Tema 6. Grafos

Estructura de Datos y Algoritmos

Contenidos

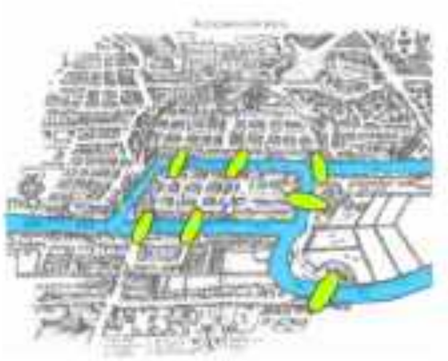
- **¿Qué es un grafo?.**
- TAD Grafo.
- Implementaciones
  - Matriz de adyacencias.
  - Lista de adyacencias
- Recorridos
  - En profundidad
  - En altura

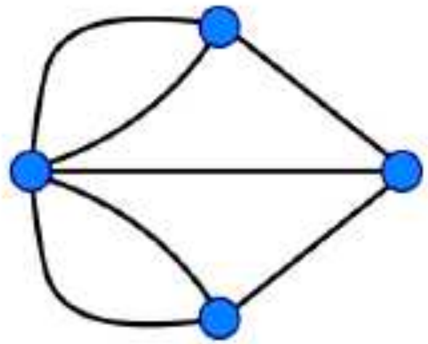
¿Qué es un grafo?

- "A Graph is a way of representing relationships that exist between pairs of objects." - Goodrich

Los orígenes de la Teoría de Grafos

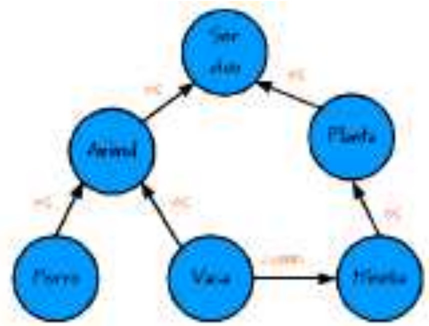
- El problema de los siete puentes de Könisberg (Euler)
  - ¿es posible dar un paseo comenzando desde cualquiera de estas regiones, pasando por todos los puentes, recorriendo sólo una vez cada uno, y regresando al mismo punto de partida?



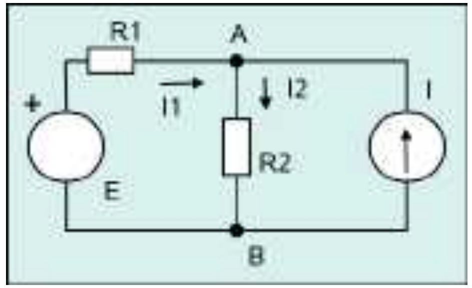


Aplicaciones

- Otras aplicaciones



Grafos conceptuales



Circuitos electrónicos

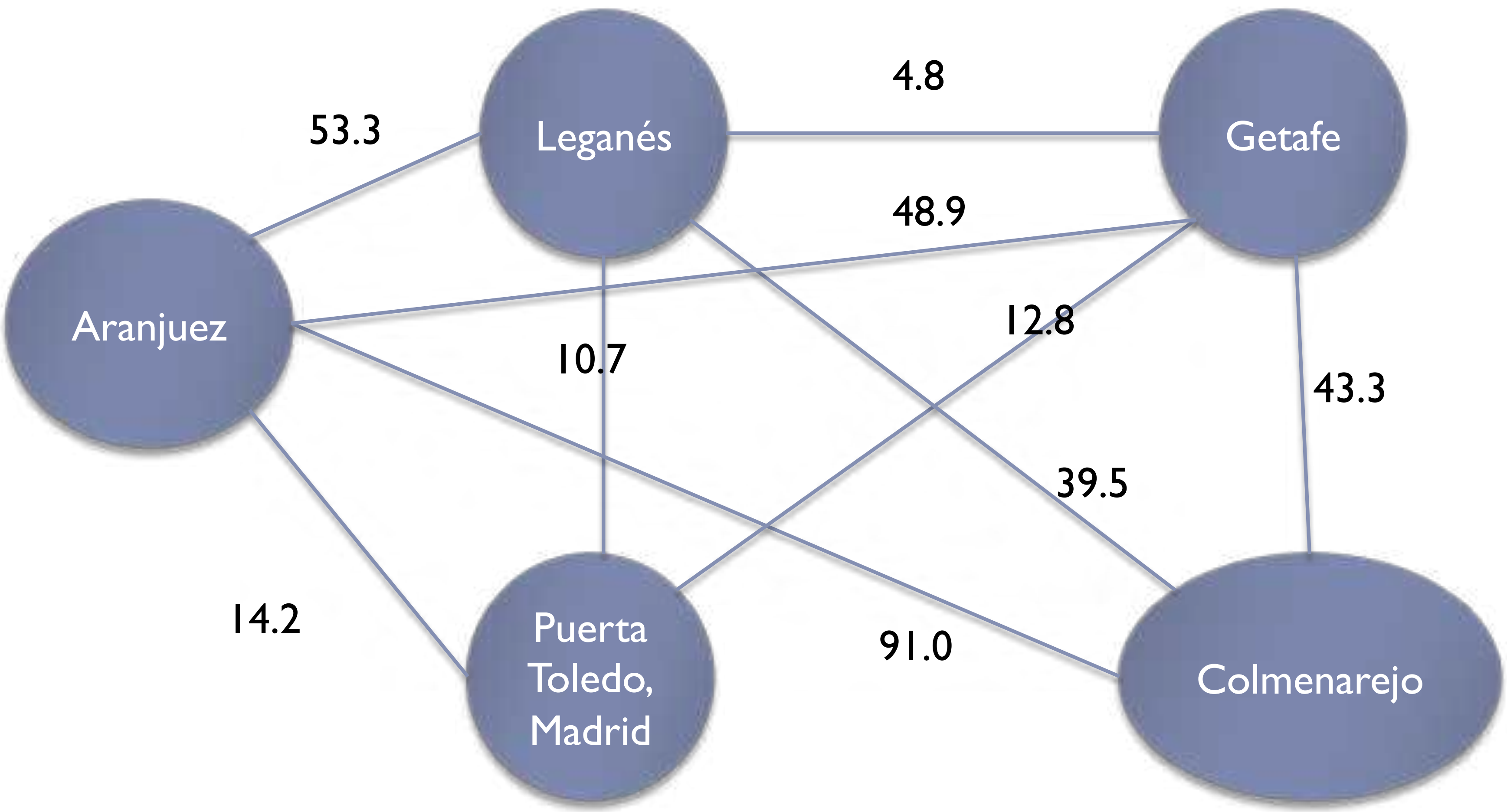
Redes de ordenadores

Organigramas

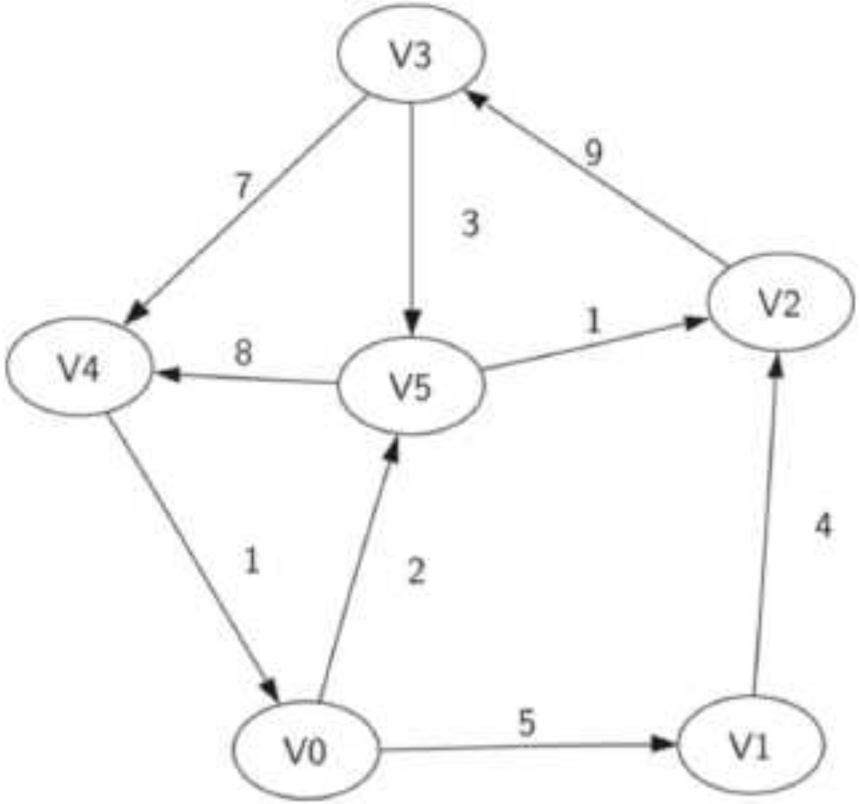
Conceptos Básicos

- Grafo *G* donde *G*=(*V*,*A*).
  - V es un conjunto de vértices (nodos)
  - A es un conjunto de aristas (arcos).
    - Una arista es una conexión entre dos vértices.
    - Cada arista puede ser representada como una tupla (v,w) donde $w,v \in V$
    - Además, cada artista puede tener un peso asociado (**grafo ponderado**). En este caso, la arista quedaría representada por una terna (v,w,p) donde p es el peso asociado a la arista entre v y w.

