Chapter 5. Graphs

Data Structures and Algorithms

FIB

Q2 2018–19

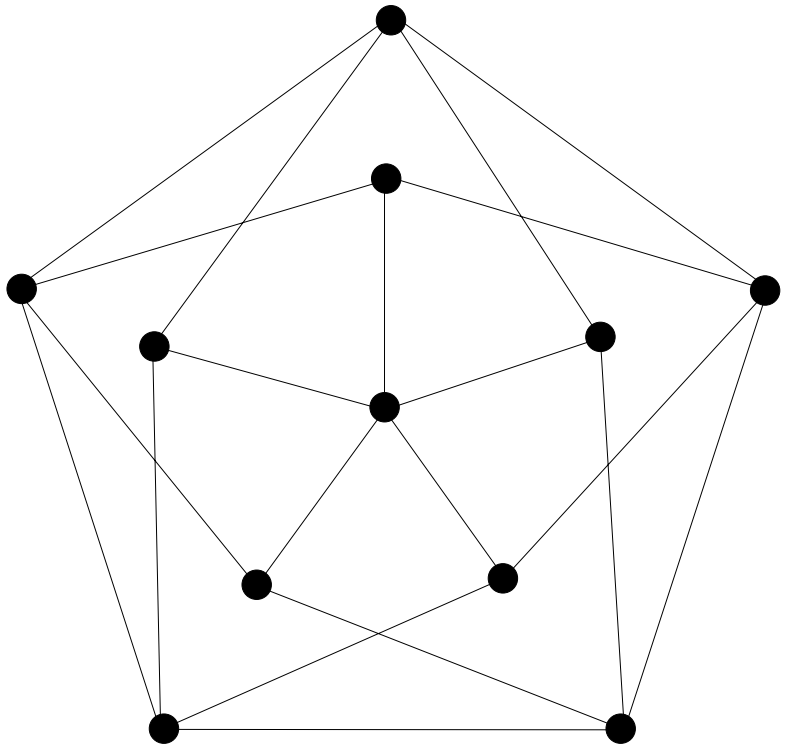
Jordi Delgado
(slides by Antoni Lozano)

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.

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?

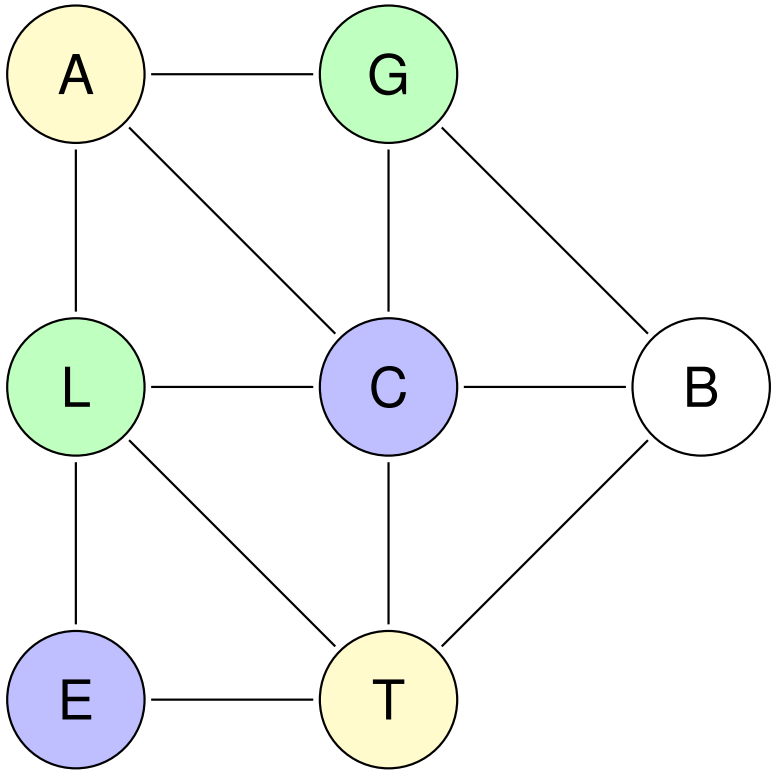
If we analyze a real map, we will find lots of irrelevant information:

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

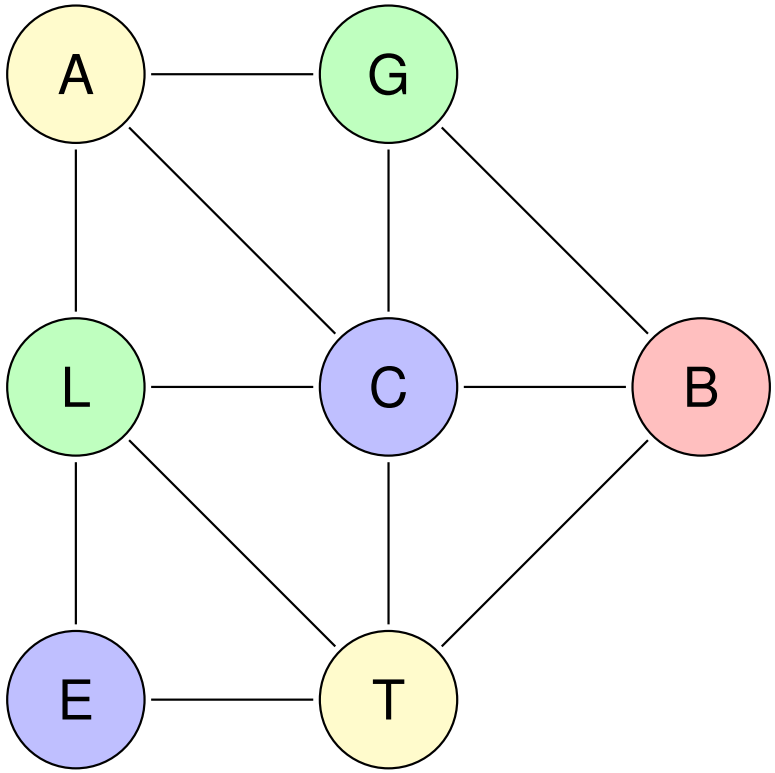
But any map can be represented as a planar graph:

- 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)

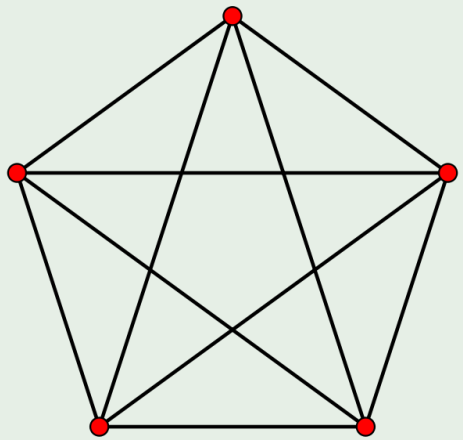
Any planar graph can be colored with 4 colors.

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.

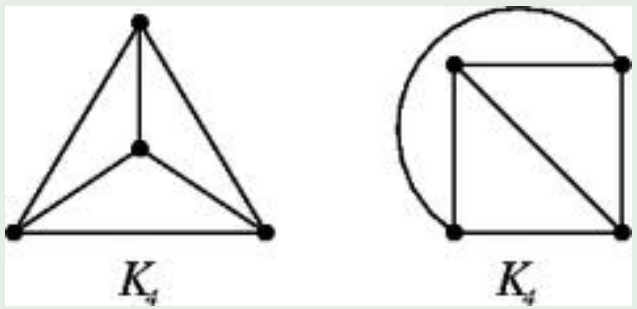
In gran theory, this is the same as saying that graph K_5



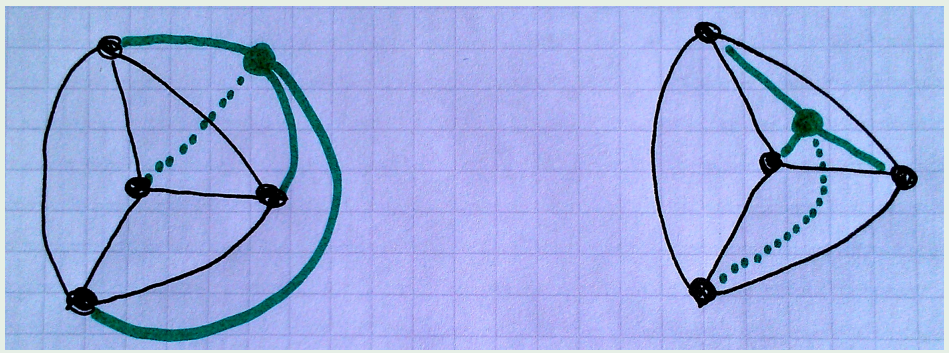
cannot be drawn in a plane without intersecting edges (it is not *planar*).

