Chapter 5. Graphs

Data Structures and Algorithms

FIB

Q2 2018–19

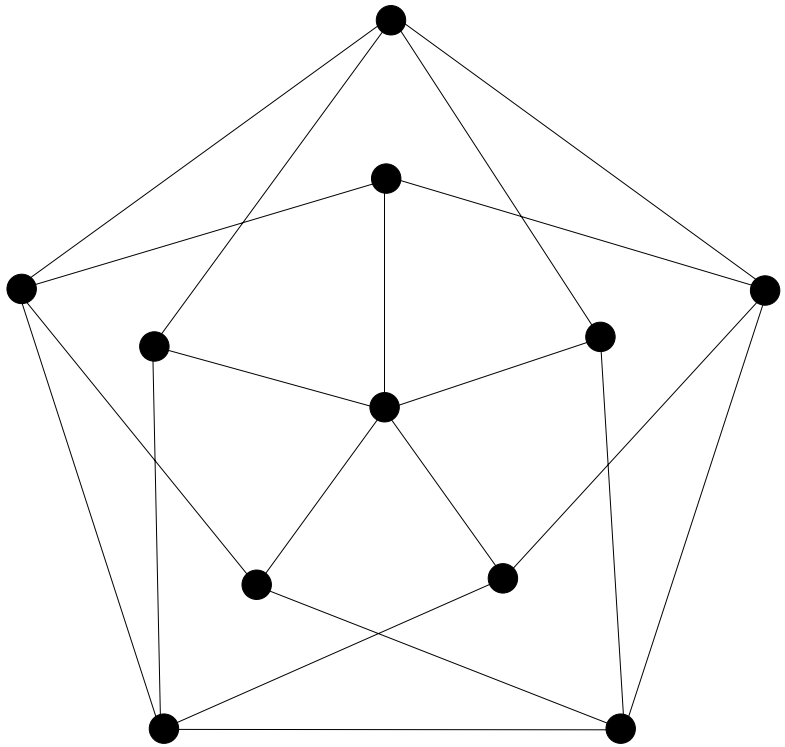
Jordi Delgado
(slides by Antoni Lozano)

Chapter 5. Graphs

Chapter 5. Graphs

Introduction

Why do we need graphs?



Because there are lots of problem that can be expressed in a clear and precise way by means of graphs.

Introduction

Example: coloring a map with the minimum number of colors

Which is the minimum number of colors needed to color a map in such a way that neighboring countries have different colors?

If we analyze a real map, we will find lots of irrelevant information:

- irregular borders,
- seas,
- points where more than two countries coincide...

But any map can be represented as a planar graph:

- A vertex corresponds to a country
- An edge corresponds to a border

Introduction

Example: coloring a map with the minimum number of colors

Which is the minimum number of colors needed to color a map in such a way that neighboring countries have different colors?

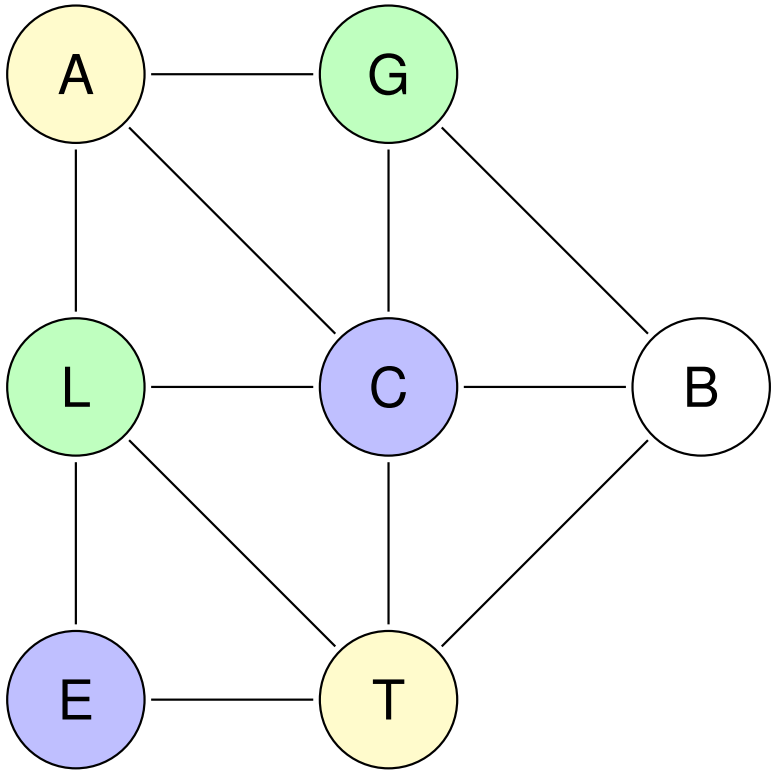
If we analyze a real map, we will find lots of irrelevant information:

- irregular borders,
- seas,
- points where more than two countries coincide...

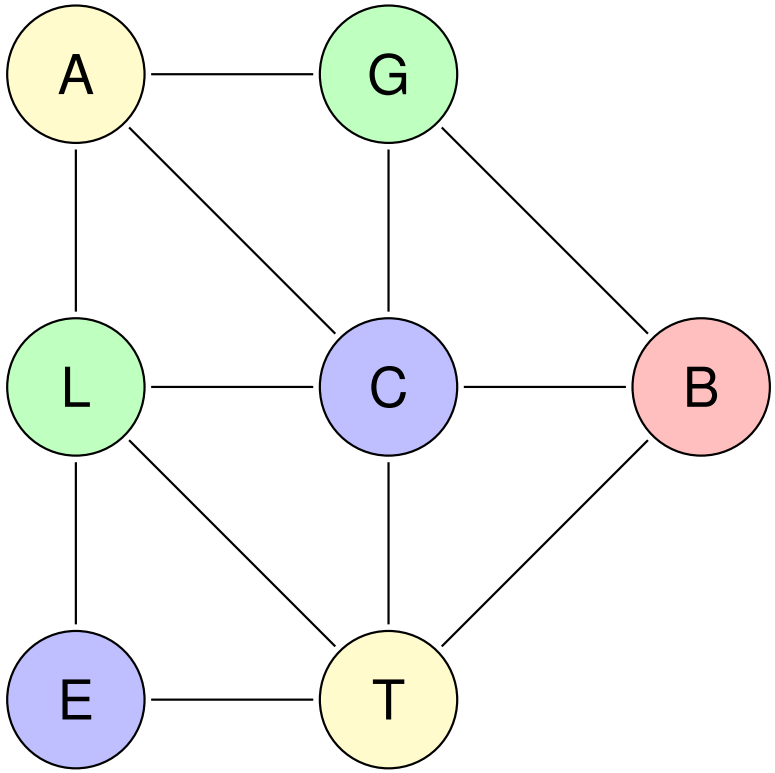
But any map can be represented as a planar graph:

- A vertex corresponds to a country
- An edge corresponds to a border

Introduction



Introduction



Introduction

Using graphs, we can use the following theorem.

Four color theorem (Apple/Haken, 1976)

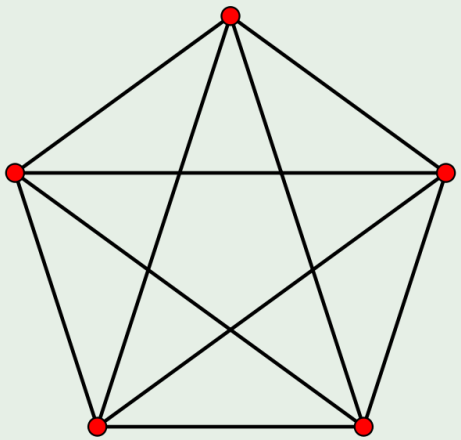
Any planar graph can be colored with 4 colors.

Hence, any map can be colored with 4 colors.

Example: connecting five objects in the plane

We cannot connect 5 objects in a plane without intersecting connections.

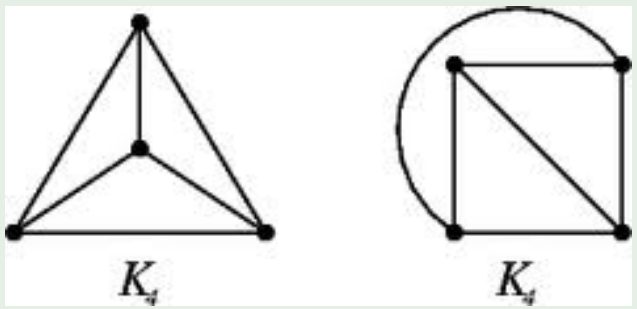
In gran theory, this is the same as saying that graph K_5



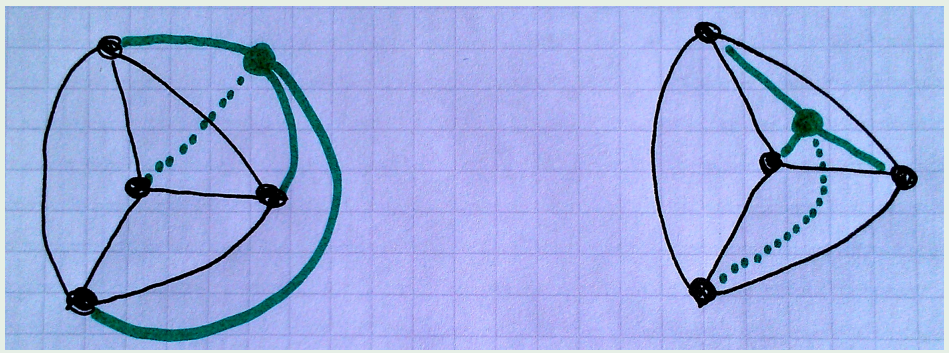
cannot be drawn in a plane without intersecting edges (it is not *planar*).

Introduction

One can see that K_5 has to contain K_4 (the complete graph with 4 vertices) that always defines 4 areas in the planes (1 external and 3 internals):



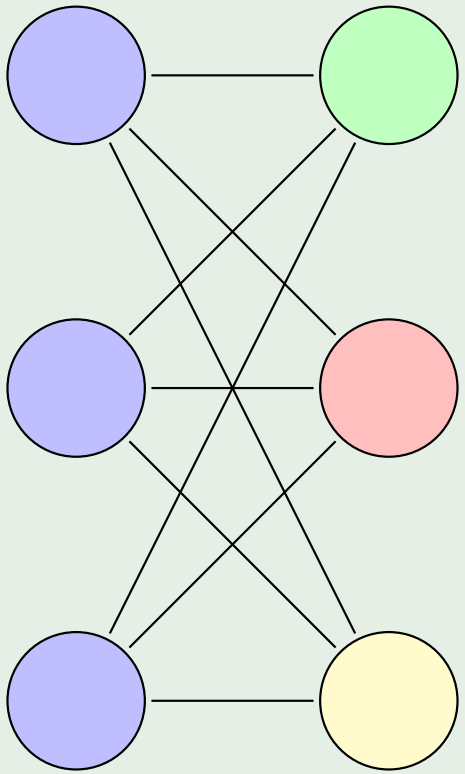
Independently of whether the fifth vertex is placed in the external or the internal area, there is always a vertex that cannot be connected without intersecting some edge.



Example: connecting three objects with three other objects

We cannot connect 3 houses to 3 resources (water, light and gas) in a plane without intersecting connections.

In graph theory, this amounts to saying that the graph $K_{3,3}$



is not planar.

But with a similar argument to the one of K_5, one can see that $K_{3,3}$ is not planar.

Theorem (Kuratowski, 1922)

A graph is planar if and only if it does not contain a subgraph that is a subdivision of K_5 or $K_{3,3}$.

