Graphs

Aims

- To study the representation of a binary relation between the data of the collection through a structure ***Graph*** and some of its most important applications
- Reuse of models already studied to represent graphs and to explore them

References

Michael T. Goodrich and Roberto Tamassia. "*Data Structures & Algorithms in Java*" (4th edition), John Wiley & Sons, 2005

(chapter 13)

Contents

1. Introduction

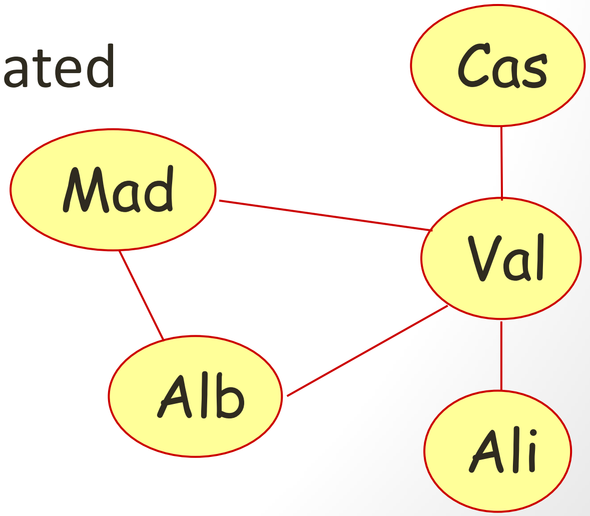
Relations between data of the collection

- Binary relation between data of the collection:
  - A relation R over a set S is defined as a set of pairs (a, b) / $a, b \in S$
  - If $(a, b) \in R$, it can be written as "$a\ R\ b$" and it denotes that a is related with b

Example: graph whose ***vertices***(S) are related via ***edges*** (R)

S = {Cas, Val, Ali, Mad, Alb}

R = {(Cas,Val), (Val,Ali), (Val,Alb), (Mad,Alb), (Mad,Val)}

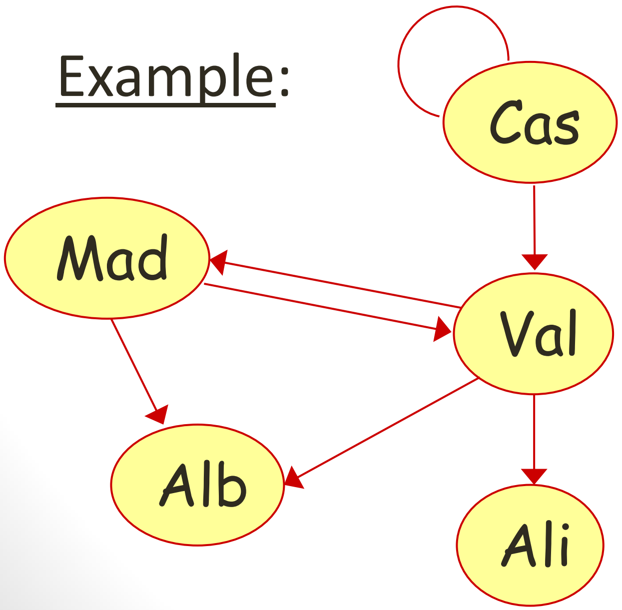


1. Introduction

Directed graphs (Digraphs)

- A **directed graph** (*dg*) is a pair $G = (V, E)$
  - V is a finite set of ***vertices*** (or nodes)
  - E is a set of directed ***edges***, where an *edge* is an ordered pair of vertices (u, v): $u \rightarrow v$

<u>Example</u>:



V = {Cas, Val, Ali, Alb, Mad}

$|V| = 5$

E = {(Cas,Cas), (Cas,Val), (Val,Mad), (Val,Alb), (Val,Ali), (Mad,Val), (Mad,Alb)}

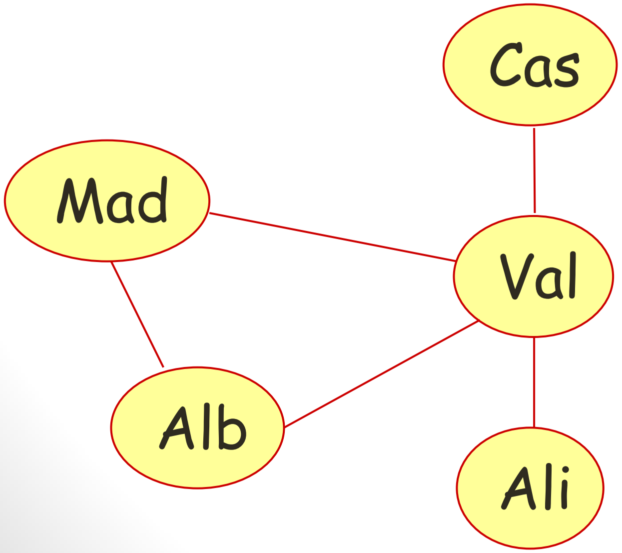
$|E| = 7$

1. Introduction

Non directed Graphs

- A **non directed graph** (*ndg*) is a pair $G = (V, E)$
  - V is a finite set of ***vertices***
  - E is a set of non directed ***edges***, where an *edge* is a pair of non directed vertices $(u,v) = (v,u)$, $u \neq v$: u —— v

Example:



V = {Cas, Val, Ali, Alb, Mad}

$|V| = 5$

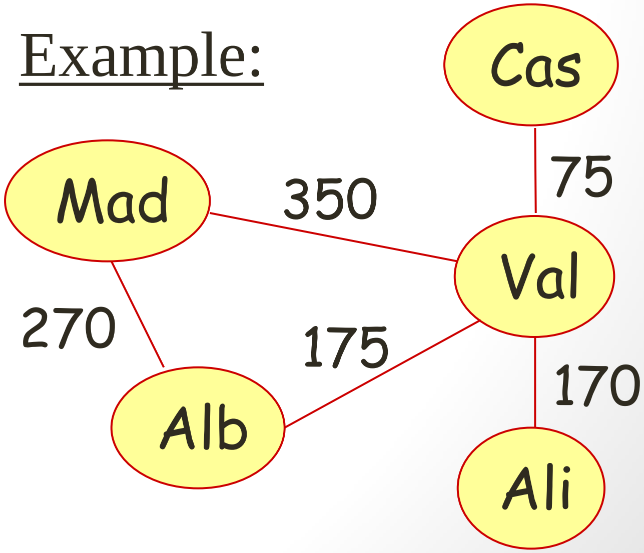
E = {(Cas,Val), (Val,Ali), (Val,Mad), (Val,Alb), (Mad,Alb)}

$|E| = 5$

1. Introduction
Labelled graphs

- A ***labelled* graph** is a graph $G = (V, E)$ where a function is defined $f: E \to L$, with L a set whose components are called ***labels***
  - *Note*: the labelling function can be defined over V, the set of vertices
- Un ***weighted* graph** is a graph labelled with real numbers ($L \equiv \Re$)

Example:



1. Introduction

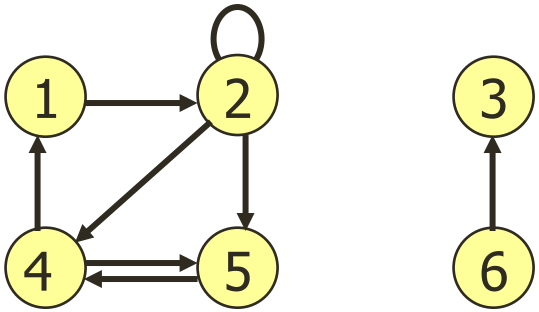
Relations of adjacency

- Be $G = (V, E)$ a graph. If $(u, v) \in E$, we say that the vertex v is adjacent to the vertex u

<u>Example</u> with the vertex 2:

2 is adjacent to 1

1 is not adjacent to 2



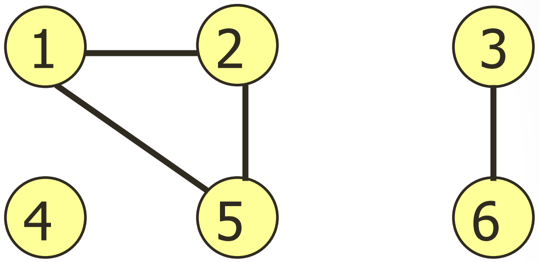
- In a non directed graph the relation is symmetrical

1. Introduction

Degree of a vertex

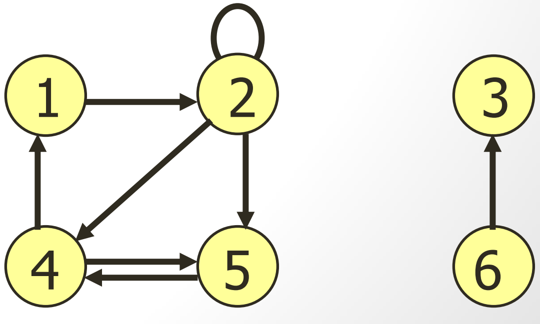
- The ***degree of a vertex*** in a non directed graph is the number of its incident edges (or adjacent vertices)

<u>Example</u>: the degree of the vertex 2 is 2



- The degree of a vertex in a directed graph is the sum of:
  - *outdegree*
  - *indegree*

<u>Example</u>:

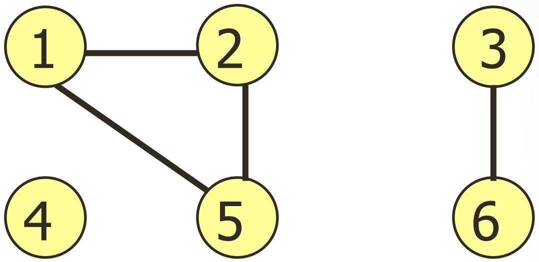


1. Introduction

Degree of a vertex

- The ***degree of a vertex*** in a non directed graph is the number of its incident edges (or adjacent vertices)

Example: the degree of the vertex 2 is 2



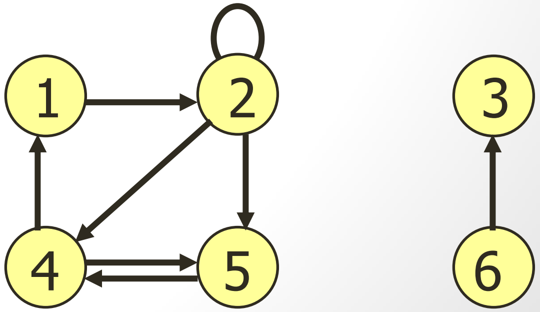
- The degree of a vertex in a directed graph is the sum of:
  - *outdegree*
  - *indegree*

