TEMA 5

Grafos.

5.1. DEFINICIÓN DE GRAFO

A menudo, cuando se observa la red de rutas aéreas de un país interesa observar cómo ir de una ciudad a otra por las rutas posibles. En consecuencia, se tiene dos conjuntos de objetos distintos: ciudades y rutas. La Figura 5.1 muestra una manera de representar la relación existente entre las ciudades y las rutas, así como la distancia entre las distintas ciudades.

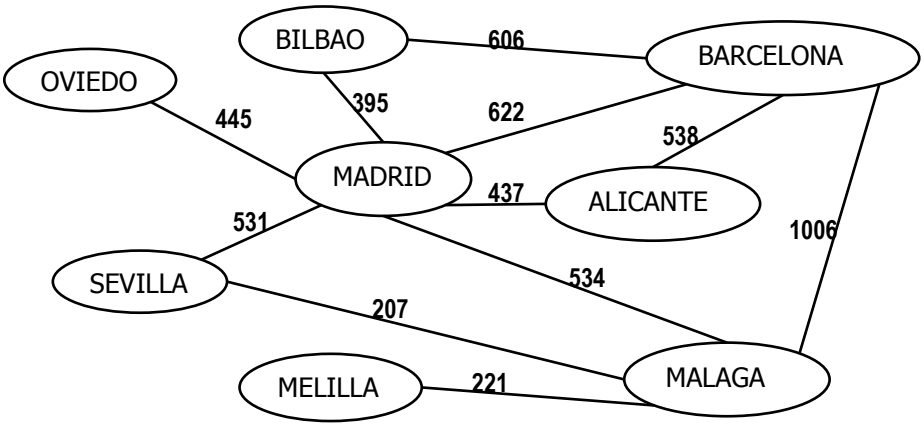


Figura 5.1. Representación de las conexiones entre ciudades.

En general, un grafo es una manera de representar relaciones que existen entre pares de objetos. Así, un grafo es un conjunto de objetos, llamados vértices[1], y relaciones entre objetos que establecen una relación entre pares de vértices, representadas por aristas.

En el ejemplo anterior, el grafo de la Figura 5.1 representa las conexiones aéreas entre ciudades. Los vértices representarían las ciudades. Las aristas representan las conexiones entre ciudades y, en este caso, se almacenan la distancia en kilómetros entre las ciudades que une.

Definición 1. Un grafo se define como un par G = (V, A), donde V es un conjunto finito no vacío de vértices y A es un conjunto de pares de vértices de V, es decir, las aristas.

Definición 2. Un grafo G se define como un par ordenado, G = (V, A), donde V es un conjunto finito y A es un conjunto que consta de dos elementos de V.

[1] La terminología de la teoría de grafos no es estándar. El concepto de vértice también se referencia como nodo. Asimismo, aristas (*edges* en inglés) y arcos denotan el mismo elemento. En algunos libros, sin embargo, se establece una diferencia entre aristas (unen vértices en un grafo no dirigido) y arcos (unen vértices en grafos dirigidos). En este capítulo, se dará preferencia a los términos vértice y arista.

5.2. TERMINOLOGÍA Y CONCEPTOS

5.2.1. Grafos dirigidos y no dirigidos

Dependiendo del tipo de relación entre los vértices del grafo, se definen distintos tipos de grafos. Así se distinguen aristas dirigidas y no dirigidas:

- **Arista dirigida**: es aquella que define un par ordenado de vértices (***u,v***), donde el primer vértice ***u*** es el origen de la arista y el segundo vértice ***v*** es el término (o vértice final). El par (***u, v***) ≠ (***v, u***).

- **Arista no dirigida:** es aquella que define un par no ordenado de vértices (***u, v***), donde (***u, v***) = (***v, u***).

De esta forma se distinguen entre **grafos dirigidos** y **grafos no dirigidos**.

- **Grafo dirigido:** Es aquel cuyas aristas son dirigidas. Los grafos dirigidos suelen representar relaciones asimétricas como por ejemplo: relaciones de herencia, los vuelos entre ciudades, etc.

- **Grafo no dirigido:** Es aquel cuyas aristas son no dirigidas. Representan relaciones simétricas como relaciones de hermandad y colaboración, conexiones de transportes, etc.

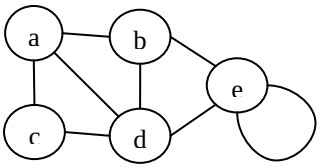
5.2.2. Incidencia, adyacencia y grado de un vértice

Sea un grafo G = (V, A), los vértices ***u*** y ***v*** pertenecientes a V; y una arista (***u,v***) perteneciente a A, se dice que:

- **Incidencia**: la arista (***u,v***) es incidente con los vértices ***u*** y con ***v***.
- **Adyacencia**: Dos vértices ***u*** y ***v*** son adyacentes si existe una arista cuyos vértices sean ***u*** y ***v***:
  - El vértice **u es adyacente a *v***
  - El vértice **v es adyacente desde *u***

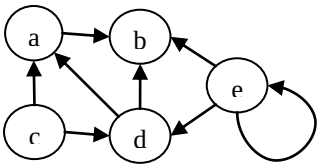
- **Grado**: El grado de un vértice ***u*** es el número de vértices adyacentes a ***u***. Para un grafo dirigido, el **grado de salida** de un vértice ***u*** es el número de vértices adyacentes desde ***u***, mientras que el grado de entrada de un vértice ***u*** es el número de vértices adyacentes a ***u.*** La Figura 5.2 muestra los grados de los vértices para un grafo no dirigido y un grafo dirigido.