But with a similar argument to the one of K_5, one can see that $K_{3,3}$ is not planar.

Theorem (Kuratowski, 1922)

A graph is planar if and only if it does not contain a subgraph that is a subdivision of K_5 or $K_{3,3}$.

Introduction

Example: King Arthur and the round table

King Arthur wants to seat his knights at the round table but wants to avoid that some of them sit next to each other.

Solution

- We create a graph where each vertex is a knight and there is an edge between each pair of knights that can sit next to each other.
- We input the graph to an algorithm that finds a cycle going through all vertices (Hamiltonian cycle): this will be the order to be seated.

Introduction

Example: King Arthur and the round table

King Arthur wants to seat his knights at the round table but wants to avoid that some of them sit next to each other.

Solution

- We create a graph where each vertex is a knight and there is an edge between each pair of knights that can sit next to each other.
- We input the graph to an algorithm that finds a cycle going through all vertices (Hamiltonian cycle): this will be the order to be seated.

Introduction

Graph applications:

- **Maps.** How to find the best way in the subway network? Which is the cheapest combination for going from Barcelona to Paris?
- **WWW.** If a website is a vertex and a link is an edge. the whole WWW is a graph. How could we compute the importance of a page with respect to information search?
- **Task scheduling.** In industrial processes, there are some tasks that should be done before others, but we want to complete the whole process in the minimum amount of time.
- **Computer networks, circuits, road maps, industrial processes,...**

Graphs

Definition

A graph is a set of vertices and a set of edges that connect pairs of different vertices (there is at most an edge between each pair of vertices).

Notation

Formally, a graph is a pair (V, E), where V is a finite set (of vertices) and E is a set of unordered pairs of vertices.

Graphs

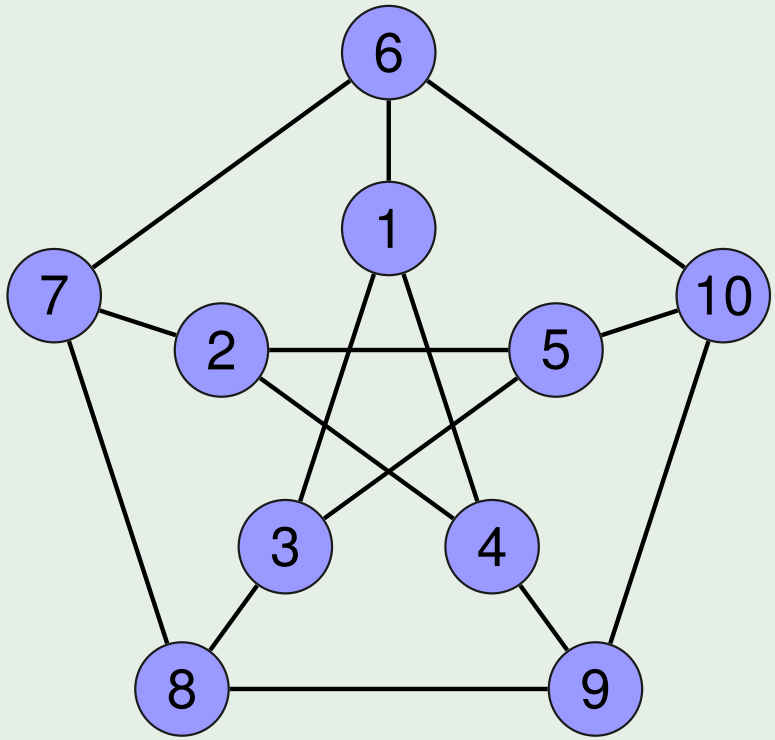
Definition

A graph is a set of vertices and a set of edges that connect pairs of different vertices (there is at most an edge between each pair of vertices).

Notation

Formally, a graph is a pair (V, E), where V is a finite set (of vertices) and E is a set of unordered pairs of vertices.

Example: Petersen's graph

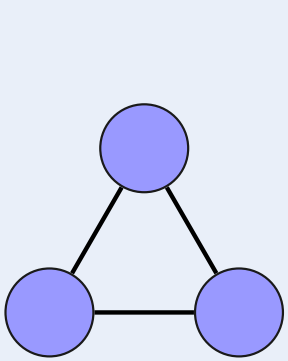


Formally, it is the pair (V, E) where

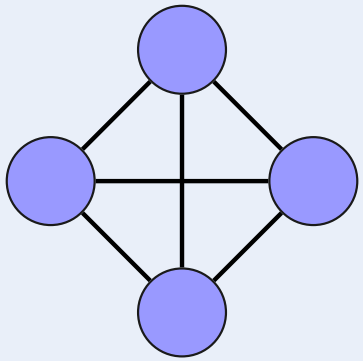
- $V = \{1, \ldots, 10\}$
- $E = \{\{1,3\},\{1,4\},\{1,6\},\{2,4\},\{2,5\},\{2,7\},\{3,5\},$ $\{3,8\},\{4,9\},\{5,10\},\{6,7\},\{6,10\},\{7,8\},\{8,9\},\{9,10\}\}$

Families of graphs

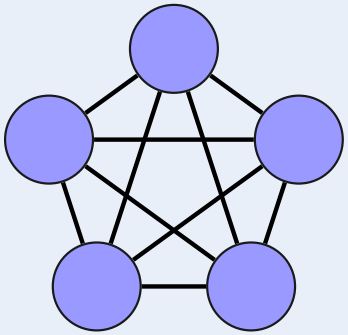
- **Complete:** K_i is the complete graph with i vertices.



K_3

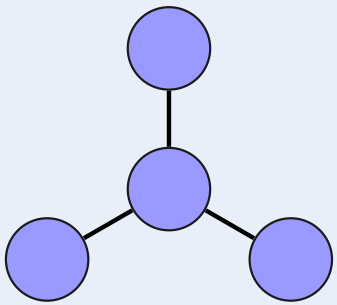


K_4

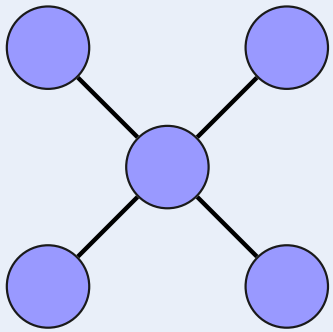


K_5

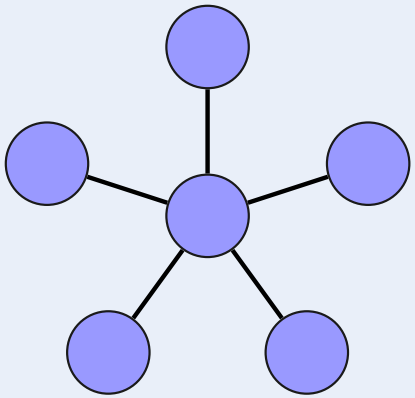
- **Stars:** S_i is the star with $i + 1$ vertices.



S_3



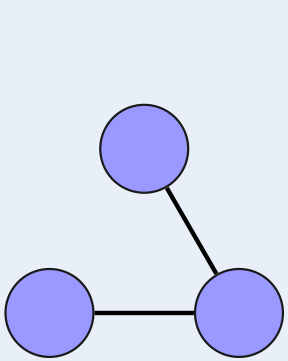
S_4



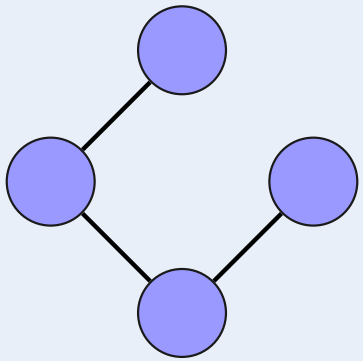
S_5

Families of graphs

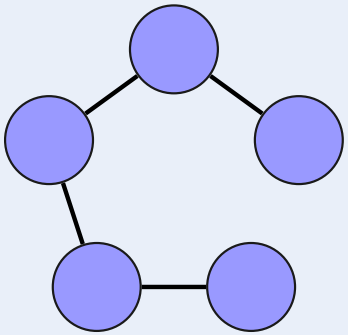
- **Paths:** P_i is the path with i vertices.



P_3

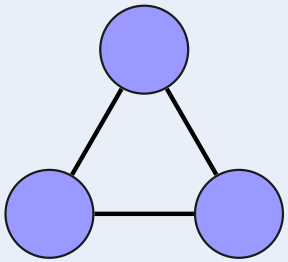


P_4

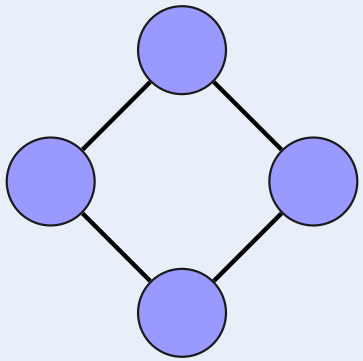


P_5

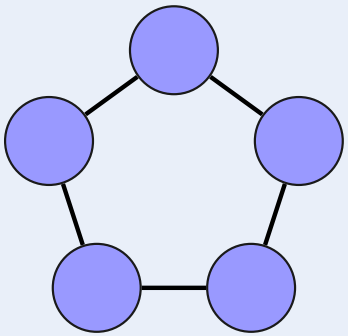
- **Cycles:** C_i is the cycles with i vertices.



C_3



C_4



C_5

Property

A graph with n vertices has at most $\frac{n(n-1)}{2}$ edges.

Proof

Each vertex can have an edge with $n-1$ vertices (but not with itself). Since each edge is counted twice, we get

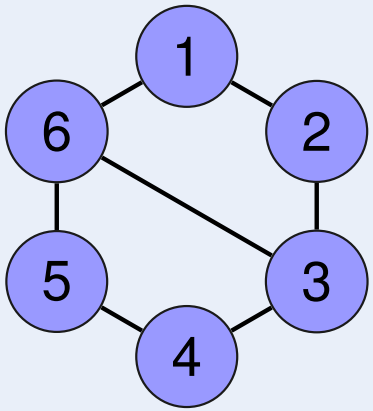
$$\frac{n(n-1)}{2}$$

different edges.

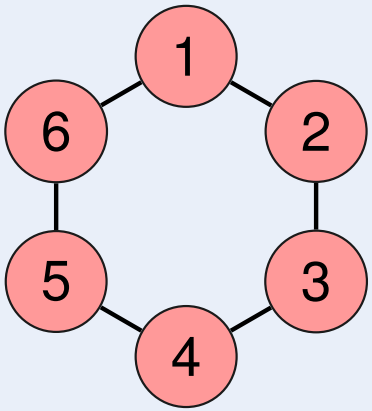
(Combinations of n elements chosen in groups of 2.)

Adjacency and subgraphs

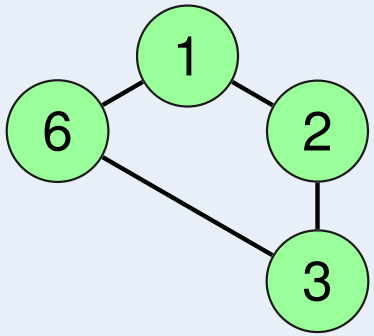
- two vertices u, v are adjacent if $\{u, v\}$ is an edge
- an edge $\{u, v\}$ is incident in u and v
- degree of a vertex u: number of incident edges in u
- a graph $H = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.
- a graph H is an induced subgraph of a graph G if H is a subgraph of G and contains all edges that G has among vertices of H.