Example: the indegree of 2 is 2

+ the outdegree of 2 is 3

---

the degree of vertex 2 is 5

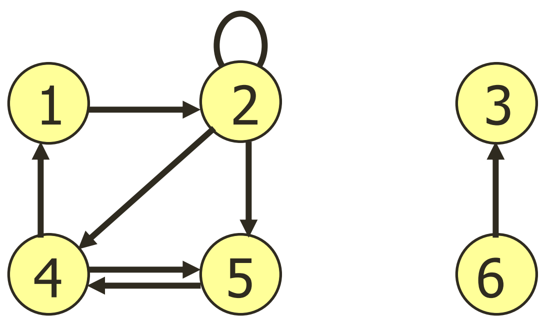


1. Introduction

Degree of a graph

- The ***degree of a graph*** is the maximum degree of its vertices

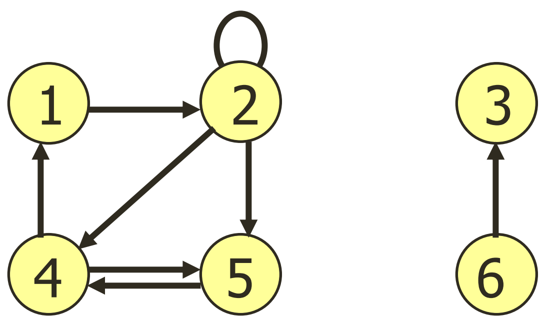
<u>Example</u>:



1. Introduction
Degree of a graph

- The ***degree of a graph*** is the maximum degree of its vertices

<u>Example</u>:



The degree of this graph is 5 (the degree of vertex 2)

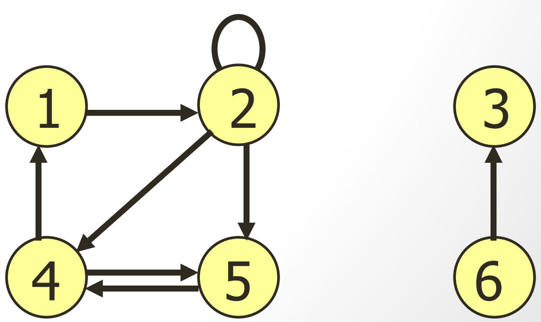
1. Introduction
Paths

- A ***path of length*** k from u to u' in a graph $G = (V, E)$ is a sequence of vertices $\langle v_0, v_1, \ldots, v_k \rangle$ such that:
  - $v_0 = u$ and $v_k = u'$
  - $\forall\, i : 1 \ldots k : (v_{i-1}, v_i) \in E$
  - The length k of the path is the number of edges
  - The length of the path with weights is the sum of the weights of the edges of the path
- If there exists a path P from u to u', we say that u' is ***reachable*** from u via P

1. Introduction
Simple paths and cycles

- A ***cycle*** is a path $\langle v_0, v_1, \ldots, v_k \rangle$:
  - starting and ending in the same vertex ($v_0 = v_k$)
  - containing at least an edge
- A path or cycle is ***simple*** if all its vertices are different
- A ***loop*** is a cycle of length 1
- A graph is ***acyclic*** if it does not contain cycles

Example:

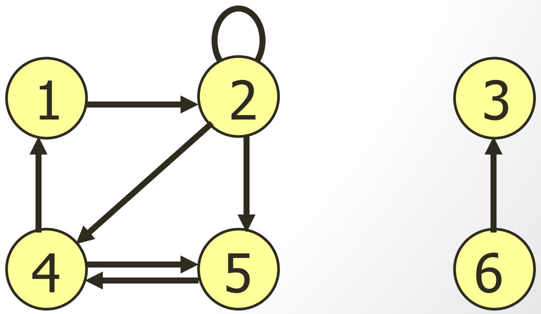


1. Introduction
Simple paths and cycles

- A ***cycle*** is a path $\langle v_0, v_1, \ldots, v_k \rangle$:
  - starting and ending in the same vertex ($v_0 = v_k$)
  - containing at least an edge
- A path or cycle is ***simple*** if all its vertices are different
- A ***loop*** is a cycle of length 1
- A graph is ***acyclic*** if it does not contain cycles

<u>Example</u>:

$\langle 1, 2, 5, 4, 1 \rangle$ is a cycle of length 4



1. Introduction

Exercise 1. Be $G = (V, E)$ a directed graph with weights:

$V = \{v_0, v_1, v_2, v_3, v_4, v_5, v_6\}$

$E = \{(v_0,v_1, 2), (v_0,v_3, 1), (v_1,v_3, 3), (v_1,v_4, 10), (v_3,v_4, 2),$
$(v_3,v_6, 4), (v_3,v_5, 8), (v_3,v_2, 2), (v_2,v_0, 4), (v_2,v_5, 5),$
$(v_4,v_6, 6), (v_6,v_5, 1) \}$

Please answer to:

a) $|V|$ and $|E|$

b) Adjacent vertices of each vertex

c) Degree of each vertex and degree of the graph

d) Simple paths from v_0 to v_6, and their length with and without weights

e) Reachable vertices from v_0

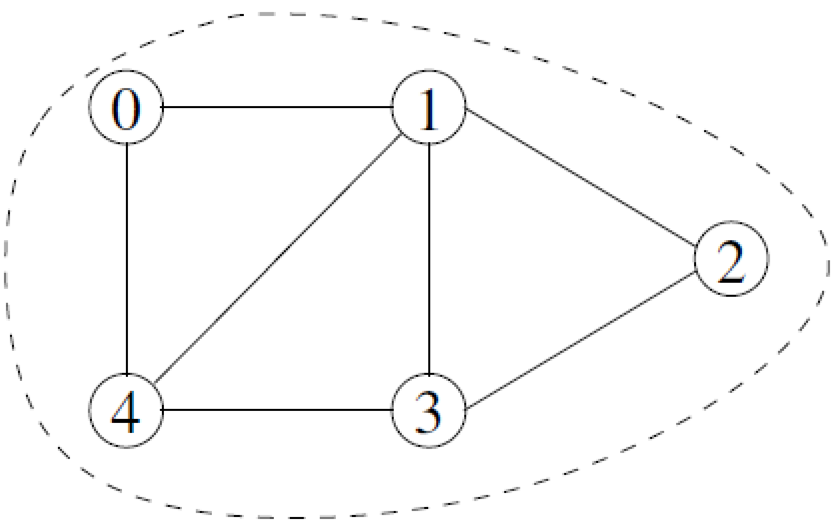
f) Minimum paths from v_0 to the rest of vertices

g) Is there any cycle?

1. Introduction

Connected components

- The ***connected components*** in a non directed graph are the equivalence classes of vertices under the relation "*being reachable*"
  - A non directed graph is connected if $\forall u, v \in V$, v is reachable from u. That is if it has just one connected component

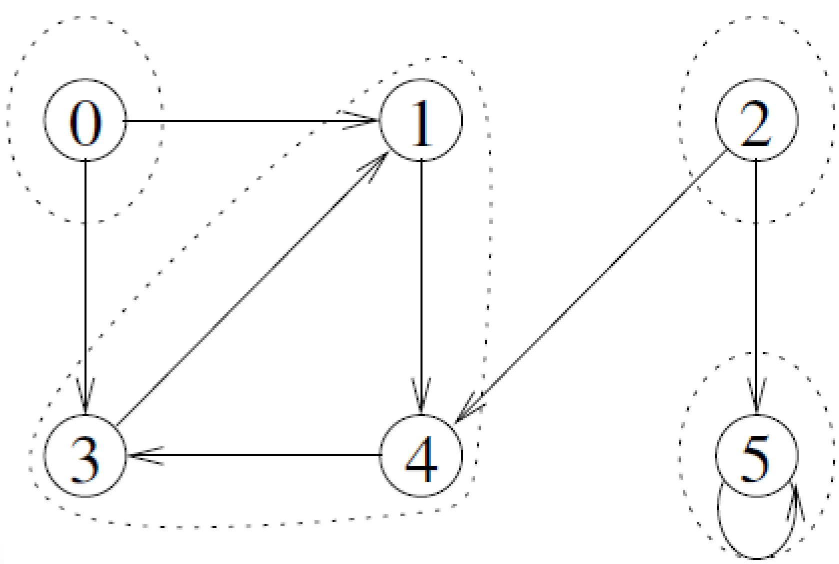


Example: connected non directed graph

1. Introduction

Strongly connected components

- The ***strongly connected components*** in a directed graph are the equivalence classes of vertices under the relation "*being mutually reachable*"
  - A directed graph is strongly connected if $\forall u, v \in V$, v is reachable from u



Example: directed graph with 4 strongly connected components

2. Representation of graphs

Representations

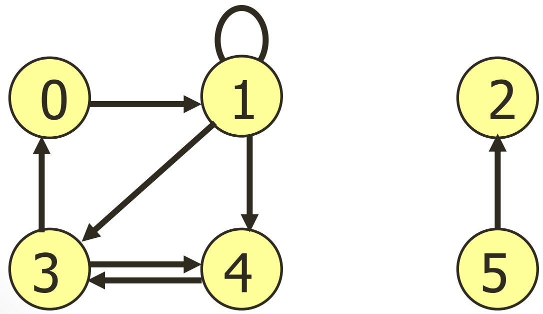
- There exist two forms for representing a graph:
  - If the graph is ***disperse*** ($|E| \lll |V|^2$):
    adjacency lists
  - If the graph is ***dense*** ($|E| \approx |V|^2$):
    adjacency matrix

2. Representation of graphs

Adjacency matrix

- A graph $G = (V, E)$ is represented as a ***matrix*** of $|V| \text{x} |V|$ of elements of type *boolean*
  - If $(u, v) \in E \rightarrow G[u, v] = true$ (otherwise $G[u, v] = false$)
  - Spatial cost …
  - Time for access …