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):



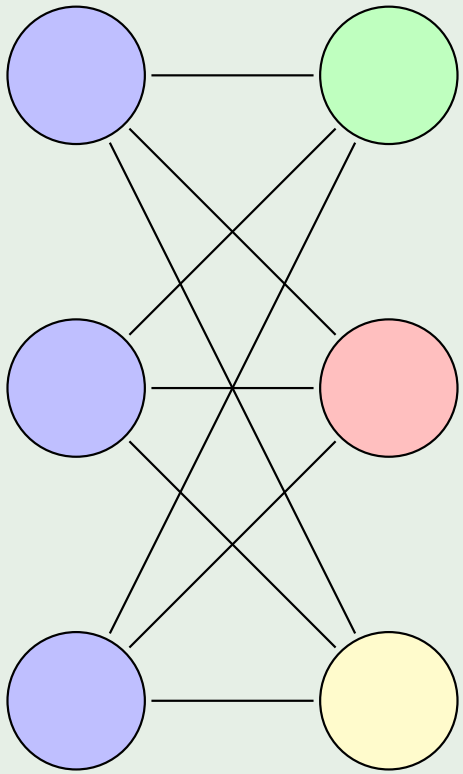
Independently of whether the fifth vertex is placed in the external or the internal area, there is always a vertex that cannot be connected without intersecting some edge.



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

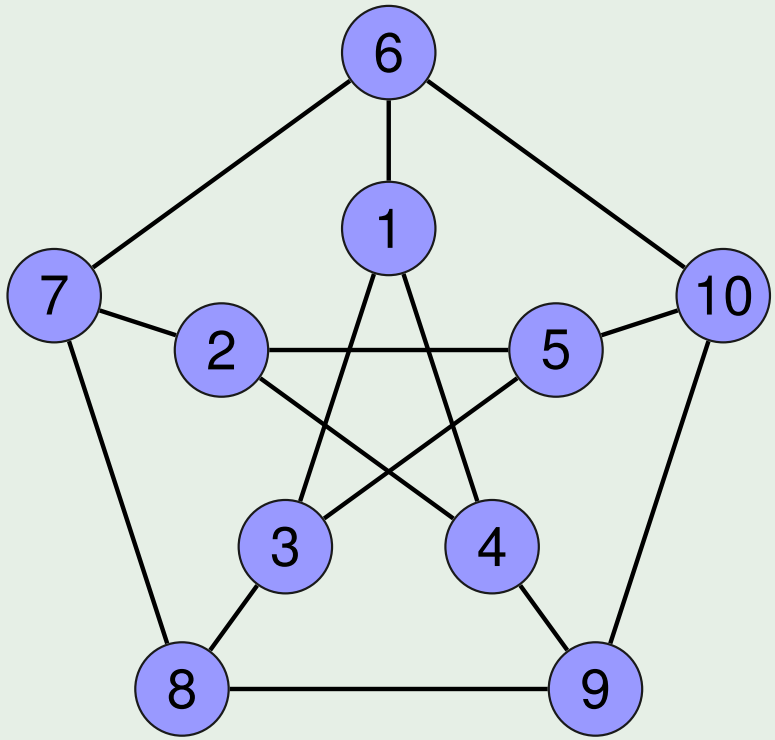
Definition

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

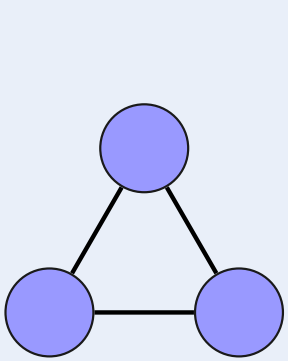


Formally, it is the pair (V, E) where

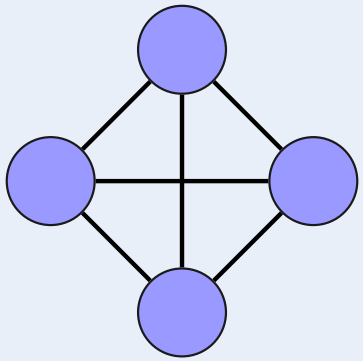
- $V = \{1, \ldots, 10\}$
- $E = \{\{1,3\},\{1,4\},\{1,6\},\{2,4\},\{2,5\},\{2,7\},\{3,5\},$ $\{3,8\},\{4,9\},\{5,10\},\{6,7\},\{6,10\},\{7,8\},\{8,9\},\{9,10\}\}$

Families of graphs

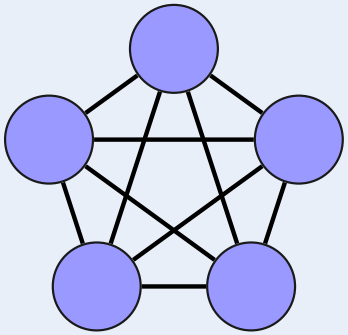
- **Complete:** K_i is the complete graph with i vertices.



K_3

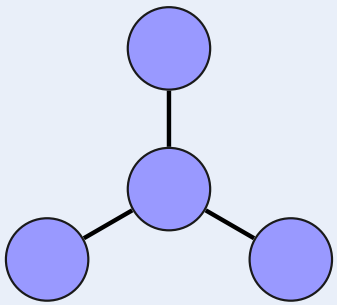


K_4

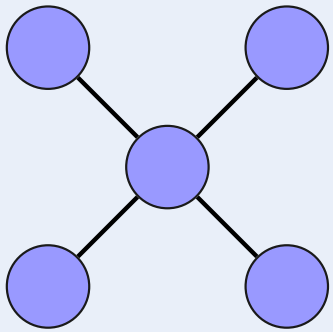


K_5

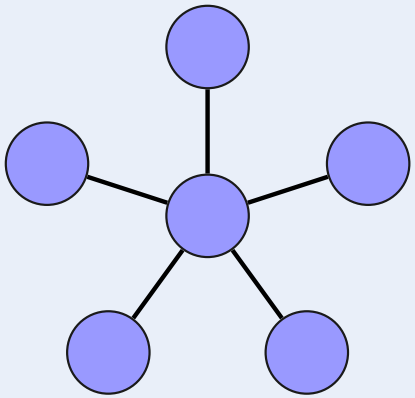
- **Stars:** S_i is the star with $i + 1$ vertices.



S_3



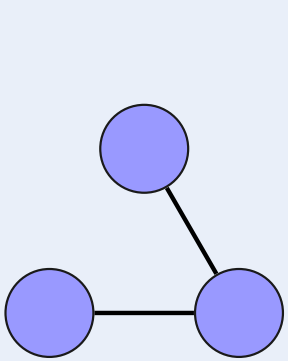
S_4



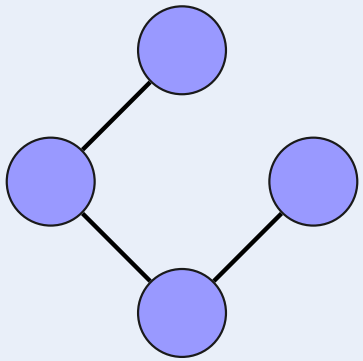
S_5

Families of graphs

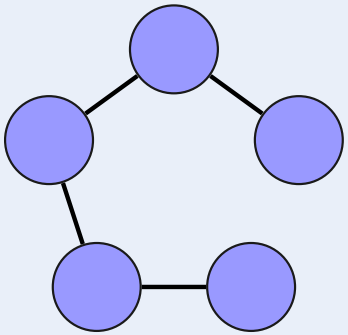
- **Paths:** P_i is the path with i vertices.



P_3

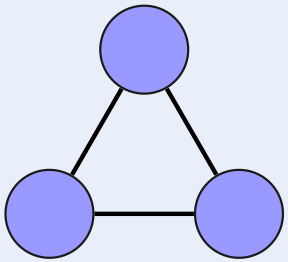


P_4

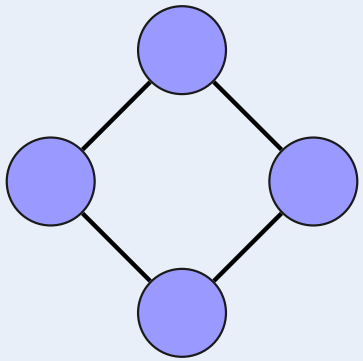


P_5

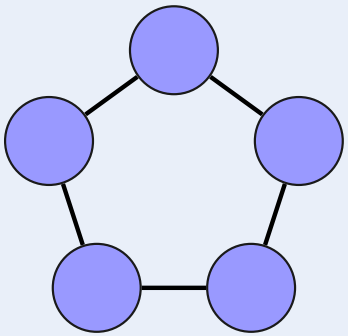
- **Cycles:** C_i is the cycles with i vertices.



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

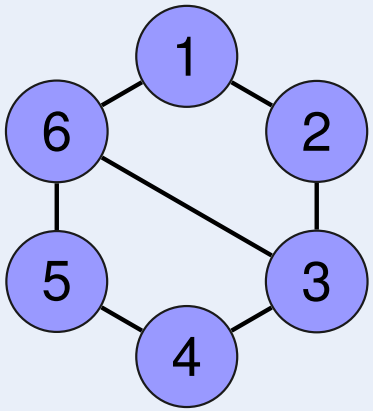
$$\frac{n(n-1)}{2}$$

different edges.

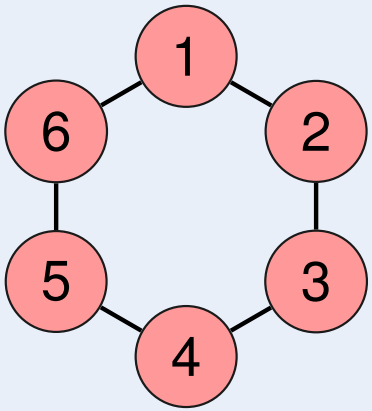
(Combinations of n elements chosen in groups of 2.)