Graph G



Subgraph of G



Induced subgraph of G

Paths and cycles

- A path in a graph is a sequence of vertices where each vertex (except for the first one) is adjacent toe its predecessor in the path.
- A simple path is a path with no repeated vertices or edges.
- A cycle is a simple path except in the initial and final vertex, that are the same.
- A graph is cyclic if it contains some cycle.

Observation

A graph G has

1. a simple path of k vertices if and only if P_k is a subgraph of G
2. a cycle of k vertices if and only if C_k is a subgraph of G

Paths and cycles

- A path in a graph is a sequence of vertices where each vertex (except for the first one) is adjacent toe its predecessor in the path.
- A simple path is a path with no repeated vertices or edges.
- A cycle is a simple path except in the initial and final vertex, that are the same.
- A graph is cyclic if it contains some cycle.

Observation

A graph G has

1. a simple path of k vertices if and only if P_k is a subgraph of G
2. a cycle of k vertices if and only if C_k is a subgraph of G

Connectivity

- A graph is connected if there is a path between any pair of vertices.
- A connected component of a graph is a connected induced subgraph that has not vertex adjacent to an external one.

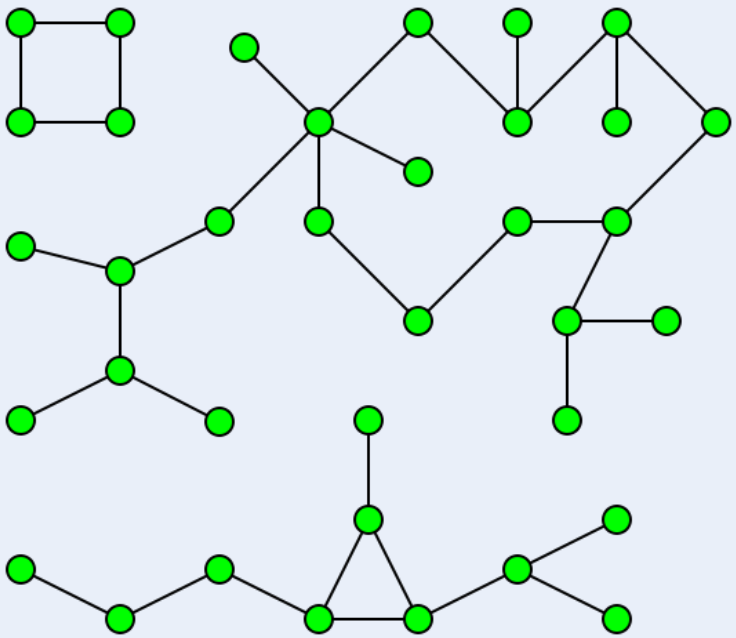


Figura: Cyclic graph with 3 connected components

Distance

- The distance between two vertices is the minimum number of edges in a path between them.
- The diameter of a graph is the maximum distance between all pairs of vertices of a graph.

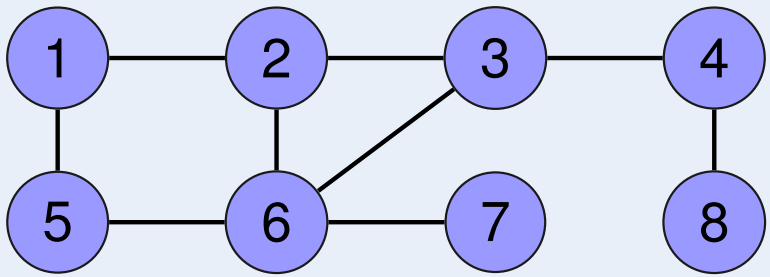
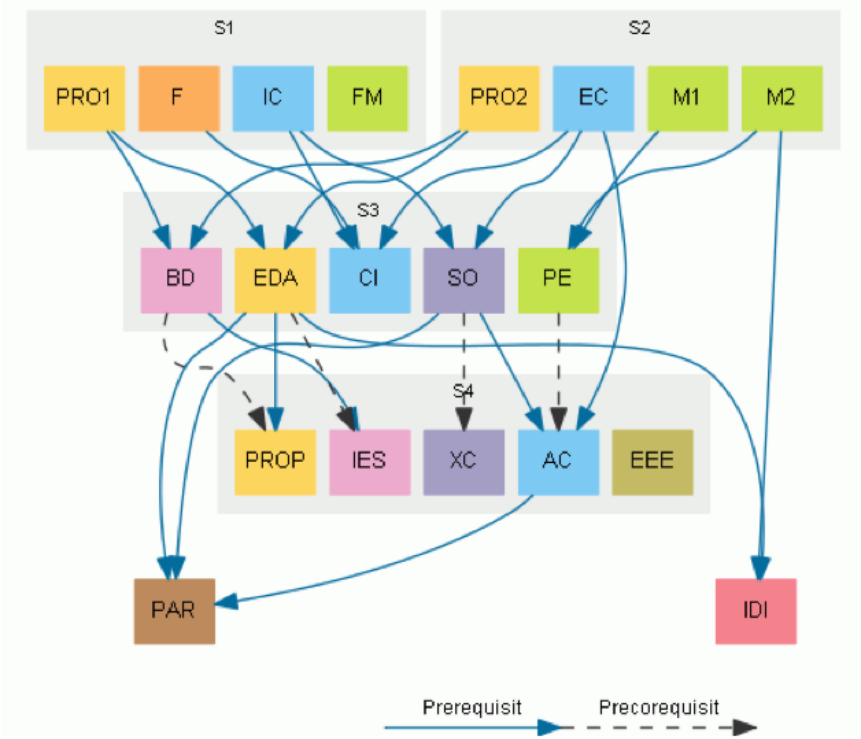


Figura: Distance between 4 and 5 is 3. The diameter of the graph is 4.

Directed and labeled graphs

Definition

- A directed graph or digraph is a pair (V, E), where
  - V is a finite set (of vertices) and
  - E is a set of ordered pairs of vertices (called edges or arcs).



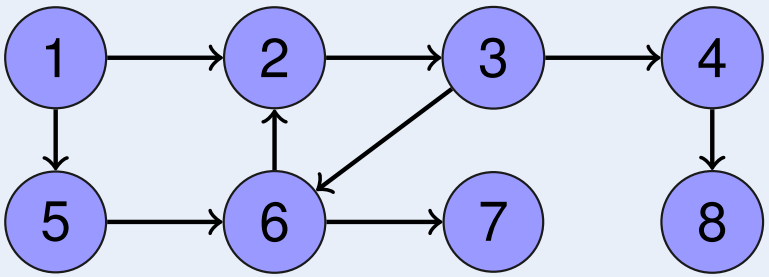
Directed graph of compulsory courses of the degree (with two types of arcs)

Directed and labeled graphs

The concepts of graph are easily adapted to digraphs.

Digraphs: degrees and distances

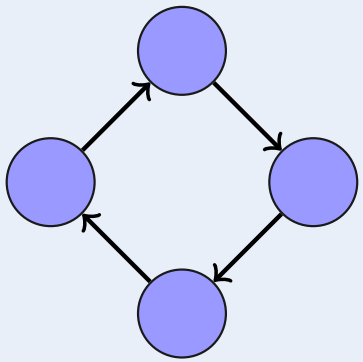
- We distinguish between indegree and outdegree.
- In a path (or directed path), all arcs go in the same direction.
- The distance between two vertices refers to directed paths.



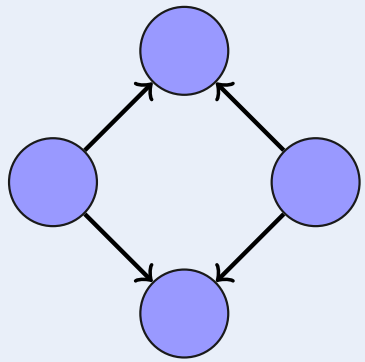
Vertex 2 has indegree 2 and outdegree 1.
The distance from 5 to 4 is 4; from 4 to 5, ∞.

Digraphs: connectivity

- A digraph is weakly connected (or connected) if the graph obtained by replacing arcs by edges is connected.
- A digraph is strongly connected if there is a directed path between any pair of vertices.



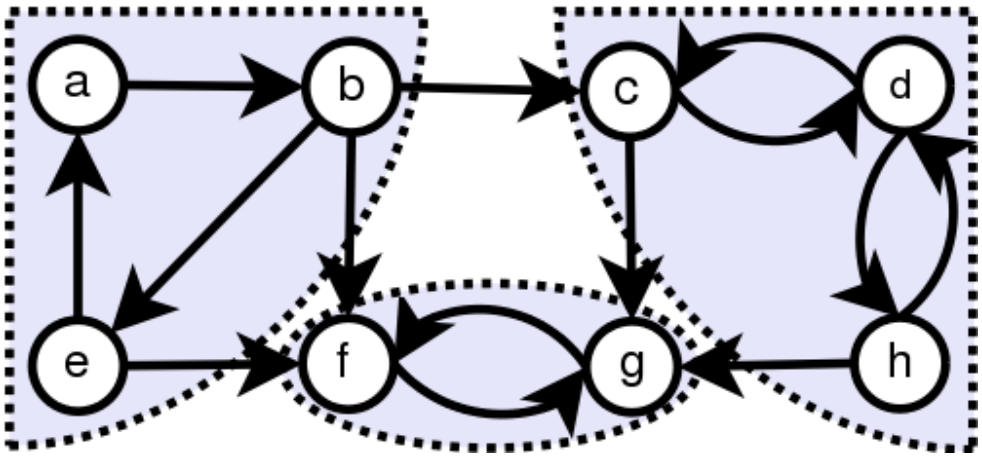
strongly connected



weakly connected

Digraphs: connectivity

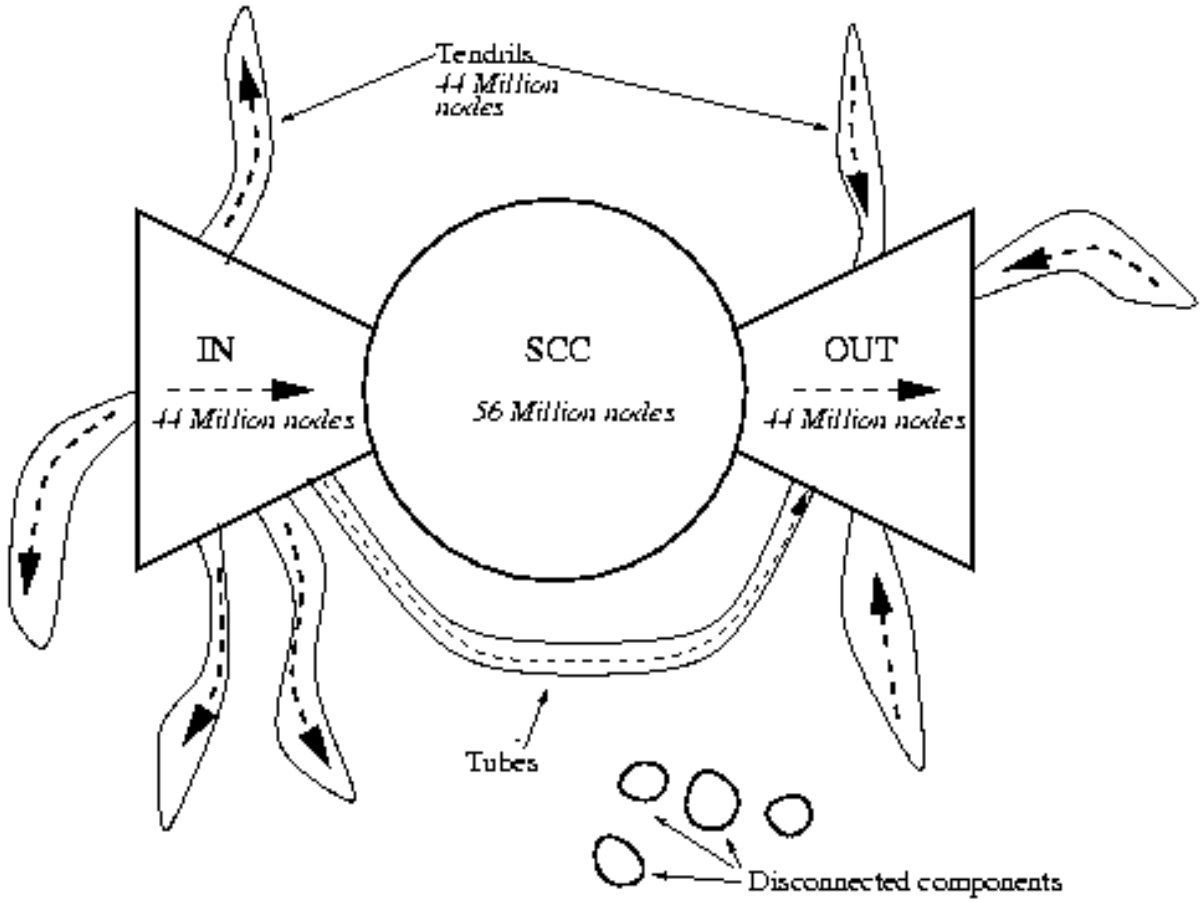
- Strongly connected components of a digraph are the maximal subgraphs that are strongly connected.