<u>Example:</u>



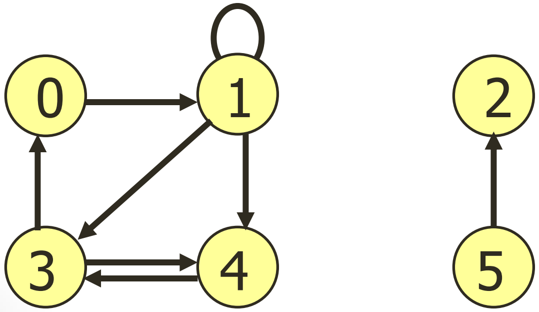
	0	1	2	3	4	5
0	false	true	false	false	false	false
1	false	true	false	true	true	false
2	false	false	false	false	false	false
3	true	false	false	false	true	false
4	false	false	false	true	false	false
5	false	false	true	false	false	false

2. Representation of graphs

Adjacency matrix

- A graph $G = (V, E)$ is represented as a ***matrix*** of $|V| \mathrm{x} |V|$ of elements of type *boolean*
  - If $(u, v) \in E \rightarrow G[u, v] = true$ (otherwise $G[u, v] = false$)
  - Spatial cost $O(|V|^2)$
  - Time for access O(1)

Example:



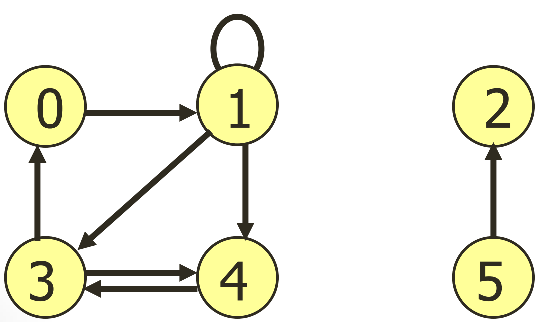
	0	1	2	3	4	5
0	false	true	false	false	false	false
1	false	true	false	true	true	false
2	false	false	false	false	false	false
3	true	false	false	false	true	false
4	false	false	false	true	false	false
5	false	false	true	false	false	false

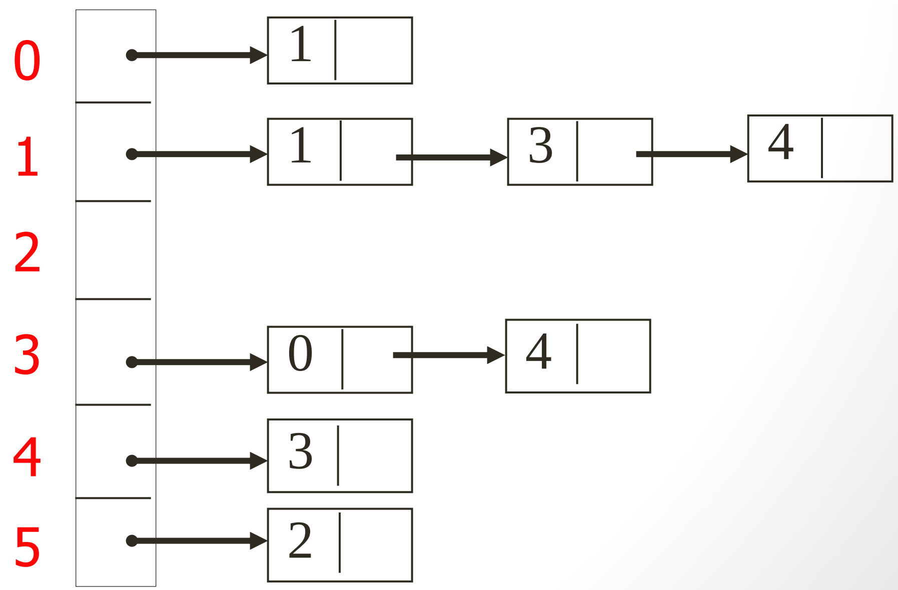
2. Representation of graphs

Adjacency lists

- A graph $G = (V, E)$ is represented as an ***array*** of $|V|$ ***lists*** of vertices
  - $G[v]$, $v \in V$, is the list of the adjacent vertices to v
  - Spatial cost …
  - Time for access …

<u>Example:</u>



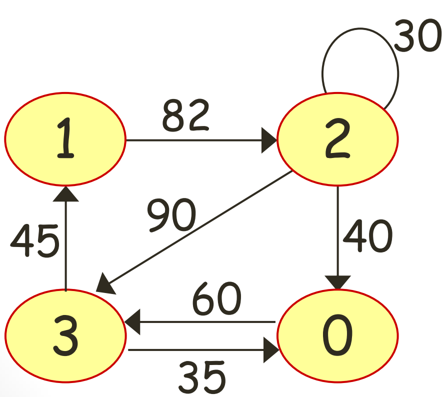


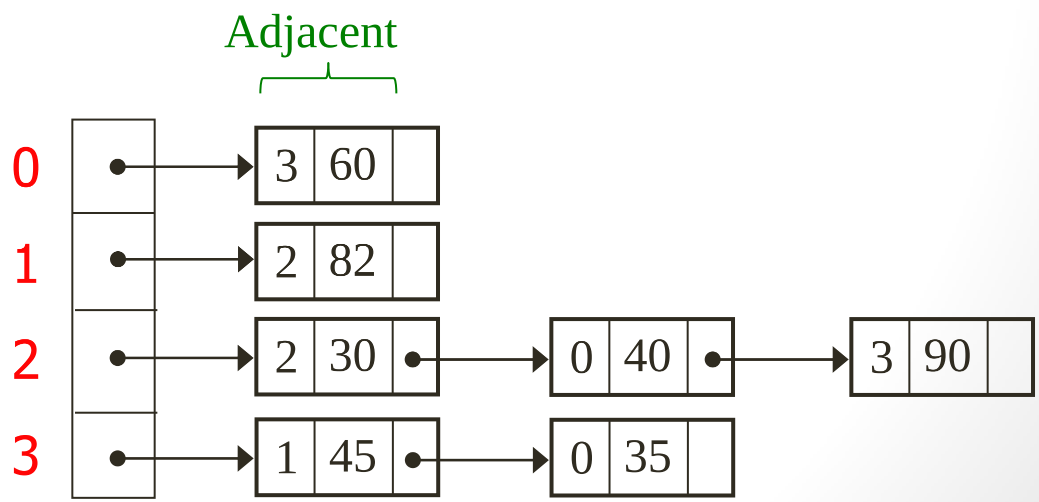
2. Representation of graphs

The class Adjacent

```
class Adjacent {
  int target; // Vertex target of the edge
  double weight;    // weight of the edge
  public Adjacent(int d, double p){ target = d; weight = p;}
  public String toString(){ return target + "(" + weight + ")";}
}
```

<u>Example:</u>

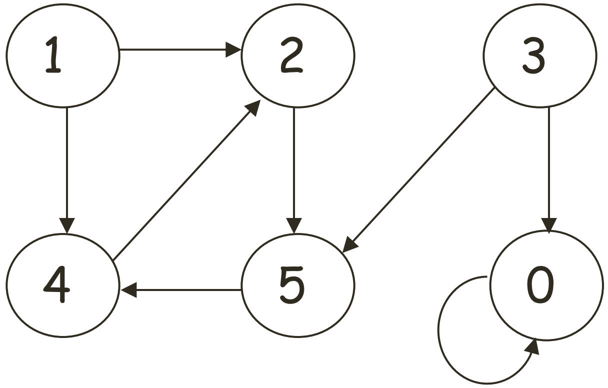




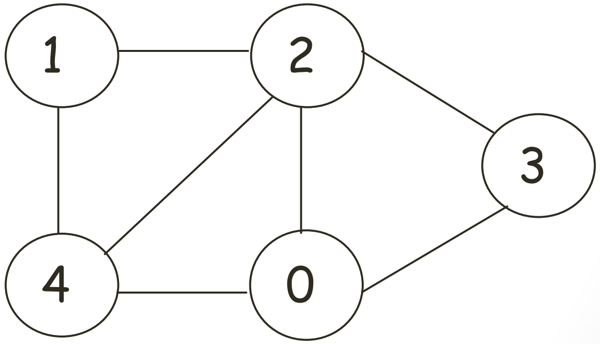
2. Representation of graphs
Exercise

<u>Exercise 2</u>: Represent the following graphs through an adjacency matrix and through adjacency lists

a)



b)



2. Representation of graphs

Basic functionality of a graph

- The abstract class *Graph* defines the basic functionality of a graph
  - We do not use an *interface* because the code of some of the methods, such as traversals ones, are type of graph-independent and implementation-independent
- The basic functionality includes :
  - Setters *(to modify; Spanish: modificadores)* : insertion of edges (with or without weights)
  - Getters *(to access; Spanish: consultores)*: number of vertices/edges, search for edges
  - Traversals: in-depth and in-breadth

2. Representation of graphs

The class Graph: getters

```
public abstract class Graph {
  // It returns the number of vertices of the graph
  public abstract int numVertices();

  // It returns the number of edges of the graph
  public abstract int numEdges();

  // It checks whether the edge (i,j) exists
  public abstract boolean existEdge(int i, int j);

  // It retrieves the weight of the edge (i,j)
  public abstract double weightEdge(int i, int j);

  // It returns a list with the adjacent vertices of vertx i
  public abstract ListWithPI<Adjacent> adjacentsOf(int i);
```

2. Representation of graphs

The class Graph: setters

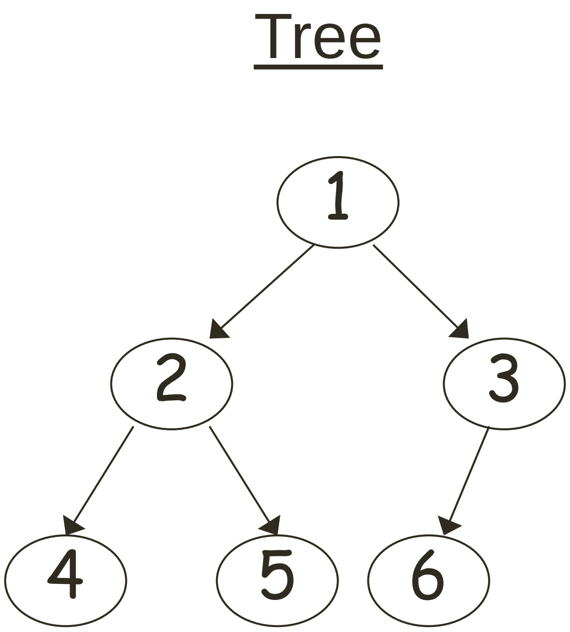
```
public abstract class Graph {
  ...

  // It adds the edge (i,j) to a non weighted graph
  public abstract void insertEdge(int i, int j);

  // It adds the edge (i,j) with weight p to a weighted graph
  public abstract void insertEdge(int i, int j, double p);
```