Adjacency and subgraphs

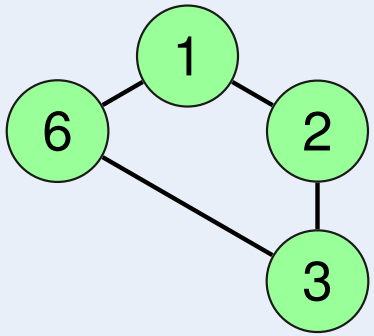
- two vertices u, v are adjacent if $\{u, v\}$ is an edge
- an edge $\{u, v\}$ is incident in u and v
- degree of a vertex u: number of incident edges in u
- a graph $H = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.
- a graph H is an induced subgraph of a graph G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

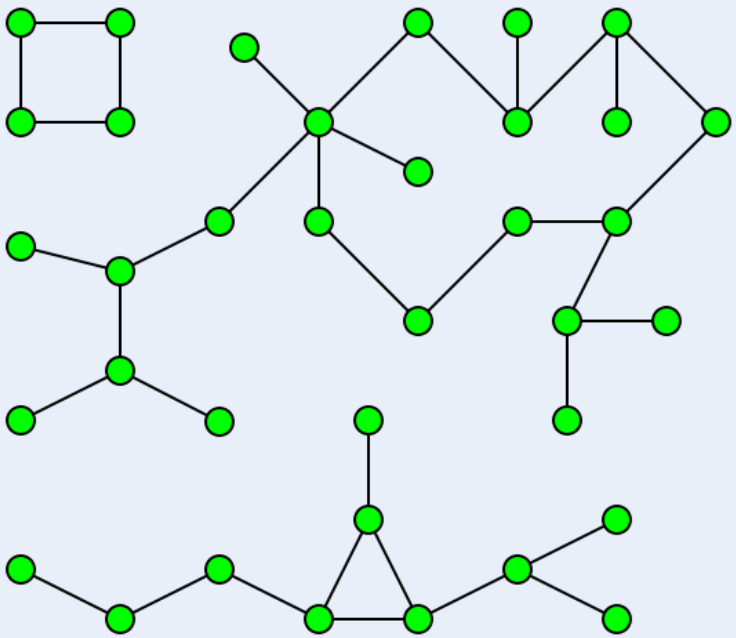


Figura: Cyclic graph with 3 connected components

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.

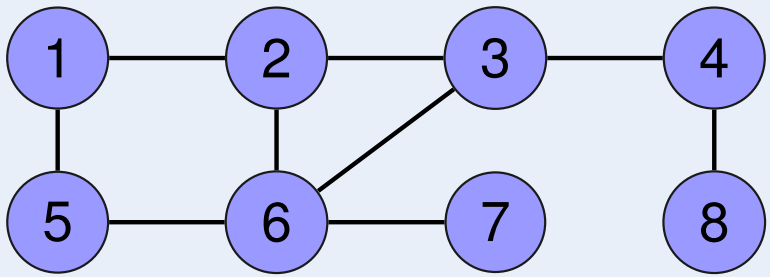
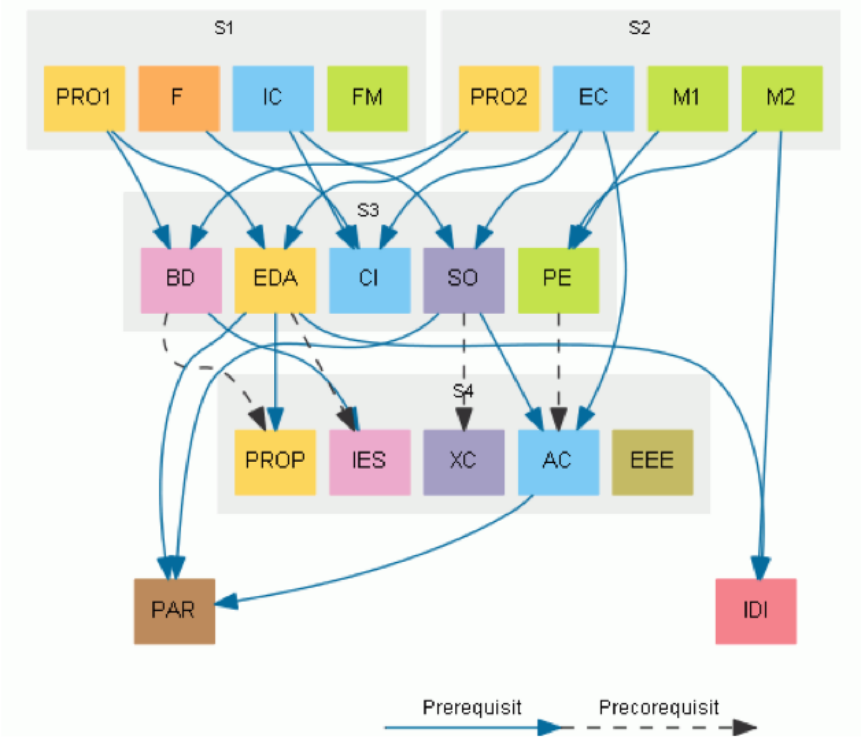


Figura: Distance between 4 and 5 is 3. The diameter of the graph is 4.

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).



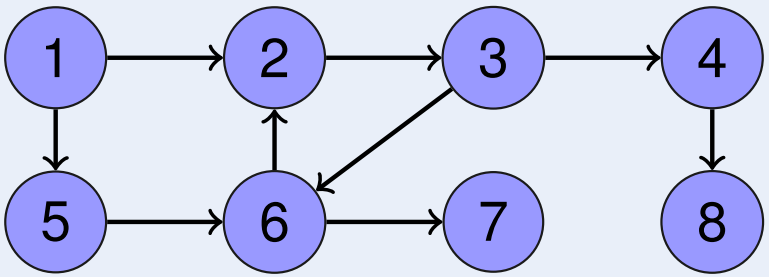
Directed graph of compulsory courses of the degree (with two types of arcs)

Directed and labeled graphs

The concepts of graph are easily adapted to digraphs.

Digraphs: degrees and distances

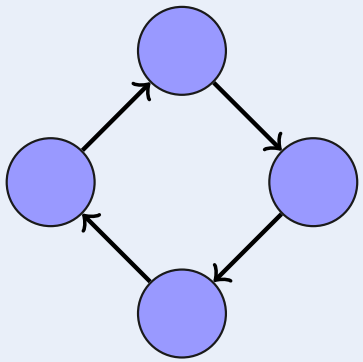
- We distinguish between indegree and outdegree.
- In a path (or directed path), all arcs go in the same direction.
- The distance between two vertices refers to directed paths.



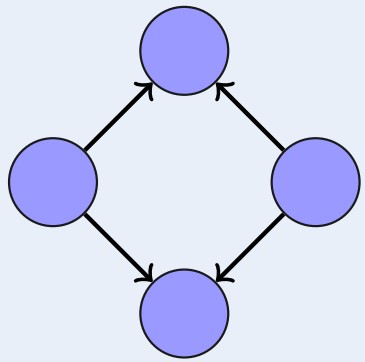
Vertex 2 has indegree 2 and outdegree 1.
The distance from 5 to 4 is 4; from 4 to 5, ∞.

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.