Digraph with 3 connected components

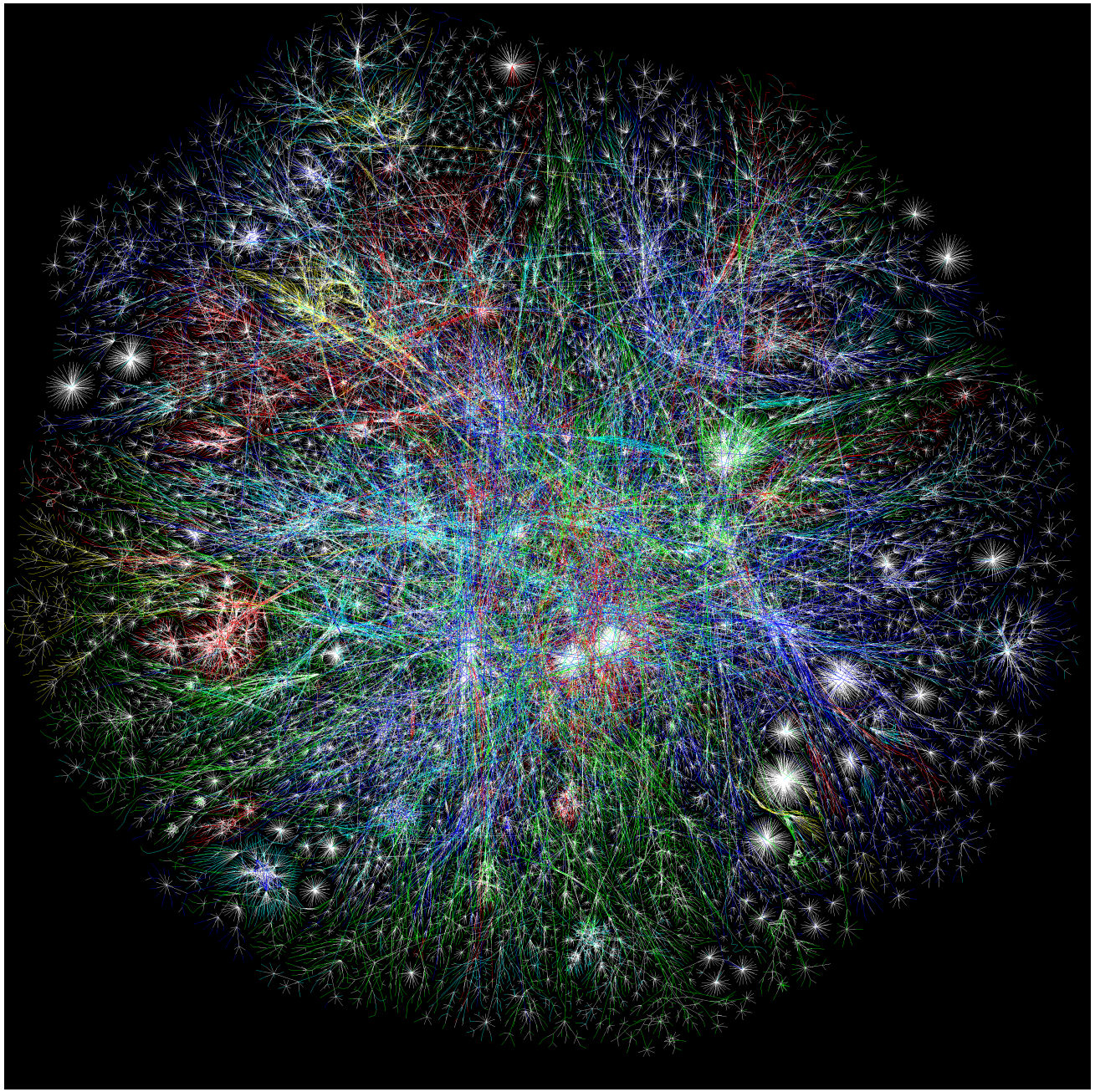
Internet graph (year 2000, Broder, Kumar et al.)

Websites are vertices; links are edges.



SCC: *(giant) strongly connected component*
diameter of the SCC: 28

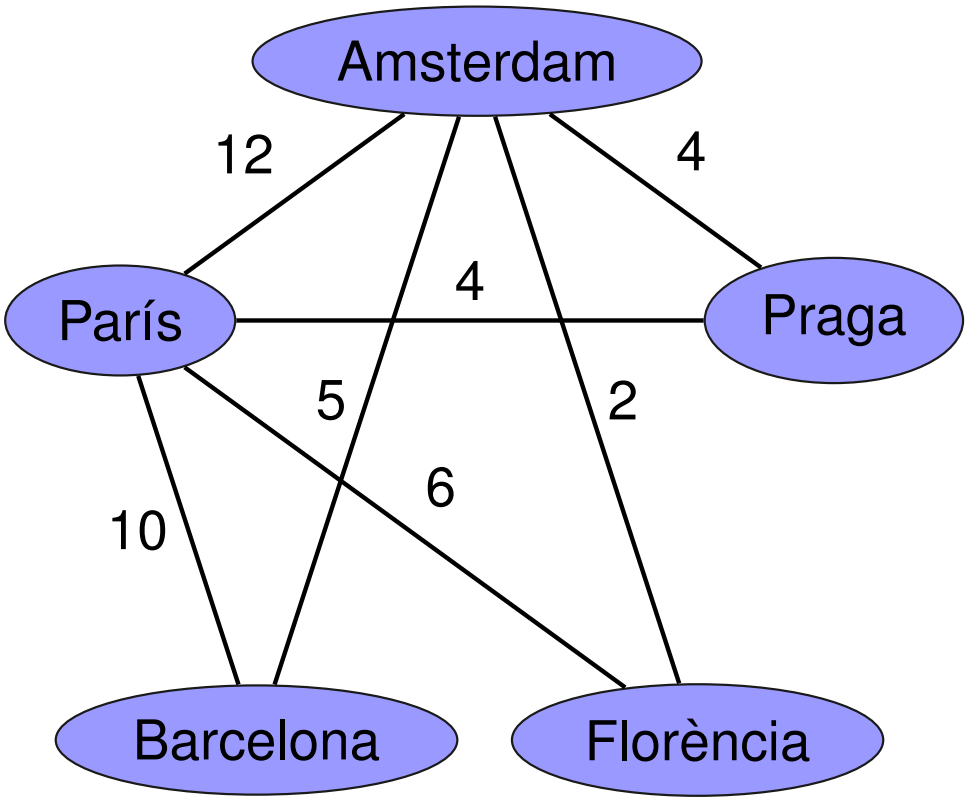
Internet graph



Directed and labeled graphs

Definition

A labeled graph (directed or undirected) is a graph where edges have associated labels. It is sometimes also called *weighted* graph.



Number of daily flights with Air France and KLM

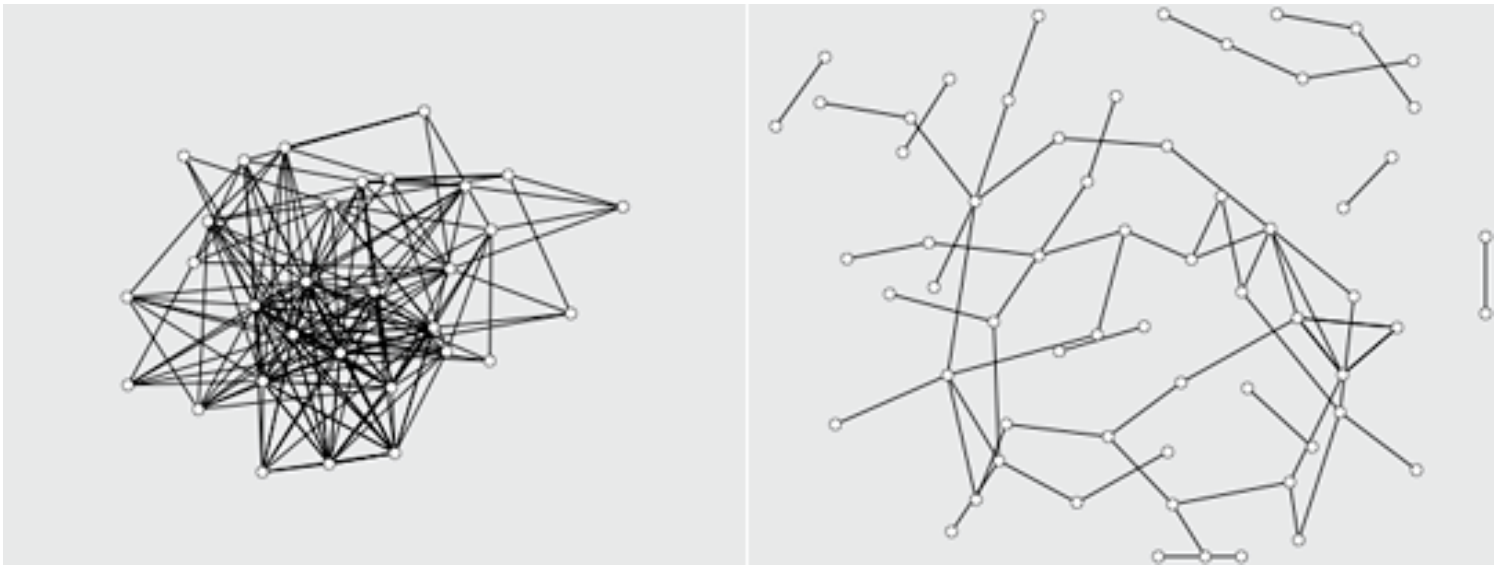
Directed and labeled graphs

All 4 combinations are useful:

- Undirected unlabeled graphs
- Undirected labeled graphs
- Directed unlabeled graphs
- Directed labeled graphs

Representations

The representation of graphs will depend on their edge density.