Grafo no dirigido:



Grado (a) = 3
Grado (b) = 3
Grado (c) = 2
Grado (d) = 4
Grado (e) = 4

Grafo dirigido:



GradoE (a) = 2
GradoE (b) = 3
GradoE (c) = 0
GradoE (d) = 2
GradoE (e) = 1

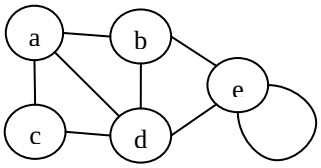
GradoS (a) = 1
GradoS (b) = 0
GradoS (c) = 2
GradoS (d) = 2
GradoS (e) = 3

Figura 5.2. Grado de los vértices en un grafo no dirigido y dirigido.

5.2.3. Grafos simples y multigrafos

Un grafo simple es aquel que no tiene aristas paralelas o múltiples que unan el mismo par de vértices. Un grafo que cuente con múltiples aristas entre dos vértices se denomina multigrafo. La Figura 5.3 muestra un ejemplo de grafo simple y de multigrafo, donde existen aristas paralelas incidentes a los vértices *a* y *d*, y *e* y *d*. En este caso, se dice que el grafo tiene multiplicidad 2 (máximo de aristas paralelas entre dos vértices).

GRAFO SIMPLE:



MULTIGRAFO:

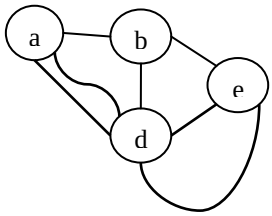


Figura 5.3. Grafo simple y grafo no simple

Si asumimos un grafo simple, se observan las siguientes propiedades:

- Si G es un grafo no dirigido con *m* vértices, entonces

$$\sum_{v \text{ en } G} \text{grado}(v) = 2m.$$

> Una arista (**u, v**) se cuenta dos veces en este sumatorio, uno como vértice final ***u*** y otro como vértice final ***v***. Entonces, la contribución total de las aristas a los grados de los vértices es dos veces el número de las aristas.

- Si G es un grafo dirigido con m vértices, entonces:

$$\sum_{v \text{ en } G} \text{grado } E(v) = \sum_{v \text{ en } G} \text{grado } S(v) = m$$

> En un grafo dirigido, una arista (***u,v***) contribuye una unidad al grado de salida de su origen ***u*** y una unidad al grado de entrada de su vértice final ***v***. Por tanto, la contribución total de las aristas al grado de salida de los vértices es igual al número de aristas, y similar para los grados de salida.

- Sea G un grafo simple con *n* vértices y *m* aristas, entonces:
  - Si G es **no dirigido**: $m \leq n(n-1)/2$.
  - Si G es **dirigido**: $m \leq n(n-1)$.

5.2.4. Camino, bucle y ciclo

Un **camino** es una secuencia que alterna vértices y aristas que comienza por un vértice y termina en vértice tal que cada arista es incidente a su vértice predecesor y sucesor. Es decir, un camino es un sucesión de vértices de $v_i \in V$: $<v_0, v_1, v_2, \ldots v_k>$ que cumple que:

$$(v_i,v_{i+1}) \in A \ \forall \ i \in \{0 \ldots k-1\}.$$

Se dice que este camino tiene **longitud *k***. Es decir, el número de aristas de un camino o ciclo es la longitud del camino.

Un **camino** es **simple** si cada vértice en el camino es distinto, excepto posiblemente por el primero y el último vértice. Un camino simple cumple la siguiente restricción:

$$v_i \neq v_j \ \forall \ i \in \{0...k\}, j \in \{1...k-1\}, i \neq j$$

Para todo vértice x, existe el camino simple <x>, que sería el camino de longitud 0.

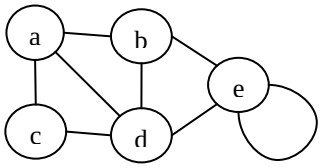
Un **bucle** es un camino de longitud 1 que comienza y termina en el mismo vértice: $<x_i, x_i>$.

Un **ciclo** es un camino simple $<v_0, \ldots v_k>$ que cumple las siguientes restricciones:

- $v_0 = v_k$
- Si es no dirigido, $k = 1$ (es un bucle) o $k \geq 3$.

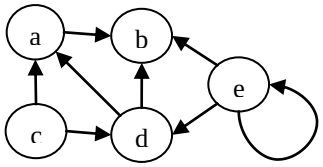
La Figura 5.4 ilustra estos conceptos para un grafo no dirigido y un grafo dirigido.

GRAFO NO DIRIGIDO:



<a,b,e,d,c>: camino simple de longitud 4.
<a,c,d,a,b,e>: camino de longitud 5.
<a,e>: no es un camino.
<e,e>: camino, bucle y ciclo

GRAFO DIRIGIDO:



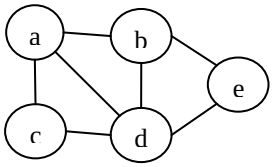
<a,b>: camino simple de longitud 1.
<e,d,a,b>: camino de longitud 3.
<a,c,d>: no es un camino.
<e,e>: camino, bucle y ciclo

Figura 5.4. Caminos, bucles y ciclos en un grafo dirigido y no dirigido.

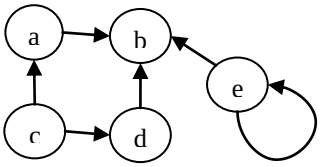
5.2.5. Grafos conexos

Sea G = (V, A) un **grafo no dirigido**, se le denomina **conexo** si existe un camino entre dos vértices cualesquiera de G. Para un **grafo dirigido** G, su grafo asociado no dirigido es aquel que se obtiene ignorando la dirección de las aristas. G se considera conexo si su grafo asociado es conexo. La Figura 5.5 muestra ejemplos de grafos conexos y no conexos.

