CAPITULO **15**

Grafos, representación y operaciones

Objetivos

Con el estudio de este capítulo, usted podrá:

- Distinguir entre relaciones jerárquicas y otras relaciones.
- Definir un grafo e identificar sus componentes.
- Conocer estructuras de datos para representar un grafo.
- Conocer las operaciones básicas que se aplican sobre grafos.
- Encontrar los caminos que puede haber entre dos nodos de un grafo.
- Realizar en Java la representación de los grafos y las operaciones básicas.

Contenido

15.1. Conceptos y definiciones.
15.2. Representación de los grafos.
15.3. Listas de adyacencia.
15.4. Recorrido de un grafo.
15.5. Conexiones en un grafo.
15.6. Matriz de caminos. Cierre transitivo.
15.7. Puntos de articulación de un grafo.
RESUMEN
EJERCICIOS
PROBLEMA

Conceptos clave

- ♦ Camino.
- ♦ Conexión y componente conexa.
- ♦ Factor de peso.
- ♦ Lista de adyacencia.
- ♦ Matriz de adyacencia.
- ♦ Relación jerárquica.
- ♦ Vértice y arco.

http://descargavariable.blogspot.com/

INTRODUCCIÓN

Este capítulo introduce al lector a conceptos matemáticos importantes denominados *grafos* que tienen aplicaciones en campos tan diversos como sociología, química, geografía, ingeniería eléctrica e industrial, etc. Los grafos se estudian como estructuras de datos o tipos abstractos de datos. Este capítulo estudia las definiciones relativas a los grafos y la representación de los grafos en la memoria del ordenador. El capítulo investiga dos formas tradicionales de implementación de grafos, matriz de adyacencia y listas de adyacencia. También se estudian operaciones importantes y algoritmos de grafos que son significativos en informática.

15.1. CONCEPTOS Y DEFINICIONES

Un grafo G agrupa *entes* físicos o conceptuales y las relaciones entre ellos. Un grafo está formado por un conjunto de vértices o nodos `V`, que representan a los *entes,* y un conjunto de arcos `A`, que representan las relaciones entre vértices. Se representa con el par `G = (V, A)`. La Figura 15.1 muestra un grafo formado por los vértices `V = {1,4,5,7,9}` y el conjunto de arcos `A = {(1,4), (4,1), (5,1), (1, 5), (7,9), (9,7), (7,5), (5,7), (4,9), (9,4)}`.

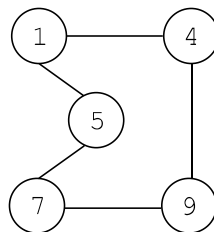

Figura 15.1 Grafo no dirigido

Un arco o arista representa una *relación* entre dos nodos. Esta relación, al estar formada por dos nodos, se representa por `(u, v)` siendo `u, v` el par de nodos. El grafo es *no dirigido* si los arcos están formados por pares de nodos no ordenados, no apuntados; se representa con un segmento uniendo los nodos, `u – v`. El grafo de la Figura 15.1 es no dirigido.

Un grafo es dirigido, también denominado *digrafo,* si los pares de nodos que forman los arcos son ordenados; se representan con una flecha que indica la dirección de la relación, u → v. El grafo de la Figura 15.2, que consta de los vértices `V = {C,D,E,F,H}` y de los arcos `A = {(C,D,), (D,F), (E,H), (H,E), (E,C)}` forma el grafo dirigido `G = {V,A}`

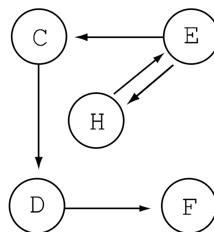

Figura 15.2 Grafo dirigido

Dado el arco `(u,v)` de un grafo, se dice que los vértices `u` y `v` son adyacentes. Si el grafo es dirigido, el vértice `u` es adyacente a `v`, y `v` es adyacente de `u`.

http://descargavariable.blogspot.com/

En los modelos realizados con grafos, a veces, una relación entre dos nodos tiene asociada una *magnitud*, denominada factor de peso, en cuyo caso se dice que es un *grafo valorado.* Por ejemplo, los pueblos que forman una comarca junto con la relación *entre un par de pueblos que están unidos por un camino:* esta relación tiene asociado el factor de peso, que es la distancia en kilómetros. La Figura 15.3 muestra un grafo valorado en el que cada arco tiene asociado un peso que es la longitud entre dos nodos.

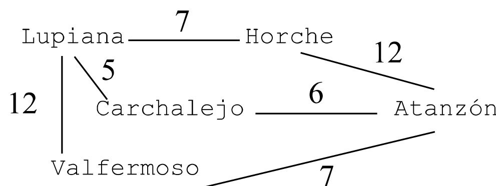

Figura 15.3 Grafo no dirigido valorado

> ***Definición***
>
> Un grafo permite modelar relaciones arbitrarias entre objetos. Un grafo `G = (V,A)` es un par formado por un conjunto de vértices o nodos, `V`, y un conjunto de arcos o aristas, `A`. Cada arco es el par `(u,w)`, siendo `u, w` dos vértices relacionados.

15.1.1. Grado de entrada, grado de salida de un nodo

El *grado* es una cualidad que se refiere a los nodos de un grafo. En un grafo no dirigido, el grado de un nodo `v`, `grado(v)`, es el número de arcos que contienen a `v`. En un grafo dirigido se distingue entre grado de entrada y grado de salida; *grado de entrada* de un nodo `v`, `gradent(v)`, es el número de arcos que llegan a `v`; *grado de salida* de `v`, `gradsal(v)`, es el número de arcos que salen de `v`.

Así, en el grafo no dirigido de la Figura 15.3, `grado(Lupiana) = 3`. En el grafo dirigido de la Figura 15.2, `gradent(D) = 1` y `gradsal(D) = 1`.

15.1.2. Camino

Un camino `P` de longitud `n` desde el vértice v_0 a v_n en un grafo `G`, es la secuencia de `n+1` vértices $v_0, v_1, v_2, \ldots, v_n$ tal que $(v_i, v_{i+1}) \in A$ *(arcos)* para $0 \le i \le n$. Matemáticamente, el camino $P = (v_0, v_1, v_2, \ldots, v_n)$.

En el grafo de la Figura 15.4 se pueden encontrar más de un camino; por ejemplo, $P_1 = (4,6,9,7)$ es un camino de longitud `3`. $P_2 = (10, 11)$ es un camino de longitud `1`. En resumen, la *longitud del camino* es el número de arcos que lo forma.

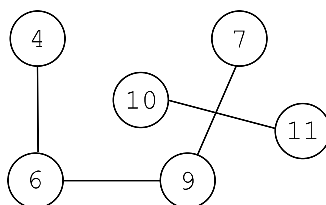

Figura 15.4 Grafo no dirigido de 5 vértices

http://descargavariable.blogspot.com/

Definición

La longitud de un camino es el número de arcos del camino. En un grafo valorado, la longitud del camino con pesos es la suma de los pesos de los arcos en el camino.

En el grafo valorado de la Figura 15.3, el camino (Lupiana, Valfermoso, Atanzón) tiene de longitud 12 + 7 = 19.

En algunos grafos se dan arcos desde un vértice a sí mismo, (v,v); entonces, el camino $v \rightarrow v$ es un bucle. Normalmente, en los grafos no hay nodos relacionados con sí mismo, no es frecuente encontrarse grafos con bucles.

Un camino $P = (v_0,v_1,v_2,..., v_n)$ es simple si todos los nodos que forman el camino son distintos, pudiendo ser iguales v_0, v_n, es decir, los extremos del camino.

En un grafo dirigido, un ciclo es un camino simple cerrado. Por tanto, un ciclo empieza y termina en el mismo nodo, $v_0 = v_n$, y además, debe tener más de un arco. Un grafo dirigido sin ciclos (acíclico) se acostumbra a denominar GDA *(Grafo Dirigido Acíclico)*. La Figura 15.5 muestra un grafo dirigido en el que los vértices (A, E, B, F, A) forman un ciclo de longitud 4. En general, un ciclo de longitud k se denomina *k-ciclo*.

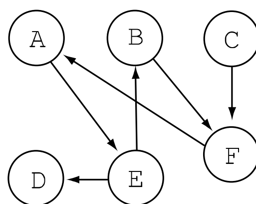

Figura 15.5 Grafo dirigido con ciclos

Un grafo no dirigido es *conexo* si existe un camino entre cualquier par de nodos que forman el grafo. Un grafo dirigido con esta propiedad se dice que es *fuertemente conexo*. Además, un *grafo completo* es aquel que tiene un arco para cualquier par de vértices.

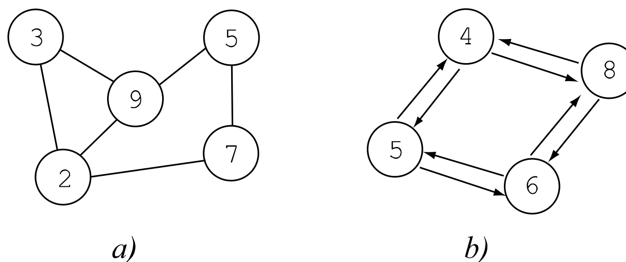

Figura 15.6 *a)* Grafo conexo; *b)* grafo fuertemente conexo

15.1.3. *Tipo Abstracto de Datos* Grafo

Es preciso definir las operaciones básicas para construir la estructura grafo y, en general, modificar sus elementos. En definitiva, especificar el *tipo abstracto de datos grafo*.

http://descargavariable.blogspot.com/

Ahora se definen operaciones básicas, a partir de las cuales se construye el grafo. Su realización depende de la representación elegida (matriz de adyacencia, o listas de adyacencia).

arista (u, v).	Añade el arco o arista `(u,v)` al grafo.
aristaPeso(u,v, w).	Para un grafo valorado, añade el arco `(u,v)` al grafo y el coste del arco, `w`.
borraArco(u,v).	Elimina del grafo el arco `(u,v)`.
adyacente(u,v).	Operación que devuelve *cierto* si los vértices `u`, `v` forman un arco.
nuevoVértice(u).	Añade el vértice `u` al grafo `G`.
borraVértice(u).	Elimina el vértice `u` del grafo `G`.