Ejemplo I – Campus UC3M



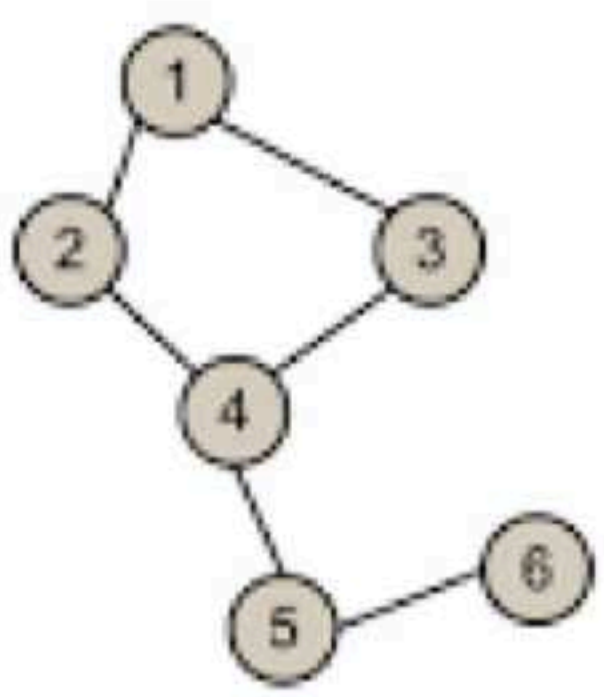
Ejemplo II



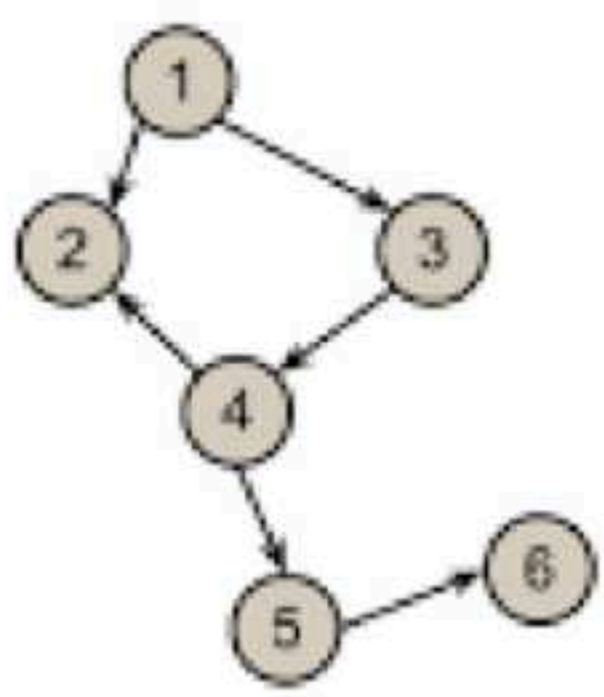
$$V = \{V0, V1, V2, V3, V4, V5\}$$

$$A = \left\{ \begin{array}{l} (v0, v1, 5), (v1, v2, 4), (v2, v3, 9), (v3, v4, 7), (v4, v0, 1), \\ (v0, v5, 2), (v5, v4, 8), (v3, v5, 3), (v5, v2, 1) \end{array} \right\}$$

Grafo No Dirigido vs Dirigido



No dirigido

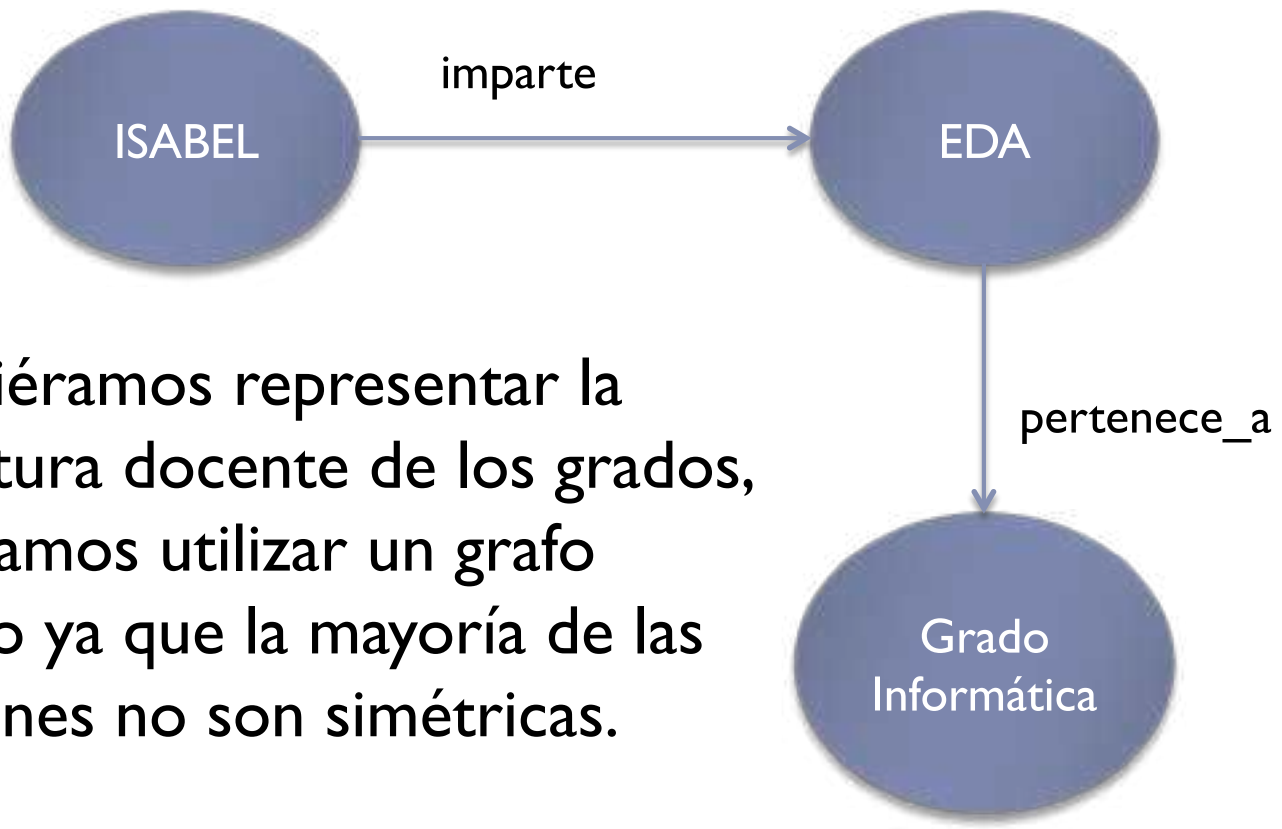


dirigido

Tipos de Grafos

- **Grafos no dirigidos.**
  - Las aristas no tiene dirección, es decir, (u,v)=(v,u). La arista se puede recorrer en ambos sentidos.
  - Nos permiten representar relaciones simétricas y de colaboración.
  - Ejemplo Grafo Campus UC3M.
- **Grafos dirigidos.**
  - Cada arista (u,v) tiene una única dirección, siendo u el origen y v el vértice final. $(u,v) \neq (v,u)$
  - Nos permiten representar relaciones asimétricas y jerárquicas.