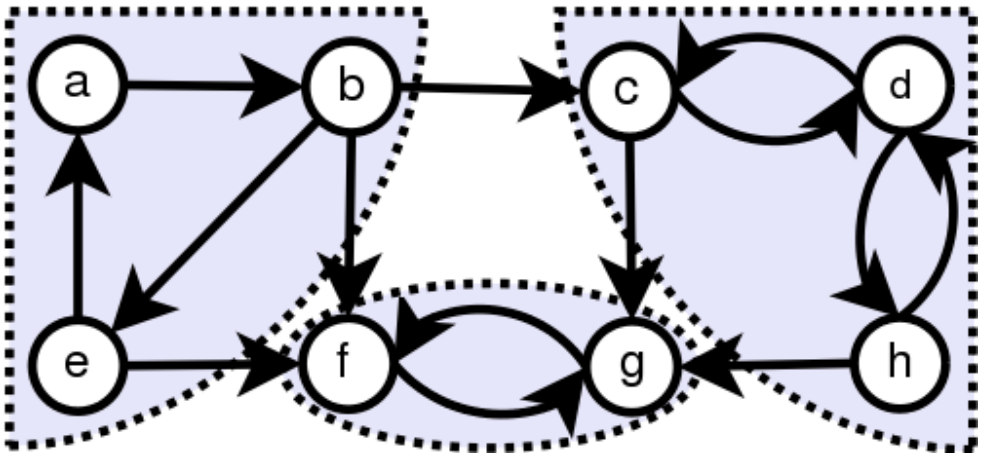
strongly connected



weakly connected

Digraphs: connectivity

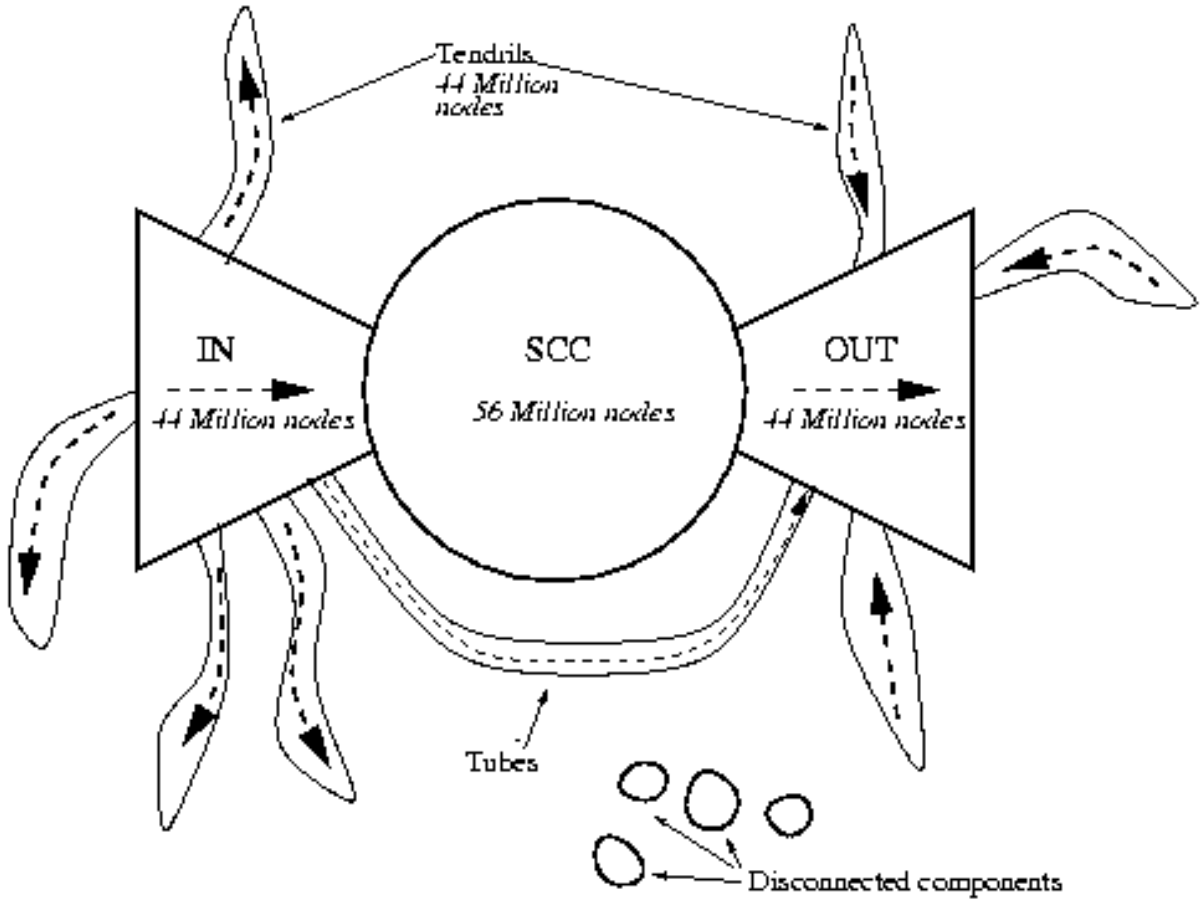
- Strongly connected components of a digraph are the maximal subgraphs that are strongly connected.



Digraph with 3 connected components

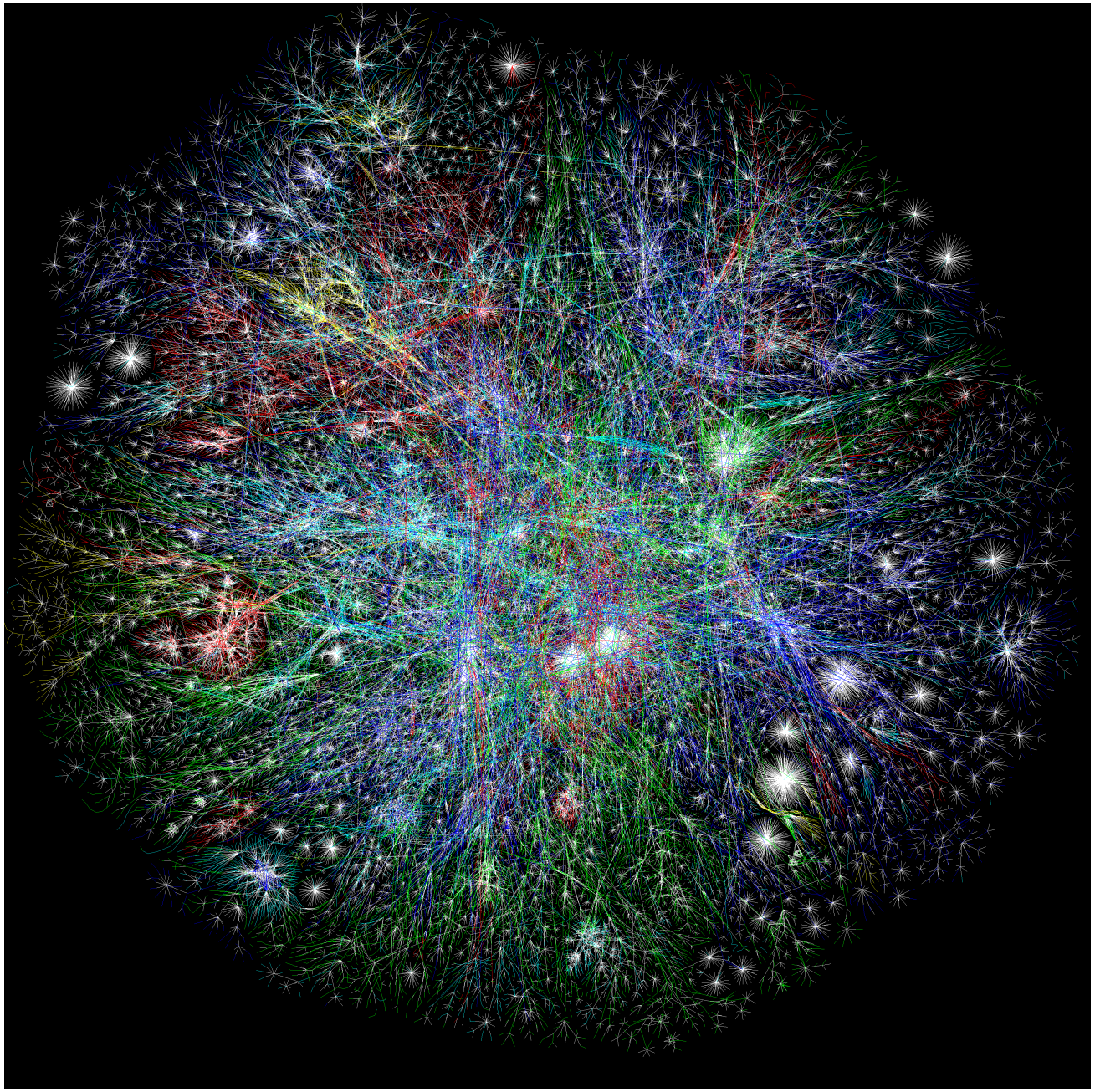
Internet graph (year 2000, Broder, Kumar et al.)

Websites are vertices; links are edges.