15.2. REPRESENTACIÓN DE LOS GRAFOS

Para trabajar con los grafos y aplicar algoritmos que permitan encontrar propiedades entre los nodos hay que pensar cómo representarlo en memoria interna, qué tipos o estructuras de datos se deben utilizar para considerar los nodos y los arcos.

Una primera simplificación es considerar los vértices o nodos como números consecutivos, empezando por el vértice `0`. Es preciso tener en cuenta que se ha de representar un número (finito) de vértices y de arcos que unen dos vértices. Se puede elegir una representación secuencial, mediante un *array* bidimensional, conocida como *matriz de adyacencia*; o bien, una representación dinámica, mediante una estructura multienlazada, denominada *listas de adyacencia*. La elección de una representación u otra depende del tipo de grafo y de las operaciones que se vayan a realizar sobre los vértices y arcos. Para un grafo denso (tiene la mayoría de los arcos posibles) lo mejor es utilizar una matiz de adyacencia. Para un grafo disperso (tiene, relativamente, pocos arcos) se suelen utilizar listas de adyacencia que se ajustan al número de arcos.

15.2.1. Matriz de adyacencia

La característica mas importante de un grafo, que distingue a uno de otro, es el conjunto de pares de vértices que están *relacionados*, o que son adyacentes. Por ello, la forma más sencilla de representación es mediante una matriz, de tantas filas/columnas como nodos, que permite modelar fácilmente esa cualidad.

Sea $G = (V, A)$ un grafo de n nodos, siendo $V = \{v_0, v_1, \ldots, v_{n-1}\}$ el conjunto de nodos, y $A = \{(v_i, v_j)\}$ el conjunto de arcos. Los nodos están numerados consecutivamente de 0 a $n-1$. La representación de los arcos se hace con una matriz A de $n \times n$ elementos, denominada matriz de adyacencia, tal que todo elemento a_{ij} puede tomar los valores:

$$a_{ij} \begin{cases} 1 & \text{si hay un arco } (v_i, v_j) \\ 0 & \text{si no hay arco } (v_i, v_j) \end{cases}$$

Ejemplo 15.1

Dado el grafo dirigido de la Figura 15.7 escribir la matriz de adyacencia.

Suponiendo que el orden de los vértices es `{D,F,K,L,R}`, entonces la matriz de adyacencia:

http://descargavariable.blogspot.com/

$$A = \begin{vmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{vmatrix}$$

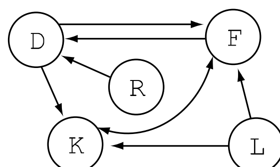

Figura 15.7 Grafo dirigido con los vértices `{D,F,K,L,R}`

Ejemplo 15.2

Dado el grafo no dirigido de la Figura 15.8 escribir la matriz de adyacencia.

El grafo está formado por `5` vértices. La matriz de adyacencia:

$$A = \begin{vmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{vmatrix}$$

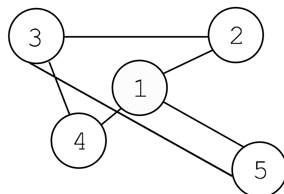

Figura 15.8 Grafo no dirigido con 5 vértices

En los grafos no dirigidos la matriz de adyacencia siempre es simétrica ya que las relaciones entre vértices no son ordenadas: si v_i *está relacionado* con v_j, entonces v_j *está relacionado* con v_i.

Los grafos que modelan problemas en los que un arco tiene asociado una magnitud, un *factor de peso*, también se representan mediante una matriz de tantas filas/columnas como nodos. Ahora un elemento cualquiera, a_{ij} representa el *coste* o *factor de peso* del arco (v_i, v_j). Un arco que no existe se puede representar con un valor imposible, por ejemplo *infinito*; también, se puede representar con el valor `0`, dependiendo de que `0` no pueda ser un factor de peso significativo de un arco. `A` esta matriz también se la denomina *matriz valorada*.

Ejemplo 15.3

Dado el grafo valorado dirigido de la Figura 15.9, escribir la matriz de pesos.

El grafo es un *grafo dirigido con factor de peso*. Si los vértices se numeran en el orden de `V = {Alicante, Barcelona, Cartagena, Murcia, Reus}`, la matriz de pesos es `P` y en ella se representa con `0` la no existencia de arco:

http://descargavariable.blogspot.com/

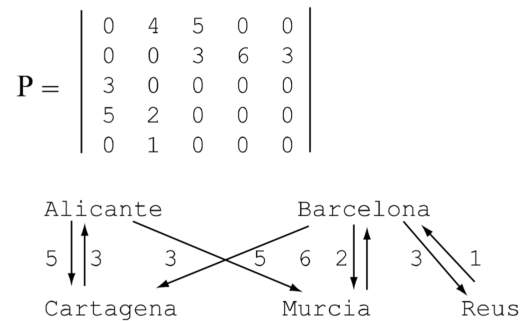

Figura 15.9 Grafo dirigido con factor de peso

> ***Nota***
>
> La matriz de adyacencia representa los arcos, relaciones entre un par de nodos de un grafo. Es una matriz de unos y ceros, que indican si dos vértices son adyacentes o no. En un grafo valorado, cada elemento representa el peso de la arista, y por ello se la denomina *matriz de pesos*.

15.2.2. Matriz de adyacencia: Clase `GrafoMatriz`

La clase `Vertice` representa un nodo del grafo, con su nombre `(String)` y número asignado. El constructor inicializa el nombre y pone como número de vértice `-1`; el método que añade el vértice al grafo establece el número que le corresponda.

```
package Grafo;

public class Vertice
{
   String nombre;
   int numVertice;
   public Vertice(String x)
   {
      nombre = x;
      numVertice = -1;
   }

   public String nomVertice() // devuelve identificador del vértice
   {
      return nombre;
   }

   public boolean equals(Vertice n) // true, si dos vértices son iguales
   {
      return nombre.equals(n.nombre);
   }

   public void asigVert(int n) // establece el número de vértices
```

http://descargavariable.blogspot.com/

```
    {
       numVertice = n;
    }

    public String toString() // características del vértice
    {
       return nombre + " (" + numVertice + ")";
    }
 }
```

La clase `GrafoMatriz` define la matriz de adyacencia, el *array* de vértices y los métodos para añadir nodos y arcos al grafo. La declaración de la clase es:

```
package Grafo;

public class GrafoMatriz
{
   int numVerts;
   static int MaxVerts = 20;
   Vertice [] verts;
   int [][] matAd;
   ...
```

Constructor

El constructor sin argumentos crea la matriz de adyacencia para un máximo de vértices preestablecido; el otro constructor tiene un argumento con el máximo número de vértices:

```
public GrafoMatriz()
{
   this(maxVerts);
}

public GrafoMatriz(int mx)
{
   matAd = new int [mx][mx];
   verts = new Vertice[mx];
   for (int i = 0; i < mx; i++)
      for (int j = 0; i < mx; i++)
         matAd[i][j] = 0;
   numVerts = 0;
}
```

Añadir un vértice

La operación recibe la etiqueta `(String)` de un vértice del grafo, comprueba si ya está en la lista de vértices, en caso negativo incrementa el número de vértices y lo asigna a la lista.

```
public void nuevoVertice (String nom)
{
   boolean esta = numVertice(nom) >= 0;
   if (!esta)
   {
      Vertice v = new Vertice(nom);
      v.asigVert(numVerts);
      verts[numVerts++] = v;
   }
}
```

http://descargavariable.blogspot.com/

`numVertice()` busca el vértice en el *array*. Devuelve `-1` si no lo encuentra:

```
boolean int numVertice(String vs)
{
   Vertice v = new Vertice(vs);
   boolean encontrado = false;
   int i = 0;
   for (; (i < numVerts) && !encontrado;)
   {
      encontrado = verts[i].equals(v);
      if (!encontrado) i++ ;
   }
   return (i < numVerts) ? i : -1 ;
}
```

Añadir un arco

El método recibe el nombre de cada vértice del arco, busca, en el *array,* el número de vértice asignado a cada uno de ellos y marca la matriz de adyacencia.

```
public void nuevoArco(String a, String b)throws Exception
{
   int va, vb;
   va = numVertice(a);
   vb = numVertice(b);
   if (va < 0 || vb < 0) throw new Exception ("Vértice no existe");
   matAd[va][vb] = 1;
}
```

Otra versión del método recibe directamente los números de vértice del arco.

```
public void nuevoArco(int va, int vb)throws Exception
{
   if (va < 0 || vb < 0) throw new Exception ("Vértice no existe");
   matAd[va][vb] = 1;
}
```

Para *grafos valorados* este método tiene un tercer argumento que es el *factor de peso* del arco.

> ***Ejecución***
>
> El tiempo de ejecución de la operación que realiza la entrada completa del grafo en memoria depende de la densidad del grafo si se considera un grafo denso el tiempo de ejecución es cuadrático, $O(n^2)$.

Adyacente

Determina si dos vértices, `v1` y `v2`, forman un arco; es decir, si el elemento de la matriz de adyacencia es `1`. Se escriben dos versiones.

```
public boolean adyacente(String a, String b)throws Exception
{
   int va, vb;
   va = numVertice(a);
```

http://descargavariable.blogspot.com/

```
      vb = numVertice(b);
      if (va < 0 || vb < 0) throw new Exception ("Vértice no existe");
      return matAd[va][vb] == 1;
   }
   public boolean adyacente(int va, int vb)throws Exception
   {
      if (va < 0 || vb < 0) throw new Exception ("Vértice no existe");
      return matAd[va][vb] == 1;
   }
```

Ejercicio 15.1

Dado un grafo no dirigido de n nodos representado por su matriz de adyacencia, escribir una aplicación que realice su entrada en memoria.