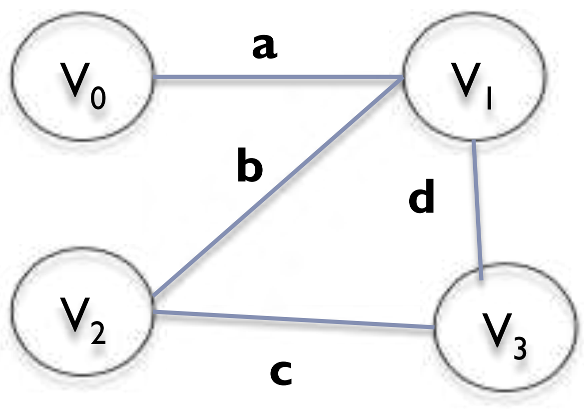
Ejemplo I Grafo Dirigido



Si quisiéramos representar la estructura docente de los grados, deberíamos utilizar un grafo dirigido ya que la mayoría de las relaciones no son simétricas.

Conceptos Básicos

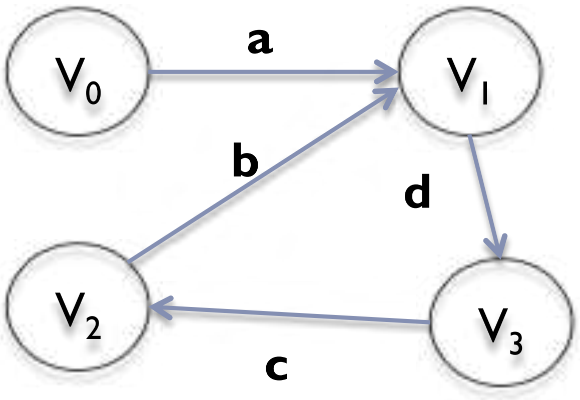
- Dos vértices son **adyacentes** si existe una arista que los conecta.
- Una arista es **incidente** a un vértice si lo une con otro vértice.
- El **grado** de un vértice v, **deg(v)**, es el número de aristas conectadas a v.



- Vértices adyacentes: (V_0,V_1), (V_1,V_2), (V_2,V_3), (V_1,V_3).
- Grado de $V_1 = 3$.
- Las aristas a, b y d son incidentes en V_1

Conceptos Básicos

- En los grafos dirigidos, podemos distinguir entre:
- ***Grado de entrada*** *de un vértice v*, **indeg(v)**, es el número de aristas que llegan al vértice v.
- **Grado de salida** de un vértice v, **outdeg(v)**, es el número de aristas que parten del vértice v.



- $indeg(V_0)=0$, $indeg(V_1)=2$, $indeg(V_2)=1$, $indeg(V_3)=1$
- $outdeg(V_0)=1$, $outdeg(V_1)=1$, $outdeg(V_2)=1$, $outdeg(V_3)=1$

Conceptos Básicos

- Proposición: Si G es un grafo con m aristas:

$$\sum_{v \in G} \deg(v) = 2m$$

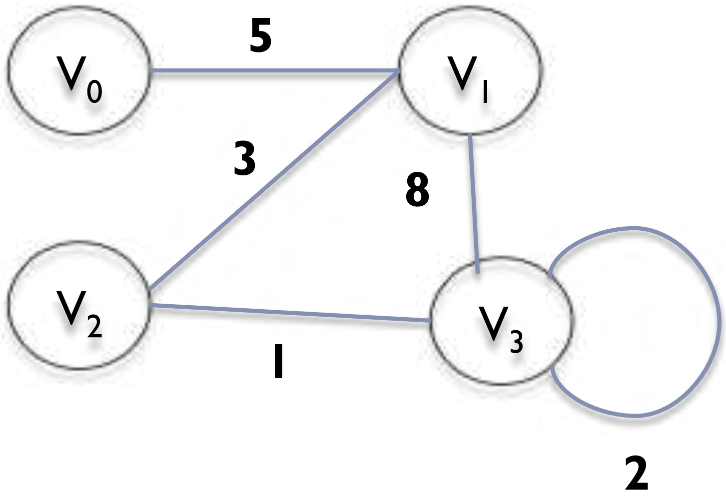
- Proposición: Si G es un grafo dirigido con m aristas:

$$\sum_{v \in G} in\deg(v) = \sum_{v \in G} out\deg(v) = m$$

Conceptos Básicos

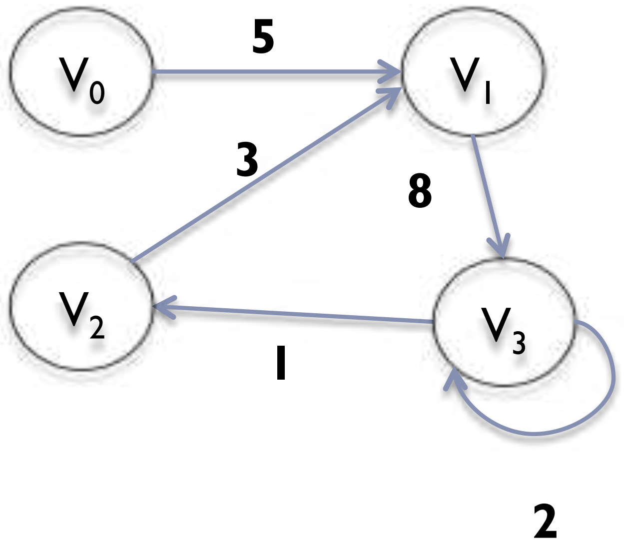
- ***Camino***: una secuencia de vértices conectados por aristas.
  - La longitud del camino sin pesos es el número de aristas en el camino.
  - La longitud del camino con pesos es la suma de los pesos de todas las aristas del camino.
- ***Ciclo***: un ciclo en un grafo dirigido es una camino que empieza y termina en el mismo nodo.

Ejemplo camino y ciclo (grafo no dirigido)



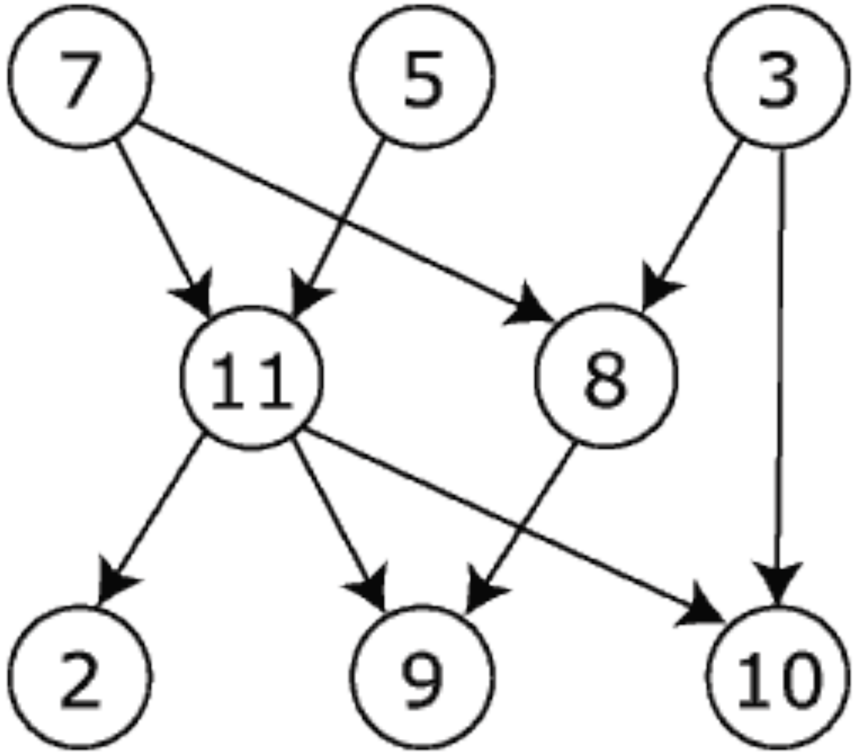
- <V_0,V_1,V_2,V_3> camino simple (no se repiten nodos) de longitud 5+3+1=9
- <V_0,V_1,V_2,V_3,V_1> camino de longitud 5+3+1+8=17
- <V_0,V_3> no es un camino
- <V_1,V_2,V_3,V_1> camino y ciclo
- <V_3,V_3> bucle

Ejemplo camino y ciclo (grafo dirigido)



- $<V_0,V_1,V_2,V_3>$ no es un camino.
- $<V_0,V_1,V_3,V_{,}>$ camino simple de longitud 5 + 8 + 1 = 14
- $<V_0,V_1,V_3,V_2,V_1>$ camino de longitud 5+8+1+3=17
- $<V_1,V_3,V_2,V_1>$ camino y ciclo
- $<V_3,V_3>$ bucle

Grafo Acíclico



- Un grafo sin ciclos es un **grafo acíclico**.
- Un grafo dirigido sin ciclos se llama **grafo dirigido acíclico (DAG)**.