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

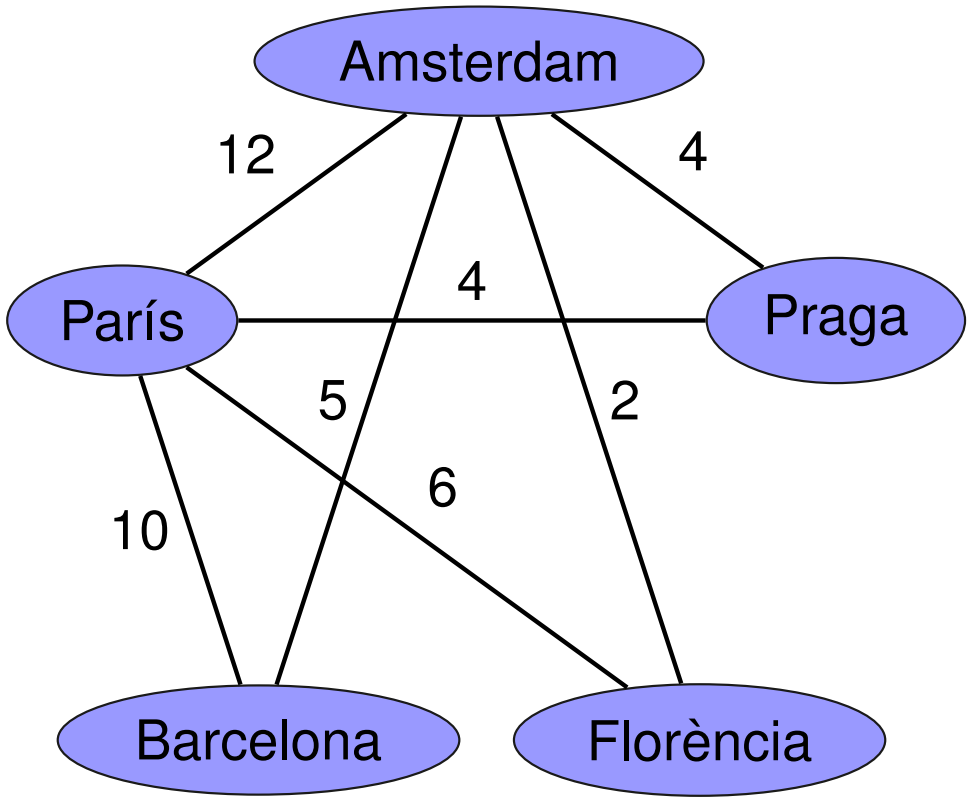
Internet graph



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

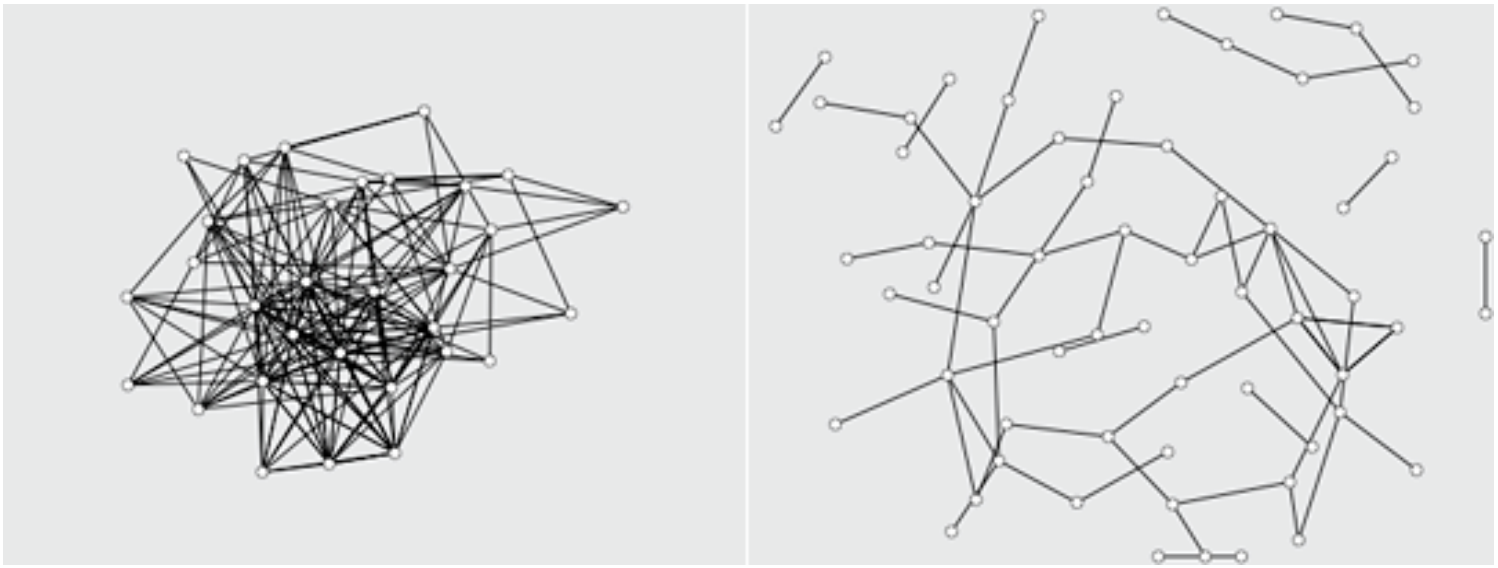
Directed and labeled graphs

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.



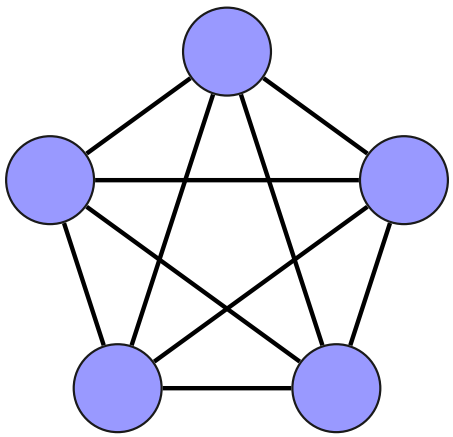
How can we define the density of a graph?

Density

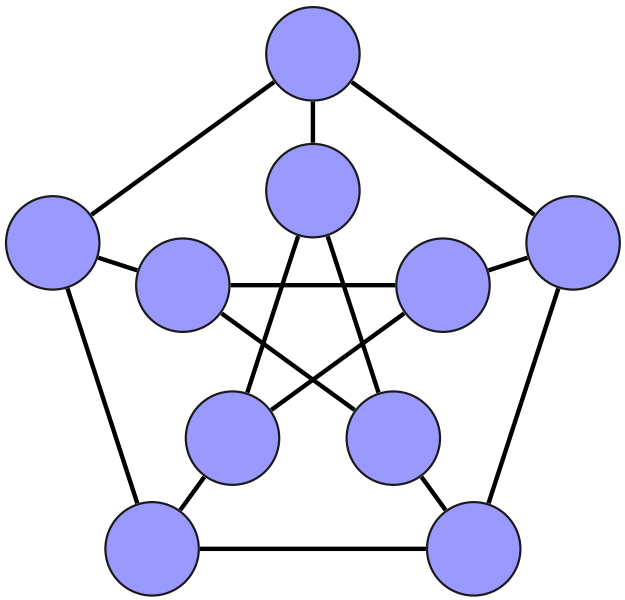
The density of a graph with n vertices and m edges is

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

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:

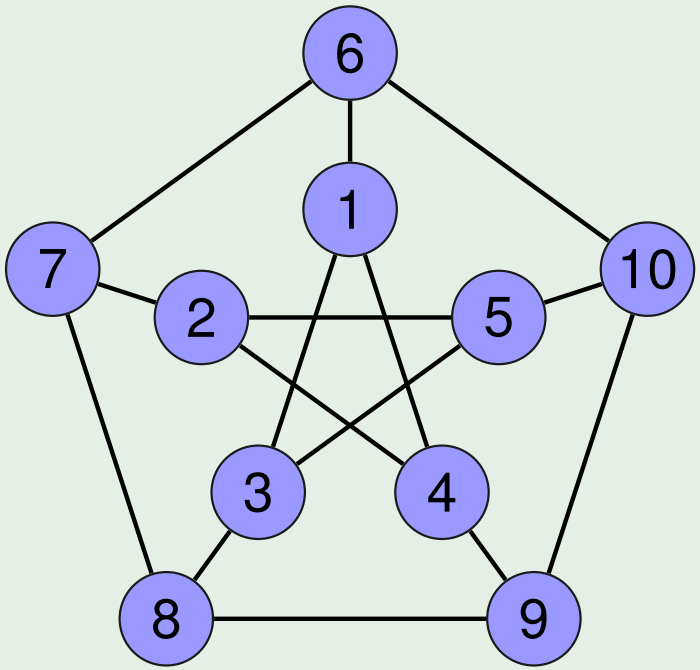
- Complete graphs (K_n) are dense because $D = 1$ for all n.
- Cycles (C_n) are sparse because $D = 2/(n-1)$ and density tends to 0 when n tends to ∞.

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



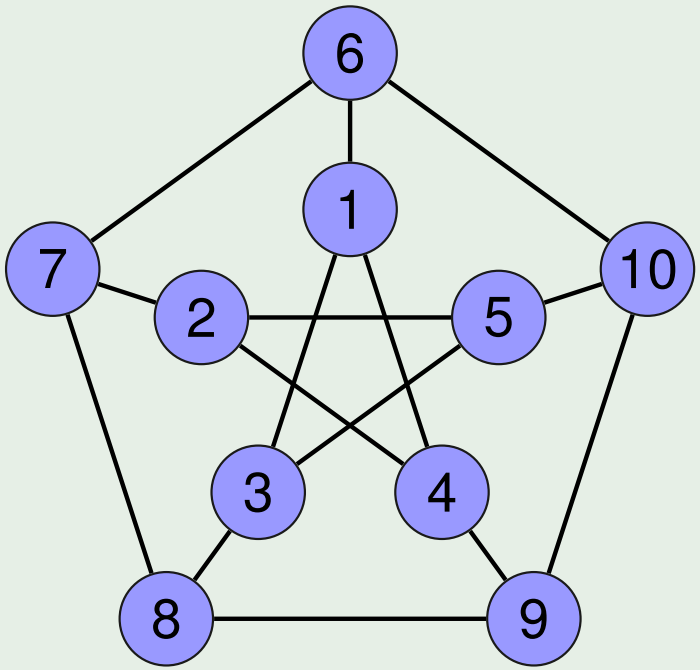
$$\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{array} \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