El grafo se guarda en memoria utilizando la clase `GrafoMatriz`. Al constructor de la clase se le pasa el número de nodos del grafo. El usuario sólo tiene que introducir los nombres de los vértices; la llamada al método `nuevoVertice()` crea el vértice y lo asigna a la estructura. A continuación se crean los arcos, para lo cual se leen los pares de vértices con sus nombres y se llama al método `nuevoArco()`.

(Codificación se encuentra en la página web del libro, archivo Ejercicio 15.1.)

15.3. LISTAS DE ADYACENCIA

La representación de un grafo con matriz de adyacencia no es eficiente cuando el grafo es poco denso (disperso), es decir, tiene pocos arcos, y por tanto la matriz de adyacencia tiene muchos ceros. Para grafos dispersos, la matriz de adyacencia ocupa el mismo espacio que si el grafo tuviera muchos arcos (grafo denso). Cuando esto ocurre, se elige la representación del grafo con listas enlazadas, denominadas *listas de adyacencia*.

Las listas de adyacencia son una estructura multienlazada formada por una tabla directorio en la que cada elemento representa un vértice del grafo, del cual emerge una lista enlazada con todos sus vértices adyacentes. Es decir, cada lista representa los arcos con el vértice origen del nodo de la lista directorio, por eso se llama lista de adyacencia.

Ejemplo 15.4

La Figura 15.10 representa un grafo dirigido. La representación mediante listas de adyacencia se encuentra en la Figura 15.11.

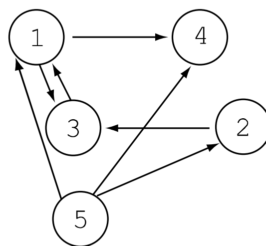

Figura 15.10 Grafo dirigido

http://descargavariable.blogspot.com/

Si se analiza el vértice `5`, es adyacente a los vértices `1, 2` y `4`; por ello su lista de adyacencia consta de tres nodos, cada uno con el vértice destino que forma el arco. El vértice `4` no es origen de ningún arco, su lista de adyacencia está vacía.

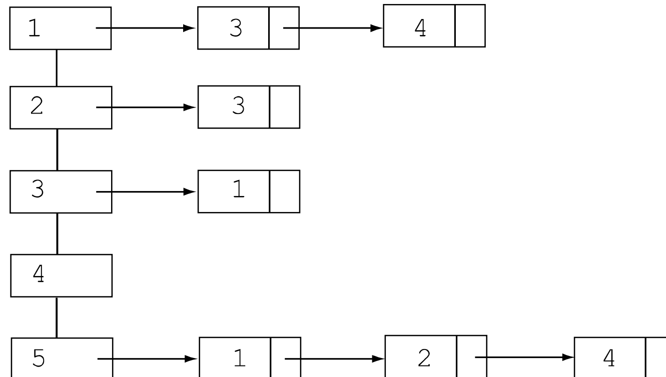

Figura 15.11 Listas de adyacencia del grafo de la Figura 15.10

15.3.1. Clase `GrafoAdcia`

Cada elemento de la tabla directorio es un vértice del grafo que guarda el identificador del vértice, su número y la lista de adyacencia. La clase `VérticeAdy` declara estos datos. Los métodos de la clase tienen la misma funcionalidad que la clase `Vertice`.

```
String nombre;
int numVertice;
Lista lad; // lista de adyacencia
           // constructor de la clase
public VerticeAdy(String x)
{
   nombre = x;
   numVertice = -1:
   lad = new Lista();
}
```

La lista de adyacencia de un vértice `u`, consta de tantos nodos como arcos tiene por origen `u`. Un nodo de la lista contiene un objeto de la clase `Arco`, en la cual se guarda el vértice destino `v` del arco que tiene su origen en `u`; además, en los grafos valorados, el *peso* asociado al arco. La clase `Arco`:

```
package Grafo;
public class Arco
{
   int destino;
   double peso;
   public Arco(int d)
   {
      destino = d;
   }

   public Arco(int d, double p)
   {
```

http://descargavariable.blogspot.com/

```
      this(d);
      peso = p;
   }
   public int getDestino()
   {
      return destino;
   }

   public boolean equals(Object n)
   {
      Arco a = (Arco)n;
      return destino == a.destino;
   }
}
```

La clase `GrafoAdcia` define la tabla de vértices con sus respectivas listas de adyacencia. Implementa las operaciones básicas para crear un grafo: añadir un vértice, insertar un arco, dar de baja un vértice y sus arcos...

```
package Grafo;
public class GrafoAdcia
{
   int numVerts;
   static int maxVerts = 20;
   VerticeAdy [] tablAdc;
```

Constructor

Crea la tabla de listas de adyacencia para un número de vértices preestablecido:

```
public GrafoAdcia(int mx)
{
   tablAdc = new VerticeAdy[mx];
   numVerts = 0;
   maxVerts = mx;
}
```

15.3.2. Realización de las operaciones con listas de adyacencia

Los métodos que implementan las operaciones forman la interfaz principal de la clase `GrafoAdcia`[1].

Nuevo vértice

Añade un nuevo vértice a la lista directorio. Si el vértice ya está en la tabla no hace nada, devuelve control; si es nuevo, se asigna a continuación del último. La implementación no considera que pueda haber *overflow*; el lector puede implementar un método auxiliar que duplique la tabla si ésta se llena. El tiempo de la operación depende de la búsqueda, en el peor caso es lineal, `O(n)` siendo `n` el número de nodos.

Arista

Esta operación da entrada a un arco del grafo. El método tiene como entrada el vértice origen y el vértice destino. El método `adyacente()` determina si la arista ya está en la lista de adyacencia,

[1] La codificación completa de la clase se encuentra en la página web del libro.

http://descargavariable.blogspot.com/

y, por último, el *Arco* se inserta en la lista de adyacencia del nodo origen. La inserción se hace como primer nodo para que el tiempo sea constante, `O(1)`.

Otra versión del método recibe, directamente, los números de vértices que forman los extremos del arco.

Para añadir una arista en un grafo valorado, se debe asignar el *factor de peso* al crear el *Arco*.

Borrar arco

La operación consiste en eliminar el nodo de la lista de adyacencia del vértice origen que contiene el arco del vértice destino. Una vez encontrada la dirección de ambos vértices en la lista de nodos se accede a la correspondiente lista de adyacencia para proceder a borrar el nodo que contiene el vértice destino.

Adyacente

La operación determina si dos vértices, `v1` y `v2`, forman un arco. En definitiva, si el vértice `v2` está en la lista de adyacencia de `v1`. El método `buscarLista()` devuelve la dirección del nodo que contiene al arco, si no está devuelve `null`.

Borrar vértice

Eliminar un vértice de un grafo es una operación que puede ser considerada costosa en tiempo de ejecución, ya que supone acceder a cada uno de los elementos de la multiestructura. En primer lugar, hay que buscar la dirección (puntero) del vértice en la lista directorio. En segundo lugar, eliminar cada uno de los nodos de la correspondiente lista de adyacencia. Y por último, recorrer cada lista de adyacencia y si encuentra algún arco con el vértice que se está borrando, se elimina de la lista. (Se deja como ejercicio la implementación de la operación.)

15.4. RECORRIDO DE UN GRAFO

En general, *recorrer una estructura* consiste en visitar (procesar) cada uno de los nodos a partir de uno dado. Se puede recorrer una lista, o un árbol en, por ejemplo, *preorden* partiendo del nodo raíz. De igual forma, *recorrer un grafo* consiste en visitar todos los vértices alcanzables a partir de uno dado. Muchos de los problemas que se plantean con los grafos exigen examinar las aristas o arcos de que consta y procesar los vértices.

Siendo `V` el conjunto de vértices del grafo, los algoritmos de *recorrido* de grafos parten de un nodo, `v`, y mantienen un conjunto de nodos *marcados* (*procesables*), `W`, que inicialmente es el nodo o vértice de partida, `v`. Cada *pasada* del algoritmo retira un nodo `w`, del conjunto `W`, que se procesa y para cada una de las aristas cuyo origen es `w, (w,u)`, su vértice adyacente `u` se añade a `W` si no está marcado. El algoritmo continúa hasta que el conjunto `W` se queda vacío, en ese momento todo vértice no marcado no es accesible desde el nodo de partida `v`. Si se necesita un recorrido completo, se puede continuar desde cualquier vértice no marcado.

Nota

Hay dos formas de recorrer un grafo: *recorrido en profundidad* y *recorrido en anchura.* Si el conjunto de nodos *marcados* se trata como una cola, el recorrido es en anchura; si se trata como una pila, el recorrido es en profundidad.

http://descargavariable.blogspot.com/

15.4.1. Recorrido en anchura

Se utiliza una cola como estructura en la que se mantienen los vértices marcados que se van a procesar posteriormente. El proceso de los elementos en una cola (*primero en entrar primero en salir*) hace que, a partir del vértice de partida `v` se procesen primero todos los vértices adyacentes a `v`, después los adyacentes de éstos que no estuvieran ya *marcados* o *visitados*, y así sucesivamente con los adyacentes de los adyacentes.

El orden en que se visitan los nodos en el recorrido en anchura se expresa de manera más concisa en los siguientes pasos:

1. *Marcar* el vértice de partida `v`.
2. *Meter* en la cola el vértice de partida `v`.
3. Repetir los pasos 4 y 5 hasta que se cumpla la condición *cola vacía*.
4. *Quitar* el nodo frente de la cola, `w`, visitar `w`.
5. *Meter* en la cola todos los vértices adyacentes a `w` que no estén marcados y, a continuación marcar esos vértices.
6. Fin del recorrido.

Ejemplo 15.5

Recorrer en anchura el grafo dirigido de la Figura 15.12.