Contenidos

- ¿Qué es un grafo?
- **TAD Grafo.**
- Implementaciones
  - Matriz de adyacencias.
  - Lista de adyacencias
- Recorridos
  - En profundidad
  - En altura

TAD Grafo

```java
//Interface for a graph whose vertices are integers
//and its weights are floats

public interface IGraph {

    //return the number of vertices
    public int sizeVertices();
    //return the number of edges
    public int sizeEdges();
    //shows the graph (vertices and edges)
    public void show();

    //returns the degree a vertex
    public int getDegree(int i);
    //return the inward-bound degree
    public int getInDegree(int i);
    //return the outward-bound degree
    public int getOutDegree(int i);
```

TAD Grafo (cont.)

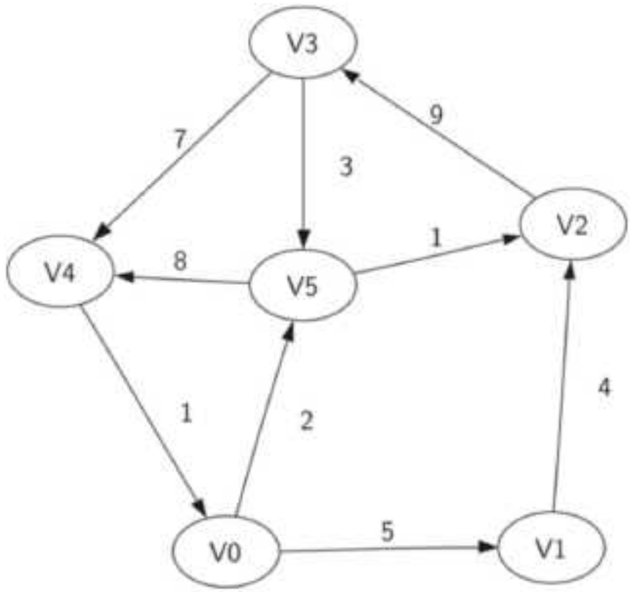
```
//create a new vertex
public void addVertex();
//create an edge between the vertices i and j
public void addEdge(int i, int j);
//create an edge between the vertices i and j with weight w
public void addEdge(int i, int j, float w);
//remove the edge between the vertices i and j
public void removeEdge(int i, int j);

//check if the pair of vertices (i,j) is an edge.
public boolean isEdge(int i, int j);
//returns the weight of the edge (i,j)
public Float getWeightEdge(int i, int j);
//returns an array with the adjacent vertices of i
public int[] getAdjacents(int i);
```

Contenidos

- ¿Qué es un grafo?
- TAD Grafo.
- **Implementaciones**
  - **Matriz de adyacencias.**
  - Lista de adyacencias
- Recorridos
  - En profundidad
  - En altura

Implementación basada en matriz

- **La matriz de adyacencias**

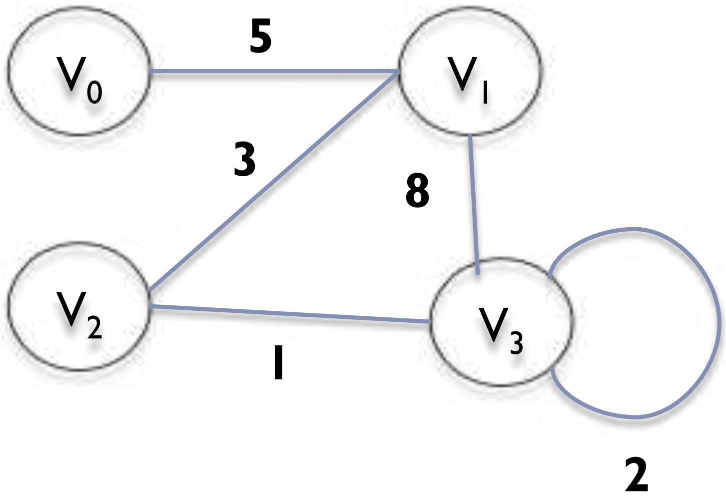


	V0	V1	V2	V3	V4	V5
V0		5				2
V1			4			
V2				9		
V3					7	3
V4	1					
V5			1		8	

- Picture taken from http://interactivepython.org/runestone/static/pythonds/Graphs/AnAdjacencyMatrix.html

Implementación – Matriz de adyacencias

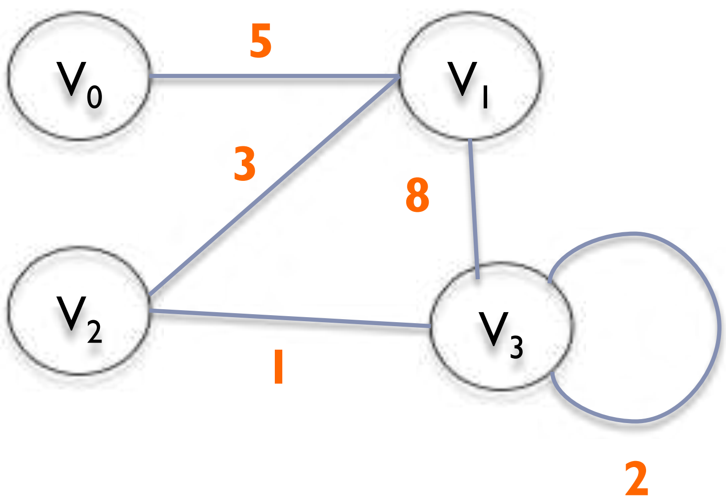
- Un grafo puede ser representado como una matriz cuadrada nxn, siendo n el número de vértices del grafo.
- Cada vértice v es representado por un entero (índice de v), en el rango {0,1,..,n-1} siendo n el número de vértices



- V_0 -> índice 0
- V_1 -> índice 1
- V_2 -> índice 2
- V_3 -> índice 3

Implementación – Matriz de adyacencias

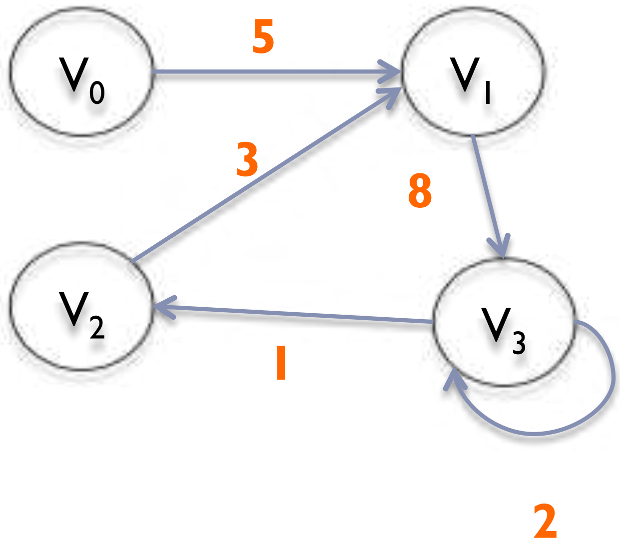
- La matriz se puede implementar como un array bidimensional n x n M, tal que el elemento **M[i,j]** guarda información sobre la arista (v,w), si existe, donde v es el vértice con índice i y w es el vértice con índice j.