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



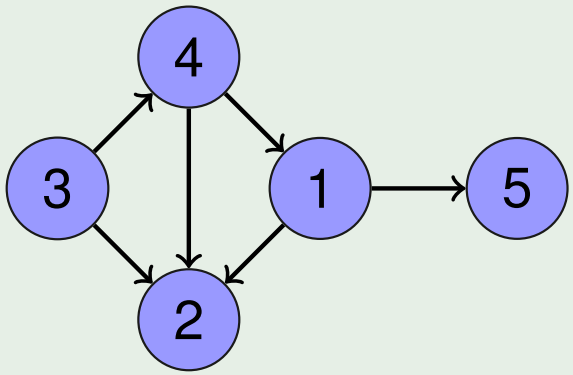
1		0	1	1	0	1	0	0	0	0
2	0		0	1	1	0	1	0	0	0
3	1	0		1	0	0	0	1	0	0
4	1	1	0		0	0	0	0	1	0
5	0	1	1	0		0	0	0	0	1
6	1	0	0	0	0		1	0	0	1
7	0	1	0	0	0	1		1	0	0
8	0	0	1	0	0	0	1		1	0
9	0	0	0	1	0	0	0	1		1
10	0	0	0	0	1	1	0	0	1	

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

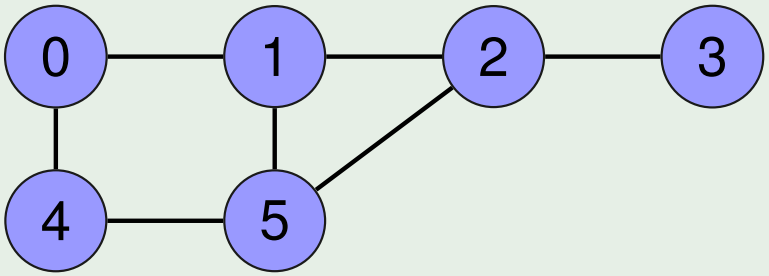
Adjacency matrices can be implemented as a vector of vectors.

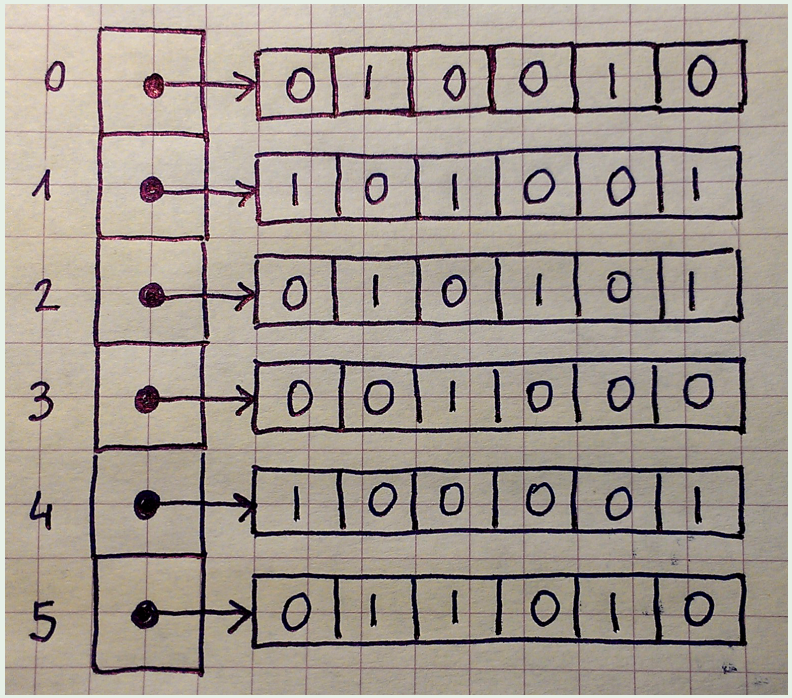
```
typedef vector< vector<bool> > graph;
```

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





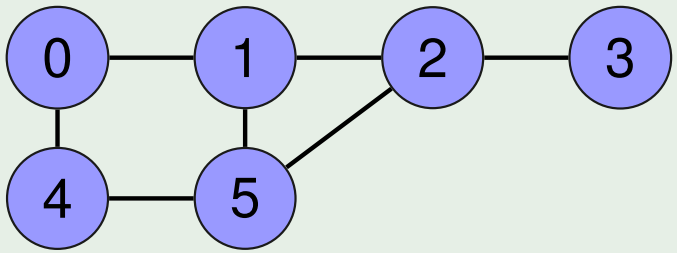
Representations

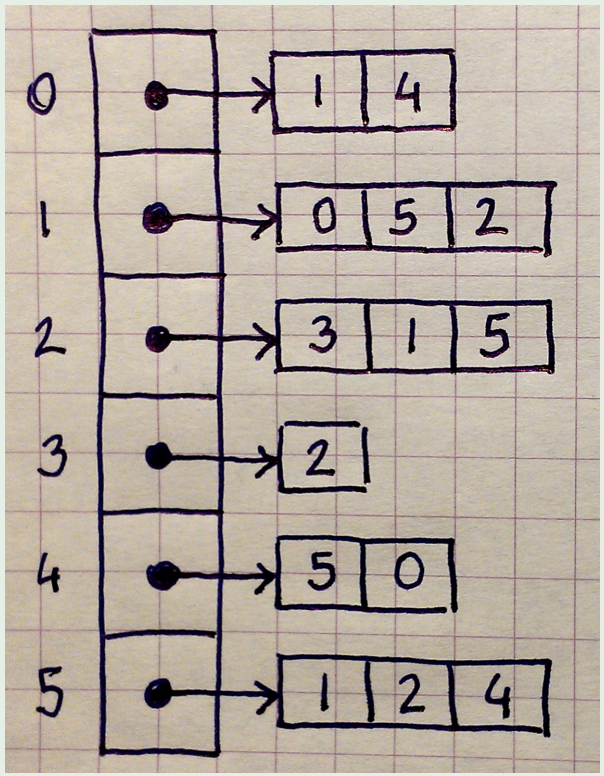
Adjacency lists

Each vertex points to the list of adjacent vertices.

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

Example





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

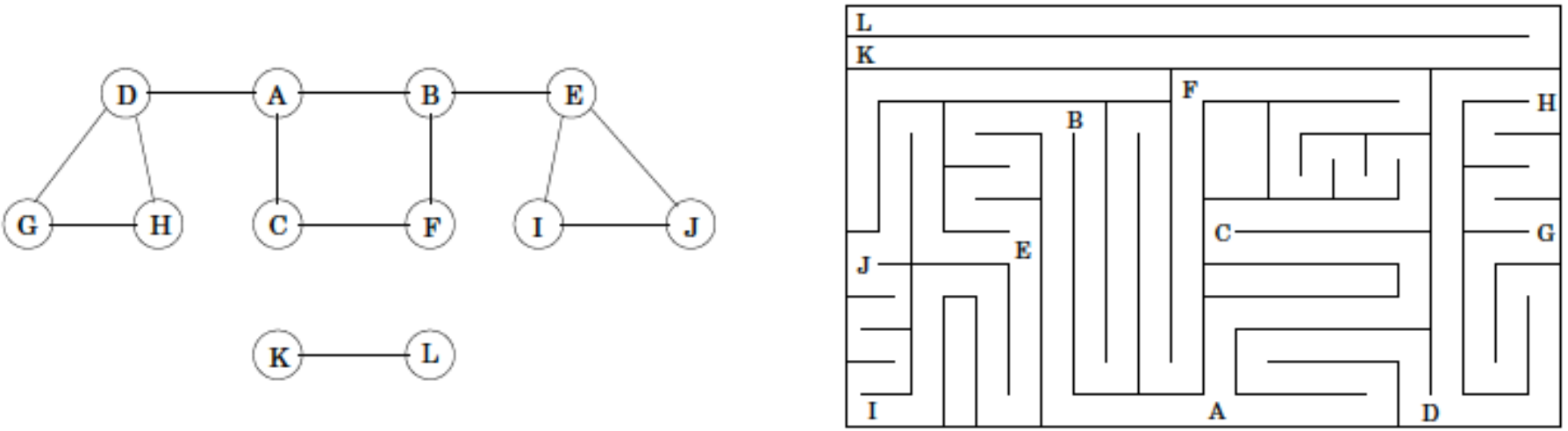


Figura: To convert a maze into a graph: we mark maze sectors (vertices) and we join them (via edges) if they are neighbors (example from *Algorithms*, S. Dasgupta, C.H. Papadimitriou and U.V. Vazirani)