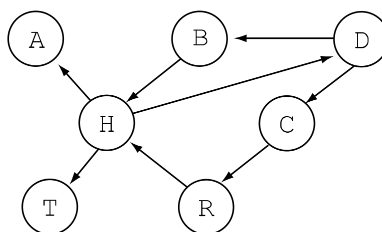

Figura 15.12 Grafo dirigido

El recorrido se inicia a partir del nodo `D`, se marca y se mete en la cola. Iterativamente, se quita el nodo frente, se procesa, se meten en la cola sus adyacentes no marcados y se marcan. Los sucesivos elementos de la cola se muestran en la Figura 15.13. En la columna de vértices procesados el vértice en negrita es el que se procesa en esa pasada, y en la cola los vértices en cursiva son los que se meten en la cola y son marcados.

COLA	Vértices procesados
D	
B C	**D**
C H	**B**
H R	**C**
R A T	**H**
A T	**R**
T	**A**
cola vacía	**T**

Figura 15.13 Recorrido en anchura del grafo de la Figura 15.12

http://descargavariable.blogspot.com/

El recorrido en anchura, a partir del nodo `D`, del grafo de la Figura 15.12 ha accedido a todos los nodos del grafo; se puede afirmar que todos sus vértices son alcanzables desde el vértice `D`.

15.4.2. Recorrido en profundidad

La búsqueda de los vértices y aristas de un grafo en profundidad persigue el mismo objetivo que el recorrido en anchura: *visitar* todos los vértices del grafo alcanzables desde un vértice dado. Difiere este recorrido con el recorrido en anchura sólo en el orden en que se procesan los vértices adyacentes. El orden en el recorrido en profundidad es el que determina la estructura pila.

El recorrido empieza por un vértice `v` del grafo, se marca como visitado y se *mete* en la pila. Después se recorre en profundidad cada vértice adyacente a `v` no visitado; hasta que no haya más vértices adyacentes no visitados. Esta estrategia de examinar los vértices se denomina en profundidad porque la dirección de *visitar* es hacia *adelante* mientras resulta posible; al contrario que la búsqueda en anchura que primero visita todos los vértices posibles en *amplitud*.

Ejemplo 15.6

Recorrer en profundidad el grafo dirigido de la Figura 15.12.

El recorrido se inicia a partir del nodo `D`, se marca y se mete en la pila. Iterativamente, se quita el nodo cabeza, se procesa, se meten en la pila sus adyacentes no marcados y se marcan. Los sucesivos elementos de la pila se muestran en la Figura 15.14. En la columna de vértices procesados en *negrita* está el que se *visita* en esa pasada, y los vértices en cursiva son los que se meten en la pila y a la vez son marcados.

FILA	Vértices procesados
D	
B C	**D**
B R	**C**
B H	**R**
B A T	**H**
B A	**T**
B	**A**
pila vacía	**B**

Figura 15.14 Recorrido en profundidad del grafo de la Figura 15.12

15.4.3. Implementación: Clase `RecorreGrafo`

La implementación de estos algoritmos se realiza con métodos `static` que reciben como argumento el grafo (con matriz o con listas de adyacencia) y el vértice de partida del recorrido.

Recorrido en anchura

Utiliza una cola en la que se guardan los vértices adyacentes al que se ha procesado. Para determinar si un vértice está o no marcado se pueden seguir varias alternativas, una de ellas utiliza el *array* `m[]`, de tantas posiciones como vértices y con una correspondencia directa entre índice

http://descargavariable.blogspot.com/

y número de vértice, para indicar si un nodo está marcado. El estado de nodo `i` *no marcado* se establece con una *clave*, si está marcado tendrá un número finito.

Al recorrer el grafo se puede obtener cierta información relativa a los vértices. En la codificación que se realiza a continuación, se guarda en `m[i]` el número mínimo de aristas para cualquier camino, desde el vértice de partida hasta el vértice `i`. El vértice de partida, `v`, se inicializa `m[v] = 0` ya que los caminos parten desde ese vértice; las otras posiciones de `m[i]` se inicializan al valor *clave* que expresa *vértice no marcado*. La implementación realizada es para un grafo representado por su matriz de adyacencia; el vértice origen viene dado por su nombre `(String)`.

```
public static
int[]recorrerAnchura(GrafoMatriz g, String org) throws Exception
{
   int w, v;
   int [] m;

   v = g.numVertice(org);
   if (v < 0) throw new Exception("Vértice origen no existe");

   ColaLista cola = new ColaLista();
   m = new int[g.numeroDeVertices()];
   // inicializa los vértices como no marcados
   for (int i = 0; i < g.numeroDeVertices(); i++)
      m[i] = CLAVE;
   m[v] = 0;     // vértice origen queda marcado
   cola.insertar(new Integer(v));
   while (! cola.colaVacia())
   {
      Integer cw;
      cw = (Integer) cola.quitar()
      w = cw.intValue();
      System.out.println("Vértice " + g.verts[w] + "visitado");
      // inserta en la cola los adyacentes de w no marcados
      for (int u = 0; u < g.numeroDeVertices(); u++)
         if ((g.matAd[w][u] == 1) && (m[u] == CLAVE))
         {
            // se marca vertice u con número de arcos hasta el
            m[u] = m[w] + 1;
            cola.insertar(new Integer(u));
         }
   }
   return m;
}
```

Para visitar todos los vértices del grafo, una vez que ha terminado el recorrido a partir de uno dado, `v`, hay que buscar si queda algún vértice sin marcar, en cuyo caso se vuelve a recorrer a partir de él; así sucesivamente hasta que todos los vértices estén marcados.

Recorrido en profundidad

La implementación utiliza una pila para establecer el orden de recorrido. También se podría realizar una implementación recursiva para evitar la utilización de la pila. Como en el recorrido

http://descargavariable.blogspot.com/

en anchura, los vértices ya visitados se marcan en el *array* m[]. Ahora, el grafo está representado mediante listas de adyacencia.

```
static
public int[] recorrerProf(GrafoAdcia g, String org) throws Exception
{
   int v, w;
   PilaLista pila = new PilaLista();
   int [] m;
   m = new int[g.numeroDeVertices()];
   // inicializa los vértices como no marcados
   v = g.numVertice(org);
   if (v < 0) throw new Exception("Vértice origen no existe");
   for (int i = 0; i < g.numeroDeVertices(); i++)
      m[i] = CLAVE;
   m[v] = 0;                       // vértice origen queda marcado

   pila.insertar(new Integer(v));
   while (!pila.pilaVacia())
   {
      Integer cw;
      cw = (Integer) pila.quitar();
      w = cw.intValue();
      System.out.println("Vértice" + g.tablAdc[w] + "visitado");
      // inserta en la pila los adyacentes de w no marcados
      // recorre la lista con un iterador
      ListaIterador list = new ListaIterador(g.tablAdc[w].lad);
      Arco ck;
      do
      {
         int k;
         ck = (Arco) list.siguiente();
         if (ck != null)
         {
            k = ck.getDestino();  // vértice adyacente
            if (m[k] == CLAVE)
            {
               pila.insertar(new Integer(k));
                  m[k] = 1;        // vértice queda marcado
            }
         }
      } while (ck != null);
   }
   return m;
}
```

15.5. CONEXIONES EN UN GRAFO

Al modelar un conjunto de objetos y sus relaciones mediante un grafo, una de las cuestiones que generalmente interesa conocer es si desde cualquier vértice se puede acceder al resto de los vértices del grafo, es decir, si todos los vértices están conectados o, simplemente, si el grafo es conexo. Para un grafo dirigido, la conectividad entre todos los vértices se denomina: *grafo fuertemente conexo*.

http://descargavariable.blogspot.com/

15.5.1. Componentes conexas de un grafo

Un *grafo* no dirigido G es *conexo* si existe un camino entre cualquier par de vértices que forman el grafo. En el caso de que el grafo no sea conexo resulta interesante determinar aquellos subconjuntos de vértices que mutuamente están conectados; es decir, las componentes conexas del mismo. El algoritmo para determinar las componentes conexas de un grafo G se basa en el recorrido del grafo (en *anchura* o en *profundidad)*. Los pasos a seguir son los siguientes:

1. Realizar un recorrido del grafo a partir de cualquier vértice w.
2. Si en el recorrido se han *marcado* todos los vértices, entonces el grafo es conexo.
3. Si el grafo no es conexo, los vértices marcados forman una componente conexa.
4. Se toma un vértice no marcado, z, y se realiza de nuevo el recorrido del grafo a partir de z. Los nuevo vértices marcados forman otra componente conexa.
5. El algoritmo termina cuando todos los vértices han sido marcados (visitados).

15.5.2. Componentes fuertemente conexas de un grafo

Un grafo dirigido fuertemente conexo es aquel en el cual existe un camino entre cualquier par de vértices del grafo. De no ser fuertemente conexo se pueden determinar componentes fuertemente conexas del grafo. La Figura 15.15 muestra un grafo dirigido y sus dos componentes fuertemente conexas.