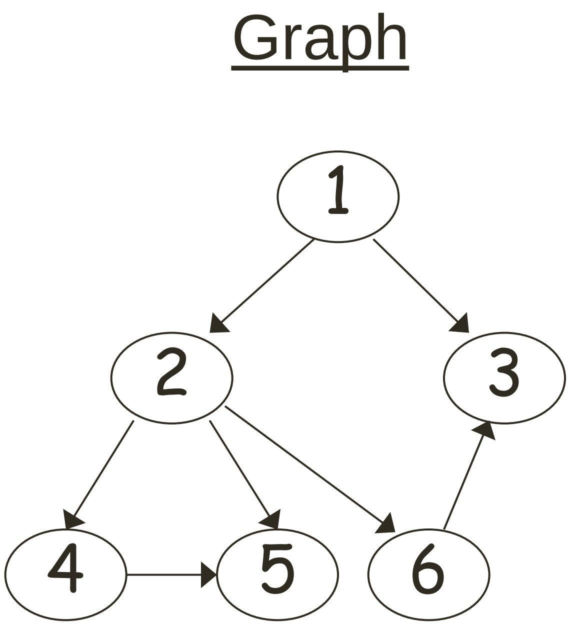
- The method to insert edges is overloaded in order to allow the insertion of edges both in graphs with and without weights

3. Graph traversals

Depth First Search

- Generalisation of the *PreOrder* traversal of a tree:



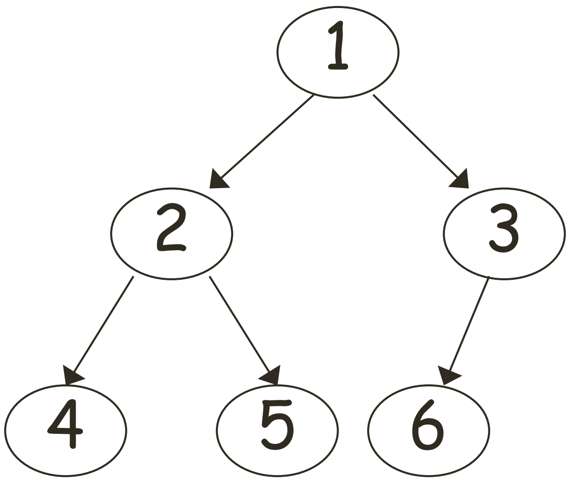


PreOrder: Father, Left, Right

3. Graph traversals
Depth First Search

- Generalisation of the *PreOrder* traversal of a tree:

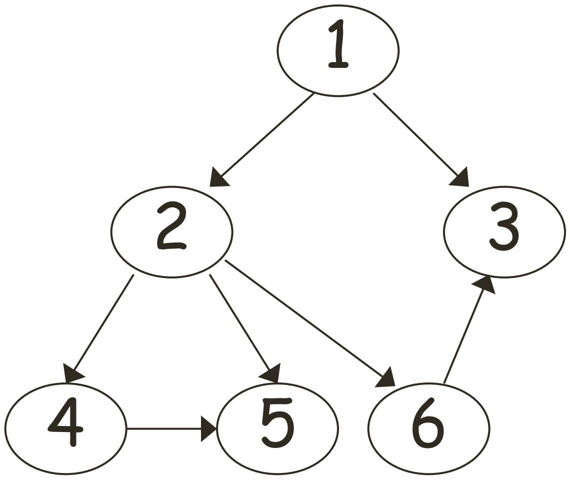
<u>Tree</u>



PreOrder: Father, Left, Right

1, 2, 4, 5, 3, 6

<u>Graph</u>

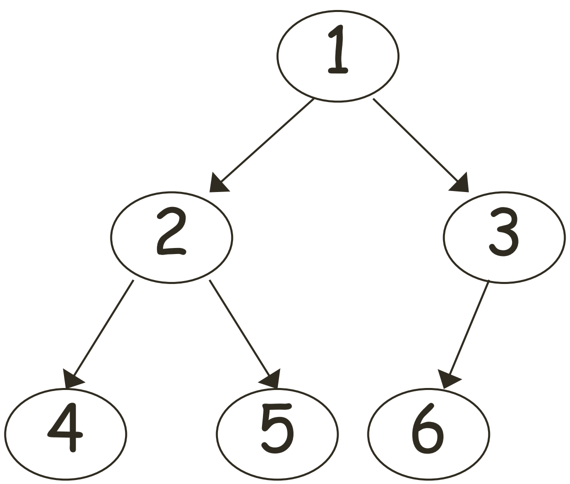


Vertices do not have to be visited twice

3. Graph traversals
Depth First Search

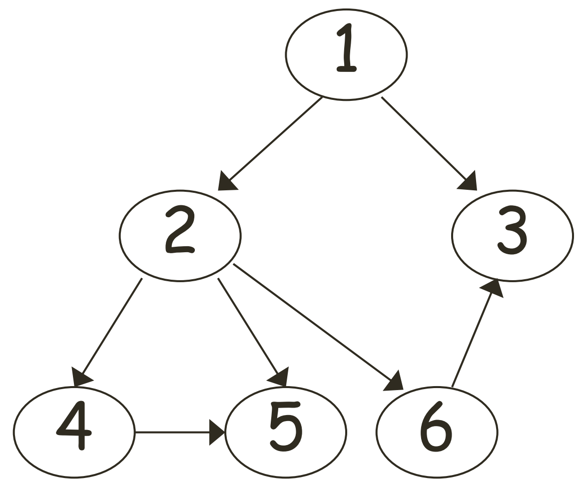
- Generalisation of the *PreOrder* traversal of a tree:

Tree



PreOrder: Father, Left, Right
1, 2, 4, 5, 3, 6

Graph



1, 2, 4, 5, 6, 3

Vertices do not have to be visited twice

3. Graph traversals

Implementation of Depth First Search (1/2)

```
public abstract class Graph {
  // DFS needs the following attributes
  protected boolean visited[]; // To not repeat vertices
  protected int orderVisit;  // Order of visit of vertices

  // DFS returns an array with visited vertices
    public int[] toArrayDFS() {
    int res[] = new int[numVertices()];
    visited = new boolean[numVertices()];
    orderVisit = 0;
    for (int  i = 0; i < numVertices(); i++)
      if (!visited[i]) toArrayDFS(i, res);
    return res;
  }
```

Initialisation to false

3. Graph traversals

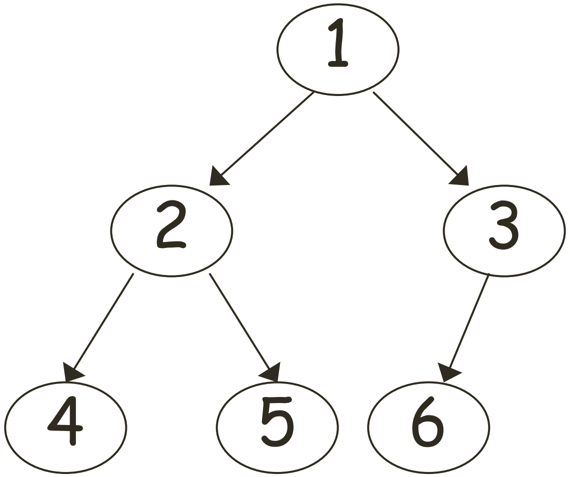
Implementation of Depth First Search (2/2)

```
// Recursive method for DFS
protected void toArrayDFS(int source, int res[]) {
  // Source vertex is added and marked as visited
  res[orderVisit++] = source;
  visited[source] = true;
  // Adjacent vertices of source are visited
  ListaConPI<Adjacent> l = adjacentsOf(source);
  for (l.inicio(); !l.esFin(); l.siguiente()) {
    Adjacent a = l.recuperar();
    if (!visited[a.target]) toArrayDFS(a.target, res);
  }
}
```

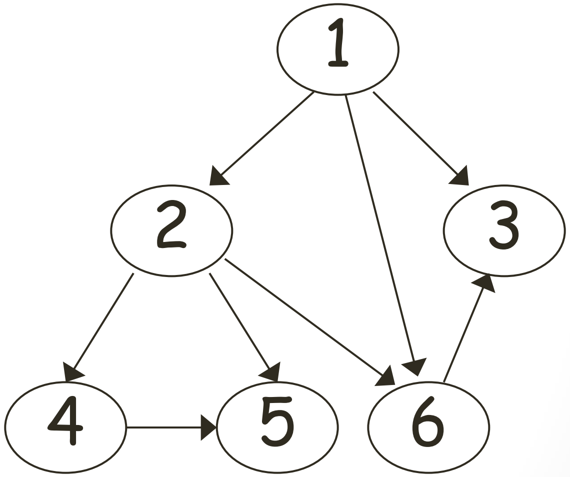
3. Graph traversals
Breadth First Search

- Generalisation of traversal by levels of a tree:

Tree



Graph

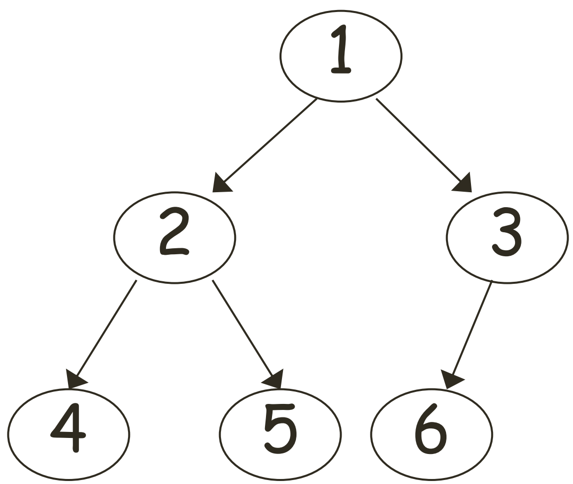


By levels

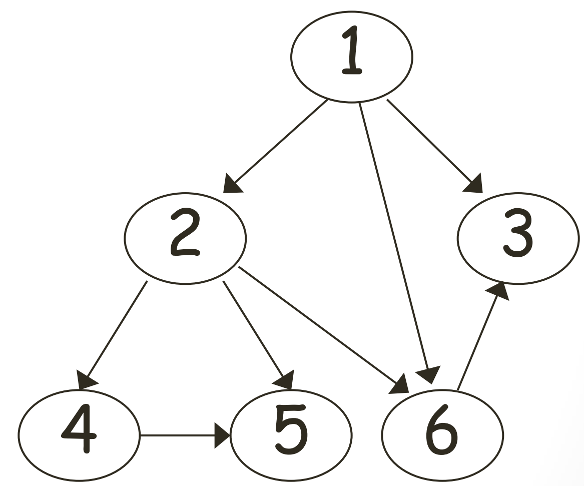
3. Graph traversals
Breadth First Search

