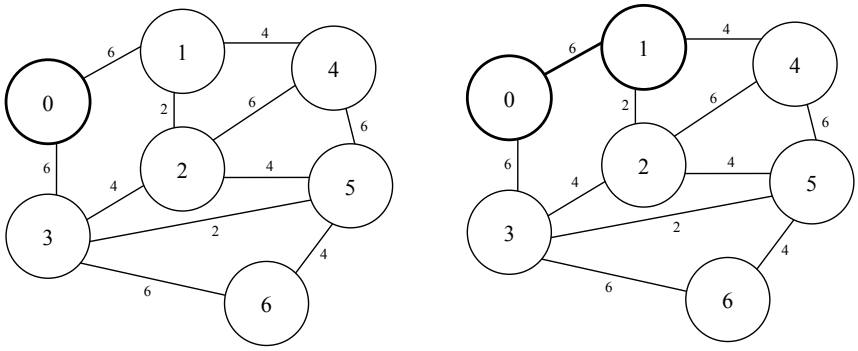
El algoritmo de Prim soluciona el problema de encontrar el árbol generador mínimo de un grafo a partir de un nodo inicial que se ubicará como la raíz del árbol generador. Luego, progresivamente, se incorporarán nuevos vértices hasta que el MST quede completo.
Comenzando por un nodo cualquiera, se evalúan las distancias directas entre este y cada uno de sus vecinos. Se selecciona el vecino que tenga la menor distancia y se lo incorpora al árbol. Luego, se repite la operación considerando los vecinos de cada uno de los vértices que ya fueron incorporados.
Así, en cada iteración el árbol incorpora un nuevo vértice. El algoritmo finaliza cuando todos los vértices del grafo hayan sido incorporados al árbol.
Nuevamente, tal como lo hicimos con Dijkstra, definiremos una *interface* para independizar la implementación del algoritmo. Así, luego de estudiar el método de Prim podremos analizar una solución alternativa para hallar el árbol generador mínimo de un grafo ponderado: el algoritmo de Kruskal.

```
package libro.cap22.mst;

import libro.cap22.Grafo;

public interface Mst
{
   // retorna el arbol generador minimo del grafo g a partir del vertice n
   public Grafo procesar(Grafo g, int n);
}
```

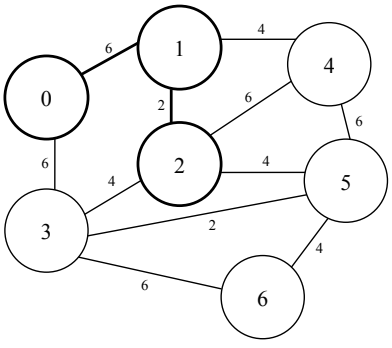
La estrategia del algoritmo de Prim consiste en armar el árbol recubridor mínimo del grafo `g`, que recibimos como parámetro en el método `procesar`, escogiendo el orden en el que vamos a incorporar cada uno de sus vértices en función de los valores de las aristas que los unen.

Algoritmo de Prim

Supongamos que el vértice de inicio será el nodo 0. Entonces, este nodo será el primero en incorporarse al árbol ocupando el lugar de la raíz. Luego, analizamos cuál de todos sus vecinos se encuentra ubicado a menor distancia. Como en este caso todos sus vecinos son equidistantes, tomaremos cualquiera de ellos, por ejemplo: 1.

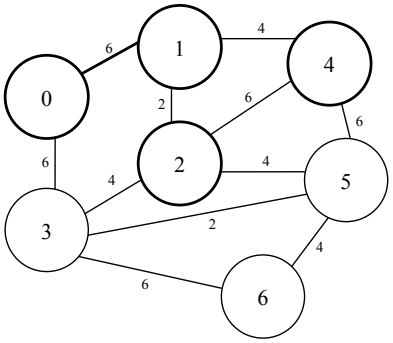



Ahora repetimos la operación, pero considerando los vecinos de 1 y los vecinos de 0, siempre y cuando no hayan sido procesados. Es decir: el conjunto de vértices a analizar es: *vecinos*(1) = {2, 4} y *vecinos*(0) = {3}. De todos estos vértices, el que se encuentra a menor distancia es 2; por lo tanto, este será el próximo nodo que debemos incorporar.