	0	1	2	3
0		5		
1	5		3	8
2		3		1
3		8	1	2

Si el grafo no es dirigido la matriz es simétrica

Implementación – Matriz de adyacencias

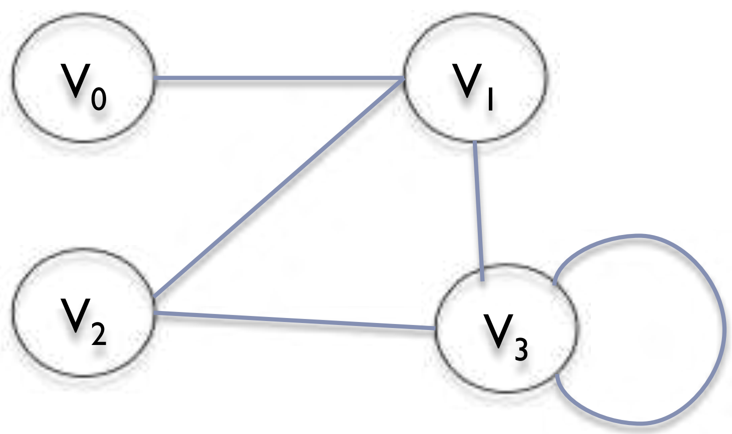


Si el grafo es dirigido la matriz NO es simétrica

	0	1	2	3
0		5		
1				8
2		3		
3			1	2

Implementación – Matriz de adyacencias

- Si es un grafo no etiquetado, el grafo se podría representar con una matriz de booleanos,



	0	1	2	3
0	false	true	false	false
1	true	false	true	true
2	false	true	false	true
3	false	true	true	true

Si el grafo no es dirigido la matriz es simétrica

Implementación – Matriz de adyacencias

- Vamos a ver una implementación para un grafo no ponderado.
- La matriz puede ser almacenada en un array bidimensional de booleanos (true indicará que existe arista y false que no existe).
- La creación de nuevos vértices podría implicar la necesidad de modificar el tamaño asignado a la matriz.
- Para evitarlo, vamos a definir un atributo que almacene el **número máximo de vértices** (en ningún caso, se permitirá añadir un nuevo vértice cuando ese umbral se haya alcanzando) y otro atributo que almacene el número actual de vértices.

Implementación – Matriz de adyacencias

```java
public class GraphMA implements IGraph {

    boolean matrix[][];
    //maximum number of vertices
    int maxVertices;
    //current number of vertices
    int numVertices;
    //true if the graph is directed, false eoc
    boolean directed;
```

Implementación – Matriz de adyacencias

```java
public GraphMAFull(int n, int max, boolean d) {
    //We checks if the values are right for the graph
    if (max<=0)
        throw new IllegalArgumentException("Negative maximum number of vertices!!!");
    if (n<=0)
        throw new IllegalArgumentException("Negative number of vertices!!!.");
    if (n>max)
        throw new IllegalArgumentException("number of vertices can never be greater than the maximum.");

    maxVertices=max;
    numVertices=n;
    matrix=new Float[maxVertices][maxVertices];
    directed=d;
}
```

- Primero deberemos comprobar que todos los argumentos del constructor reciben valores apropiados: tanto el número máximo como el número de vértices debe ser siempre un número positivo.
- Además, el número de vértices nunca deberá sobrepasar el número máximo de vértices.
- El constructor crea el array bidimensional. Por defecto, todos las posiciones son inicializadas a false.

Implementación – Matriz de adyacencias

```java
public void addVertex() {
    if (numVertices==maxVertices) {
        System.out.println("Cannot add new vertices!!!");
        return;
    }
    numVertices++;
}
```

- Lo primero que tenemos que hacer es comprobar que el nuevo número total de vértices no va a sobrepasar el número máximo permitido.
- Por último, sólo tendremos que incrementar en uno el número actual de vértices.
- No hace falta inicializar la matriz para el nuevo vértice, porque por defecto todas sus posiciones en la matriz son false.

Implementación – Matriz de adyacencias

```
//check if i is a right vertex
private boolean checkVertex(int i) {
    if (i>=0 && i<numVertices) return true;
    else return false;
}
```

- Vamos a usar un método auxiliar para comprobar si un índice representa o no un vértice en el grafo.
- Para que sea un vértice del grafo siempre deberá ser positivo y menor que numVertices, porque los vértices del grafo toman valores en el rango [0, numVertices-1].

Implementación – Matriz de adyacencias

```
public void addEdge(int i, int j) {
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);
    if (!checkVertex(j))
        throw new IllegalArgumentException("Nonexistent vertex  " + j);

    matrix[i][j]=true;
    if (!directed) matrix[j][i]=true;
}
```

Una vez comprobadas que ambas índices son correctos, simplemente lo que tenemos que hacer es actualizar la posición matrix[i,j] a true. Si no es dirigido, también tendremos que poner su posición simétrica

Implementación – Matriz de adyacencias

```java
@Override
public boolean isEdge(int i, int j) {
    //checks if the indexes are right
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);
    if (!checkVertex(j))
        throw new IllegalArgumentException("Nonexistent vertex  " + j);
    return matrix[i][j];
}
```

- En primer lugar, tenemos que comprobar que los índices i y j son correctos.
- El par (i,j) es un arista si matrix[i,j] guarda true.

Implementación – Matriz de adyacencias

```java
@Override
public void removeEdge(int i, int j) {
    //checks if the indexes are right
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);
    if (!checkVertex(j))
        throw new IllegalArgumentException("Nonexistent vertex  " + j);
    matrix[i][j]=false;
    if (!directed) matrix[j][i]=false;

}
```

- Primero tenemos que comprobar que son índices válidos
- Una vez comprobado que son índices válidos, basta con modificar el valor del array en esa posición (i,j) a false.
- Si no es dirigido, también tendremos que hacerlo en su elemento simétrico (j,i)

Implementación – Matriz de adyacencias

```
public int sizeVertices() {
    return numVertices;
}

public int sizeEdges() {
    int numEdges=0;
    if (directed) {
        for (int i=0;i<numVertices;i++) {
            for (int j=0;j<numVertices;j++) {
                if (matrix[i][j]!=false) numEdges++;
            }
        }
    } else {
        for (int i=0;i<numVertices;i++) {
            for (int j=i;j<numVertices;j++) {
                if (matrix[i][j]!=false) numEdges++;
            }
        }
    }
    return numEdges;
}
```

- Equivale a contar todos los elementos true en la matriz.

- Si no es dirigido, como la matriz es simétrica, sólo necesitaremos visitar una de las dos partes divididas por la diagonal.

Implementación – Matriz de adyacencias

```java
public int getOutDegree(int i) {
    if (!directed) {
        System.out.println("Graph non directed!!!");
        return 0;
    }
    //checks if the vertex is right
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);
    int outdeg=0;
    for (int col=0;col<numVertices;col++) {
        if (matrix[i][col]!=false) outdeg++;
    }
    return outdeg;
}
```

Las filas representan los vértices de origen y las columnas los vértices destino

- Incrementamos 1 por cada columna cuyo índice tenga una arista con i, es decir, matrix[i,col].

Implementación – Matriz de adyacencias

```java
public int getInDegree(int i) {
    if (!directed) {
        System.out.println("Graph non directed!!!");
        return 0;
    }
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);
    int indeg=0;
    for (int row=0;row<numVertices;row++) {
        if (matrix[row][i]!=false) indeg++;
    }

    return indeg;
}
```

Las filas representan los vértices de origen y las columnas los vértices destino

- Incrementamos 1 por cada fila cuyo índice tenga una arista con i, es decir, matrix[row,i].

Implementación – Matriz de adyacencias

```
public int getDegree(int i) {
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);
    int degree=0;
    if (directed) degree=getInDegree(i)+getOutDegree(i);
    else {
        for (int row=0;row<numVertices;row++) {
            if (matrix[row][i]!=false) degree++;
        }
    }
    return degree;
}
```

Si el grafo no es dirigido, el grado será la suma del grado de entrada y el grado de salida.
En otro caso, bastará con que contemos las aristas de entrada en ese vértice. También se podría hacer contando las aristas de salida (pero nunca ambas).