- Generalisation of traversal by levels of a tree:

Tree



Graph



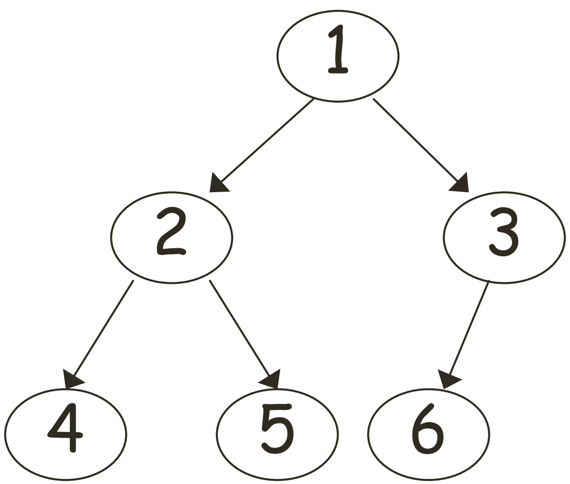
By levels

1, 2, 3, 4, 5, 6

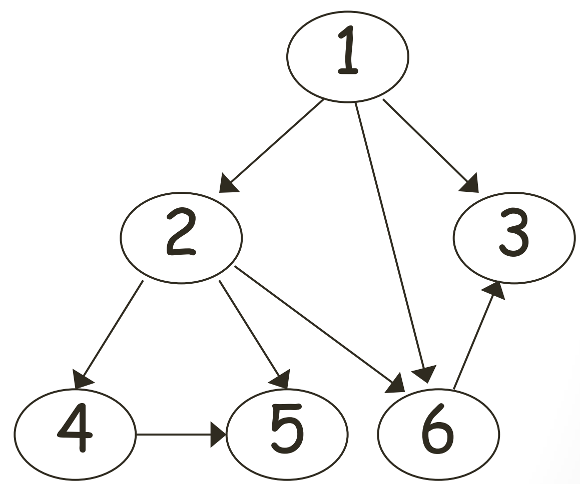
3. Graph traversals
Breadth First Search

- Generalisation of traversal by levels of a tree:

Tree



Graph



By levels

1, 2, 3, 4, 5, 6

1, 2, 6, 3, 4, 5

3. Graph traversals

Implementation of Breadth First Search (1/2)

```
public abstract class Graph {
  ... // Apart from the attributes visited and orderVisit,
      // BFS needs an auxiliar queue
     // (the algorithm is iterative)
  protected Queue<Integer> q;

  // BFS
  public int[] toArrayBFS() {
    int res[] = new int[numVertices()];
    visited = new boolean[numVertices()];
    orderVisit = 0;
    q = new ArrayQueue<Integer>();
    for (int  i = 0; i < numVertices(); i++)
      if (!visited[i]) toArrayBFS(i, res);
    return res;
  }
```

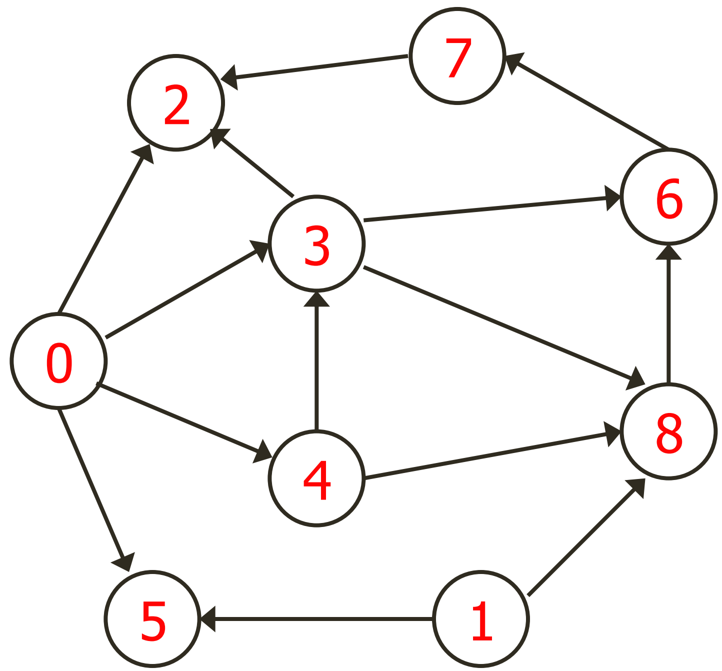
3. Graph traversals

Implementation of Breadth First Search (2/2)

```
protected void toArrayBFS(int source, int res[]) {
  res[orderVisit++] = source;
  visited[source] = true;
  q.encolar(source);
  while (!q.esVacia()) {
    int u = q.desencolar().intValue();
    ListaConPI<Adjacent> l = adjacentsOf(u);
    for (l.inicio(); !l.esFin(); l.siguiente()) {
      Adjacent a = l.recuperar();
      if (!visited[a.target]) {
        res[orderVisit++] = a.target;
        visited[a.target] = true;
        q.encolar(a.target);
      }
    }
  }
}
```

3. Graph traversals
Exercise

Exercise 3. Show the result of DFS and BFS graph traversals on the following graph:



4. Implementation

The class GraphDirected (1/3)

```
// Implementation of a Directed Graph
public class GraphDirected extends Graph {

  // Nomber of Vertices and Edges
  protected int numV, numE;
   // The array of list fo adjeacents of each vertex
  protected ListaConPI<Adjacent> elArray[];

  // Construction of the graph (number of vertices)
  public GraphDirected(int numVertices) {
    numV = numVertices;
    numE = 0;
    elArray = new ListaConPI[numVertices];
    for (int i = 0; i < numV; i++)
      elArray[i] = new LEGListaConPI<Adjacent>();
  }
```

4. Implementation

The class GraphDirected (2/3)

```
// Getters (in Spanish: consultores)
public int numVertices() { return numV; }
public int numEdges()  { return numE; }

public ListaConPI<Adjacent> adjacentsOf(int i) {
  return elArray[i];
}

public boolean existEdge(int i, int j) {
  ListaConPI<Adjacent> l = elArray[i];
  boolean isin = false;
  for (l.incio(); !l.esFin()&& !isin; l.siguiente())
    if (l.recuperar().target == j) isin = true;
  return isin;
}
```

4. Implementation

The class GraphDirected (3/3)

```
public double weightEdge(int i, int j) {
  ListaConPI<Adjacent> l = elArray[i];
  for (l.inicio(); !l.esFin(); l.siguiente())
    if (l.recuperar().target == j)
      return l.recuperar().weight;
  return 0.0;
}

// Insert edge
public void insertaEdge(int i, int j) {
  insertEdge(i, j, 1.0);  // Its weight is 1.0 by default
}
public void insertEdge(int i, int j, double p) {
  if (!existEdge(i,j)) {
    elArray[i].insert (new Adjacentj,p));
    numE++;
  }
}
```

4. Implementation
Exercises

Exercise 4. Define the following methods in the class *GraphDirected*:

a) Get the outdegree of a given vertex

b) Get the indegree of a given vertex

c) On the basis of the above methods, design a method that returns the degree of a graph

d) Check if a vertex is *source*, that is, if a vertex has indegree==0 and outdegree>0

e) Check if a vertex is a *sink* (i.e., *sumidero* in Spanish) has outdegree==0 and has incoming edges from all other vertices of the graph

4. Implementation
Exercises

Exercise 5: Implement a method in the class *Graph* that checks if a vertex is reachable from another given vertex

Exercise 6: A transpose graph *T* of a graph *G* has the same set of vertices but with all of the edges reversed compared to the orientation of the corresponding edges in *G.* That is, an edge (u, v) in *G* corresponds to an edge (v, u) in *T*.

Design a method *GraphDirected* that allows for obtaining its transpose graph:

```
public GraphDirected graphTranspose()…
```

5. Minimum spanning tree

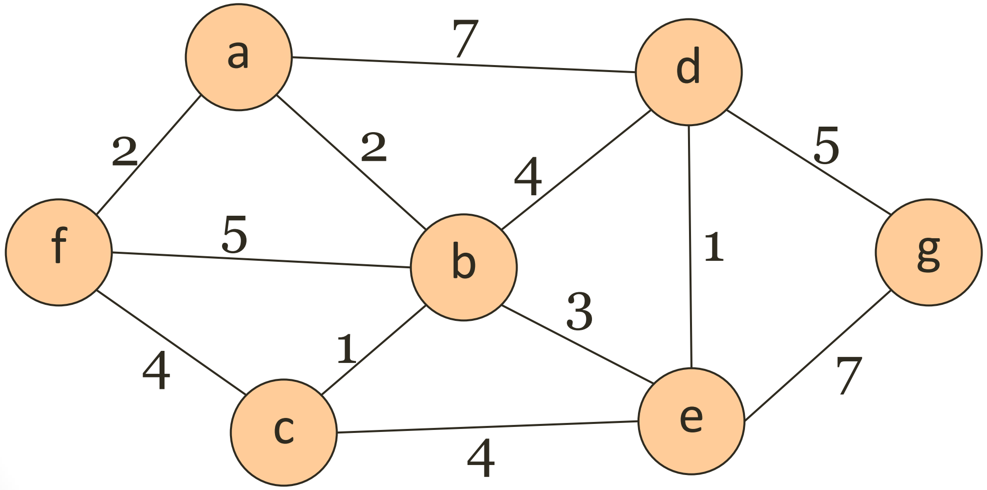
Introduction

- A undirected graph is **connected** if each pair of vertices is connected via a path
- An acyclic undirected and connected graph is a **tree**
- A spanning tree (in Spanish árbol generador or árbol de recubrimiento) of a graph (V, E) is a tree (V', E') such that:
  - $V' = V$
  - $E' \subseteq E$
- The problem of obtaining the minimum spanning tree is very important in many applications (e.g. design of networks, of roads, astronomy, medicine, etc.)