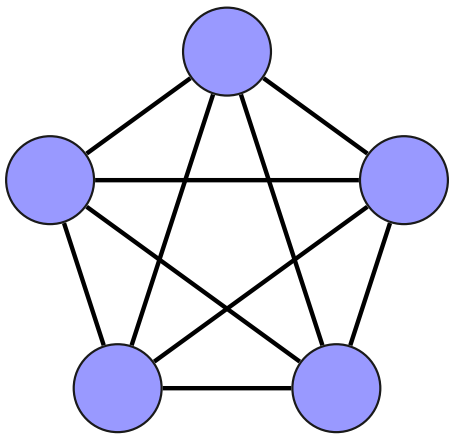
How can we define the density of a graph?

Density

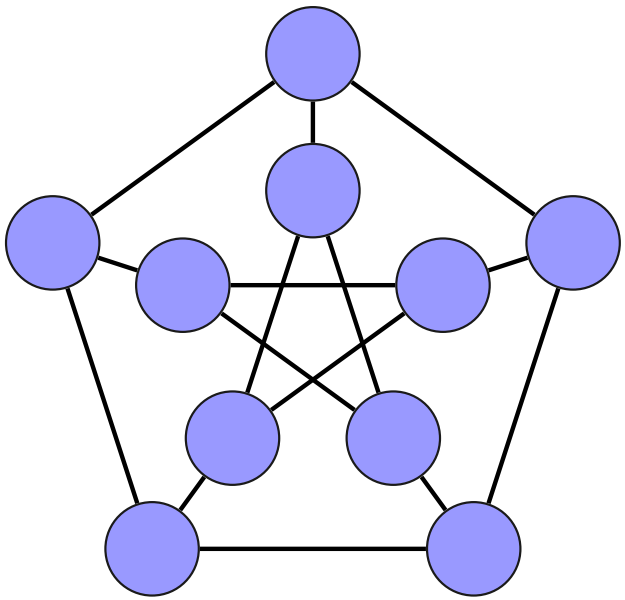
The density of a graph with n vertices and m edges is

$$D = \frac{2m}{n(n-1)}$$

We have seen that the maximum number of edges in a graph with n vertices is $\frac{n(n-1)}{2}$. Hence, $0 \leq D \leq 1$.



D=1



D=1/3

Density

A graph with n vertices and m edges is dense is $m \approx n^2$ (i.e., if D is close to 1). Otherwise, it is sparse.

The concept is more formal when we consider families of graphs. For example:

- Complete graphs (K_n) are dense because $D = 1$ for all n.
- Cycles (C_n) are sparse because $D = 2/(n-1)$ and density tends to 0 when n tends to ∞.

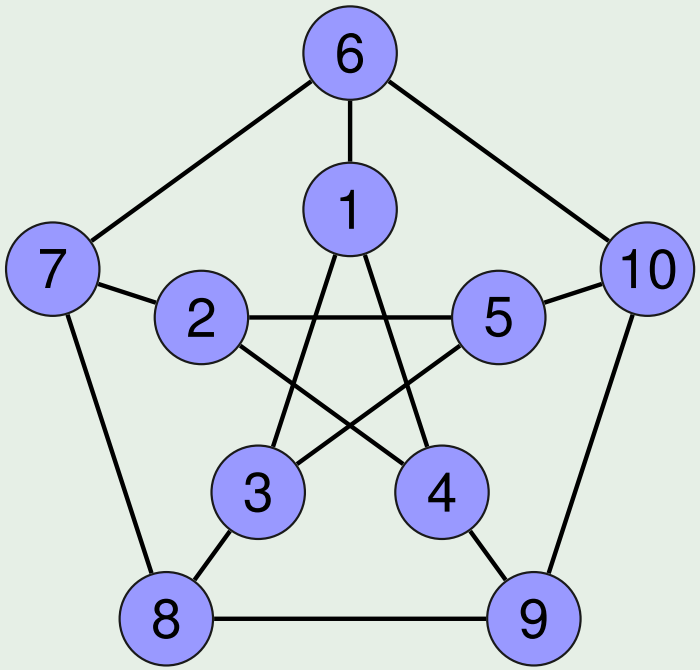
Representations

Adjacency matrix / undirected graphs

The adjacency matrix of an undirected graph $G = (V, E)$ is a matrix M with $n \times n$ boolean values such that

$$M_{ij} = \begin{cases} 1, & \text{if } \{i, j\} \in E \\ 0, & \text{if } \{i, j\} \notin E \end{cases}$$

Example



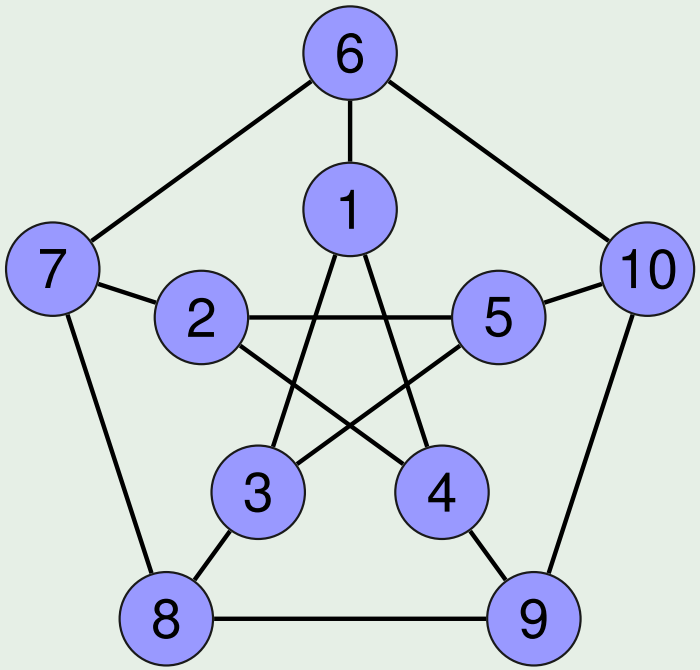
1	0	0	1	1	0	1	0	0	0
2	0	0	0	1	1	0	1	0	0	0
3	1	0	0	1	0	0	0	1	0	0
4	1	1	0	0	0	0	0	0	1	0
5	0	1	1	0	0	0	0	0	0	1
6	1	0	0	0	0	0	1	0	0	1
7	0	1	0	0	0	1	0	1	0	0
8	0	0	1	0	0	0	1	0	1	0
9	0	0	0	1	0	0	0	1	0	1
10	0	0	0	0	1	1	0	0	1	0

Adjacency matrix / undirected graphs

The adjacency matrix of an undirected graph $G = (V, E)$ is a matrix M with $n \times n$ boolean values such that

$$M_{ij} = \begin{cases} 1, & \text{if } \{i,j\} \in E \\ 0, & \text{if } \{i,j\} \notin E \end{cases}$$

Example



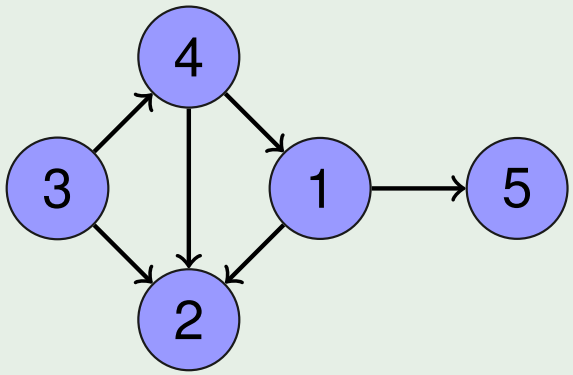
1		0	1	1	0	1	0	0	0	0
2	0		0	1	1	0	1	0	0	0
3	1	0		1	0	0	0	1	0	0
4	1	1	0		0	0	0	0	1	0
5	0	1	1	0		0	0	0	0	1
6	1	0	0	0	0		1	0	0	1
7	0	1	0	0	0	1		1	0	0
8	0	0	1	0	0	0	1		1	0
9	0	0	0	1	0	0	0	1		1
10	0	0	0	0	1	1	0	0	1	

Adjacency matrix / directed graphs

The adjacency matrix of a directed graph $G = (V, E)$ is a matrix M with $n \times n$ boolean values such that

$$M_{ij} = \begin{cases} 1, & \text{if } (i,j) \in E \\ 0, & \text{if } (i,j) \notin E \end{cases}$$

Example



$$\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Adjacency matrix: data structure

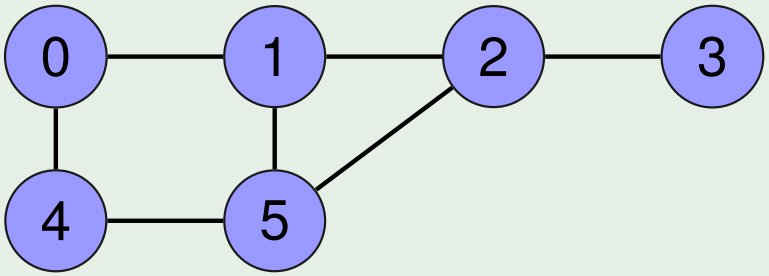
Adjacency matrices can be implemented as a vector of vectors.

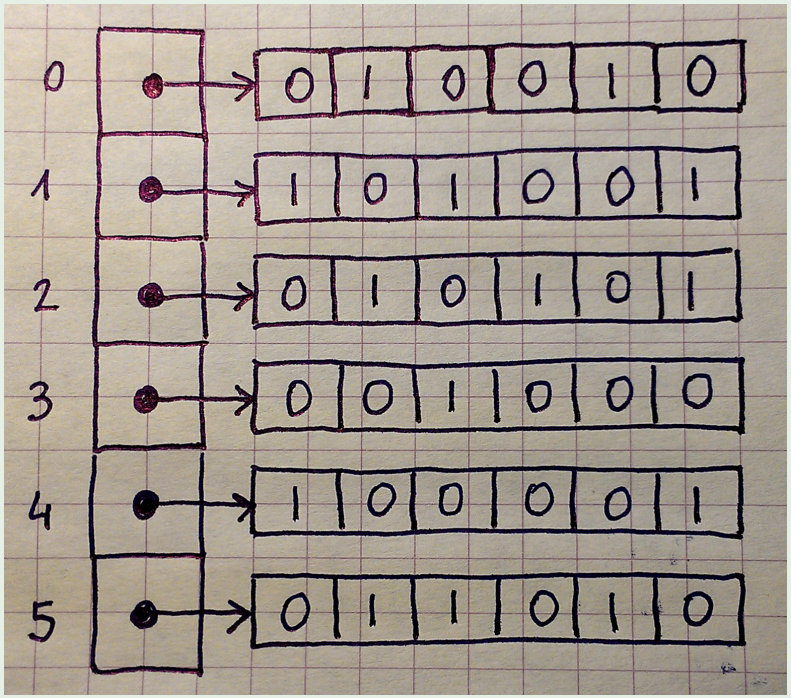
```
typedef vector< vector<bool> > graph;
```

If `g` is of type `graph` (n vertices):

- Space cost is $\Theta(n^2)$.
- Initialization of `g` is $\Theta(n^2)$.
- Vector `g[i]` contains the information about the neighbors of `i`.
- Processing vertices adjacent to `i` is $\Theta(n)$.
- If `g` is undirected, then `g[i][j] == g[j][i]` (information is duplicated).
- Traversing all edges is $\Theta(n^2)$.
- The adjacency matrix is appropriate for dense graphs.