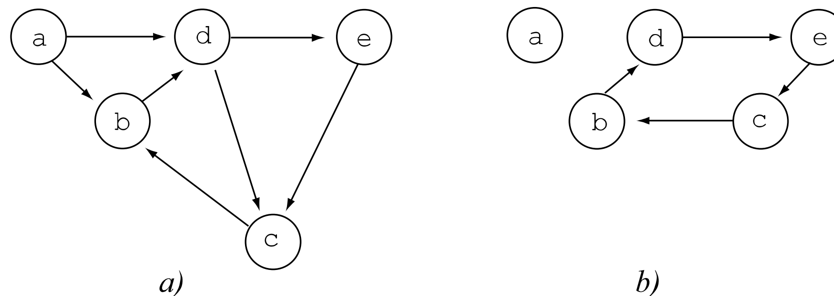

Figura 15.15 *a)* Grafo dirigido; *b)* componentes fuertes del grafo

El recorrido en profundidad a partir de un vértice dado permite diseñar un algoritmo para encontrar si un grafo es fuertemente conexo o, en su caso, determinar las componentes fuertemente conexas. A continuación se indican los pasos que sigue:

1. Obtener el conjunto de vértices descendientes del vértice de partida v, D(v), incluido el propio vértice v.
2. Obtener el conjunto de ascendientes de v, A(v), incluido el propio vértice v.
3. Los vértices comunes del conjunto de descendientes y ascendientes de v: $D(v) \cap A(v)$, es el conjunto de vértices que forman la componente fuertemente conexa a la que pertenece v. Si este conjunto es el conjunto de todos los vértices del grafo, entonces es fuertemente conexo.
4. Si no es un grafo fuertemente conexo se selecciona un vértice cualquiera, w, que no esté en ninguna componente fuerte de las encontradas ($w \neq D(v) \cap A(v)$) y se procede de la misma manera, es decir, se repite a partir del primer paso hasta obtener todas las componentes fuertes del grafo.

http://descargavariable.blogspot.com/

Para buscar los vértices descendientes de `v` se realiza un recorrido en profundidad del grafo a partir de `v`. Los vértices que son alcanzados se guardan en el conjunto `D`.

Los vértices ascendientes de `v` son aquellos desde los que se puede alcanzar a `v`. Por consiguiente, se obtiene el grafo inverso, cambiando la dirección de las aristas, y a continuación se recorre en profundidad el grafo inverso a partir de `v`. Los vértices alcanzados en el recorrido son los ascendientes de `v`.

Al recorrer el grafo de la Figura 15.15 en profundidad, a partir del vértice `d`, el conjunto de vértices que alcanza es: `{d, c, b, e}`. Repitiendo el recorrido en el grafo inverso, Figura 15.16, se obtiene los vértices ascendientes: `{d, b, c, e}`. Los vértices comunes `(d, b, c, e)`, forman una componente fuertemente conexa.

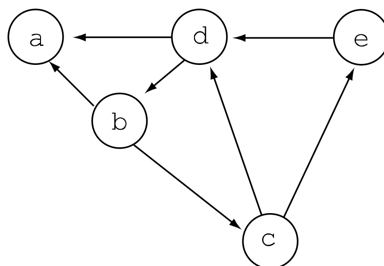

Figura 15.16 Grafo inverso del de la Figura 15.15

Ejercicio 15.2

Se tiene un grafo dirigido de n nodos, representado por su matriz de adyacencia, A. Se desea determinar si el grafo es fuertemente conexo, o bien en el caso de que no lo sea, encontrar las componentes fuertemente conexas. Las compones fuertes se mostrarán por pantalla.

El algoritmo recorre el grafo a partir de un vértice `v`, y también recorre el grafo inverso a partir del mismo vértice. El método `grafoInverso()` crea el grafo inverso, cambiando la dirección de los arcos originales.

Se parte de cualquier vértice, por ejemplo el primero, para encontrar el conjunto de vértices que están interconectados. Si son todos, el grafo es fuertemente conexo; en caso contrario, se repite el proceso a partir de un vértice que no esté formando parte de componentes anteriores. El recorrido en profundidad a partir de `v` encuentra los vértices descendientes de `v`; el recorrido se repite a partir del mismo vértice, pero con el grafo inverso, los vértices marcados forman los vértices ascendientes de `v`. Los vértices marcados en ambos recorridos forman una componente fuerte del grafo.

Al haber una correspondencia biunívoca entre el número de vértices y los índices de los *arrays* de vértices, `descendientes[]` y `ascendientes[]`, si un vértice `i` está presente se activa (se pone a *true*) la misma posición del *array;* de esa forma se guarda el vértice en el correspondiente conjunto. Esto facilita la operación de intersección (vértices comunes) ya que, simplemente, si se cumple `ascendientes[i] && descendientes[i]` el nodo `i` pertenece a ambos conjuntos. Además, en el *array* `bosque[]` se marcan los vértices que ya están formando parte de una componente conexa. El método `todosArboles()` devuelve un vértice que todavía no forma parte de componente conexa, y el proceso termina cuando devuelve el valor clave `-1`.

El método que da entrada a los componentes del grafo: vértices y arcos, se deja como ejercicio al lector.

http://descargavariable.blogspot.com/

```
import java.io.*;
import Grafo.*;
public class ComponentesFuertes
{
   static BufferedReader entrada = new BufferedReader(
                              new InputStreamReader(System.in));
   static final int CLAVE = 0xffff;
   public static void main(String [] a)throws Exception
   {
      int n, i, v;
      GrafoMatriz ga;
      GrafoMatriz gaInverso;

      System.out.print("Número de vértices del grafo: ");
      n = Integer.parseInt(entrada.readLine());

      ga = new GrafoMatriz(n);
      gaInverso = new GrafoMatriz(n);
      int []m = new int [n];
      int []descendientes = new int[n];
      int []ascendientes = new int[n];
      int []bosque = new int[n];

      entradaGrafo(ga, n);
      grafoInverso(ga, gaInverso, n);
      Vertice [] vs = new Vertice[n];
      vs = ga.vertices();
      for (i = 0; i < n; i++)
         bosque[i] = 0;

      v = 0;   // vértice de partida
      do {
         m = RecorreGrafo.recorrerProf(ga, vs[v].nomVertice());
         // se obtiene conjunto de vértices descendientes
         for (i = 0; i < n; i++)
         {
            descendientes[i] = m[i]!= CLAVE ? 1 : 0;
         }
         // recorre el grafo inverso y obtiene ascendientes
         m = RecorreGrafo.recorrerProf(gaInverso, vs[v].nomVertice());
         // se obtiene conjunto de vértices descendientes
         for (i = 0; i < n; i++)
         {
            ascendientes[i] = m[i]!= CLAVE ? 1 : 0;
         }
         System.out.print("\nComponente conexa { ");
         for (i = 0; i < n; i++)
         {
            if (descendientes[i] * ascendientes[i] == 1)
            {
               System.out.print(" " + vs[i].nomVertice());
               bosque[i] = 1;
            }
         }
         System.out.println(" }");
```

http://descargavariable.blogspot.com/

```
        // vértice a partir del cual se obtiene otra componente
        v = todosArboles(bosque,n);
      } while (v != -1);
   } // fin del método main
   static void
   grafoInverso(GrafoMatriz g, GrafoMatriz x, int n) throws Exception
   {
      Vertice [] vr = g.vertices();
      for (int i = 0; i < n; i++)
         x.nuevoVertice(vr[i].nomVertice());
      for (int i = 0; i < n; i++)
         for (int j = 0; j < n; j++)
          if (g.adyacente(i,j)) x.nuevoArco(j,i);
   }
   static int todosArboles(int [] bosque, int n)
   {
      int i, w;
      w = i = -1;
      do
      {
         if (bosque[++i] == 0)
            w = i;
      } while ((i < n - 1)  && (w == -1));
      return w;
   }
   static void entradaGrafo(GrafoMatriz gr, int n)
                                      throws Exception{...}
}
```

15.6. MATRIZ DE CAMINOS. CIERRE TRANSITIVO

Dado un grafo `G`, un camino de longitud `n` desde el vértice v_0 hasta el vértice v_n es una secuencia de `n+1` vértices $v_0, v_1, v_2, \ldots, v_n$ tal que el vértice inicial es v_0, el vértice final v_n y son adyacentes (v_i, v_{i+1}) *para* $0 \le i \le n$. Encontrar caminos entre un par de vértices, de una determinada longitud es una tarea relativamente sencilla, aunque poco eficiente, si se tiene la matriz de adyacencia del grafo.

Consideremos por un momento que la matriz de adyacencia, `A`, es de tipo `boolean`, la expresión $A_{i,k}$ `&&` $A_{k,j}$ es *verdadera si y sólo si* los valores de ambos operandos lo son. Esta hipótesis implica que hay un arco desde el vértice `i` al vértice `k` y otro desde el vértice `k` al `j`. También se puede afirmar, que si $A_{i,k}$ `&&` $A_{k,j}$ es *verdadera* existe un camino de longitud `2` desde el vértice `i` al `j`. Ese sencillo razonamiento se puede ampliar a la siguiente expresión:

$$(A_{i1} \ \&\& \ A_{1j}) \ || \ (A_{i2} \ \&\& \ A_{2j}) \ || \ \ldots \ ||(A_{in} \ \&\& \ A_{nj})$$

Si la expresión es *verdadera* implica que hay al menos un camino de longitud `2` desde el vértice `i` al vértice `j` que pase por el vértice `1`, o por el `2` ... o por el vértice `n`. Si ahora en la expresión se cambia el operador `&&` por el operador *producto* y `||` por el operador *suma*, la expresión es el elemento A_{ij} de la matriz producto A^2. La conclusión es inmediata: los elementos (A_{ij}) de la matriz A^2 son distintos de cero si existe un camino de longitud `2` desde el vértice `i` al vértice `j`, $\forall i,j = 1..n$.

http://descargavariable.blogspot.com/

El razonamiento se puede extender para encontrar caminos de longitud 3; realizando el producto matricial $A^2 \times A = A^3$ se encuentran caminos de longitud 3 entre cualquier par de vértices del grafo.

A tener en cuenta

Una manera de encontrar los caminos de longitud m entre cualquier par de vértices de un grafo es mediante el producto matricial de A^{m-1} por la matriz de adyacencia A.

A continuación se siguen los mismos razonamientos para obtener caminos del grafo de la Figura 15.17, cuya matriz de adyacencia es la indicada en la citada figura:

$$A = \begin{vmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{vmatrix}$$

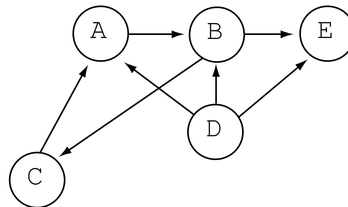

Figura 15.17 Grafo dirigido de 5 vértices

El producto matricial A x A, considerando todo valor distinto de 0 como 1 (*verdadero*):

$$A^2 = \begin{vmatrix} 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{vmatrix}$$

Si, por ejemplo, A_{15} toma el valor 1 significa que hay un camino de longitud 2, desde A – E, formado por los arcos: (A → B → E).