5. Minimum spanning tree

Example

- The vertices of the following graph represent the light spots in a factory, and the edges the length of cable that is necessary to link the two light spots:



Problem:

How to use all the light spots using the minimum quantity of cable?

5. Minimum spanning tree
Kruskal's algorithm

Step 1: store the edges in a priority queue

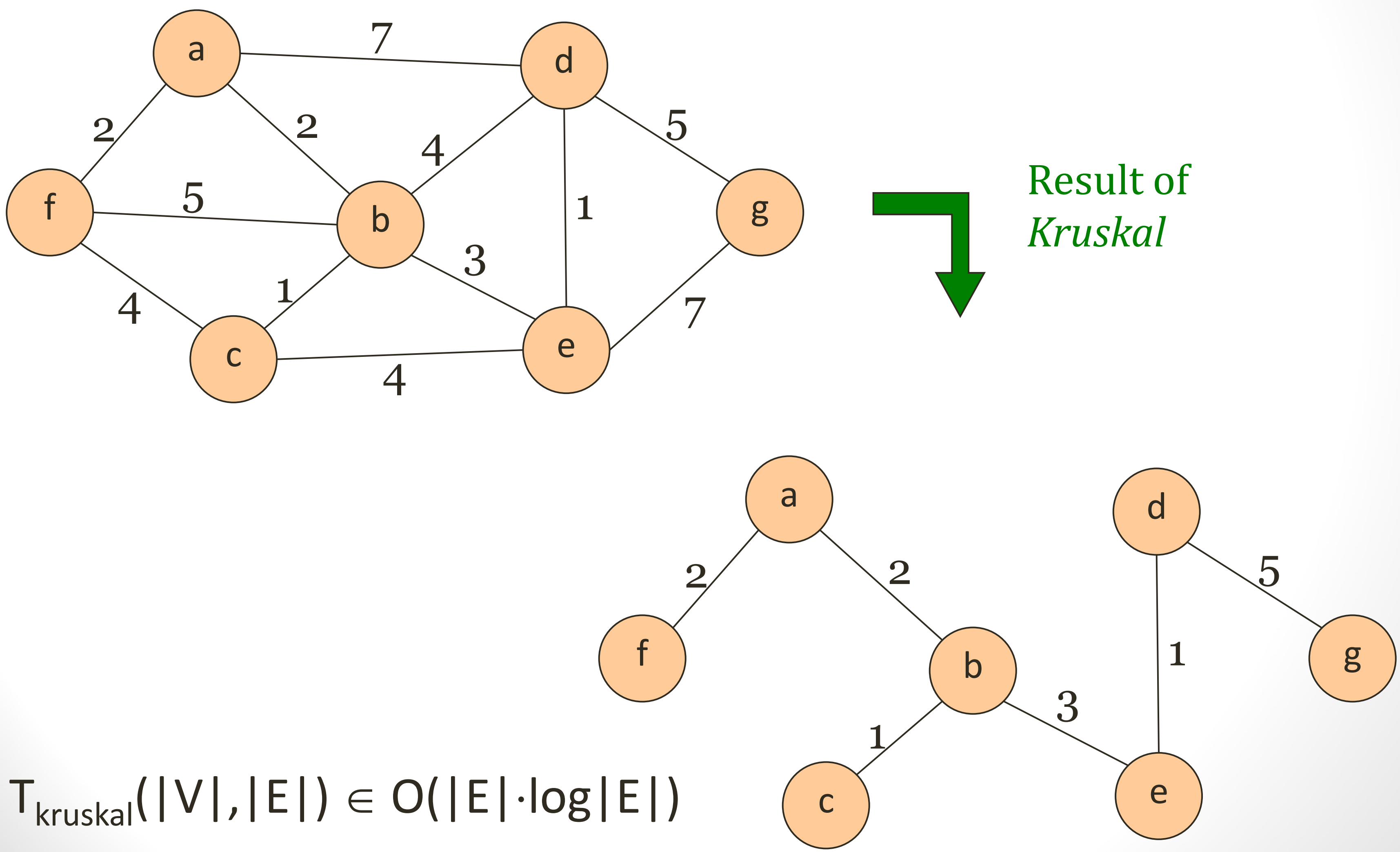
(an edge will be smaller than another one if its cost is smaller)

Step 2: start from a graph without edges (just vertices)

Step 3: *while* |E| < |V| - 1 *do*:

- Retrieve and delete from the priority queue the edge with the smallest cost
- Insert the edge in the graph if not cycles occur

5. Minimum spanning tree
Kruskal's algorithm



$T_{kruskal}(|V|,|E|) \in O(|E| \cdot \log|E|)$

5. Minimum spanning tree

/* This is just an example: a complete implementation, also for the Kruskal algorithm, will be given in the lab */

Class for the priority queue with the weights of the edges:

```
public class Edge
       implements Comparable< Edge> {
  int source, target;
  double weight;
  public Edge(int o, int d, double p) {
    source = o;
    target = d;
    weight = p;
  }
  public int compareTo(Edge p) {
    if (weight < p. weight) return -1;
    return weight > p. weight ? 1 : 0;
  }
}
```

5. Minimum spanning tree: Kruskal

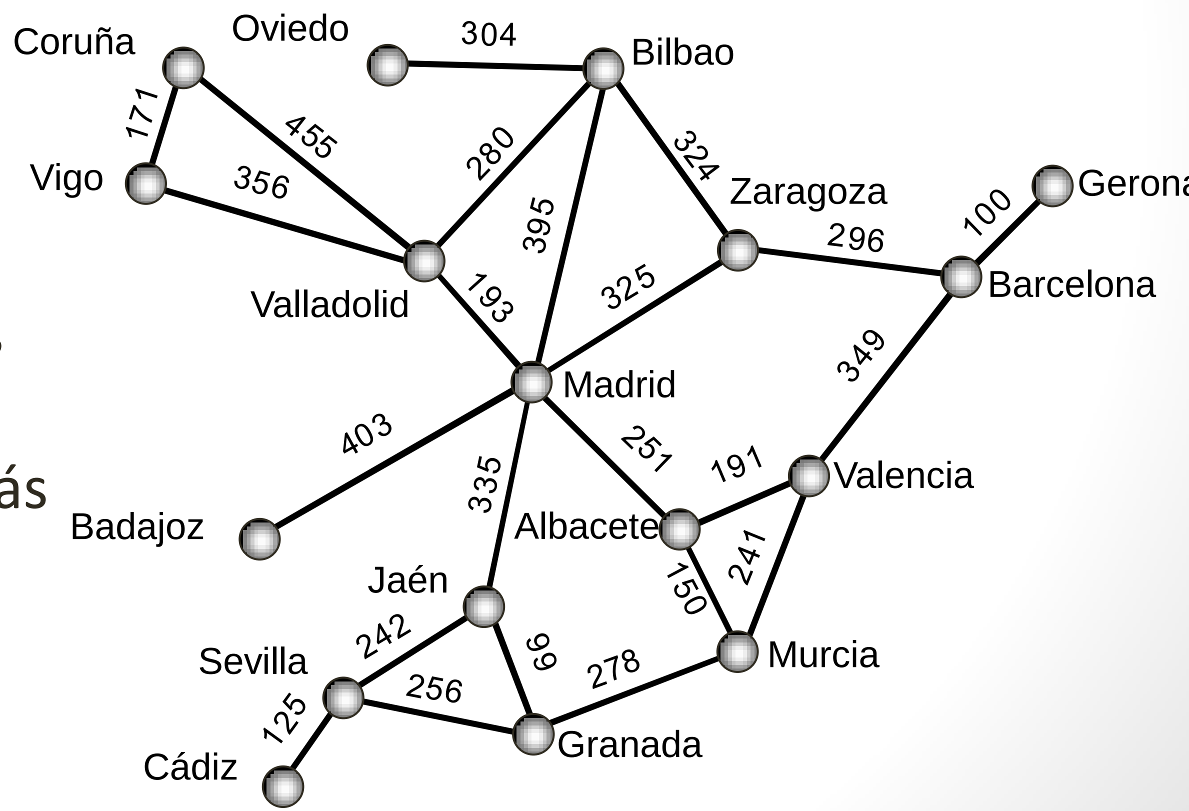
```
public Edge[]  Kruskal() {
 Edge[] res = new Edge[numVertices()-1];
 PriorityQueue<Edge> qPrior = new MinHeap<Edge>();
 MFset m = new ForestMFset(numV);
 // The priority queue with the edges of the graph is created
 for (int i = 0; i < numVertices(); i++){
  ListaConPI<Adjacent> l = adjacentsOf(i);
  for (l.inicio(); !l.esFin(); l.siguiente()){
      Adjacent a = l.recuperar();
      qPrior.insert(new Edge (i, a.target, a.weight));
  }
 }
 int numE = 0; // Construction of the minimum spanning tree
 while (numE < numVertices() - 1 && !qPrior.isEmpty()) {
   Edge a = qPrior.deleteMin();
   if (m.find(a.source) != m.find(a.target)) {
     m.merge(a.source, a.target); res[numE++] = a;
   }
 }
 return res;
}
```

6. Shortest path problem

Definition of the problem

- ***Peso de un camino***: suma de los pesos de las aristas por las que pasa: $p(v_0, v_1, \ldots, v_k) = \sum_{i=1}^{k} p(v_{i-1}, v_i)$

- ***Problema***: calcular el camino de mínimo peso entre dos nodos o entre un nodo y todos los demás



6. Shortest path problem

Dijkstra's algorithm

- ***Dijkstra***: it calculates the shortest paths from a vertex to the rest of vertices. It requires the weigths of the edges to be positive.
- It stores the information in two arrays:
  - ***distanceMin***: it stores the minimum distance from the source vertex to the rest of vertices
  - ***pathMin***: for each vertex it stores the previous vertex in the shortest path from the source vertex

6. Shortest path problem

Example of Dijkstra: minimum paths from v_0

- We want to calculate the minimum paths from the vertex v_0, for instance, to the rest of vertices
- First step: What is the information wrt the vertex v_0?