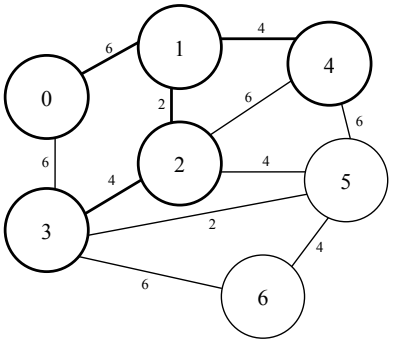

Ahora, el conjunto de vecinos a analizar es el siguiente: *vecinos*(0) = {3}, *vecinos*(1) = {4} y *vecinos*(2) = {3, 4, 5}. Recordemos que descartamos aquellos nodos que ya fueron incorporados al árbol.

Luego, de todo el conjunto de vecinos, los más cercanos son 4 (llegando por 1), 5 (llegando por 2) y 3 (llegando por 2). Todos estos vecinos son equidistantes por lo que podemos seleccionar cualquiera de ellos. Optaremos por incorporar a 4 a través de 1.




Claramente, el próximo vecino que debemos incorporar será 3 o 5, ambos pasando por 2. Estos nodos se conectan a 2 con una distancia de 4, la menor entre todas las posibilidades.

La elección entre estos dos nodos será arbitraria. Escogeremos a 3.




Visiblemente, el próximo vecino que debemos incorporar será 5 a través de 3 y finalmente conectaremos a 6 a través de 5.

Con esto habremos formado un árbol con todos los vértices del grafo original, pero conservando solo aquellas aristas que minimizan el camino entre estos.

Comparemos entonces el grafo original con su árbol generador mínimo hallado mediante la aplicación del algoritmo de Prim.

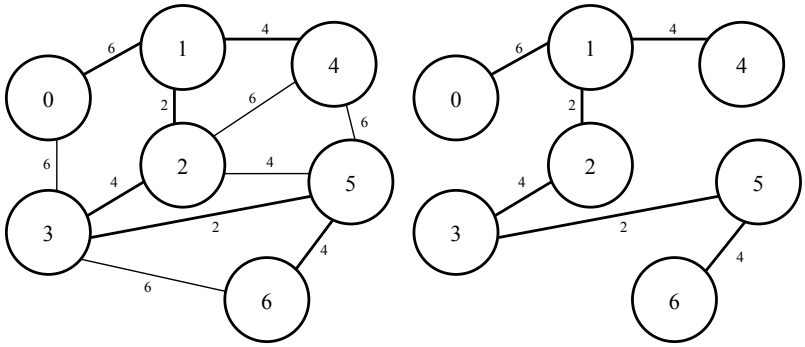



Fig. 22.16 Comparación del grafo original con el árbol generador mínimo hallado por Prim.

Veamos entonces la clase `MstImplePrim` que implementa la *interface* `Mst`.

```
package libro.cap22.mst.imple;

import java.util.ArrayList;

import libro.cap22.Arista;
import libro.cap22.Grafo;
import libro.cap22.mst.Mst;
```

```
public class MstImplePrim implements Mst
{
   public Grafo procesar(Grafo g, int s)
   {
      // nodos incorporados al arbol, inicialmente vacio
      ArrayList<Integer> incorporados = new ArrayList<Integer>();

      // arbol recubridor minimo implementado como una instancia de Grafo
      Grafo solu = new Grafo(g.size());

      // en cada iteracion procesamos al nodo n, comenzando por s
      int n = s;
      for(int i=0; i<g.size()-1; i++)
      {
         // marco a n como "procesado" y lo incorporo al array
         g.setProcesado(n);
         incorporados.add(n);

         // de todos los nodos incorporados busca el nodo no imporporado
         // cuya distancia sea la menor...
         int incorporar[] = buscarMinimos(g,incorporados);
         int desde = incorporar[0]; // nodo ya incorporado
         int hasta = incorporar[1]; // nodo a incorporar...
         int dist = g.getDistancia(desde,hasta);

         // agrego al arbol la arista que conecta el nuevo nodo
         solu.addArista(new Arista(desde, hasta, dist));

         // en la proxima iteracion marcaremos como "procesado"
         // al nodo que acabamos de incorporar al arbol solucion
         n = hasta;
      }

      return solu;
   }

   // sigue mas abajo
   // :
```