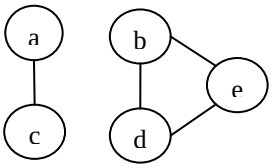
GRAFO 1: NO DIRIGIDO CONEXO



GRAFO 2: DIRIGIDO CONEXO



GRAFO 3: NO DIRIGIDO NO CONEXO



GRAFO 4: DIRIGIDO NO CONEXO

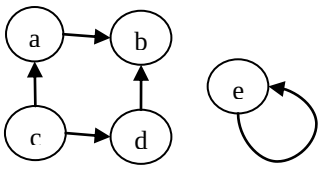


Figura 5.5. Ejemplos de grafos conexos y no conexos

5.2.6. Grafos valorados y grafos etiquetados[2]

Un **grafo valorado** (o ponderado) es una terna <V, A, f> donde <V, A> es un grafo y f es una función cualquiera, denominada función de coste, que asocia un valor o peso a cada arista en el grafo. El **peso de un camino** en un grafo con pesos es la suma de los pesos de todas las aristas atravesadas. En un **grafo etiquetado**, la función f tiene como imagen un conjunto de etiquetas no numéricas.

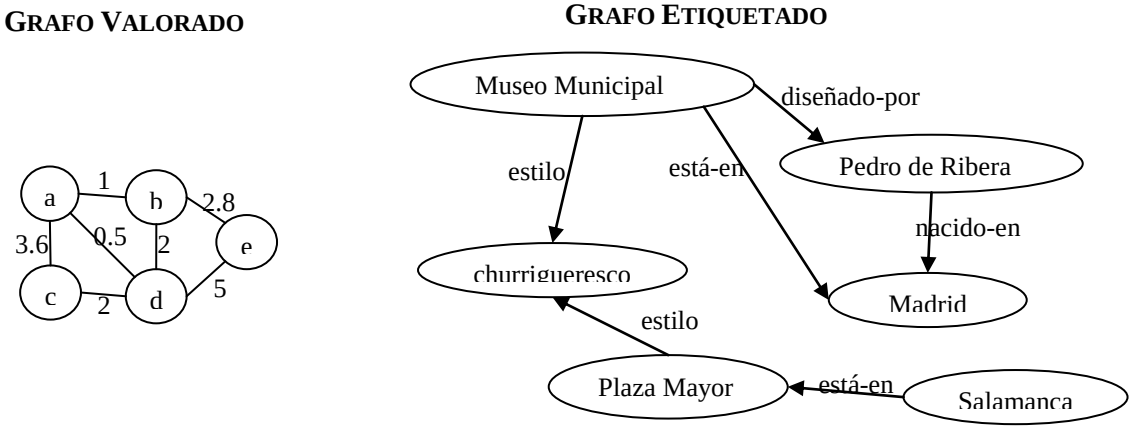


Figura 5.6. Grafo valorado y grafo etiquetado

[2] En este curso no se van a tratar los grafos valorados ni etiquetados.

5.3. IMPLEMENTACIONES DE GRAFOS

Los dos tipos de implementación más frecuentes (independientemente del lenguaje de programación) para la representación de grafos son las **matrices de adyacencias** y las **listas de adyacencias**. En este tema, se detallarán ambas implementaciones en el lenguaje Java.

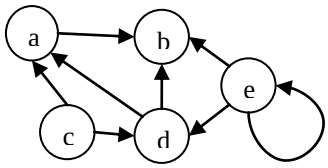
5.3.1. Interfaz del TAD Grafo.

Los métodos de objeto que podremos utilizar sobre las variables de la clase Grafo (no etiquetado ni ponderado) aparecen en la siguiente interfaz:

```
import java.io.IOException;
public interface Grafo {
   public void insertaVertice(int n);
   /** Inserta un vértice en el grafo siempre que no se supere el número máximo
de
       nodos permitidos **/
   public void eliminarVertice(int v)
   /** Elimina un vértice del grafo **/
   public void insertaArista(int i, int j);
   /** Inserta una arista entre los vértices i y j **/
   public void eliminarArista(int i, int j);
   /** Elimina la arista entre los vértices i y j **/
   public boolean esVacio(Grafo g);
   /** Devuelve true si el grafo no contiene ningún vértice **/
   public boolean existeArista(int i, int j);
   /** Devuelve true si existe una arista que una los vértices i y j. **/
   public int gradoIn(int i)      ;
   /** Devuelve el grado de entrada del vértice i **/
   public int gradoOut(int i);
   /** Devuelve el grado de salida del vértice i **/
   public int incidencia (int i)
   /** Devuelve la incidencia del vértice i **/
   public int tamano();
   /** Devuelve el tamaño (número de aristas) del grafo **/
   public boolean esDirigido (Grafo g) ;
   /** Devuelve true si el grafo g es dirigido **/
   public void ponerMaxNodos(int n);
   /** Asigna el número máximo de nodos o vértices permitidos en el grafo**/
   public void ponerDirigido(boolean d);
   /** Determina si es un grafo dirigido o no dirigido **/
}
```

5.3.2. Matriz de adyacencias

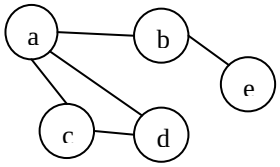
Una matriz de adyacencias A (implementada como una matriz bidimensional) determina las adyacencias entre pares de vértices de un grafo. En una matriz de adyacencias, los vértices se conciben como enteros en el conjunto {0,1,…,n-1} y las aristas como pares de tales enteros. Esto permite almacenar referencias a las aristas en las celdas de la matriz bidimensional de nxn. Cada fila y cada columna representan un vértice del grafo y cada posición representa una arista (o la ausencia de esta) cuyo vértice origen se encuentra en la fila y vértice final se encuentra en la columna. La Figura 5.7 muestra la representación gráfica de un grafo y su matriz de adyacencias.