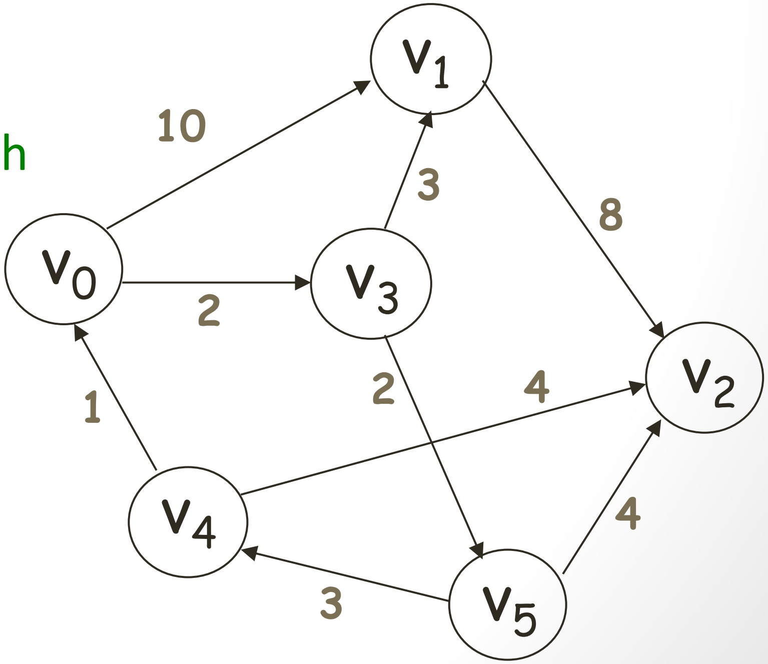
// To reach v_0 from v_0 *cost nothing*

distanceMin[0] = 0

// There is not a previous vertex: the path

// starts in v_0

pathMin[0]   = -1



6. Shortest path problem

Example of Dijkstra: minimum paths from v_0

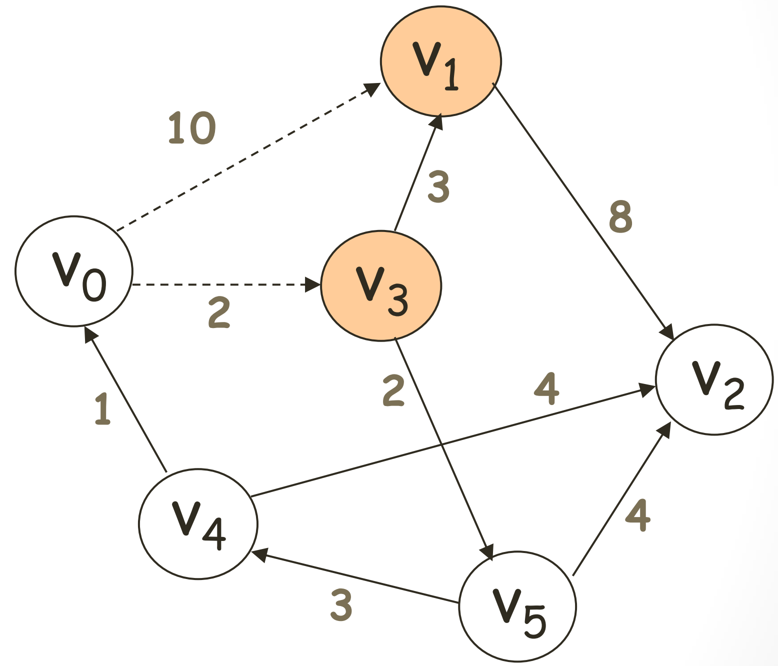
- Second step: we calculate the distance of the adjacent vertices of v_0

distanceMin[1] = 10

pathMin[1] = 0

distanceMin[3] = 2

pathMin[3] = 0



What vertex do we continue with? v_1 or v_3?

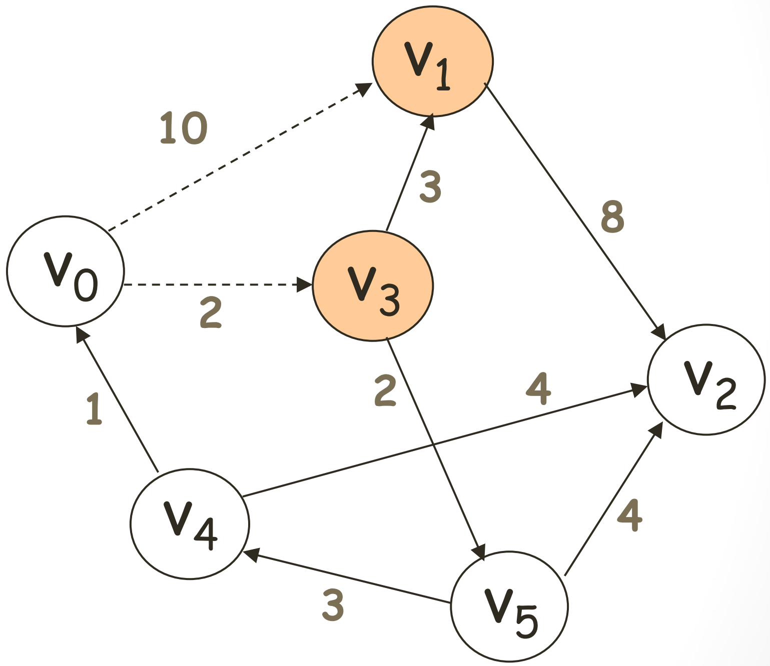
6. Shortest path problem

Example of Dijkstra: minimum paths from v_0

- Second step: we calculate the distance of the adjacent vertices of v_0

distanceMin[1] = 10
pathMin[1] = 0

distanceMin[3] = 2
pathMin[3] = 0



What vertex do we continue with? v_1 or v_3?

⇒ v_3 : its distance from v_0 is the shortest one

6. Shortest path problem

Example of Dijkstra: minimum paths from v_0

- Third step: we calculate the distance of the adjacent vertices of v_3

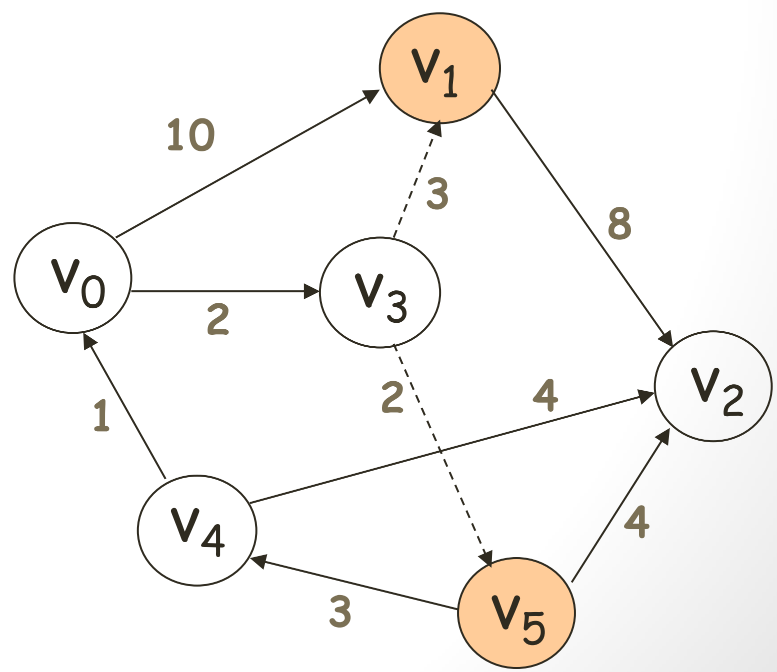
Vertex v_1 has been reached already:

distanceMin[1] = *min*(distanceMin[1], distanceMin[3] + 3) = min(10, 2 + 3) = 5

pathMin[1] = 3

distanceMin[5] =
distanceMin[3] + 2 = 4

pathMin[5] = 3



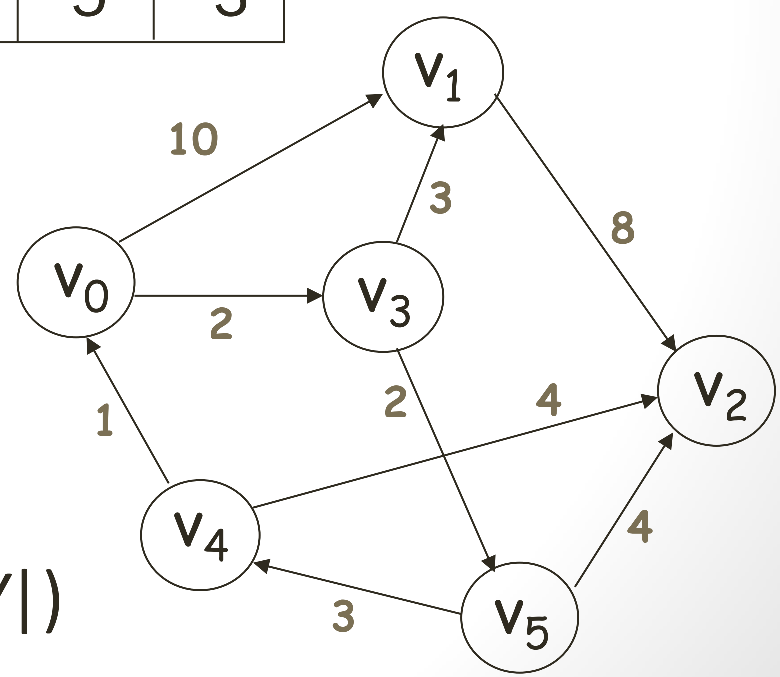
- Etc...