El método `buscarMinimos` analiza los vecinos de todos los nodos que contiene el *arraylist* `incorporados` y retorna aquel que se encuentre ubicado a menor distancia. El método retorna un `int[2]` donde el primer elemento es el "nodo desde" el cual nos conectaremos hacia el próximo nodo que debemos incorporar. Este último ubicado en la segunda posición del *array*.

```
   // :
   // viene de mas arriba

   private int[] buscarMinimos(Grafo g, ArrayList<Integer> incorporados)
   {
      int ret[]=new int[2] ;

      int min = Integer.MAX_VALUE;
      for(int i=0; i<incorporados.size(); i++)
      {
         int desde = incorporados.get(i);
         ArrayList<Integer> vecinos = g.getVecinos(desde);
```

```
            for(int j=0; j<vecinos.size();j++)
            {
               int vecino = vecinos.get(j);
               if( vecino!=desde && g.getDistancia(desde,vecino)<min )
               {
                  min = g.getDistancia(desde,vecino);
                  ret[0]=desde;
                  ret[1]=vecino;
               }
            }
         }

         return ret;
      }
   }
```

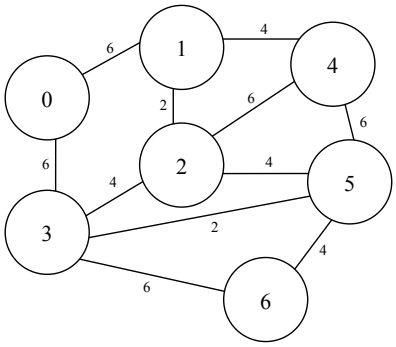
Joseph B. Kruskal (1928 2010). Fue un matemático y estadístico estadounidense.
Investigador en el Bell-Labs, en 1956, descubrió un algoritmo para la resolución del problema del árbol recubridor mínimo, el cual es un problema típico de optimización combinatoria.

Algoritmo de Kruskal

### 22.4.2 Algoritmo de Kruskal

El algoritmo de Kruskal también ofrece una solución al problema de hallar el árbol recubridor mínimo de un grafo, pero en este caso el análisis pasa por las aristas.

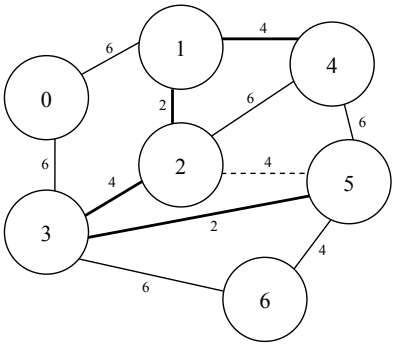
La estrategia consiste en incorporar, una a una, las aristas de menor peso, siempre y cuando no formen ciclos. Dado que buscamos formar un árbol la cantidad de aristas que debemos incorporar será *n*-1 siendo *n* la cantidad de vértices que tiene el grafo original.




Comenzaremos analizando el conjunto de aristas del grafo, ordenándolo de menor a mayor según su distancia. A igual distancia la elección es arbitraria pero nosotros daremos prioridad a las que conecten nodos con etiquetas de menor valor.