Example





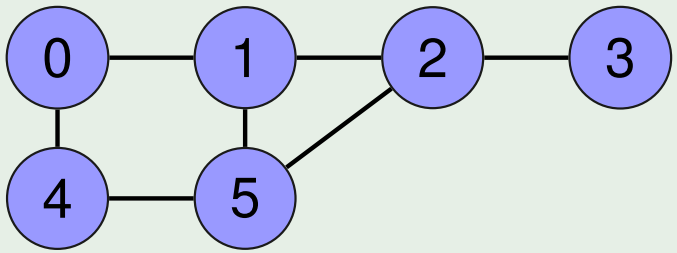
Representations

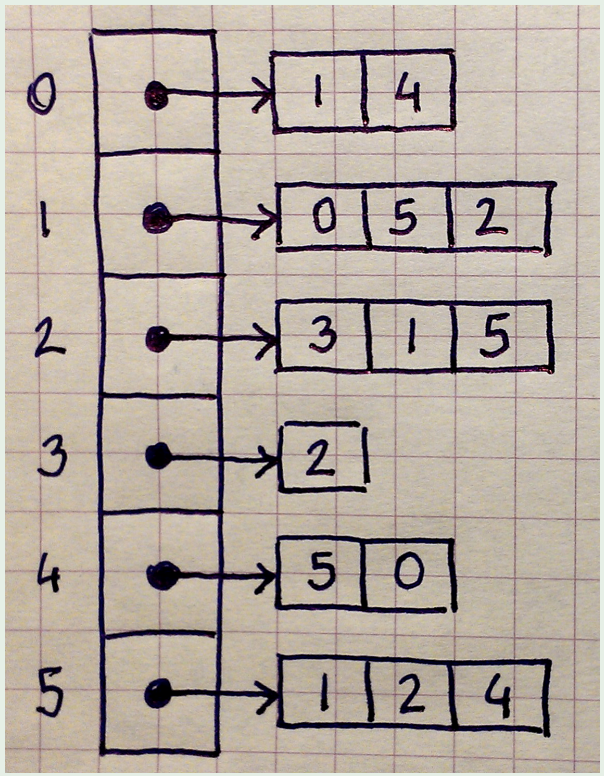
Adjacency lists

Each vertex points to the list of adjacent vertices.

```
typedef vector< vector<int> > graph;
```

Example





Representations

Adjacency lists

Each vertex points to the list of adjacent vertices

```
typedef vector< vector<int> > graph;
```

If `g` is of type `graph` (n vertices and m edges):

- Space cost is $\Theta(n+m)$.
- Initialization is $\Theta(n)$.
- Adjacent vertices to `i` are linked with no particular order.
- Processing vertices adjacent to `i` is $\mathcal{O}(m)$.
- If `g` is undirected, information is duplicated.
- Traversing all edges is $\Theta(n+m)$.
- Adjacency lists are appropriate for sparse graphs.

Cost of operations

n: number of vertices / m: number of edges

operations	adjacency matrix	adjacency lists
space	$\Theta(n^2)$	$\Theta(n+m)$
create	$\Theta(n^2)$	$\Theta(n)$
add vertex	$\Theta(n)$	$\Theta(1)$
add edge	$\Theta(1)$	$\mathcal{O}(n)$
remove edge	$\Theta(1)$	$\mathcal{O}(n)$
consult vertex	$\Theta(1)$	$\Theta(1)$
consult edge	$\Theta(1)$	$\mathcal{O}(n)$
is v isolated?	$\Theta(n)$	$\Theta(1)$
successors*	$\Theta(n)$	$\mathcal{O}(n)$
predecessors*	$\Theta(n)$	$\mathcal{O}(m)$
adjacents+	$\Theta(n)$	$\mathcal{O}(n)$

* only in directed graphs

+ only in undirected graphs

Chapter 5. Graphs

Depth-first search

Depth-First Search (DFS) answers the question:

Which parts of the graph are accessible from a given vertex?

An algorithm con only check the adjacencies: whether it is possible to go from one vertex to another one. The situation is similar to the exploration of a maze.

Depth-first search

To explore a maze, we need chalk and a rope:

- The chalk prevents from walking in circles (we know what we have visited).
- The rope allows one to go back and see parts not yet visited.

How could this be implemented?

- the chalk is a vector of booleans
- the rope is a stack

Depth-first search

To explore a maze, we need chalk and a rope:

- The chalk prevents from walking in circles (we know what we have visited).
- The rope allows one to go back and see parts not yet visited.

How could this be implemented?

- the chalk is a vector of booleans
- the rope is a stack

Depth-first search

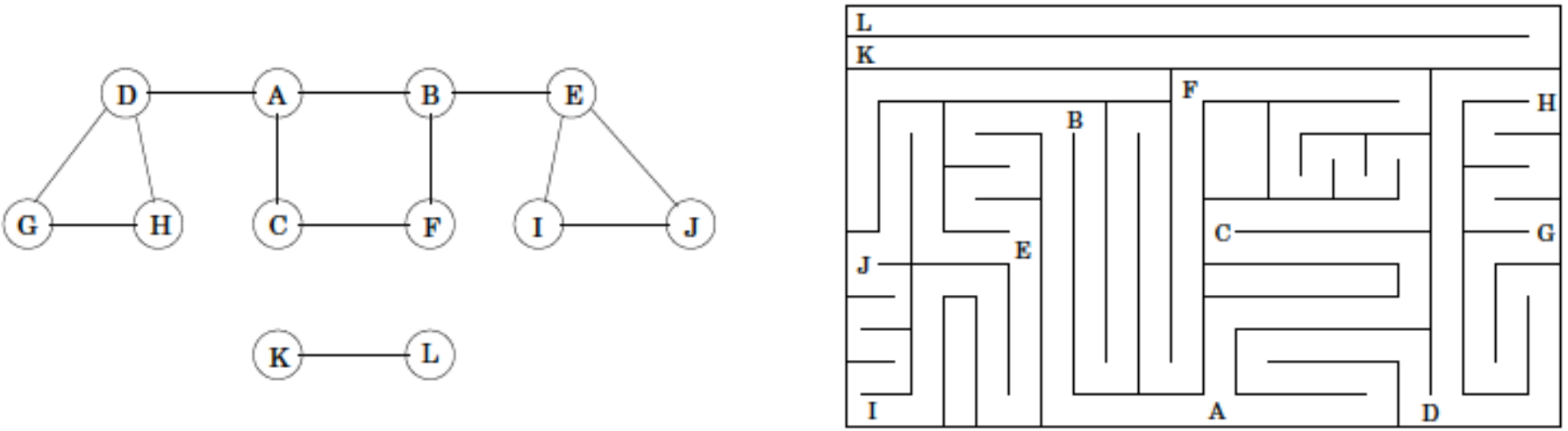


Figura: To convert a maze into a graph: we mark maze sectors (vertices) and we join them (via edges) if they are neighbors (example from *Algorithms*, S. Dasgupta, C.H. Papadimitriou and U.V. Vazirani)

Depth-first search

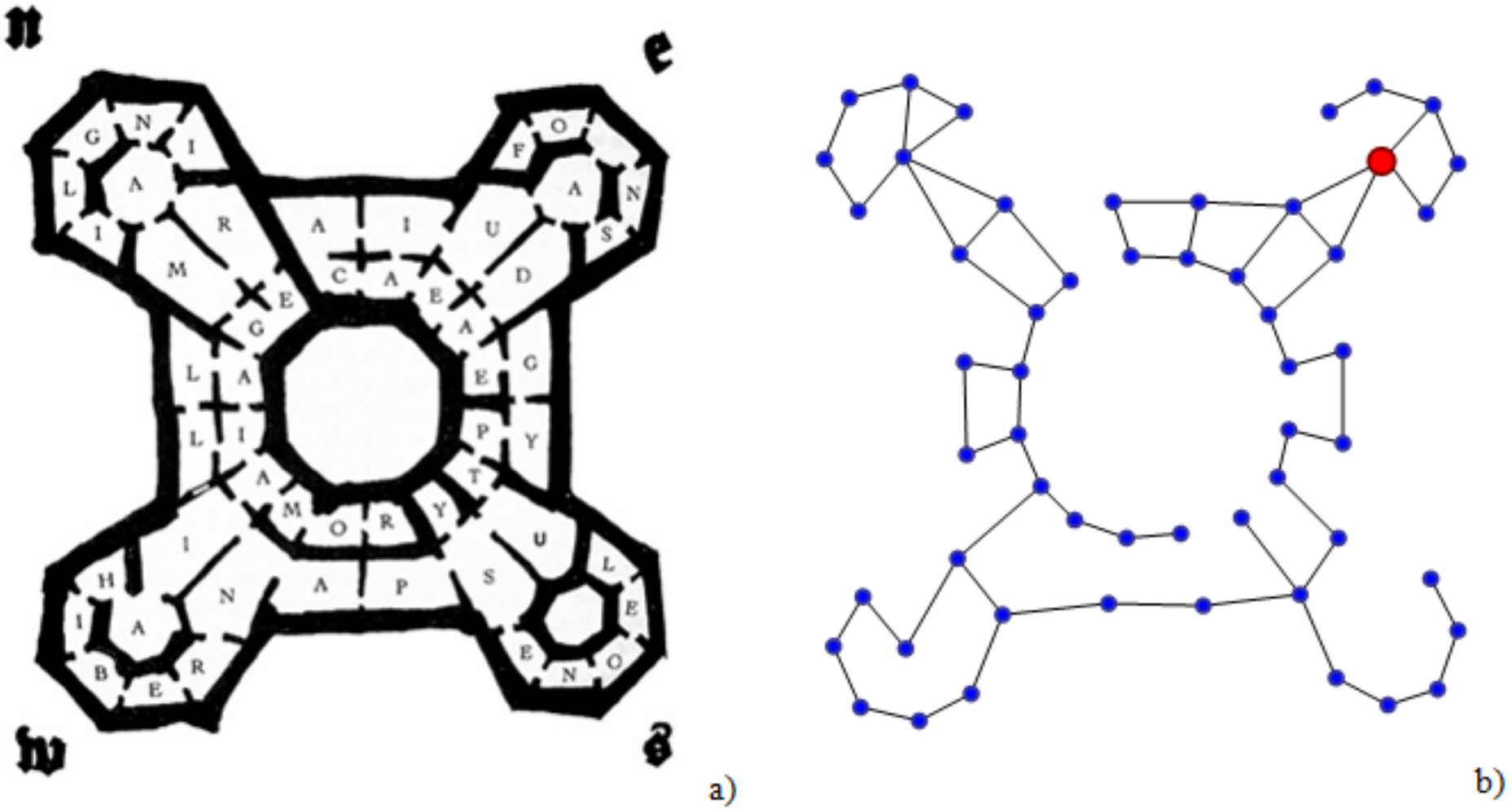


Figura: Library maze from *The name of the rose*, U. Eco

Depth-first search

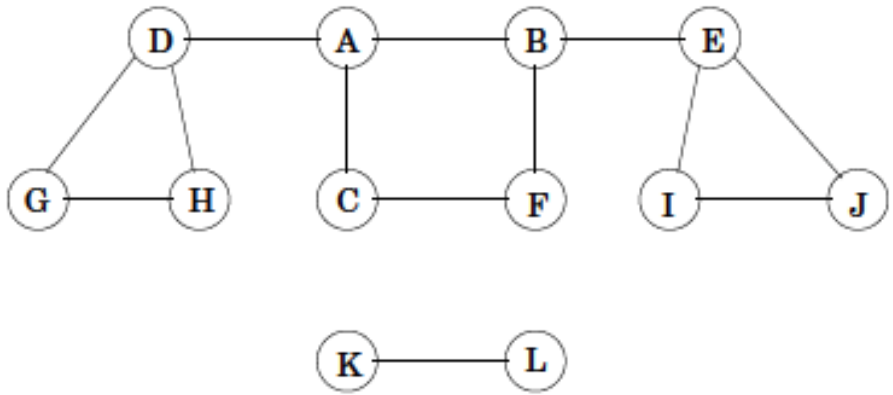
Recursive depth-first search (from a given vertex)

Visit all vertices reachable from a given vertex `u`.