6. Shortest path problem

Example of Dijkstra: minimum paths from v_0

- <u>Final result:</u>

	v_0	v_1	v_2	v_3	v_4	v_5
distanceMin	0	5	8	2	7	4
pathMin	-1	*3*	*5*	*0*	*5*	*3*



$T_{dijkstra}(|V|, |E|) \in O(|E| \cdot \log|V|)$

6. Shortest path problem

Decode the shortest path

- How do we calculate now the shortest path between v_0 and another vertex, for instance v_4?

	v_0	v_1	v_2	v_3	v_4	v_5
pathMin	-1	3	5	0	5	3

1.- The shortest way to reach $\boldsymbol{v_4}$ is through $\boldsymbol{v_5}$

2.- The shortest way to reach $\boldsymbol{v_5}$ is through $\boldsymbol{v_3}$

3.- The shortest way to reach $\boldsymbol{v_3}$ is through $\boldsymbol{v_0}$

4.- $\boldsymbol{v_0}$ es el origen, pues *pathMin*[*0*] = -1

$<v_4, v_5, v_3, v_0>$ Note: the path needs to be inverted $\rightarrow$

$<v_0, v_3, v_5, v_4>$

6. Shortest path problem

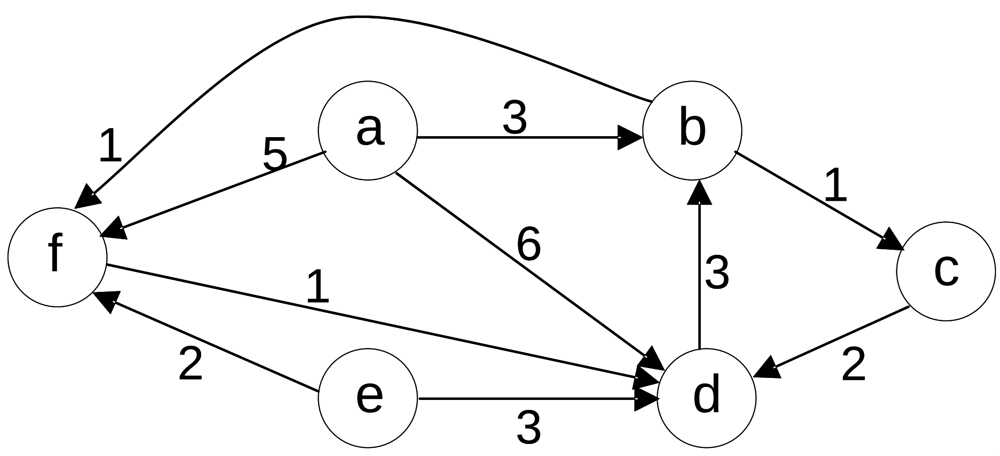
Dijkstra's algorithm (pseudo code)

```
void dijkstra(int vSource) {
  pathMin[v] = -1,    ∀v∈V       // Initialisations
  distanceMin[v] = ∝, ∀v∈V
  distanceMin[vSource] = 0
  qPrior ← (vSource,0)
  while qPrior ≠ ∅{  // While there are vertices to explore
    v ← qPrior   // Next vertex to explore: the one with shortest distance
    if !visited[v]{    // Repetitions are avoided
      visited[v] = true
      for each a ∈ adjacentsOf(v){// The vertices are explored
        w = a.target                  // adjacent of v
        weightW = a.weight
        // Is it the best way to reach w through v ?
        if distanceMin[w] > distanceMin[v] + weightW {
          distanceMin[w] = distanceMin[v] + weightW;
          parthMin[w] = v;
          qPrior ← (w, distanceMin[w])
}}}}}
```

6. Shortest path problem
Exercises

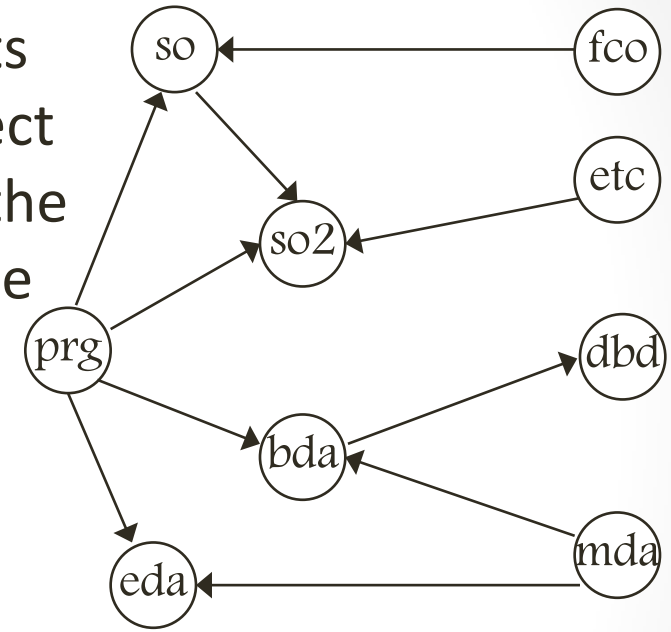
<u>Exercise 7</u>. The vertices of the following graph represent people (Ana, Begoña, Carmen, Daniel, Eliseo and Francisco) and the edges indicate if a person has the mobile number of the other one. The weight of an edge is the cost to send an SMS (for instance, Ana can send an SMS to Begoña for 3 cents).

Show the steps of Dijkstra to see what would be the cheapest way for Ana to send an SMS to Francisco.



7. Topological sorting
Introduction

- <u>Example</u>: the following graph represents the previous subjects that a given subject requires. An edge (u, w) indicates that the subject u has to be passed in order to be allowed to enroll in w.

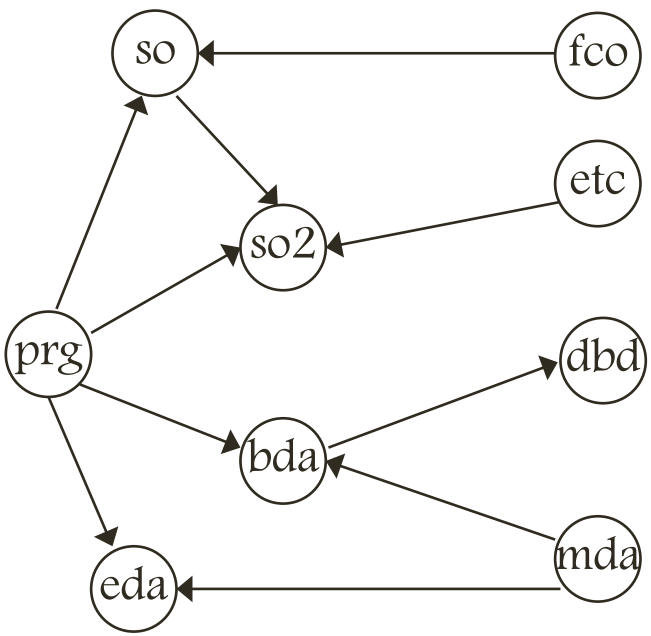


- ⟨prg, so, so2⟩, ⟨prg, bda, dbd⟩, ⟨mda, bda, dbd⟩, ⟨mda, eda⟩, etc. are **topological sortings**

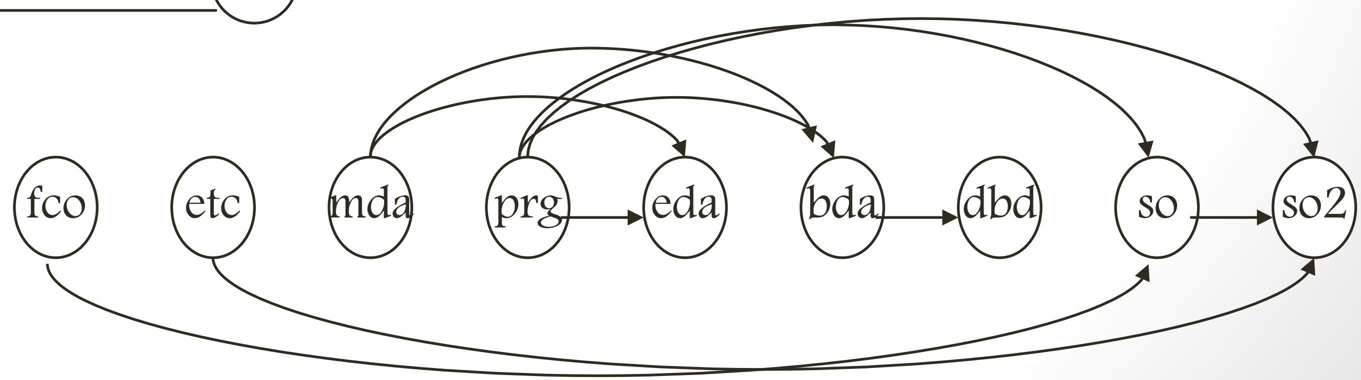
- A ***topological sorting*** is a linear sorting of a given directed <u>acyclic</u> graph, preserving the original partial sorting

7. Topological sorting
Introduction

<u>Example</u>: find a sorting (i.e., an order) to study ALL subjects:



In-depth traversal
+
Usage of a stack of vertices



7. Topological sorting

Initial method

```
// It returns an array with the topological sorting of the codes
  of the vertices
public int[] toArrayTopologic() {
  visited = new boolean[numVertices()];
  Pila<Integer> pVExplored = new ArrayPila<Integer>();
  // Traversal of vertices
  for (int vSource = 0; vSource < numVertices(); vSource++)
    if (!visited[vSource])
        topologicalSorting(vSource, pVExplored);
  // Result of topological sorting is copied in an array
  int res[] = new int[numVertices()];
  for (int i = 0; i < numVertices(); i++)
    res[i] = pVExplored.desapilar();
  return res;
}
```

7. Topological sorting
Recursive method

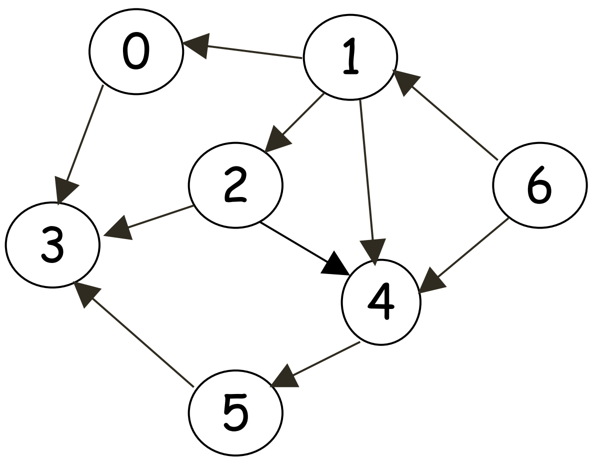
```
protected void topological sorting(int source,
                Stack<Integer> pVExplored) {
  visited[source] = true;
  // The adjacent vertices are explored
  ListaConPI<Adjacent> aux = adjacentsOf(source);
  for (aux.inicio(); !aux.esFin(); aux.siguiente()) {
    int target = aux.recuperar().target;
    if (!visited[target])
       topologicalSorting(target, pVExplored);
  }
  // the vertex is pushed
 pVExplored.apilar(source);
}
```

$T_{topologicalSorting}(|V|, |E|) \in O(|V| + |E|)$

7. Topological sorting

Exercise

Exercise 8: On the basis of the method *topologicalSorting*, show the topological sorting of the following directed acyclic graph:



Is the topological sorting unique? In case of negative answer show another valid topological sorting.