Implementación – Matriz de adyacencias

```
//returns an array with the adjacent vertices for i
public int[] getAdjacents(int i) {
    if (!checkVertex(i))
            throw new IllegalArgumentException("Nonexistent vertex  " + i);

    //obtains the number of adjacent vertices,
    //which will be the size of the array
    int numAdjacents=0;
    if (directed) numAdjacents=getOutDegree(i);
    else numAdjacents=getDegree(i);

    int[] adjacents=new int[numAdjacents];

    if (numAdjacents>0) {
        int j=0;
        //gets the edges (i,col) and saves col into adjacents
        for (int col=0; col<numVertices;col++) {
            if (matrix[i][col]!=null) {
                adjacents[j]=col;
                j++;
            }
        }
    }
    //return an array with the adjacent vertices of i
    return adjacents;
}
```

Problema

- ¿Cómo representarías un grafo ponderado con valores reales (es decir, un grafo donde todas las aristas tienen un valor real asociado)?. Implementa una clase para representar grafos ponderados.

Contenidos

- ¿Qué es un grafo?
- TAD Grafo.
- **Implementaciones**
  - Matriz de adyacencias.
  - **Lista de adyacencias**
- Recorridos
  - En profundidad
  - En altura

Implementación – Lista de adyacencias

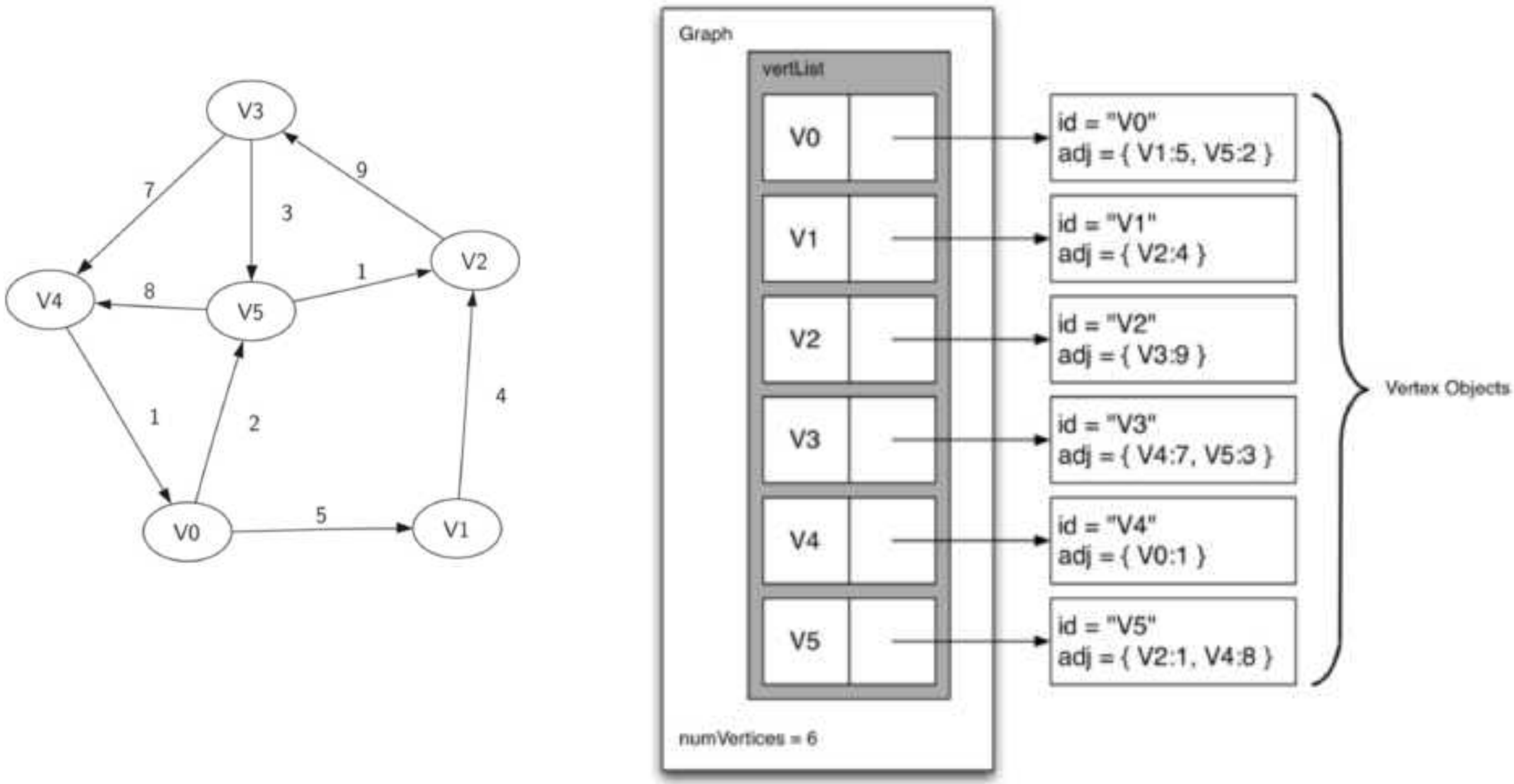
- La matriz de adyacencia consume memoria y la complejidad de las operaciones con la matriz es alta (por ejemplo, el método que muestra la matriz tiene complejidad cuadrática).

- Una lista de adyacencia sólo almacena la información para los aristas existentes, en lugar de almacenar todas las posibles combinaciones como ocurría en la matriz de adyacencias.

- Necesita menos espacio de memoria y su coste computacional es menor.

Implementación – Lista de adyacencias

- Lista adyacencias



Picture taken from http://interactivepython.org/runestone/static/pythonds/Graphs/AnAdjacencyList.html

Implementación – Lista de adyacencias

- Si tenemos un número fijo de vértices, el grafo se puede representar como un array de listas enlazadas (tema 2).
- Cada posición del array representa un vértice y almacena la referencia a la lista de vértices adyacentes a dicho vértice (implementada como lista enlazada).
- Cada uno de los nodos almacenará la información sobre el vértice adyacente. Si el grafo es ponderado, también debería almacenar su peso.

Implementación – Lista de adyacencias

```java
import dlist.DListVertex;

public class GraphLAFull implements IGraph {

    int numVertices;
    int maxVertices;

    DListVertex[] vertices;
    boolean directed;
```

Implementación – Lista de adyacencias

```
public GraphLAFull(int n, int max, boolean d) {
    if (max<=0)
        throw new IllegalArgumentException("Negative maximum number of vertices!!!");
    if (n<=0)
        throw new IllegalArgumentException("Negative number of vertices!!!.");
    if (n>max)
        throw new IllegalArgumentException("number of vertices can never "
                + "be greater than the maximum.");

    maxVertices=max;
    vertices=new DListVertex[maxVertices];
    numVertices=n;
    //creates each list
    for (int i=0; i<numVertices;i++) {
        vertices[i]=new DListVertex();
    }
    directed=d;
}
```

Implementación – Lista de adyacencias

Comprobamos que los índices son correctos.

```
public void addEdge(int i, int j, float w) {
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);
    if (!checkVertex(j))
        throw new IllegalArgumentException("Nonexistent vertex  " + j);

    vertices[i].addLast(j,w);
    //if it is a non-directed graph
    if (!directed) vertices[j].addLast(i,w);
}
```

Tenemos que añadir el vértice j a la lista de vértices adyacentes del vértice i (que está almacenada en vertices[i]).

Si el grafo no es dirigido, deberemos también almacenar el vértice i como adyacente del vértice j

Implementación – Lista de adyacencias

```java
public void removeEdge(int i, int j) {
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);
    if (!checkVertex(j))
        throw new IllegalArgumentException("Nonexistent vertex  " + j);

    int index=vertices[i].getIndexOf(j);
    vertices[i].removeAt(index);

    if (!directed) {
            index=vertices[j].getIndexOf(i);
            vertices[j].removeAt(index);
    }
}
```

Si el grafo no es dirigido, deberemos también borrar el vértice i como adyacente del vértice j

Implementación – Lista de adyacencias

```java
public boolean isEdge(int i, int j) {
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);
    if (!checkVertex(j))
        throw new IllegalArgumentException("Nonexistent vertex  " + j);

    boolean result=vertices[i].contains(j);
    return result;
}
```

Implementación – Lista de adyacencias

```java
public int getOutDegree(int i) {
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);

    int outdegree=0;
    outdegree=vertices[i].getSize();
    return outdegree;
}
```

Sólo para grafos dirigidos

```java
public int getInDegree(int i) {
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);
    int indegree=0;
    for (int j=0; j<numVertices;j++) {
        if (vertices[j].contains(i)) indegree++;
    }
    return indegree;
}
```

Implementación – Lista de adyacencias

```
public int getDegree(int i) {
    int degree=0;
    if (directed) {
        degree=getOutDegree(i)+getInDegree(i);
    } else degree=vertices[i].getSize();
    return degree;
}
```

El grado de un vértice en un grado dirigido es igual a la suma de su grado de entrada y de su grado de salida.
En un grado no dirigido, es suficiente con obtener el número de vértices adjacentes a dicho vértice.