	a	b	c	d	e
a	0	1	0	0	0
b	0	0	0	0	0
c	1	0	0	1	0
d	1	1	0	0	0
e	0	1	0	1	1

Figura 5.7. Grafo y matriz de adyacencias correspondiente

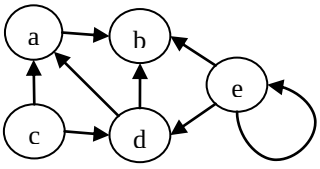
Nótese que la matriz de adyacencias para un grafo no dirigido es una matriz simétrica como se puede apreciar en la Figura 5.8:



	a	b	c	d	e
a	0	1	1	1	0
b	1	0	0	0	1
c	1	0	0	1	0
d	1	0	1	0	0
e	0	1	0	0	0

Figura 5.8. Grafo y matriz de adyacencias para un grafo no dirigido

En una tabla de adyacencias, los vértices se representan mediante índices. Así, sea un grafo con los vértices {a,b,c,d,e}, estos serán representados mediante sus índices {0,1,2,3,4} tal y como se muestra en la Figura 5.9.



VÉRTICES:	**a**	**b**	**c**	**d**	**e**
ÍNDICES:	**0**	**1**	**2**	**3**	**4**

Figura 5.9. Relación entre vértices e índices

De esta manera, podemos definir una matriz A bidimensional de n x n donde la celda [***i, j***] guarda información referente a la arista (***v, w***), en caso de que exista, donde ***v*** es el vértice con índice i y ***w*** es el vértice con índice j. Existen varias posibilidades para representar la arista (***v, w***) en su correspondiente celda A [i, j]:

- Si no es etiquetado: un valor booleano que toma el valor 1 si existe la arista y 0 en caso contrario.
- Si es valorado: un real con el valor f (i, j) si existe la arista. En caso contrario, toma el valor ∞. Se obtendría la matriz de costes del grafo.
- Si es etiquetado: una etiqueta con el valor f (i, j) si existe la arista. En caso contrario, toma el valor "etiqueta imposible". Se obtendría la matriz de etiquetas del grafo.

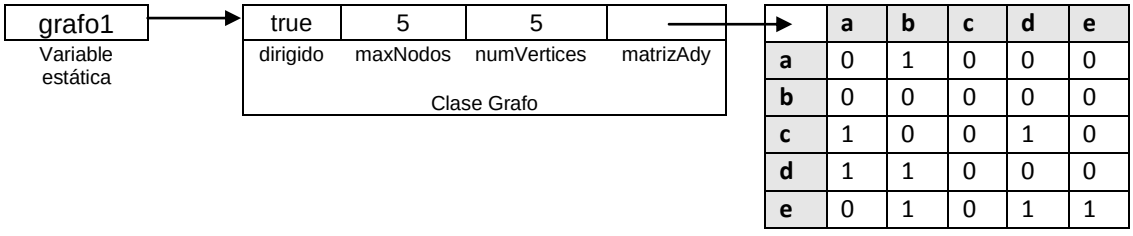
A continuación, se muestra una posible implementación de un grafo en una matriz de adyacencias en Java. Se asume que se trata de un grafo simple no etiquetado (tanto dirigido como no dirigido).

5.3.2.1. Representación de la clase Grafo

La siguiente sintaxis muestra la implementación de la clase grafo con un matriz de adyacencias:

```
public class GrafoMA implements Grafo {
   private boolean dirigido;              // Indica si es dirigido o no.
   private int maxNodos;                  // Tamaño máximo de la tabla.
   private int numVertices;               // Número de vértices del grafo.
   private boolean matrizAdy [ ] [ ];     // Matriz de adyacencias del grafo.
}
```

La representación gráfica de una variable de la clase Grafo llamada "grafo1" que represente el grafo de la Figura 5.7 sería:



	a	b	c	d	e
a	0	1	0	0	0
b	0	0	0	0	0
c	1	0	0	1	0
d	1	1	0	0	0
e	0	1	0	1	1

Figura 5.10. Representación gráfica de la variable "grafo1"

5.3.2.2. Constructores de la clase Grafo[3].

Se utiliza para crear un grafo vacío, con un tamaño máximo y número de vértices igual a 0. Se pasa como argumento un booleano que indica si se trata de un grafo dirigido o no dirigido.

```
public GrafoMA (boolean d) {
   maxNodos = numVertices = 0;
   dirigido = d;
}
```

También se puede incluir un constructor al que se le pasen como argumentos el número de vértices del grafo y el valor para el campo *dirigido*:

```
public GrafoMA (int n, boolean d) {
   dirigido = d;
   maxNodos = n;
   numVertices = 0;
   matrizAdy = new boolean[n][n];
}
```

5.3.2.3. Algoritmos básicos de modificación: insertar y eliminar aristas.

La inserción de una arista (***i, j***) en la matriz supone asignar a la celda correspondiente el valor `true`. Si se trata de un grafo dirigido, debe tenerse en cuenta que el vértice origen ***i*** corresponde a fila, mientras que el vértice destino ***j*** corresponde la columna. En caso de que se trate de un grafo no dirigido, debe recordarse que la arista (***i, j***) es igual a la arista (***j, i***) para que la matriz mantenga la propiedad de la simetría.

```
public void insertaArista (int i, int j) {
   matrizAdy [i] [j] = true;
   if (!dirigido)
      matrizAdy [j] [i] = matrizAdy [i] [j];
}
```

La eliminación de la arista (***i, j***) es sencilla, consiste en asignar el valor `false` a la celda correspondiente de la matriz. En caso de que se trate de un grafo no dirigido, habrá de modificarse igualmente el valor correspondiente a la arista (***j, i***):

```
public void eliminarArista (int i, int j) {
   matrizAdy [i] [j] = false;
   if (!dirigido)
      matrizAdy [j] [i] = false;
}
```

[3] Antes de ejecutar esta operación deberá asegurarse de que no existe previamente ningún grafo con el mismo nombre pues su ejecución haría que se perdiese el contenido del grafo preexistente.