De igual manera, el producto matricial $A^2 \times A$:

$$A^3 = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{vmatrix}$$

Si nos fijamos en A_{41}, su valor es 1 ya que hay un camino longitud 3 desde D – A, formado por los arcos: (D → B → C → A).

http://descargavariable.blogspot.com/

Realizando el producto matricial, el valor almacenado en cualquier posición, A_{ij}, no sólo indica si hay camino de longitud `m`, sino además el número de caminos de dicha longitud entre los vértices `i` y `j`.

A recordar

La forma de obtener el número de caminos de longitud `k` entre cualquier par de vértices de un grafo es obtener el producto matricial A^2, A^3 ... A^k. Entonces, el elemento $A_k(i,j)$ contiene el número de caminos de longitud `k` desde el vértice `i` hasta el vértice `j`.

Ejecución

La eficiencia del algoritmo para encontrar el número de caminos de longitud `k` es muy pobre. El producto matricial se realiza con tres bucles anidados, complejidad cúbica $O(n^3)$, además, este producto se realiza `k-1` veces.

15.6.1. Matriz de caminos y cierre transitivo

Sea `G` un grafo con `n` vértices, la matriz de caminos de `G` es la matriz `P` de `n*n` elementos, definida como:

$$P_{ij} \begin{cases} 1 & \textit{si hay un camino desde } (v_i, v_j) \\ 0 & \textit{si no hay camino desde } (v_i, v_j) \end{cases}$$

Se encuentra la siguiente relación entre la matriz de caminos `P`, la matriz de adyacencia `A` y las sucesivas potencias de `A` para encontrar los caminos de longitud `m`:

"dada la matriz $B_n = A + A_2 + A_3 + \ldots + A_n$, la matriz de caminos `P` es tal que un elemento $P_{ij} = 1$ *si y sólo si* $B_n(i,j) = 1$ y en otro caso $P_{ij} = 0$ ".

El **cierre transitivo** o contorno transitivo de un grafo `G` es otro grafo, `G´`, con los mismos vértices y cuya matriz de adyacencia es la matriz de caminos del grafo `G`.

15.7. PUNTOS DE ARTICULACIÓN DE UN GRAFO

Un *punto de articulación* de un grafo no dirigido es un vértice `v` que tiene la propiedad de que si se elimina junto a sus arcos, el componente conexo en que está el vértice se divide en dos o más componentes. Por ejemplo, el grafo de la Figura 15.18 tiene dos puntos de articulación, el vértice `A` y el vértice `C`. En la Figura 15.18*b* se observa que al retirar el vértice `A` y sus arcos, el grafo se divide en dos componentes conexas: `{C,B,E,F}` y `{D}`. De igual forma, se observa en la Figura 15.18*c* que al eliminar el vértice `C` el grafo se divide en dos componentes conexas: `{B,E,F}` y `{A,D}`. Sin embargo, al suprimir cualquier otro vértice del grafo, éste no se divide en componentes.

http://descargavariable.blogspot.com/

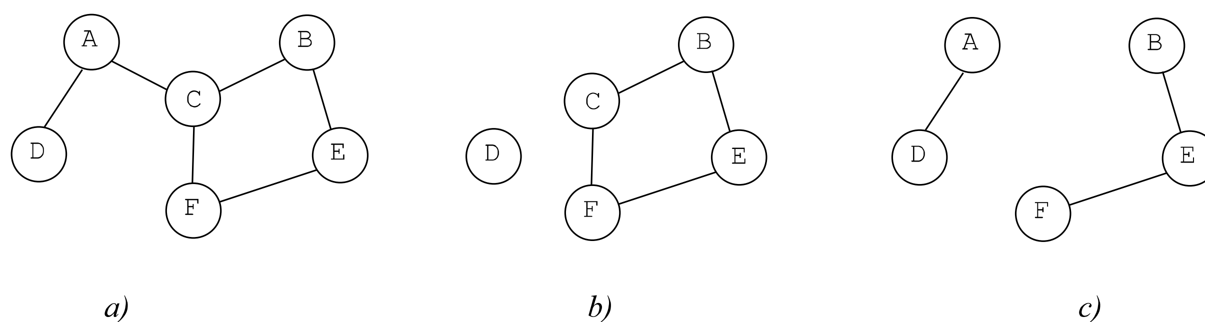

Figura 15.18 *a)* Grafo origen; *b)* después de quitar el nodo A; *c)* después de quitar el nodo C

Se estudian los *puntos de articulación* debido a que los grafos tiene propiedades relativas a ellos. Así, un grafo sin puntos de articulación se dice que es un grafo *biconexo*. De no ser el grafo biconexo, es interesante encontrar componentes biconexos. Un grafo tiene *conectividad* `k` si la eliminación de `k-1` vértices cualesquiera no divide al grafo en componentes conexas (no lo desconecta).

A considerar

El estudio de los puntos de articulación de grafos, como las redes, es importante porque determinan el grado de *conectividad* del grafo y cuanto mayor es la conectividad del grafo, mayor probabilidad tiene de mantener la estructura ante el *fallo* (eliminación) de alguno de sus vértices.

15.7.1. Búsqueda de los puntos de articulación

El algoritmo de búsqueda se basa en el recorrido en profundidad para encontrar todos los puntos de articulación. Los sucesivos vértices por los que se pasa en el recorrido en profundidad de un grafo se puede representar mediante un árbol de expansión. La raíz del árbol es el vértice de partida, `A` y cada arco del grafo será una arista en el árbol.

Se aprovecha el recorrido para encontrar aristas del grafo *hacia adelante* y aristas *hacia atrás*. Así, si en el recorrido por los vértices adyacentes de `v`, arcos `(v,u)`, el vértice `u` no es visitado, entonces `(v,u)` es un arco *hacia delante* y pasa a ser una arista del árbol. Si el vértice `u` es ya visitado pero no se han terminado de procesar todos sus adyacentes, entonces `(v,u)` es un arco *hacia atrás*, no será una arista verdadera del árbol, y por ello se dibuja con una línea discontinua.

La Figura 15.19 muestra un grafo no dirigido y el árbol al que da lugar el recorrido en profundidad a partir del vértice `A`. El recorrido empieza visitando los vértices `B, C, D` que forman las correspondientes aristas del árbol, Figura 15.19*b*. Los vértices adyacentes de `D` no procesados son `E, F` y `A` del que no se ha procesado todos sus adyacentes (el vértice `D`). Los arcos `(D,E),(D,F)`forman las correspondientes aristas del árbol; el arco `(D,A)`es considerado *hacia atrás,* ya que, `A`, fue marcado anteriormente. En el árbol de la Figura 15.19*b* se señala la arista `D - A` con línea discontinua.

http://descargavariable.blogspot.com/

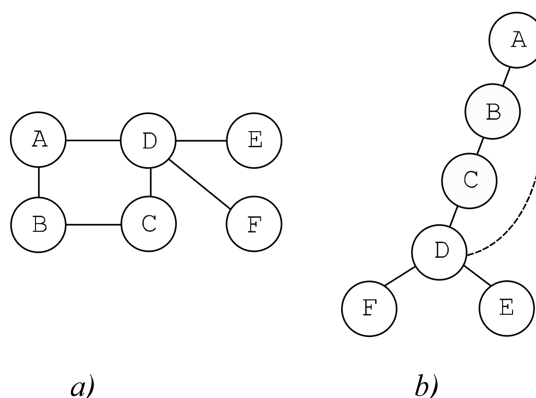

Figura 15.19 *a)* Grafo no dirigido; *b)* árbol de expansión

En el recorrido para formar el árbol se numeran los nodos del árbol, y así se obtiene el orden en que han sido visitados. El algoritmo para encontrar los puntos de articulación de un grafo conexo sigue estos pasos:

1. Recorrer el grafo en profundidad a partir de cualquier vértice. Se numeran en el orden en que son visitados los vértices y esta numeración queda asociada a cada vértice con `num(v)`.
2. Para cada nodo `v` del árbol obtenido en el recorrido se determina el vértice de numeración *más baja*, denominado `bajo(v)`, que es alcanzable desde `v` a través de cero o más aristas del árbol y como mucho una arista *hacia atrás*.
3. Una vez se tienen los valores `num(v)` y `bajo(v)`, se determinan los puntos de articulación aplicando estas reglas:
   - La raíz del árbol (vértice de partida) es punto de articulación *si y sólo* si tiene dos o más hijos.
   - Cualquier otro vértice `w` es punto de articulación *si y sólo si* `w` tiene al menos un *hijo* `u` tal que `bajo(u) ≥ num(w)`.
4. Fin del algoritmo.

En el paso 2 se introduce el concepto de numeración *más baja*, `bajo(v)`, magnitud asociada a cada vértice. Su cálculo se expresa matemáticamente como el mínimo de los siguientes tres valores:

a) `num(v)`.

b) El menor valor de `bajo()` para los vértices `a` de las aristas *hacia atrás* `(v,a)` del árbol.

c) El menor valor de `bajo()` para los vértices `w` de las aristas `(v,w)` del árbol.

En el Ejemplo 15.7 se obtienen las *numeraciones*, `num()` y `bajo()`, de los vértices del grafo no dirigido de la Figura 15.20. Aplicando las reglas descritas en el paso 3 del algoritmo, tiene dos puntos de articulación, el vértice `A` y el `C`.

Ejemplo 15.7

Dado el grafo de la Figura 15.20a encontrar las numeraciones de cada uno de sus vértices y comprobar que coinciden con las señaladas en 15.20b.