Implementación – Lista de adyacencias

```java
public int[] getAdjacents(int i) {
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);
    //gets the number of adjacent vertices
    int numAdj=vertices[i].getSize();
    //creates the array
    int[] adjVertices=new int[numAdj];
    //saves the adjacent vertices into the array
    for (int j=0; j<numAdj; j++) {
        adjVertices[j]=vertices[i].getVertexAt(j);
    }
    //return the array with the adjacent vertices of i
    return adjVertices;
}
```

Problemas a resolver

- Añade un método que muestre el contenido del grafo. Es decir, por cada vértice, que muestra su lista vértices adyacentes.
- Implementa un método que compruebe si el grafo es dirigido o no. (No puedes usar el atributo directed!!!)

Contenidos

- ¿Qué es un grafo?.
- TAD Grafo.
- Implementaciones
  - Matriz de adyacencias.
  - Lista de adyacencias
- **Recorridos**
  - **En amplitud**
  - En profundidad

Recorridо en amplitud

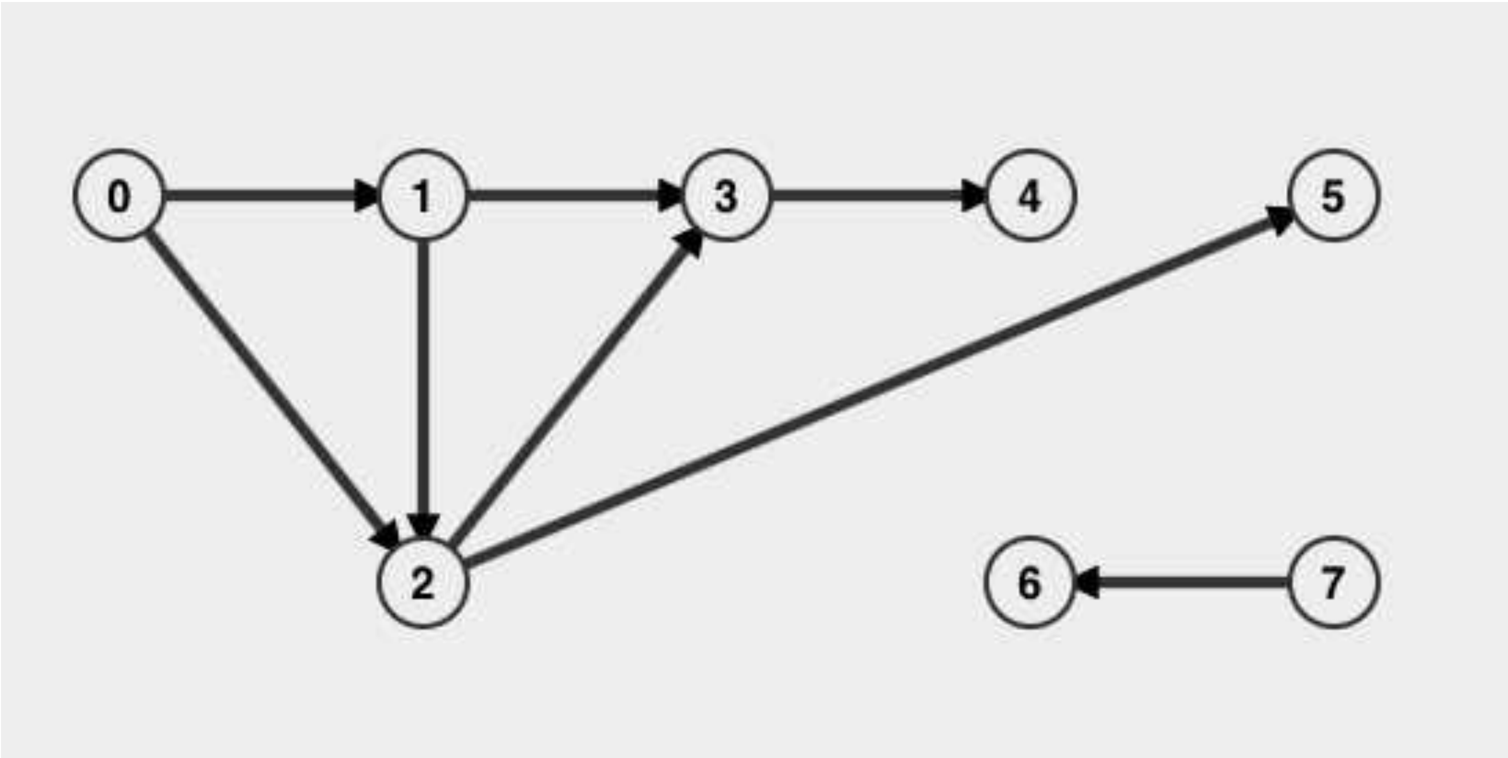
- Recorrido basado en estructura FIFO (first in first out).
- Se toma un vértice como inicial, y se visitan todos sus adyacentes. Una vez visitados éstos, se continúa visitando los adyacentes de cada uno de ellos, hasta que no se pueden alcanzar más vértices.
- Se repite el proceso mientras haya nodos sin visitar.
- Es necesario utilizar una estructura de cola para ir almacenando los vértices a medida que se llega a ellos.
- Ver animación http://visualgo.net/dfsbfs.html (bfs)

Recorrido en amplitud

- Pasos del algoritmo:

1. Se toma un vértice v como inicial y se imprime.
2. Se visitan cada uno de sus nodos adyacentes y se almacenan en la cola.
3. Se desencola el primer elemento de la cola y se marca como visitado.
4. Se repiten los pasos 2 y 3 mientras haya elementos en la cola.
5. Si la cola no está vacía y quedan vértices sin recorrer se debe elegir un vértice no visitado y repetir los puntos 2-3-4.
6. El algoritmo termina cuando todos los vértices del grafo han sido visitados.

Recorrido en amplitud



Recorrido en amplitud

- Tomamos el índice inicial: 0 y lo visitamos. (Salida=0)
- Recuperamos sus adyacentes (1,2) y los encolamos: (cola=1,2)
- Desencolamos el 1 (c=2) y lo visitamos (Salida = 0,1).
- Recuperamos sus adyacentes (2,3). Sólo encolamos el 3 porque el 2 ya fue añadido (c=2,3).
- Desencolamos el 2 (c=3) y lo visitamos (Salida=0,1,2).
- Recuperamos sus adyacentes (3,5). Sólo encolamos el 5 porque el 3 ya fue añadido (c=3,5).
- Desencolamos el 3 (c=5) y lo visitamos (Salida=0,1,2,3).
- Recuperamos sus adyacentes (4). Encolamos el 4 (c=5,4).

Recorrido en amplitud. Ejemplo (cont)

- Desencolamos el 5 (c=4) y lo visitamos (Salida=0,1,2,3,5).
- Como 5 no tiene adyacentes no hacemos nada y continuamos.
- Desencolamos el 4 (c=empty) y lo visitamos (Salida=0,1,2,3,5,4).
- La cola está vacía.
- Como en el grafo existen nodos sin visitar (6 y 7), continuamos.
- Elegimos el 6 y lo visitamos (Salida=0,1,2,3,5,4,6). Como no tiene adyacentes no encolamos nada.
- Elegimos el 7 y lo visitamos (Salida=0,1,2,3,5,4,6,7). Su único adyacente ya ha sido visitado.
- Hemos terminado porque no quedan nodos por visitar.