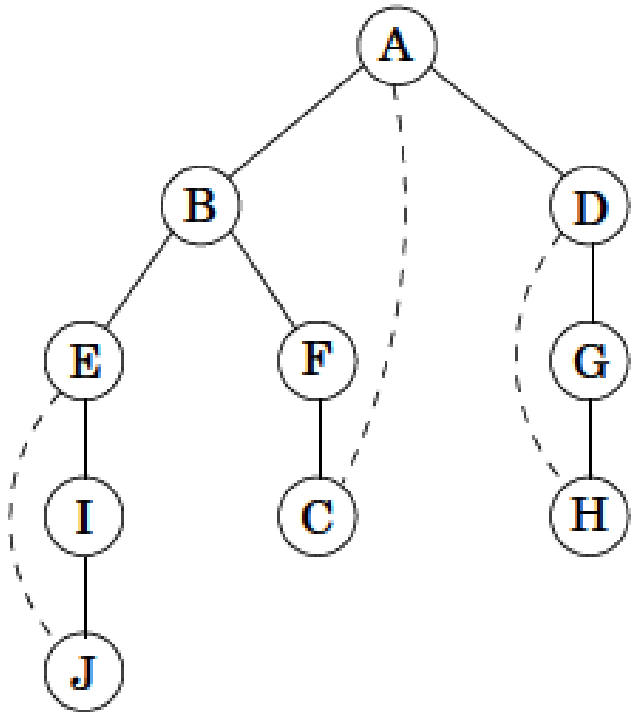
```
void dfs_rec (const graph& G, int u,
              vector<boolean>& vis, list<int>& L) {
    if (not vis[u]) {
        vis[u] = true; L.push_back(u);
        for (int i = 0; i < G[u].size(); ++i) {
            dfs_rec(G, G[u][i], vis, L);
}   }   }
```

Depth-first search

Previous example :
(*Algorithms.* Dasgupta et al.)



Depth-first search from vertex A:
(vertices are visited in alphabetical order)



Depth-first search

Recursive depth-first search

Depth-first search of the whole (possibly unconnected) graph

```
list<int> dfs_rec (const graph& G) {
    int n = G.size();
    list<int> L;
    vector<boolean> vis(n, false);
    for (int u = 0; u < n; ++u) {
        dfs_rec(G, u, vis, L);
    }
    return L;
}
```

Depth-first search

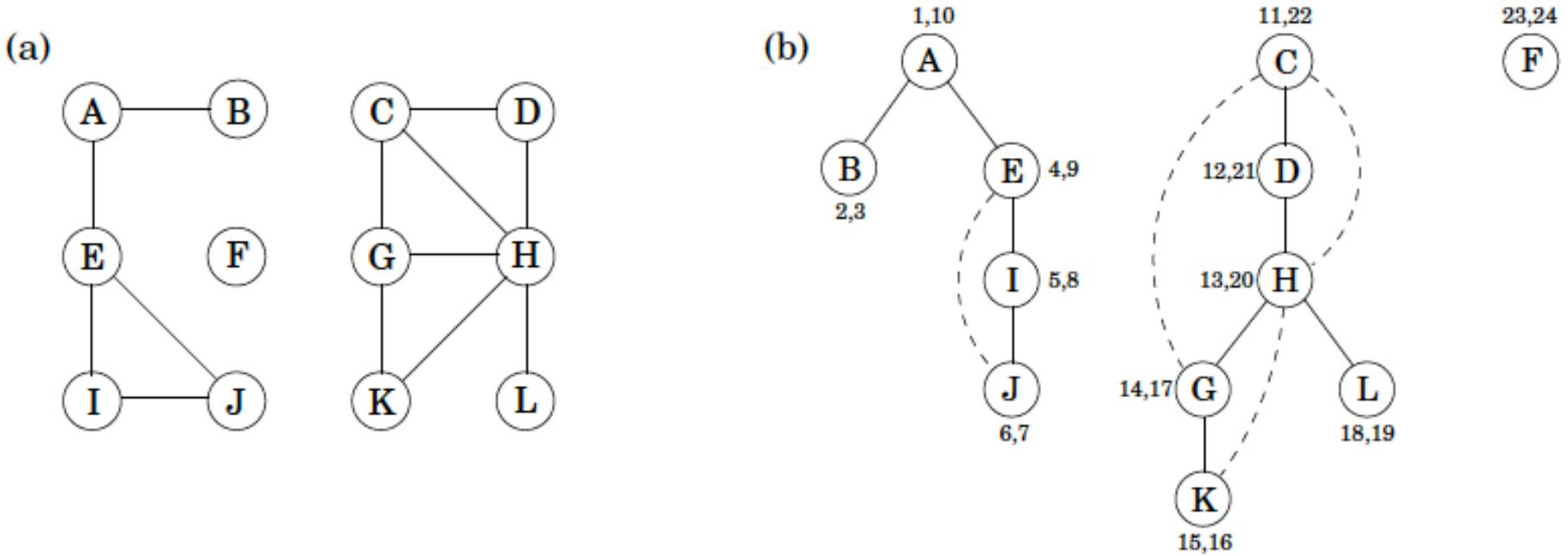


Figura: Depth-first search in an undirected and unconnected graph (*Algorithms.* Dasgupta et al.)

Depth-first search

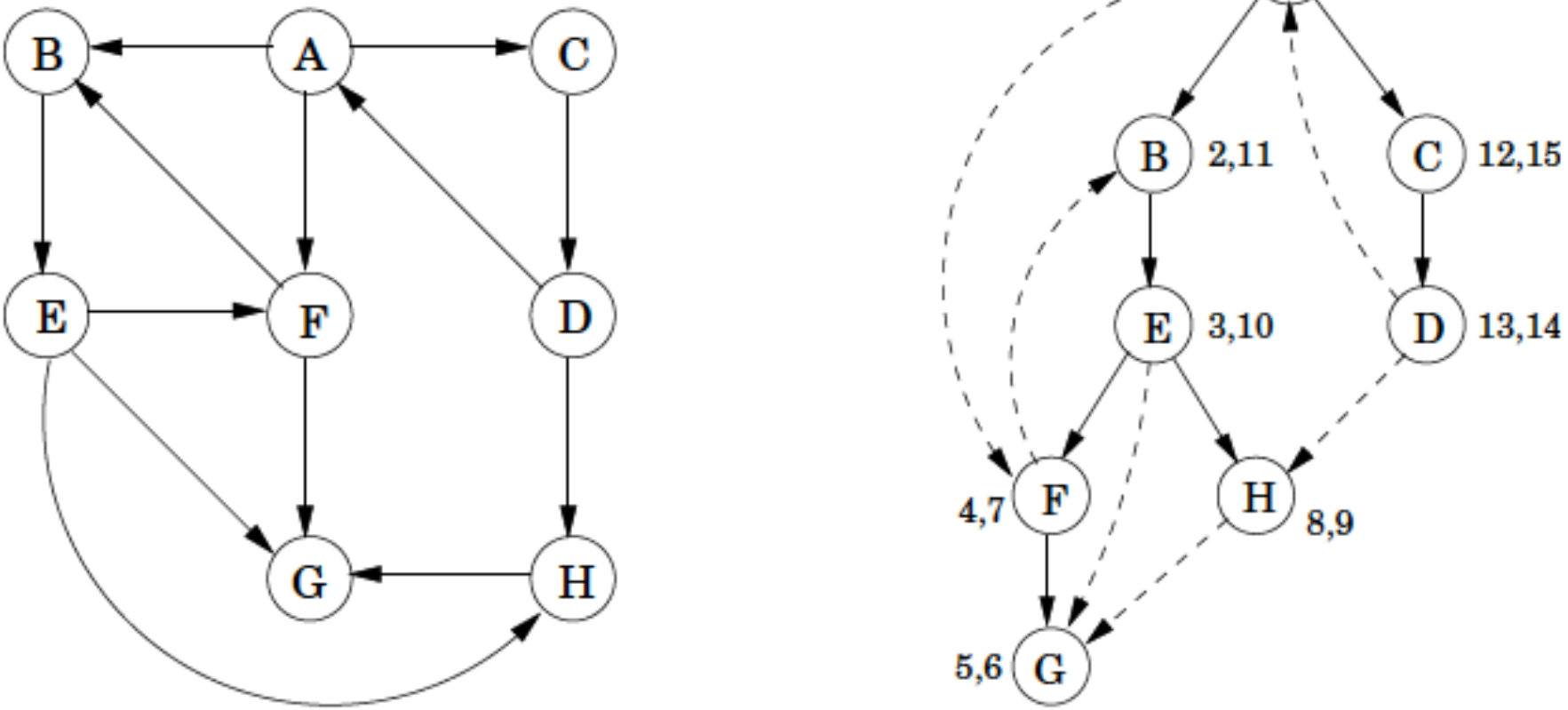


Figura: Depth-first search in a directed graph (*Algorithms.* Dasgupta et al.)

Depth-first search

Cost of depth-first search from a vertex

```
void dfs_rec (const graph& G, int u,
              vector<boolean>& vis, list<int>& L) {
    if (not vis[u]) {
        vis[u] = true; L.push_back(u);
        for (int i = 0; i < G[u].size(); ++i) {
            dfs_rec(G, G[u][i], vis, L);
}   }   }
```

- A fixed amount of work is done (first 2 lines): $\Theta(1)$
- Recursive calls to the neighbors are made. In total,
  - each vertex of the connected component is marked once
  - each edge $\{i, j\}$ is visited twice: from `i` and from `j`

  Hence, $\mathcal{O}(n+m)$.

Total cost: $\mathcal{O}(n+m)$.

Depth-first search

Cost of depth-first search

```
list<int> dfs_rec (const graph& G) {
    int n = G.size();
    list<int> L;
    vector<boolean> vis(n, false);
    for (int u = 0; u < n; ++u)
        dfs_rec(G, u, vis, L);
    return L; }
```

If graph `G` has

- k connected components (c.c.),
- $n = \sum_{i=1}^{k} n_i$ vertices (c.c. i has n_i vertices) and
- $m = \sum_{i=1}^{k} m_i$ edge (c.c. i has m_i edges),

then the cost is

$$\sum_{i=1}^{k} \Theta(n_i + m_i) = \Theta(\sum_{i=1}^{k} n_i + \sum_{i=1}^{k} m_i) = \Theta(n + m).$$

Depth-first search

Iterative depth-first search

```
list<int> dfs_ite (const graph& G) {
    int n = G.size();
    list<int> L;
    stack<int> S;
    vector<bool> vis(n, false);

    for (int u = 0; u < n; ++u) {
        S.push(u);
        while (not S.empty()) {
            int v = S.top(); S.pop();
            if (not vis[v]) {
                vis[v] = true; L.push_back(v);
                for (int i = 0; i < G[v].size(); ++i) {
                    S.push(G[v][i]);
    }   }   }   }
    return L;
}
```

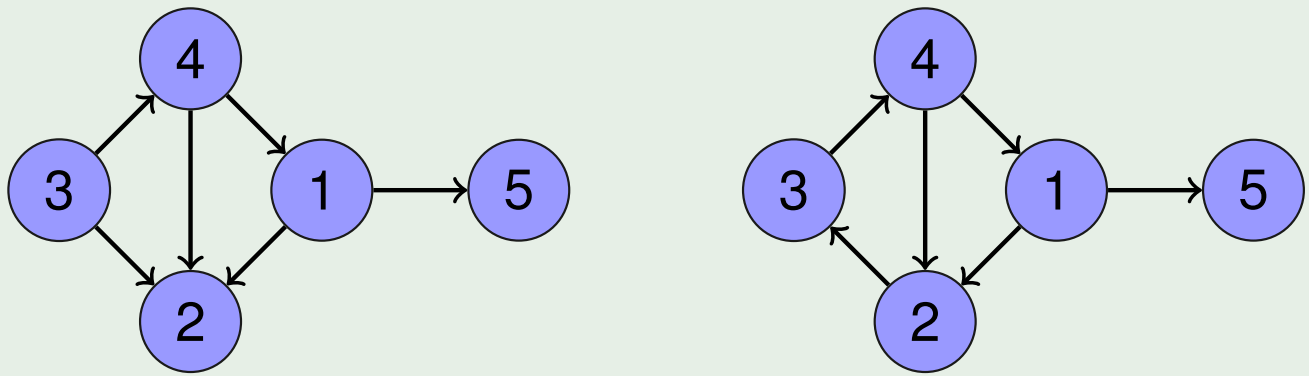
Topological sorting

Definition

A directed acyclic graph is usually called dag.