5.3.2.4. Algoritmos básicos de modificación: insertar vértices

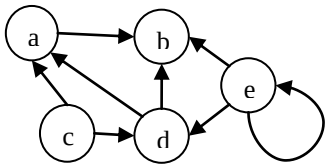
La modificación de los vértices del grafo supone un grado de complejidad mayor que el tratamiento de las aristas. Para la inserción de un vértice, el código aquí mostrado simplifica el método de manera que no deja insertar vértices si se supera el límite de vértices del grafo (valor del campo `maxNodos`). En caso de que no se supere el número máximo de vértices, simplemente se asigna el valor `false` a las celdas correspondientes y se actualiza el campo `numVertices`:

```
public void insertaVertice (int n) {
   if ( n > maxNodos - numVertices )
      System.out.println ("Error, se supera el número de nodos máximo");
   else {
      for (int i=0; i < numVertices + n; i++) {
         for (int j = numVertices; j < numVertices + n; j++)
            matrizAdy [i] [j] = matrizAdy [j] [i] = false;
      }
   numVertices = numVertices + n;
   }
}
```

La eliminación de un vértice supondría cambiar los índices de los vértices. De manera que no se va a reproducir esta operación. Se deja al alumno como ejercicio la posible implementación de eliminación de un vértice.

5.3.2.5. Otros métodos

Puede ser interesante conocer el grado de entrada y de salida de un vértice. Retómese el grafo y su matriz de la Figura 5.7. En un grafo dirigido, el grado de entrada de un vértice, por ejemplo ***d,*** viene dado por la suma de los valores de la columna d; mientras que su grado de salida vendría dado por la suma de los valores de la fila d:



	a	b	c	d	e
a	0	1	0	0	0
b	0	0	0	0	0
c	1	0	0	1	0
d	1	1	0	0	0
e	0	1	0	1	1

Grado de salida (b) = 0

Grado de entrada (b) = 3

Figura 5.11. Grados de entrada y salida en una matriz de adyacencias.

```
public int gradoIn(int j) {
   int gIn = 0;
   for (int i = 0; i < numVertices; i++)       //recorrido por filas
      if (matrizAdy [i] [j])
         gIn++;            //manteniendo la posición de la columna en [j]
   return gIn;
}
```

```
public int gradoOut(int i) {
   int gOut = 0;
   for (int j= 0; j < numVertices; j++)
      if (matrizAdy [i][j])
         gOut++;          // manteniendo la posición de la fila en [i]
   return gOut;
}
```

En un grafo no dirigido, la incidencia de un vértice vendría dada por su grado de entrada:

```
public int incidencia (int i) {
   if (!dirigido)
      return gradoIn (i);
   else return gradoIn (i) + gradoOut (i);
}
```

El tamaño de un grafo viene dado por el número de aristas. Debe notarse que las aristas de un grafo no dirigido se cuentan dos veces.

```
public int tamano() {
   int tm = 0;
   for (int i = 0; i < numVertices; i++)
      for (int j=0; j < numVertices; j++)
         if (matrizAdy [i] [j])
            tm++;
   if (dirigido)
      return tm;
   else return tm/2;
}
```

Para comprobar si un grafo es dirigido o no, basta con comprobar si se trata de una matriz simétrica, donde la posición [i, j] = [j, i]:

```
public boolean esDirigido (Grafo g) {
   boolean dir = true;
   for (int i = 0; i < numVertices; i++)
      for (int j = 0; j < numVertices; j++)
         if (matrizAdy [i] [j] != matrizAdy [j] [i])
            dir = false;
   return dir;
}
```

Por último, el siguiente método imprime una matriz de adyacencias de un grafo:

```
public void imprimirTabla () {
   System.out.println ("La matriz contiene " + numVertices + " vértices: \n");
   for (int i = 0; i < numVertices; i++) {
      for (int j = 0; j < numVertices; j++) {
         if (matrizAdy [i] [j])
            System.out.print ("1 ");
         else System.out.print ("0 ");
      }
   }
}
```

5.3.3. Lista de adyacencias

Otra implementación frecuente para la estructura grafo es la lista de adyacencias. En una lista de adyacencias, a cada vértice ***i*** se le asocia una lista enlazada con todos los vértices ***j*** adyacentes a ***i***. De esta forma sólo se reserva memoria para las aristas adyacentes a ***i*** y no para todas las posibles aristas que pudieran tener como origen ***i*** (como ocurre en una matriz de adyacencias).

El grafo, por tanto, se representa por medio de un vector de *n* componentes, siendo n el número de vértices del grafo, donde cada componente constituye la lista de adyacencias correspondiente a cada uno de los vértices del grafo. Cada nodo de la lista consta de un campo indicando el vértice adyacente. En caso de que el grafo sea etiquetado o valorado, habrá que añadir un segundo campo para mostrar el valor de la etiqueta o el peso de la arista.

La Figura 5.12 muestra un grafo y su representación en una lista de adyacencias.