Así, el conjunto ordenado de aristas será el siguiente: {(1,2,2), (3,5,2), (1,4,4), (2,3,4), (2,5,4), (5,6,4), (0,1,6), (0,3,6), (2,4,6), (3,6,6), (4,5,6) }

Cada terna representa *nodoDesde*, *nodoHasta* y la distancia entre ambos.




Las aristas que agregaremos al conjunto solución serán (1,2,2), (3,5,2), (1,4,4) y (2,3,4). La siguiente arista en el conjunto ordenado es (2,5,4), indicada con una línea punteada. Claro que al incorporar esta arista formaríamos un ciclo, por esta razón la descartamos y analizamos las siguientes.

El árbol se completa con las aristas (5,6,4), (0,1,6) que no forman ciclos y suman la cantidad de *n*-1 aristas necesarias para unir a todos los vértices del grafo original.

El lector podrá observar que el árbol que obtuvimos con el algoritmo de Kruskal coincide con el árbol arrojado por el algoritmo de Prim.

Para simplificar la lógica del algoritmo, antes de comenzar agregaremos a la clase `Grafo` el método `getAristas`.

```
// :
public class Grafo
{
    // :

    public ArrayList<Arista> getAristas()
    {
        ArrayList<Arista> arr = new ArrayList<Arista>();
        for( int i=0; i<matriz.length; i++ )
        {
            for(int j=0; j<matriz.length; j++ )
            {
                if( matriz[i][j] != Integer.MAX_VALUE )
                {
                    Arista a = new Arista(i, j, matriz[i][j]);

                    // recordemos que sobrescribimos el metodo
                    // equals indicando que la arista (a,b)
                    // identica a la arista (b,a)
                    if( !arr.contains(a) )
                    {
                        arr.add(a);
                    }
                }
            }
        }

        return arr;
    }

    // :
}
```

Ahora sí, veamos la clase `MstImpleKruskal` que implementa la *interface* `Mst` siguiendo estos lineamientos.

```
package libro.cap22.mst.imple;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

import libro.cap22.Arista;
import libro.cap22.Grafo;
import libro.cap22.mst.Mst;

public class MstImpleKruskal implements Mst
{
   public Grafo procesar(Grafo g, int s)
   {
      // tomo el conjunto de aristas del grafo original
      ArrayList<Arista> aristas = g.getAristas();
```

```
            // las ordeno segun su distancia, de menor a mayor
            Collections.sort(aristas, new ComparaArista());

            // instancio el grafo solucion (arbol, conexo y sin ciclos)
            Grafo solu = new Grafo(g.size());

            // agrego exactamente n-1 aristas
            int i=0;
            while( !aristas.isEmpty() && i<g.size()-1 )
            {
               // tomo la primer arista del array, la de menor peso
               Arista a = aristas.get(0);

               // si no hay ciclo la agrego y cuento "una arista mas"
               if( !solu.hayCiclo(a.getN1(),a.getN2()) )
               {
                  solu.addArista(a);
                  i++;
               }

               // la remuevo del conjunto de aristas disponibles
               aristas.remove(0);
            }

            return solu;
         }

         class ComparaArista implements Comparator<Arista>
         {
            public int compare(Arista a1, Arista a2)
            {
               return a1.getDistancia()-a2.getDistancia();
            }
         }
      }
```

La clase `ComparaArista` es una *inner class*. Java permite definir clases dentro de otras clases con el objetivo de reducir la cantidad de archivos de código fuente. La idea de incluir una clase adentro de otra es útil cuando la clase interna es de uso exclusivo de la clase contenedora. En nuestro caso, al menos por ahora, la clase `ComparaArista` solo es utilizada por la implementación del algoritmo de Kruskal.

Notemos que el hecho de que una clase esté codificada dentro del cuerpo de otra no afecta para nada la relación de herencia que pueda existir entre ambas. En el caso de las clases, `MstImpleKruskal` y `ComparaArista`, ambas heredan de `Object`. Además, la primera implementa la *interface* `Mst` mientras que la segunda implementa la *interface* `Comparator`. Solo eso.