Dags express precedences or causalities and are used in a variety of areas (computer science, economy, medicine, ...)

Example

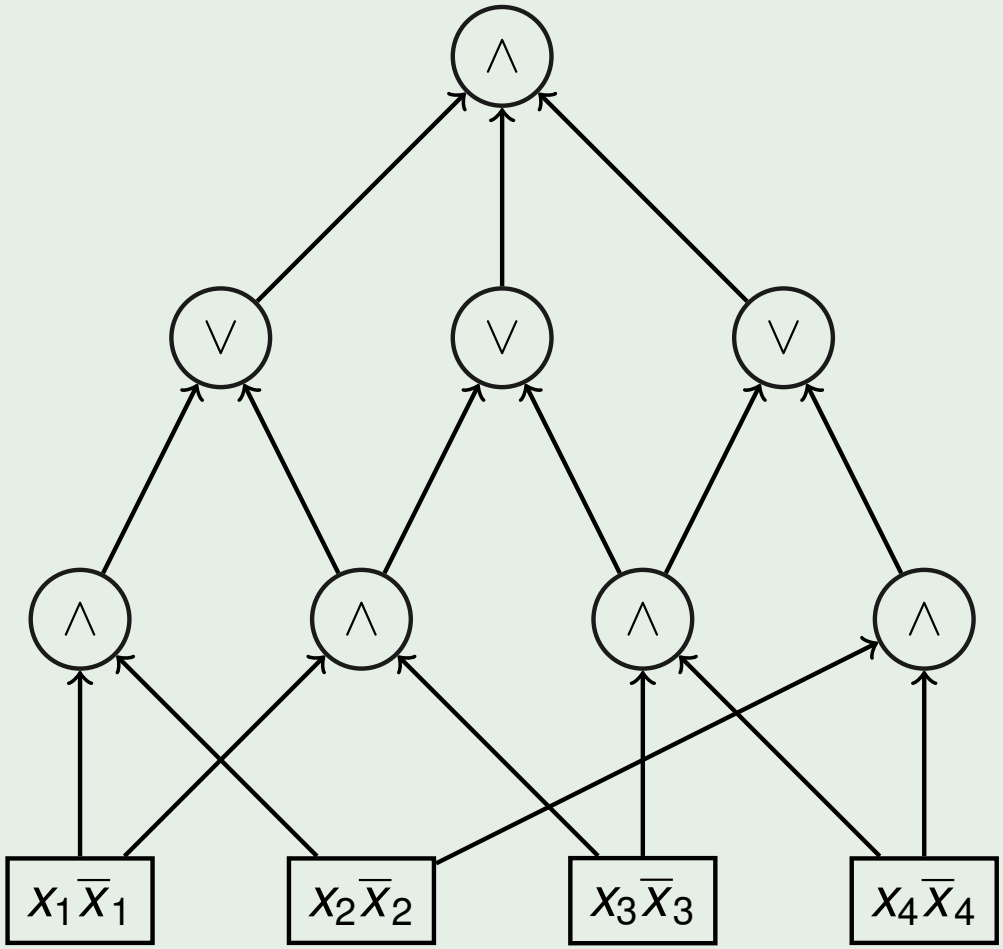


The digraph on the left is a dag; the one in the right is not.

Topological sorting

Example

A circuit is also a dag.



Topological sorting

Definition

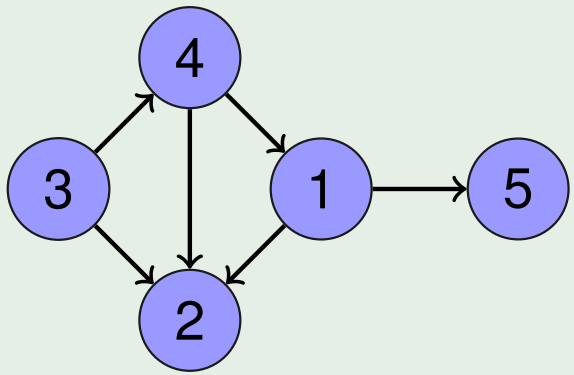
A topological sorting of a dag $G = (V, E)$ is a sequence

$$v_1, v_2, v_3, \ldots, v_n$$

such that $V = \{v_1, \ldots, v_n\}$ and if $(v_i, v_j) \in E$, then $i < j$.

Example

A dag can have more than one topological sorting.



3,4,1,2,5, as well as 3,4,1,5,2 are topological sortings.

Topological sorting

Definition

A topological sorting of a dag $G = (V, E)$ is a sequence

$$v_1, v_2, v_3, \ldots, v_n$$

such that $V = \{v_1, \ldots, v_n\}$ and if $(v_i, v_j) \in E$, then $i < j$.

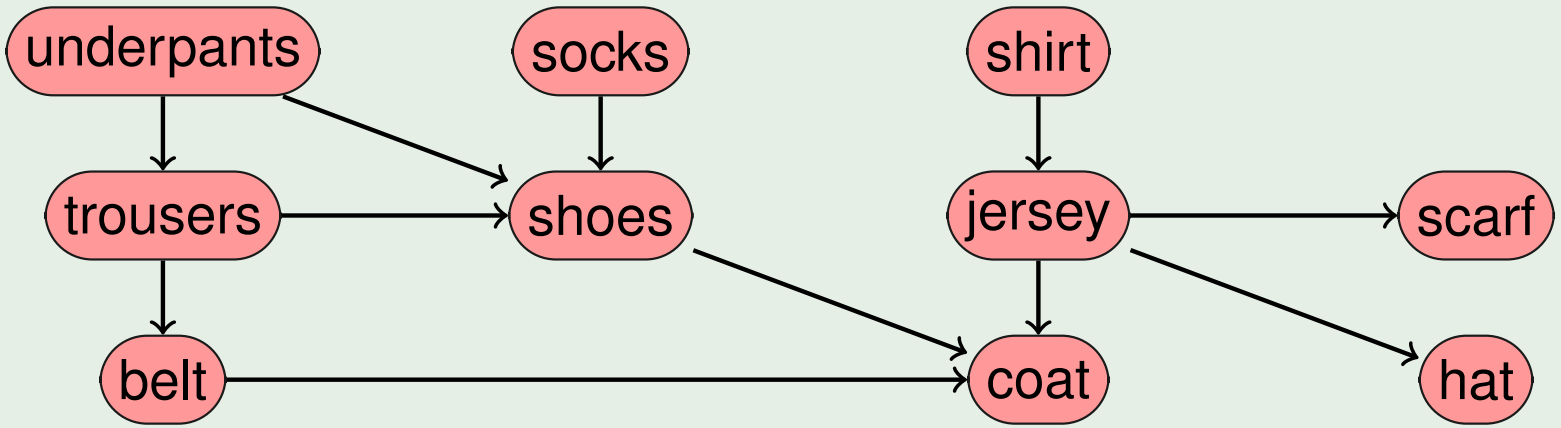
Exercises

1. Give examples of dags with n vertices such that they have
   - a unique topological sorting
   - $(n-1)!$ topological sortings
2. Explain why if a dag with n vertices has $n!$ topological sortings, it consists of n isolated vertices.

Topological sorting

Example

Male winter clothes



Possible sortings:

- underpants, socks, trousers, shirt, belt, jersey, shoes, coat, scarf, hat
- socks, shirt, underpants, jersey, trousers, belt, scarf, hat, shoes, coat

Topological sorting

Topological sorting problem

Given a dag, find a topological sorting.

Algorithm (greedy)

Repeat until no vertices are left:

1. Find a vertex u with indegree 0.
2. Write u and remove it from the graph.

Cost

If the graph has n vertices

- look for a vertex with indegree 0 costs $\Theta(n)$
- previous step is repeated n times

Total cost: $\Theta(n^2)$.

Topological sorting

Topological sorting problem

Given a dag, find a topological sorting.

Algorithm (greedy)

Repeat until no vertices are left:

1. Find a vertex u with indegree 0.
2. Write u and remove it from the graph.

Cost

If the graph has n vertices

- look for a vertex with indegree 0 costs $\Theta(n)$
- previous step is repeated n times

Total cost: $\Theta(n^2)$.

Topological sorting

Topological sorting problem

Given a dag, find a topological sorting.

Algorithm (greedy)

Repeat until no vertices are left:

1. Find a vertex u with indegree 0.
2. Write u and remove it from the graph.

Cost

If the graph has n vertices

- look for a vertex with indegree 0 costs $\Theta(n)$
- previous step is repeated n times

Total cost: $\Theta(n^2)$.

Topological sorting

The cost can be improved if we search in an efficient way for a vertex with indegree 0. We will need:

- A vector to store the indegree of each vertex.
- A structure (stack or queue) containing all vertices of indegree 0.

We initialize a stack with all vertices of indegree 0. While it is not empty:

1. We remove a vertex from the stack, write it and adjust the indegrees.
2. We push onto the stack new vertices with indegree 0.

Topological sorting

The cost can be improved if we search in an efficient way for a vertex with indegree 0. We will need:

- A vector to store the indegree of each vertex.
- A structure (stack or queue) containing all vertices of indegree 0.

We initialize a stack with all vertices of indegree 0. While it is not empty:

1. We remove a vertex from the stack, write it and adjust the indegrees.
2. We push onto the stack new vertices with indegree 0.

Topological sorting

Initialization of the vector and the stack for a graph with n vertices and m edges. Cost $\Theta(n+m)$.

```
list<int> topological_sorting (graph& G) {
    int n = G.size();
    vector<int> indegree(n, 0);
    for (int u = 0; u < n; ++u) {
        for (int i = 0; i < G[u].size(); ++i) {
            ++indegree[G[u][i]];
    }  }

    stack<int> S;
    for (int u = 0; u < n; ++u) {
        if (indegree[u] == 0) {
            S.push(u);
    }  }
```

Topological sorting

Main loop.

```
    list<int> L;
    while (not S.empty()) {
        int u = S.top(); S.pop();
        L.push_back(u);
        for (int i = 0; i < G[u].size(); ++i) {
            int v = G[u][i];
            if (--indegree[v] == 0) {
                S.push(v);
    }   }   }
    return L;
}
```

We visit

- each vertex once
- each edge once

If the graph is given as adjacency lists, the cost is $\Theta(n+m)$.

Topological sorting

Exercise

Given the following graph, compute the evolution of the vector **indegree**, the stack **S** and the list **L**.