Implementación Recorrido en amplitud

```java
public void breadth() {
    System.out.println("breadth traverse of the graph:");

    //to mark when a vertex has already been shown
    boolean visited[]=new boolean[numVertices];

    //we have to traverse all vertices
    for (int i=0;i<numVertices;i++) {
        if (!visited[i]) { //we only process the non-visited vertex
            breadth(i,visited);
            System.out.println();
        }
    }
}
```

Llama a otro método auxiliar que va a hacer el recorrido en amplitud de cada vértice (no visitado previamente).
Para registrar que vértices ya han sido visitados o no, usamos un array de booleanos

```java
//breadth order for the vertex i
protected void breadth(int i, boolean[] visited) {
    //this array helps to mark what vertices have been stored into the queue
    boolean stored[]=new boolean[numVertices];
    System.out.println("breadth traverse for " + i);
    //we use a queue to save the adjacent vertices that we visit
    SQueue q=new SQueue();
    //enqueue the first
    q.enqueue(i);
    //while the queue is not empty
    while (!q.isEmpty()) {
        //gets the first
        int vertex=q.dequeue();
        //shows the vertex and marks it as visited
        System.out.print(vertex+"\t");
        visited[vertex]=true;
        //gets its adjacent vertices
        int[] adjacents=getAdjacents(vertex);
        for(int adjVertex:adjacents) {
                //enqueue only those that have not been visited or stored yet
                if (!visited[adjVertex] && !stored[adjVertex]) {
                    q.enqueue(adjVertex);
                    stored[adjVertex]=true;
                }
        }
    }
}
```

Recorrido en profundidad

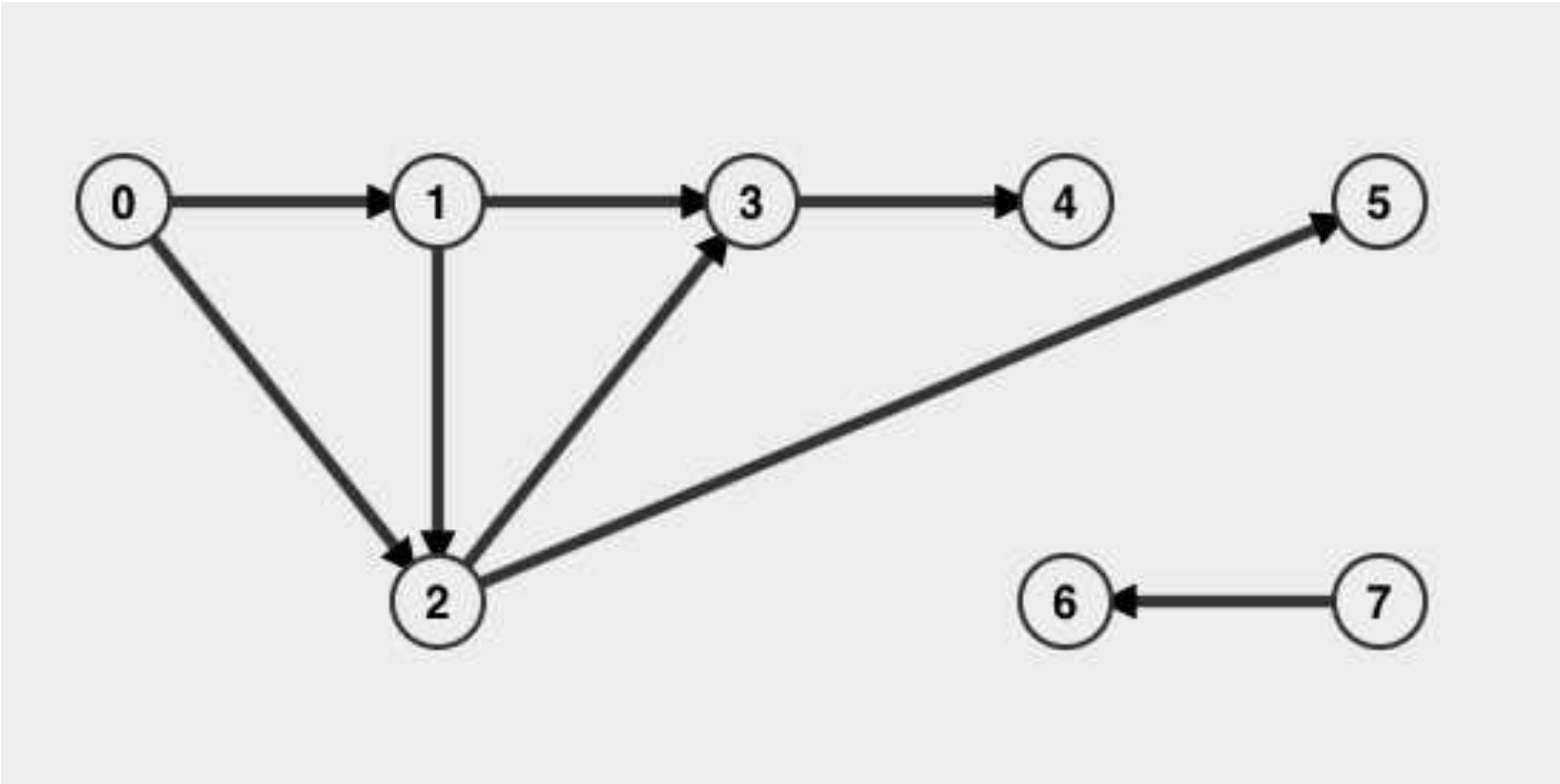
- Va localizando los posibles caminos y en el caso de no poder continuar, vuelve al punto donde existen nuevos caminos posibles con el fin de visitar todos los vértices.
- Ver animación http://visualgo.net/dfsbfs.html (dfs)

Recorrido en profundidad

- Pasos del algoritmo:

1. Se toma un vértice v como inicial y se marca como visitado.
2. Por cada vértice w, no visitado y adyacente a v, deberemos hacer una llamada recursiva sobre w. Repetir mientras haya vértices no visitados y adyacentes a v.
3. Repetir 1-2 mientras haya vértices no visitados. El algoritmo termina cuando todos los vértices del grafo han sido visitados.

Recorrido en profundidad



Recorrido en profundidad

- Comenzamos con el vértice inicial v=0. Lo visitamos (salida =0) y obtenemos sus adyacentes son {1,2}.
  - Empezamos con 1, lo visitamos (salida=0,1). Obtenemos sus adyacentes {2,3}.
    - Tomamos el 2, lo visitamos (salida=0,1,2). Obtenemos sus adyacentes {3,5}.
      - Visitamos el 3 (salida =0,1,2,3) y recuperamos sus adyacentes {4}.
      - Visitamos al 4 (salida=0,1,2,3,4) como no tiene más adyacentes, continuamos con los adyacentes de 3. Como 3 no tiene más adyacentes, continuamos con los adyacentes de 2.
      - Visitamos el 5 (salida =0,1,2,3,4,5) y como no tiene más adyacentes regresamos al 2.
      - El 2 no tiene más adyacentes por lo que continuar, así que debemos regresar al 1.
    - El 1 tiene otro adyacente, 3, pero esté ya ha sido visitado, así que no debemos continuar por ese camino. Continuamos con los adyacentes de 1ya no tiene más adyacentes por recorrer.
  - El otro adyacente de 0, es 2, que no tenemos que visitar porque ya ha sido visitado, y por tanto, no debemos continuar por ese camino.
  - Como aún quedan nodos por visitar, elegimos otro de los no visitados, 6 y lo visitamos. (Salida=0,1,2,3, 4,5,6). 6 no tiene adyacentes.
  - Visitamos el único que queda por visitar. (salida=0,1,2,3, 4,5,6,7). Tiene un adyacente (6) pero ya ha sido visitado.
  - Hemos terminado.

Recorrido en profundidad

```java
public void depth() {
    System.out.println("depth traverse of the graph:");
    //to mark when a vertex has already been shown
    boolean visited[]=new boolean[numVertices];
    //we have to traverse all vertices
    for (int i=0;i<numVertices;i++) {
        if (!visited[i]) depth(i,visited);
    }
}
```

Llama a otro método auxiliar que va a hacer el recorrido en profundidad de cada vértice (no visitado previamente).
Para registrar que vértices ya han sido visitados o no, usamos un array de booleanos

Recorrido en profundidad

```java
protected void depth(int i,boolean[] visited) {
    //prints the vertex and marks as visited
    System.out.print(i+"\t");
    visited[i]=true;
    //gets its adjacent vertices
    int[] adjacents=getAdjacents(i);
    for (int adjV:adjacents) {
        if (!visited[adjV]) {
            //only depth traverses those adjacent vertices
            //that have not been visited yet
            depth(adjV,visited);
        }
    }
}
```