Los vértices se numeran como `0, 1, ... 5` correspondiéndose con `A .. F`. El recorrido empieza en `A` (vértice `0`), su numeración es `num(0) = bajo(0) = 1`. Los vértices que siguen en el recorrido, `D` y `C`, tienen como numeración `num(3) = bajo(3) = 2` y `num(2) = bajo(2)`

http://descargavariable.blogspot.com/

`= 3`. Los siguientes vértices por los que se pasa en el recorrido son, `F, E` y `B`; tienen, respectivamente, `num(5) = 4, num(4) = 5, num(1) = 6`. El nodo `B`, Figura 15.20*b* tiene una arista *hacia atrás* cuyo destino es `C`, por consiguiente se puede calcular su magnitud `bajo()` como `mínimo(num(1),num(3))= 3`. Entonces, `bajo(1) = 3;` como el vértice `E` tiene de descendiente a `B`, también `bajo(4) = 3` y lo mismo con el vértice `C`.

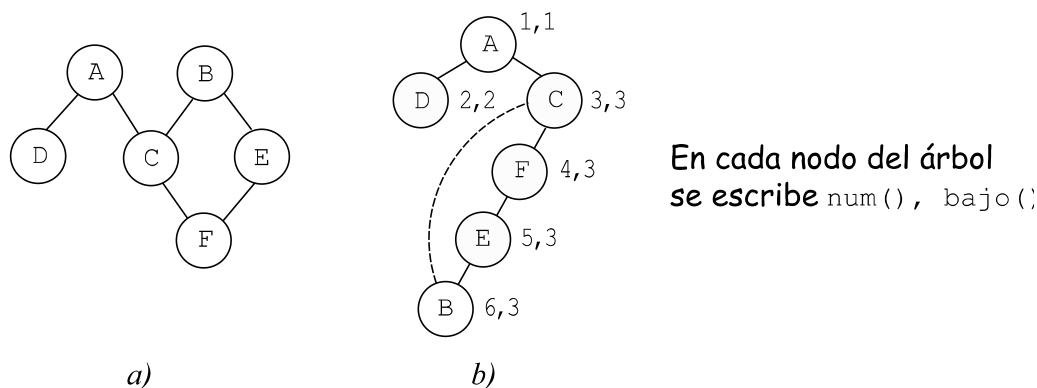

Figura 15.20 *a)* Grafo no dirigido; *b)* árbol de expansión con *numeraciones*

15.7.2. Implementación

En esta ocasión se implementa el algoritmo recursivamente. La forma de asignar los correspondientes valores de `num(v)` a cada vértice del grafo, consiste en incrementar, en cada llamada recursiva, a la función de recorrido el contador de llamadas, que es la magnitud `num(v)` para el vértice *actual*. A la vez, en el *array* `arista[]` se guarda `v` en la posición que se corresponde con el siguiente vértice adyacente, `w`, por el que seguirá el recorrido: `arista[w] = v`; los vértices `(v,w)` forman una arista del árbol de expansión.

Con `arista[]` se determinan las aristas hacia atrás aplicando este razonamiento: *si al analizar los vértices adyacentes de v resulta que "si w está ya visitado y arista[v] != w" es que (v,w) es una arista hacia atrás.*

El método recibe el grafo, no dirigido y conexo, con su matriz de adyacencia. La implementación utiliza el *array* `visitado[]`, para marcar un vértice cuando se *pase* por él (se *visita*).

```
static void puntosArticulacion(
   GrafoMatriz g, int v, int []num, int paso,
   boolean [] visitado, int [] arista, int []bajo) throws Exception
{
   visitado[v] = true;
   num[v] = ++paso;
   bajo[v] = num[v];        // valor inicial para cálculo de bajo()
   for (int w = 0; w < g.numeroDeVertices(); w++)
   {
      if (g.adyacente(v,w))  // adyacente w
      {
         if (!visitado[w])
         {
            arista[w] = v;  // arista del árbol de expansión
            puntosArticulacion(g, w, num, paso, visitado,
                                          arista, bajo);
            if (bajo[w] >= num[v]) // v cumple la regla 3
```

http://descargavariable.blogspot.com/

```
              System.out.println("Vértice " + v +
                                         " es punto de articulación");
            bajo[v] = Math.min(bajo[v], bajo[w]);
         }
         else if (arista[v]!= w)                   // arco hacia atrás
            bajo[v] = Math.min(bajo[v],num[w]);
      }
   }
}
```

Los *arrays* deben de crearse antes de llamar al método; además, el *array* `visitado` se inicializa a `false`:

```
int [] num = new int [g.numeroDeVertices()];
int [] bajo = new int [g.numeroDeVertices()];
int [] arista = new int [g.numeroDeVertices()];
boolean [] visitado = new boolean[g.numeroDeVertices()];

for (i = 0; i < g.numeroDeVertices(); i++)
   visitado[i] = false;
```

RESUMEN

Un grafo `G` consta de un par de conjuntos `(G = (V,E))`: conjunto `V` de vértices o nodos y conjunto `E` de *aristas* (que conectan dos vértices). Si las parejas de vértices que forman una arista no están ordenadas, `G` se denomina *grafo no dirigido;* si los pares están ordenados, entonces `G` se denomina grafo *dirigido*. El término *grafo dirigido* se suele también denominar como *dígrafo* y el término grafo sin calificación significa *grafo no dirigido*.

El método natural para dibujar un grafo es representar los vértices como puntos o círculos y las aristas como segmentos de líneas o arcos que conectan los vértices. Si el grafo está *dirigido*, entonces los segmentos de línea o arcos tienen puntas de flecha que indican la dirección.

Los grafos se implementan de dos formas: *matriz de adyacencia* y *listas de adyacencia*. Cada una tiene sus ventajas y desventajas relativas. La elección depende, entre otros factores, de la densidad del grafo; para grafos *densos*, con muchos arcos, se recomienda representarlos con la *matriz de adyacencia*. Los grafos poco densos y que experimenten modificaciones en sus componentes se representan con *listas de adyacencia*.

Dos vértices de un grafo *no dirigido* se llaman adyacentes si existe una *arista* desde el primero al segundo. Un *camino* es una secuencia de vértices distintos, cada uno adyacente al siguiente. Un *ciclo* es un *camino* que contiene al menos tres vértices, de tal modo que el último vértice en el camino es *adyacente* al primero. Un grafo se denomina *conectado* si existe un camino desde un vértice a cualquier otro vértice.

Un grafo *dirigido* se denomina *conectado fuertemente* si hay un *camino* dirigido desde un vértice a cualquier otro. Si se suprime la dirección de los arcos y el grafo *no dirigido* resultante se conecta, se denomina grafo dirigido *débilmente conectado*.

El *recorrido* de un grafo *visita* de cada uno de sus vértices, puede ser, en analogía con los árboles, *recorrido en profundidad* y *recorrido en anchura*. El *recorrido en profundidad* es una generalización del recorrido en *preorden* de un árbol. El *recorrido en anchura* visita los vértices por *niveles*, al igual que el recorrido por anchura de un árbol y permite encontrar el número mínimo de aristas para alcanzar un vértice cualquiera desde un vértice de partida.

Los *puntos de articulación* de un grafo *conexo* son aquellos vértices que si se retiran del grafo, junto a sus aristas, dividen a éste en dos *componentes conexas*. El algoritmo que permite encontrar los *puntos de articulación*, construye el *árbol de expansión* del grafo a partir de un vértice origen; recorre en profundidad el grafo buscando *aristas de árbol* y *aristas hacia atrás*.

http://descargavariable.blogspot.com/

EJERCICIOS

15.1. Dado el grafo no dirigido, `G` (Figura 15.21).

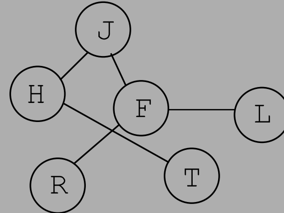

Figura 15.21 Grafo no dirigido `G`

a) Describir `G` formalmente en términos del conjunto de nodos, `V`, y del conjunto `A` de arcos.
b) Encontrar el grado de cada nodo.

15.2. Dado el grafo dirigido, `G` (Figura 15.22).

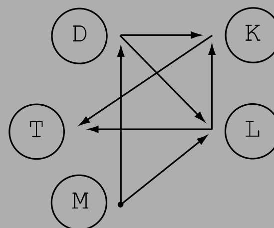

Figura 15.22 Grafo dirigido `G`

a) Describir `G` formalmente en términos de su conjunto de nodos, `V`, y de su conjunto `A` de arcos.
b) Encontrar el grado de entrada y el grado de salida de cada vértice.
c) Escribir la secuencia de vértices que forman los caminos simples del vértice `M` al vértice `T`.

15.3. Dado el grafo no dirigido, `G` (Figura 15.23).

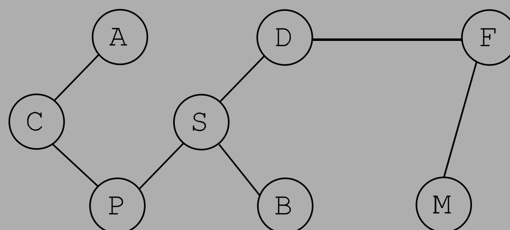

Figura 15.23 Grafo no dirigido `G`

a) Escribir la secuencia de vértices que forman los caminos simples del vértice `A` al vértice `F`.
b) Encontrar el camino más corto (en cuanto a número de aristas) del vértice `C` al vértice `D`.
c) ¿Es un grafo conexo?

http://descargavariable.blogspot.com/

15.4. Dado el grafo dirigido y valorado, `G` (Figura 15.24).

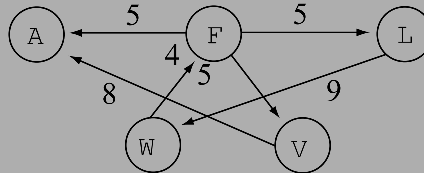

Figura 15.24 Grafo valorado

a) Escribir la matriz de pesos del grafo.
b) Representar el grafo mediante listas de adyacencia.