Depth-first search

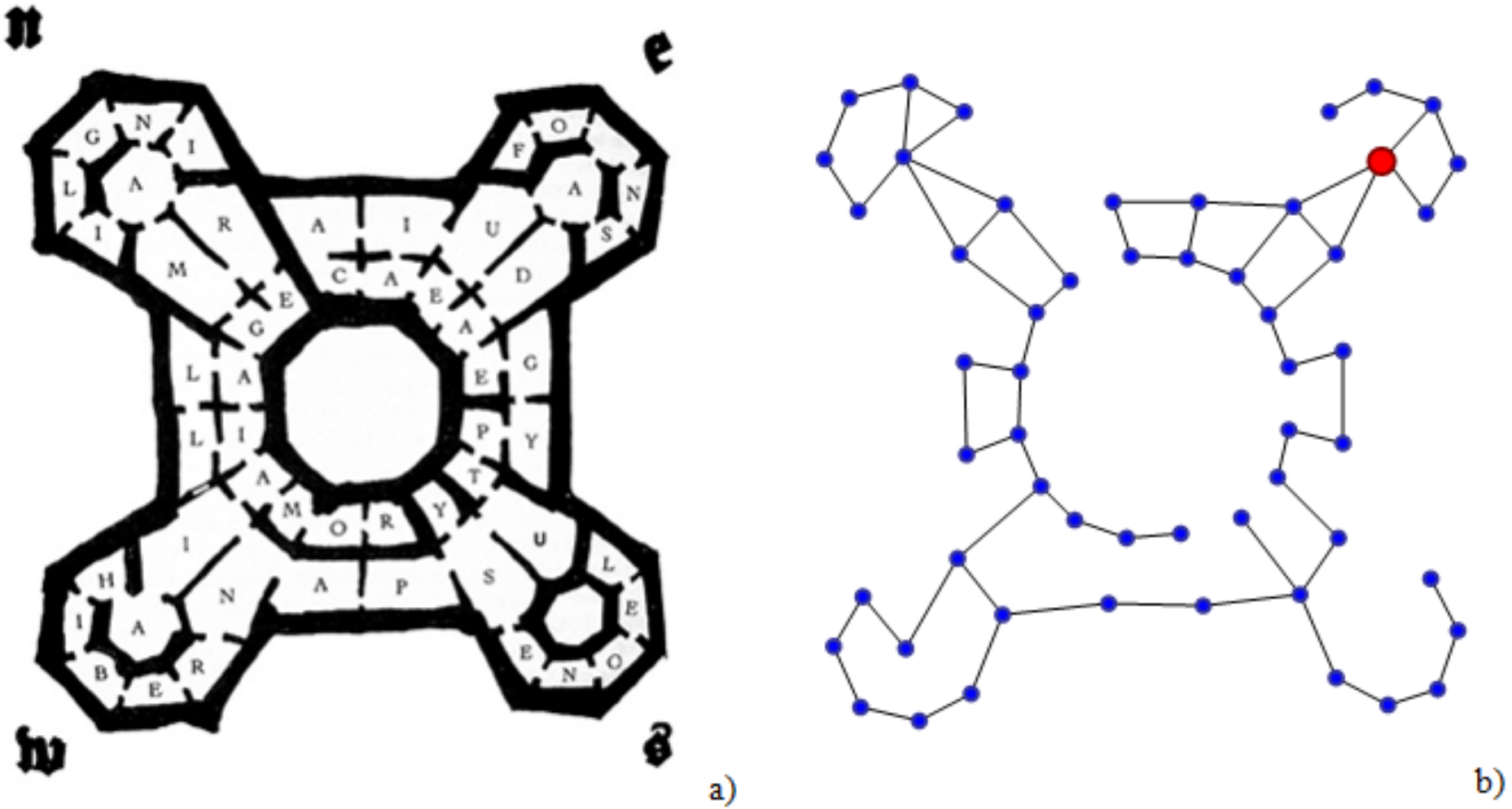


Figura: Library maze from *The name of the rose*, U. Eco

Depth-first search

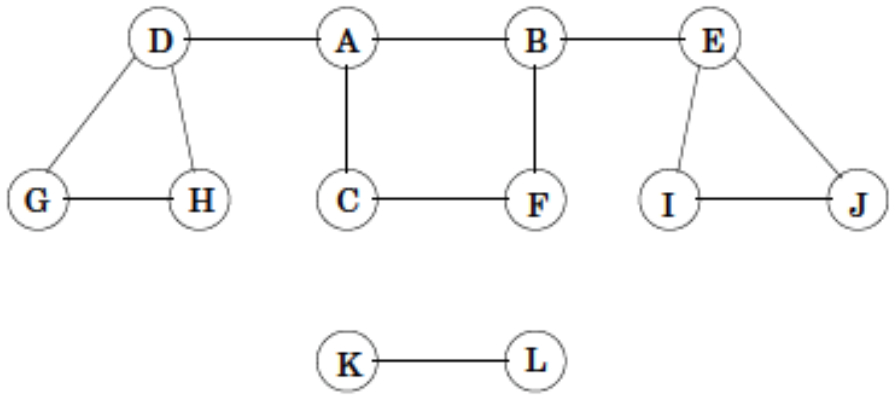
Recursive depth-first search (from a given vertex)

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

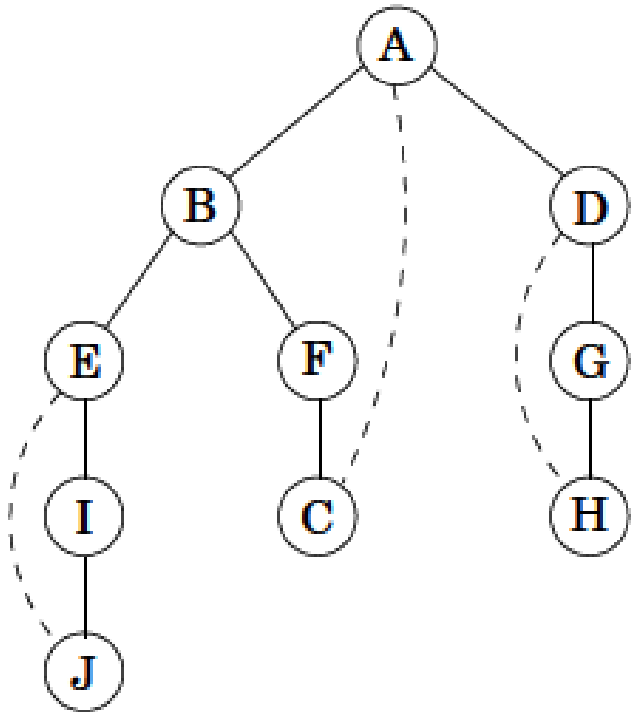
```
void dfs_rec (const graph& G, int u,
              vector<boolean>& vis, list<int>& L) {
    if (not vis[u]) {
        vis[u] = true; L.push_back(u);
        for (int i = 0; i < G[u].size(); ++i) {
            dfs_rec(G, G[u][i], vis, L);
}   }   }
```

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

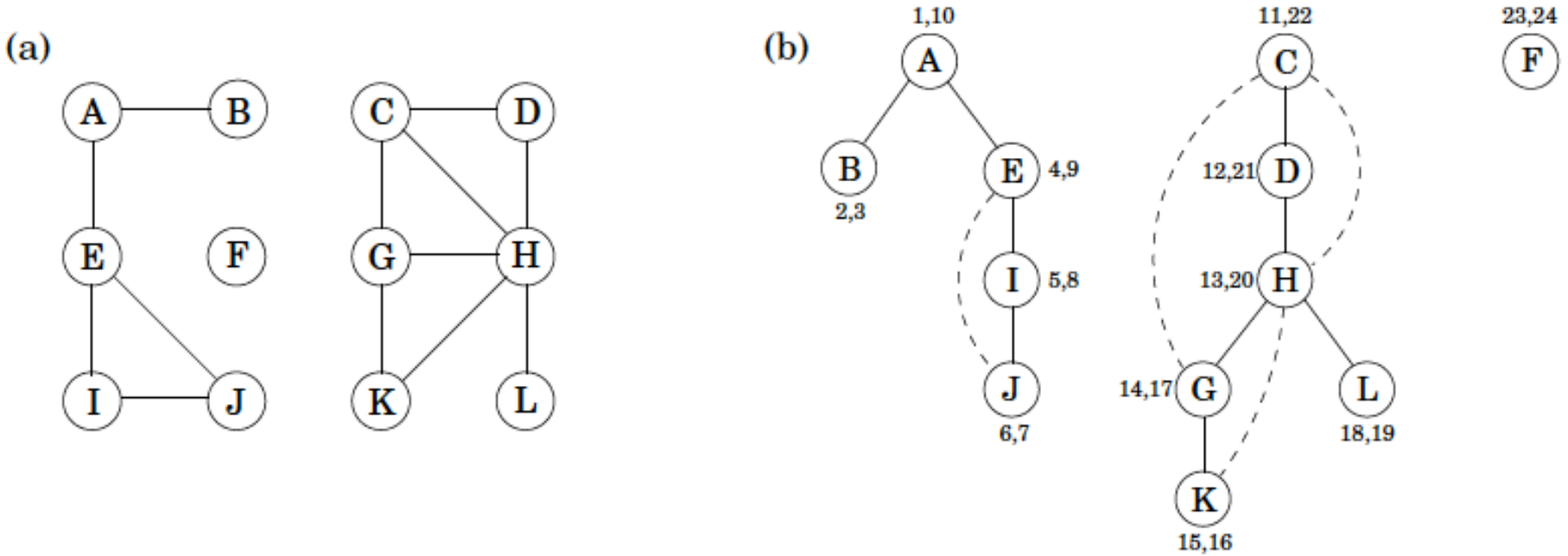


Figura: Depth-first search in an undirected and unconnected graph (*Algorithms.* Dasgupta et al.)