Volviendo al algoritmo de Kruskal, notemos lo simple que resultó su implementación. En parte, esto se debe a que el método `hayCiclo` que desarrollamos en la clase `Grafo` casi al principio del capítulo nos permite determinar si, al conectar dos nodos con una arista, estaremos o no formando un ciclo en nuestro grafo.

Para terminar, veamos un programa donde el usuario ingresa un vértice inicial y la implementación que prefiere utilizar para hallar el árbol de cubrimiento mínimo del grafo que hemos utilizado en este capítulo.

```
package libro.cap22.mst.test;

import java.util.Scanner;

import libro.cap22.Arista;
import libro.cap22.Grafo;
import libro.cap22.mst.Mst;

public class Test
{
   public static void main(String[] args) throws Exception
   {
      Scanner scanner = new Scanner(System.in);

      int i = Integer.MAX_VALUE;
      int[][] mAdy = { { i, 6, i, 6, i, i, i }
                     , { 6, i, 2, i, 4, i, i }
                     , { i, 2, i, 4, 6, 4, i }
                     , { 6, i, 4, i, i, 2, 6 }
                     , { i, 4, 6, i, i, 6, i }
                     , { i, i, 4, 2, 6, i, 4 }
                     , { i, i, i, 6, i, 4, i } };

      // instancio el grafo a partir de su matriz de adyacencias
      Grafo g = new Grafo(mAdy);

      System.out.print("Ingrese el nodo inicial: ");
      int s = scanner.nextInt();

      System.out.println("Ingrese la implementacion: ");
      String sClassImple = scanner.next();

      // obtengo una instancia de la implementacion indicada por el usuario
      Mst mst = (Mst) Class.forName(sClassImple).newInstance();

      // proceso
      Grafo solu = mst.procesar(g, 0);

      for(Arista a: solu.getAristas())
      {
         System.out.println(a);
      }
   }
}
```

Al ejecutar el programa veremos los siguientes resultados:

```
Ingrese el nodo inicial: 0
Ingrese la implementacion:
libro.cap22.mst.imple.MstImpleKruskal
Desde: 0, Hasta: 1 Distancia: 6
Desde: 1, Hasta: 2 Distancia: 2
Desde: 1, Hasta: 4 Distancia: 4
Desde: 2, Hasta: 3 Distancia: 4
Desde: 3, Hasta: 5 Distancia: 2
Desde: 5, Hasta: 6 Distancia: 4
```

Para utilizar la implementación de Prim haremos:

```
Ingrese el nodo inicial: 0
Ingrese la implementacion:
libro.cap22.mst.imple.MstImplePrim
Desde: 0, Hasta: 1 Distancia: 6
Desde: 1, Hasta: 2 Distancia: 2
Desde: 1, Hasta: 4 Distancia: 4
Desde: 2, Hasta: 3 Distancia: 4
Desde: 3, Hasta: 5 Distancia: 2
Desde: 5, Hasta: 6 Distancia: 4
```

## 22.5 Resumen

En este capítulo analizamos la estructura `Grafo` como una estructura no lineal y recursiva. Observamos la importancia de "recordar" qué nodos del grafo ya hemos procesado para no incurrir en *loops* infinitos.

Los algoritmos sobre grafos son complejos y la mejor forma de abordarlos es encapsulando esta complejidad utilizando clases y objetos, tal como lo hicimos con la clase `Grafo`.

## 22.6 Contenido de la página Web de apoyo


El material marcado con asterisco (*) solo está disponible para docentes.

### 22.6.1 Mapa conceptual

### 22.6.2 Autoevaluaciones

### 22.6.3 Videotutoriales

22.6.3.1 Algoritmo de Dijkstra por greddy

22.6.3.2 Algoritmo de Dijkstra por dinámica

22.6.3.3 Algoritmo de Prim

22.6.3.4 Algoritmo de Kruskal

### 22.6.4 Presentaciones*