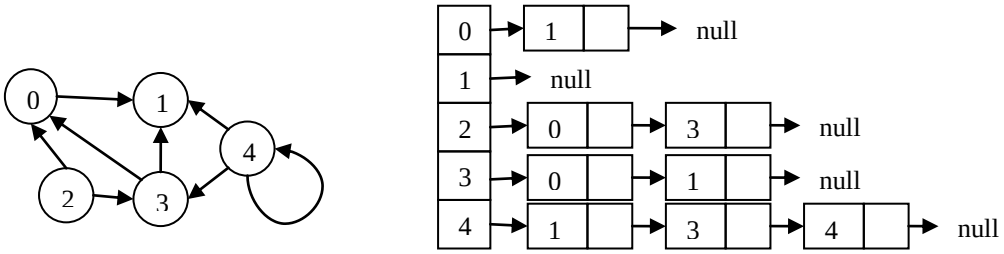


Figura 5.12. Grafo dirigido y lista de adyacencias asociada

5.3.3.1. Representación de la clase Grafo como lista de adyacencias

Para la implementación de la clase Grafo como una lista de adyacencias, se hará uso de la clase Lista Calificada Ordenada (tal y como se vio en el tema 3.6.), así como la clase NodoLista con las siguientes variables miembro y constructores:

```
public class NodoLista {
   public int clave;
   public NodoLista sig;
   public NodoLista (int dato, NodoLista siguiente) {
      clave = dato;
      sig = siguiente;
   }
}
```

```
public class Lista {
   public NodoLista inicio;
   public Lista () {
      inicio = null;
}
```

La implementación de los distintos métodos de la clase Grafo aquí propuesta utiliza los siguientes métodos de Lista:

```
   void insertar (int x) ;
   void eliminar (int x);
   boolean busqueda (int x);
```

La siguiente sintaxis muestra la implementación de la clase grafo con una lista de adyacencias:

```
public class GrafoLA implements Grafo {
   boolean dirigido;          // Indica si es dirigido o no.
   int     maxNodos;          // Tamaño máximo de la tabla.
   int     numVertices;       // Número de vértices del grafo.
   Lista   listaAdy [];       // Vector de listas de adyacencias del grafo.
}
```

Figura 5.13. Clase Grafo

5.3.3.2. Constructores de Grafo

El constructor `Grafo(boolean d)` se utiliza para crear un grafo vacío, con un tamaño máximo y número de vértices igual a 0. El argumento booleano indica si se trata de un grafo dirigido o no dirigido.

```
public GrafoLA (boolean d) {
   maxNodos = numVertices = 0;
   dirigido = d;
}
```

También se puede incluir un constructor al que se le pasen como argumentos el número de vértices del grafo y el valor para el campo *dirigido*. Se crea un vector de *n* listas inicializadas.

```
public GrafoLA (int n, boolean d) {
   dirigido = d;
   maxNodos = n;
   numVertices = 0;
   listaAdy = new Lista[n];
}
```

5.3.3.3. Algoritmos básicos de modificación: insertar aristas.

La inserción de una arista (***i, j***) en la lista de adyacencias supone insertar un nodo con clave ***j*** en la lista con índice ***i***. En caso de que se trate de un grafo no dirigido, debe recordarse que la arista (***i, j***) es igual a la arista (***j, i***) de forma que habrá que insertar en la lista con índice ***j*** el nodo con clave ***i***.

```
public void insertaArista (int i, int j) {
   if (i >= numVertices)
      System.out.println ("Error, no existe el vértice en el grafo");
   else {
      listaAdy[i].insertar (j);
      if (!dirigido)
         listaAdy[j].insertar (i);
   }
}
```

La eliminación de la arista (***i, j***) consiste en eliminar el nodo con clave ***j*** de la lista con índice ***i***. Si el grafo es no dirigido, habrá que eliminar el nodo con clave ***i*** de la lista con índice ***j***:

```
public void eliminaArista (int i, int j) {
   if (j >= numVertices)
      System.out.println("Error, no existe el vértice en el grafo");
   else {
      listaAdy[i].eliminar (j);
      if (!dirigido)
         listaAdy[j].eliminar (i);
   }
}
```

5.3.3.4. Algoritmos básicos de modificación: insertar vértices

Al igual que en la matriz de adyacencias, no se permite insertar vértices si se supera el límite de vértices del grafo (valor del campo `maxNodos`). El método aquí propuesto tiene como argumento un entero que indica el número de vértices que se desea añadir al grafo. En caso de que no se supere el número máximo de nodos del grafo, se inicializan las *n* listas correspondientes a los vértices que se añaden al grafo.

```
public void insertaVertice (int n) {
   if (n > maxNodos - numVertices)
      System.out.println ("Error, se supera el número de nodos máximo del grafo");
   else
      for (int i = numVertices; i < numVertices + n; i++)
         listaAdy[i]= new Lista();
   numVertices += n;
}
```

Dado que la eliminación de un vértice supondría cambiar los índices de los vértices, no se va a reproducir esta operación.

5.3.3.5. Otros métodos

Para conocer el **grado de entrada** de un vértice ***v*** en un grado dirigido, se deben contar las veces que aparece el nodo con clave ***v*** en las distintas listas de adyacencia. Para ello, se utiliza el método de objeto *busqueda* de la clase Lista.

```
public int gradoIn (int v) {
   int gIn = 0;
   for (int i = 0; i < numVertices; i++)
      if (i != v)
         if (listaAdy[i].busqueda(v))
            gIn++;
   return gIn;
}
```

El **grado de salida** de un vértice ***v*** viene dado por el número de elementos de su lista de adyacencia (lista con índice ***v***):

```
public int gradoOut (int i) { //contar los elementos de la lista
   int gOut = 0;
   NodoLista aux = listaAdy[i].inicio;
   while (aux != null){
      gOut++;
      aux = aux.sig;
   }
   return gOut;
}
```

En un grafo no dirigido, la **incidencia** de un vértice vendría dada por su grado de entrada. En cambio, si se trata de un grafo dirigido, la incidencia de un vértice ***i*** es la suma de su grado de entrada y salida:

```
   public int incidencia (int i) {
      if (!dirigido)
         return gradoIn (i);
      else return gradoIn (i) + gradoOut (i);
   }
```