15.5. Un grafo está formado por los vértices `V = {A, B, C, D, E}`, su matriz de adyacencia, suponiendo los vértices numerados del 0 al 4 respectivamente:

$$M = \begin{vmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \end{vmatrix}$$

a) Dibujar el grafo que le corresponde.
b) Representar el grafo mediante listas de adyacencia.

15.6. Un grafo no dirigido conexo tiene la propiedad de ser biconexo si no hay ningún vértice que al suprimirlo del grafo haga que este se convierta en no conexo.

a) Dibujar un grafo de 6 nodos biconexo.
b) Determinar si los grafos de la Figura 15.21 y 15.22 son biconexos.

15.7. Dado el grafo no dirigido del ejercicio 15.5 realizar el recorrido en profundidad a partir del vértice `C`.

15.8. Dado el grafo no dirigido del ejercicio 15.5 realizar el recorrido en anchura a partir del vértice `C` y la longitud de los caminos mínimos a los demás vértices.

15.9. En la Figura 15.25 se representan dos grafos dirigidos. Teniendo en cuenta que un grafo dirigido se considera *acíclico* si no tiene ciclos, también se denominan **gda**, indicar si los grafos de la figura son **gda**, en el caso de no ser **gda**, buscar los ciclos.

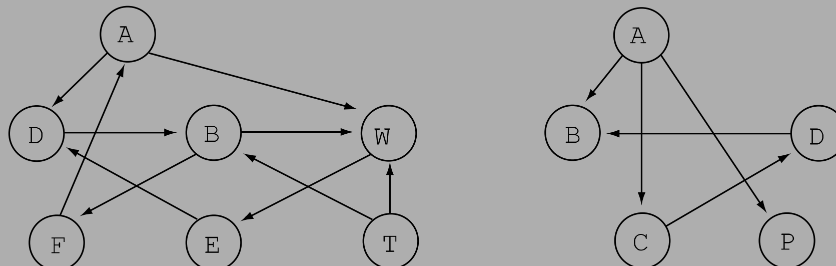

Figura 15.25 Grafos dirigidos

http://descargavariable.blogspot.com/

15.10. Dado el grafo de la Figura 15.26, encontrar la componentes fuertemente conexas.

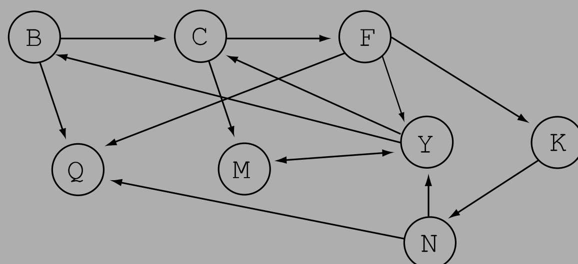

Figura 15.26 Grafo dirigido

15.11. Dado el grafo `G` de la Figura 15.27:

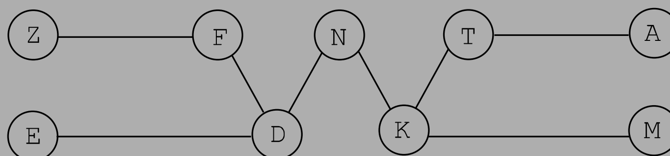

Figura 15.27 Grafo no dirigido

a) Escribir la matriz de adyacencia del grafo.
b) Escribir la matriz de caminos de `G`.

15.12. Dado un grafo dirigido en el que los vértices son números enteros positivos y el par `(x,y)` es una arista en el caso de que `x-y` sea múltiplo de `3`.

a) Representar el grafo formado por los vértices 3 al 14.
b) Determinar el grado de entrada y el grado de salida de cada vértice.

15.13. En los grafos no dirigidos de las figuras 15.23 y 15.27:

a) Dibujar los correspondientes árboles de expansión. Asociar a cada nodo su *numeración* `num()` y `bajo()`.
b) Encontrar los puntos de articulación.

15.14. Dibujar un grafo dirigido en el que los vértices son números enteros positivos del 3 al 15 para cada una de las siguientes relaciones:

a) `v` es adyacente de `w` si `v+2w` es divisible entre 3.
b) `v` es adyacente de `w` si `10v+w < v*w`.

PROBLEMAS

15.1. Diseñar un grafo valorado con los siguientes requisitos:

- Consta de 10 vértices que son números aleatorios del `11` al `99`.
- Dos vértices `v, w,` están relacionados si `v+w` es múltiplo de 3.

a) Escribir un programa en el que se genere un grafo con las condiciones descritas. El grafo se ha de representar con una matriz de adyacencia.
b) Añadir al programa los métodos necesarias para determinar la matriz de caminos utilizando las potencias de la matriz de adyacencia.

http://descargavariable.blogspot.com/

15.2. Un grafo valorado está formado por los vértices `4, 7, 14, 19, 21, 25`. Las aristas siempre van de un vértice de mayor valor numérico a otro de menor valor, y el peso es el módulo del vértice origen y el vértice destino.

a) Escribir un programa que represente el grafo con listas de adyacencia.
b) Añadir al programa el código necesario para realizar un recorrido en anchura desde un vértice dado.

15.3. Una región está formada por 12 comunidades. Se establece la relación de desplazamiento de personas en las primeras horas del día. Así, la comunidad `A` está relacionada con la comunidad `B` si desde `A` se desplazan `n` personas a `B`, de igual forma puede haber relación entre `B` y `A` si se desplazan `m` personas de `B` hasta `A`.

a) Escribir un programa que represente el grafo descrito mediante listas de adyacencia.
b) Determinar si el grafo formado tiene fuentes o sumideros.

15.4. Dado el grafo descrito en el Problema 15.3. Escriba un programa para representarlo mediante listas enlazadas de tal forma que cada nodo de la lista directorio contenga dos listas: una que contiene los arcos que salen del nodo, y otra que contiene los arcos que terminan en el nodo.

15.5. Un grafo en el que los vértices son regiones y los arcos relacionan dos regiones entre las cuales hay un flujo de emigrantes (tiene factor de peso), está representado mediante una lista directorio que contiene a cada uno de los vértices y de las que sale una lista circular con los vértices adyacentes. Ahora se quiere representar el grafo mediante una matriz de pesos, de tal forma que si entre dos vértices no hay arco su posición en la matriz tiene `0`, y si entre dos vértices hay arco su posición contiene el factor de peso que le corresponde.
Escribir las funciones necesarias de modo que partiendo de la representación mediante listas se obtenga la representación mediante la matriz de pesos.

15.6. Representar la información sobre un Centro de enseñanza que se enumera a continuación:

- Nombre del centro, Ubicación, Nombre del Director.
- Alumnos divididos en clases de:
  - Enseñanza Primaria: *n* grupos de un máximo de 25 alumnos.
  - Enseñanza Secundaria: *m* grupos de un máximo de 30 alumnos.
  - Bachillerato: *b* grupos con un máximo de 40 alumnos.
- Profesores de:
  - Enseñanza Primaria se asigna un profesor a cada clase.
  - Enseñanza Secundaria y Bachillerato, hay un profesor por cada asignatura del curso.
- Asignaturas: en cada curso hay un máximo de *max* asignaturas.

Se pide:

1. Lectura de los alumnos de cada uno de los grupos.
2. Lectura del profesorado de cada uno de los grupos.
3. Lectura de las notas de una clase.
4. Clasificación del alumnado de dicha clase en alumnos aprobados, suspensos y de muy bajo rendimiento (3 asignaturas suspensas o más).
5. Ordenación alfabética de una clase.
6. Ordenación alfabética del alumnado del colegio.

Nota: Evitar la presencia de información redundante. Hacer uso estructuras de datos dinámicas.

http://descargavariable.blogspot.com/

15.7. Durante el recorrido en profundidad de un grafo dirigido, cuando se recorren ciertos arcos, se llega a vértices aún sin visitar. Los arcos que llevan a vértices nuevos se conocen como arcos de árbol y forman un bosque abarcador en profundidad para el grafo dirigido dado. Además de los arcos del árbol, existen otros tipos de arcos diferentes que se llaman arcos de retroceso, arcos de avance y arcos cruzados:

- Un arco se dice que es de retroceso si va de un nodo del árbol a otro que es su predecesor.
- Un arco se dice que es de avance si va de un nodo del árbol a otro nodo del árbol ya construido, pero que es un descendiente de él.
- Un arco se dice que es cruzado si va de un nodo del árbol a otro que no está relacionado por la relación jerárquica definida en el árbol.

Escriba un programa que lea un grafo, calcule el bosque abarcador y los arcos de avance, retroceso y cruzado

15.8. Se dispone de un grafo no dirigido poco denso. Se elige la representación en memoria mediante listas de adyacencia. Escribir un programa en el que se de entrada a los vértices del grafo y sus arcos y se determine si el grafo tiene ciclos. En caso positivo, listar los vértices que forman un ciclo.

15.9. Supóngase un grafo no dirigido y conexo, escribir un programa que encuentre un camino que vaya a través de todas las aristas exactamente una vez en cada dirección.

15.10. Se denominan caminos hamiltonianos a aquéllos que contienen exactamente una vez a todos y cada uno de los vértices del grafo. Se trata de escribir un programa que lea un grafo e imprima todos sus caminos hamiltonianos.

15.11. Escriba un programa que compruebe si un grafo dirigido, leído del teclado, tiene circuitos (ciclos) mediante el siguiente algoritmo:

1. Obtener los sucesores de todos los vértices.
2. Buscar un vértice sin sucesores y *tachar* (eliminar) ese vértice donde aparezca (conjuntos de sucesores).
3. Se continúa el proceso en el paso 2 siempre que haya algún vértice sin sucesores.
4. Si todos los vértices del grafo han sido eliminados, no tienen circuitos.

http://descargavariable.blogspot.com/