El número de aristas define el **tamaño** de un grafo. En una implementación basada en listas de adyacencias, el tamaño del grafo viene dado por el número de nodos total de las listas de adyacencias. Para ello, se hace uso del método auxiliar *numElems* que recibe como argumento una lista y devuelve el número de nodos de la lista de entrada. En el caso de que el grafo sea no dirigido, habría que dividir por dos la suma del número de nodos de las listas de adyacencias puesto que las aristas de un grafo no dirigido se cuentan dos veces:

```
public int tamanno () {
   int tm = 0;
   for (int i=0; i<numVertices; i++)
      tm += numElementos (listaAdy[i]);
   if (!dirigido)
      tm = tm / 2;
   return tm;
}
static int numElementos (Lista lista)  {
   NodoLista aux = lista.obtenerInicio();
   int resul = 0;
   while (aux != null) {
      resul += 1;
      aux = aux.obtenerSig();
   }
   return resul;
}
```

El siguiente método booleano devuelve un valor verdadero en caso de que el grafo sea no dirigido. En un grafo dirigido simple el par (i, j) es necesariamente distinto del par (j, i), mientras que en un grafo no dirigido se cumple que el par (i, j) = (j, i). El método propuesto comprueba para, cada par de vértices ***i, j***, que el vértice ***j*** se encuentre en la lista de adyacencias del vértice ***i***; e igualmente que el vértice ***i*** se encuentre en la lista de adyacencias del vértice ***j***.

```
public boolean esNoDirigido () {
   boolean dir = true;
   for (int i=0; i<numVertices; i++)
      for (int j=0; j<numVertices; j++)
         if (listaAdy[i].busqueda (j) != listaAdy[j].busqueda (i))
            dir = false;
   return dir;
}
```

Por último, el siguiente método imprime la lista de adyacencias para un grafo, dicho método utiliza el método de clase *escribir (Lista)*:

```
public void imprimirGrafo () {
   System.out.println("Tamaño máximo del grafo: " + maxNodos + "\n");
   System.out.println("El grafo contiene " + numVertices + " vértices: \n");
   for (int i = 0; i < numVertices; i++) {
      System.out.print ("vértice " + i + ": ");
      escribir (listaAdy [i]);
   }
}
static void escribir (Lista lista) {
   NodoLista aux;
   aux = lista.inicio;
   while (aux != null) {
      System.out.print (aux.clave +", ");
      aux = aux.sig;
   }
   System.out.println ("FIN");
}
```

5.3.4. Consideraciones sobre la implementación del grafo como matriz o como lista de adyacencias

La principal ventaja de la matriz de adyacencias es el rápido acceso a la información de aristas. Es una estructura recomendable si se van a realizar consultas frecuentes del estilo: ¿existe la arista (u, v) en el grafo? O, en caso de ser un grafo valorado, ¿cuál es el coste de la arista (u, v)? También es una estructura recomendable para grafos con pocos vértices o bien grafos con muchos vértices y muchas aristas.

La principal desventaja que presenta la matriz es que requiere un almacenamiento proporcional al cuadrado del número de vértices del grafo, aunque el grafo contenga muchas menos aristas de los posibles.

Respecto a las listas de adyacencias, estas ahorran memoria en la representación de grafos con muchos vértices y pocas aristas. Asimismo, si el procesamiento se centra en el tratamiento de los vértices adyacentes a uno dado, la lista de adyacencias es preferible a la matriz.

5.4. RECORRIDOS DE GRAFOS

El concepto de recorrido del grafo, como se ha visto en el tema 4 de árboles, consiste en visitar todos los vértices del grafo sucesivamente de manera sistemática de manera que cada vértice se visite una única vez.

En el recorrido de un grafo, existen dos tipos de vértices:

- Vértices visitados: vértices ya visitados en el recorridos
- Vértices frontera: vértices que aun no ha sido visitados pero están conectados con algún vértice visitado (están pendientes de visitar).

Al igual que un árbol (de hecho, un árbol es un tipo de grafo orientado sin ciclos), un grafo puede ser recorrido en profundidad o en amplitud. Existen dos diferencias fundamentales a la hora de recorrer un grafo respecto de un árbol:

a) Puesto que un árbol es un grafo orientado sin circuitos, al avanzar en el recorrido no cabe la posibilidad de que se vuelva a visitar un vértice ya visitado. En el recorrido de un grafo sí cabe la posibilidad de al avanzar visitar un vértice ya visitado. Se deberá implementar algún mecanismo que evite esta situación.
b) Partiendo de la raíz de un árbol se pueden visitar todos los vértices, mientras que en un grafo se puede dar la posibilidad de que no se alcancen todos los vértices desde un vértice. Habría que comenzar el recorrido en otro vértice para poder alcanzar todos los vértices.

5.4.1. Recorrido en profundidad

En el recorrido en profundidad, se da preferencia a visitar a los vértices conectados con el último visitado. Dado un grafo G, en el que inicialmente ningún vértice ha sido visitado, el recorrido en profundidad selecciona un vértice ***v*** de G como vértice inicial, que se marca como visitado. Entonces, se busca un vértice no visitado adyacente a ***v***, ***w***, que se marca como visitado y se selecciona como vértice inicial para reiniciar el recorrido. Una vez que se han visitado todos los vértices alcanzables de ***w*** se vuelve a ***v*** y se selecciona un nuevo vértice no visitado. Cuando se han visitado todos los vértices alcanzables desde ***v***, se da por terminado el recorrido de ***v***. Si quedan vértices por visitar, se selecciona uno de ellos como nuevo vértice de partida y se reproduce todo el proceso. Es decir:

```
Sea un grafo G y un vértice v que pertenece a G,
     marcar v como visitado
     mientras queden vértices sin visitar hacer
       seleccionar un vértice no visitado, v,
       avanzarEnProfundidad a partir de v
     fin mientras
```

Donde `avanzarEnProfundidad` consiste en:

```
     marcar el vértice v como visitado
     seleccionar un vértice adyacente a v no visitado, w
     avanzar en profundidad a partir de w.
```

Por ejemplo, sea el siguiente grafo G:

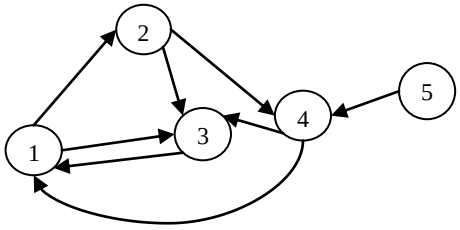


Figura 5.14. Ejemplo de grafo

Se toma como vértice inicial 1,

v1 ← visitado *(el conjunto de vértices adyacentes a v1 es 2,3)*

Se toma vértice 2:

v2 ← visitado *(el conjunto de vértices adyacentes a v2 es 3,4)*

Se toma vértice 3

v3 ← visitado *(el conjunto de vértices adyacentes a v3 es 1)*

//v1 ya está visitado por lo que se termina el recorrido en profundidad a partir del vértice 3.

Se toma el vértice 4

v4 ← visitado *(el conjunto de vértices adyacentes a v4 es 3)*

//v3 ya está visitado por lo que se termina el recorrido en profundidad a partir del vértice 4.

//v3 ya está visitado por lo que termina el recorrido en profundidad a partir del vértice 3.

No es posible alcanzar más vértices desde v1, de manera que hay que seleccionar un nuevo vértice desde el que recomenzar la exploración en profundidad, se elige el vértice 5:

v5 ← visitado *(el conjunto de vértices adyacentes a v5 es 4)*

//v4 ya está visitado de manera que se termina el recorrido en profundidad a partir del vértice 5

--- FIN---

El algoritmo de recorrido en profundidad cuenta con una parte recursiva que recorre parcialmente un subgrafo a partir de un vértice de inicio y una parte no recursiva que se encarga de relanzar el proceso en cada vértice no visitado. Se utiliza además un vector de valores lógicos para marcar los vértices visitados. Se utilizan los índices de los vértices para iniciar y marcar el proceso de recorrido.

El código en Java para un recorrido en profundidad es el siguiente:

```
//procedimiento recursivo
public void recorrerProfundidad (Grafo g, int v, boolean [ ] visitados) {
   //se marca el vértice v como visitado
   visitados [v] = true;
   //el tratamiento del vértice consiste únicamente en imprimirlo en pantalla
   System.out.println (v);
   //se examinan los vértices adyacentes a v para continuar el recorrido
   for (int i = 0; i < g.numVertices; i++) {
      if ((v != i) && (!visitados [i]) && (g.existeArista (v, i)) )
         recorrerProfundidad (g, i, visitados);
   }
}
//procedimiento no recursivo
public void profundidad (Grafo g) {
   boolean visitados [ ] = new boolean [g.numVertices];
```

```
    for (int i = 0; i < g.numVertices; i++)  //inicializar vector con campos false
       visitados [i] = false;
    for (int i = 0; i < g.numVertices; i++) {  //Relanza el recorrido en cada
       if (!visitados [i])                     //vértice visitado
          recorrerProfundidad (g, i, visitados);
    }
}
```

5.4.2. En amplitud

En un recorrido en amplitud, se elige un vértice no visitado ***v***, como punto de partida y se pasa a visitar cada uno de sus vértices adyacentes, para continuar posteriormente visitando los adyacentes a estos últimos y así sucesivamente hasta que no se puedan alcanzar más vértices. Si queda algún vértice sin visitar, se selecciona y se vuelve a relanzar el proceso.

Para realizar un recorrido en amplitud de un grafo es necesario utilizar una estructura de datos cola (cf. el recorrido en amplitud de la clase árbol). En la cola se van almacenando los vértices a medida que se llega a ellos. Los vértices se marcan en la cola como visitados y son tratados cuando se extraen de la cola al tiempo que se introducen en la cola los adyacentes al vértice tratado.

El código en Java para el recorrido en amplitud es el siguiente:

```
public static void amplitud (Grafo g) {
   Cola cola = new Cola ();
   boolean visitados [ ] = new boolean [g.obtenerNumVertices()];
   int v; //vértice actual
   //Se inicializa el vector visitados [] a false
   for (int i = 0; i < g.obtenerNumVertices (); i++)
      visitados [i] = false;
   //El recorrido en amplitud se inicia en cada vértice no visitado
   for (int i = 0; i < g.obtenerNumVertices (); i++) {
      //se pone en la cola el vértide de partida y se marca como visitado
      if (!visitados [i]){
         cola.encolar (i);
         visitados [i] = true;
         while (!cola.estaVacia ()) {
            v = cola.desencolar ();          //desencolar y tratar el vértice
            System.out.println (v);
            //y encolo los nodos adyacentes a v.
            for (int j = 0; j < g.obtenerNumVertices (); j++){
               if ((v !=j) && (g.existeArista (v, j) && (!visitados [j])) {
                  cola.encolar ( j );
                  visitados [j] = true;
               }
            }
         }
      }
   }
}
```

TEMA 5 173

5.1. definición de grafo 173

5.2. terminología y conceptos 175

5.2.1. Grafos dirigidos y no dirigidos 175

5.2.2. Incidencia, adyacencia y grado de un vértice 175

5.2.3. Grafos simples y multigrafos 176

5.2.4. Camino, bucle y ciclo 177

5.2.5. Grafos conexos 178

5.2.6. Grafos valorados y grafos etiquetados 179

5.3. Implementaciones de Grafos 180

5.3.1. Interfaz del TAD Grafo. 180

5.3.2. Matriz de adyacencias 181

5.3.3. Lista de adyacencias 186

5.3.4. Consideraciones sobre la implementación del grafo como matriz o como lista de adyacencias 191

5.4. Recorridos de grafos 192

5.4.1. Recorrido en profundidad 192

5.4.2. En amplitud 194