Depth-first search

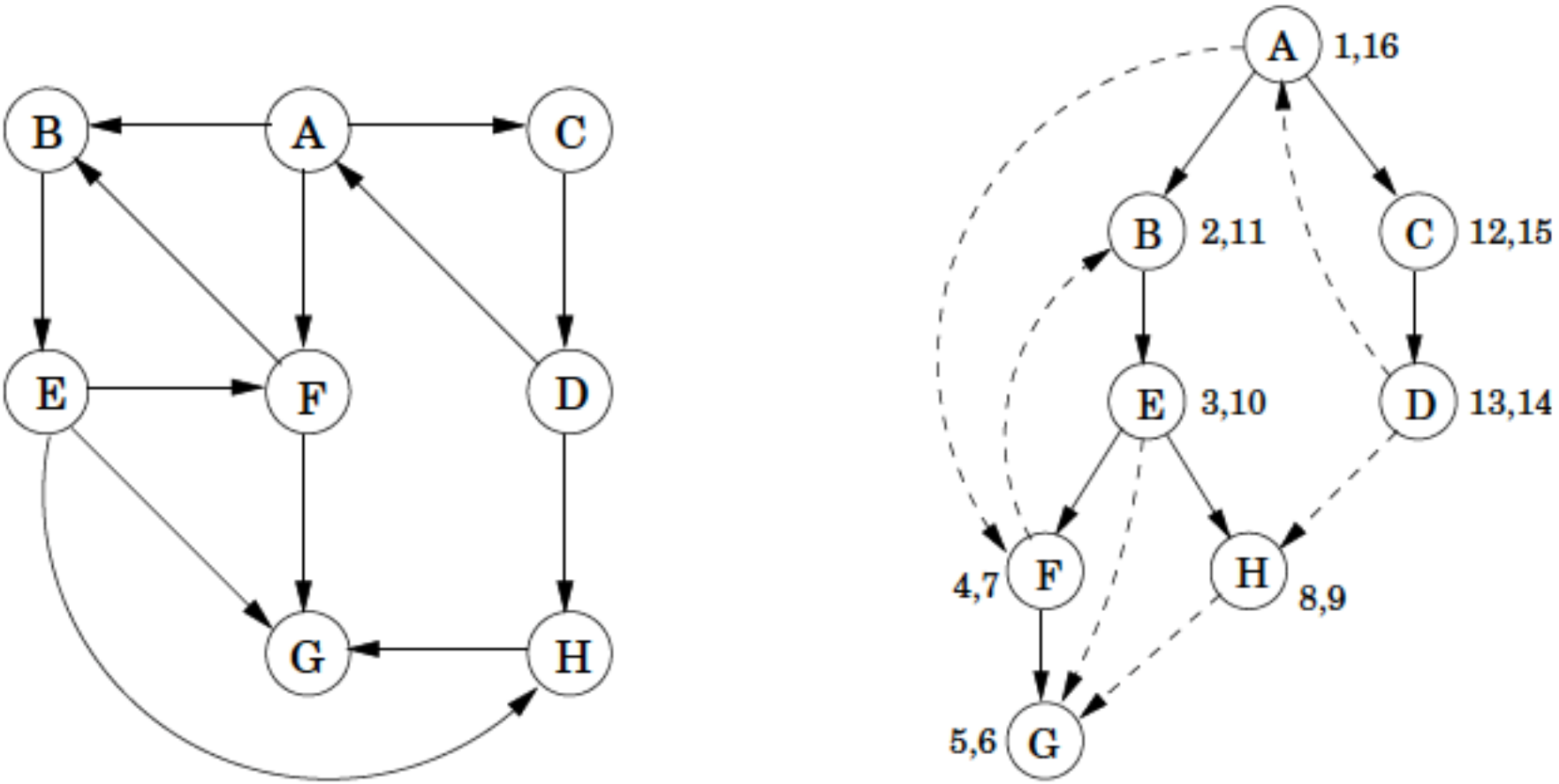


Figura: Depth-first search in a directed graph (*Algorithms.* Dasgupta et al.)

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;
}
```

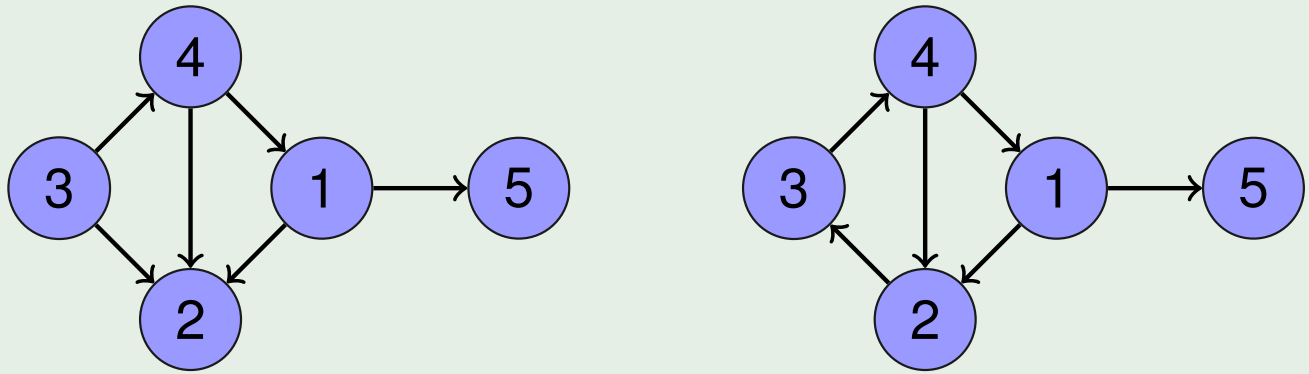
Topological sorting

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

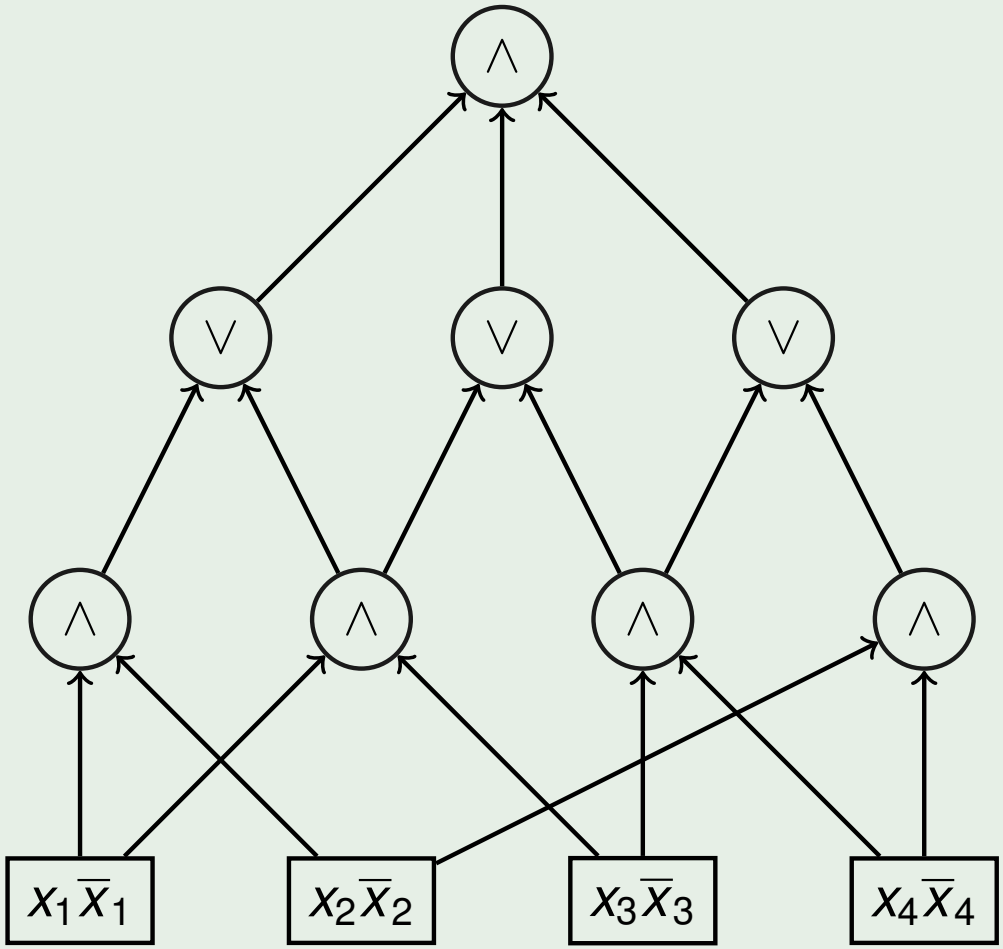


The digraph on the left is a dag; the one in the right is not.

Topological sorting

Example

A circuit is also a dag.



Topological sorting

Definition

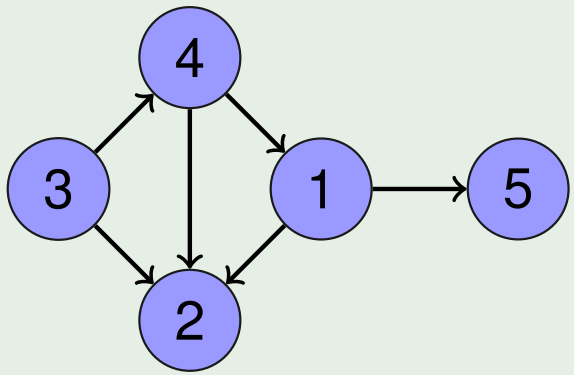
A topological sorting of a dag $G = (V, E)$ is a sequence

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

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 $i < j$.

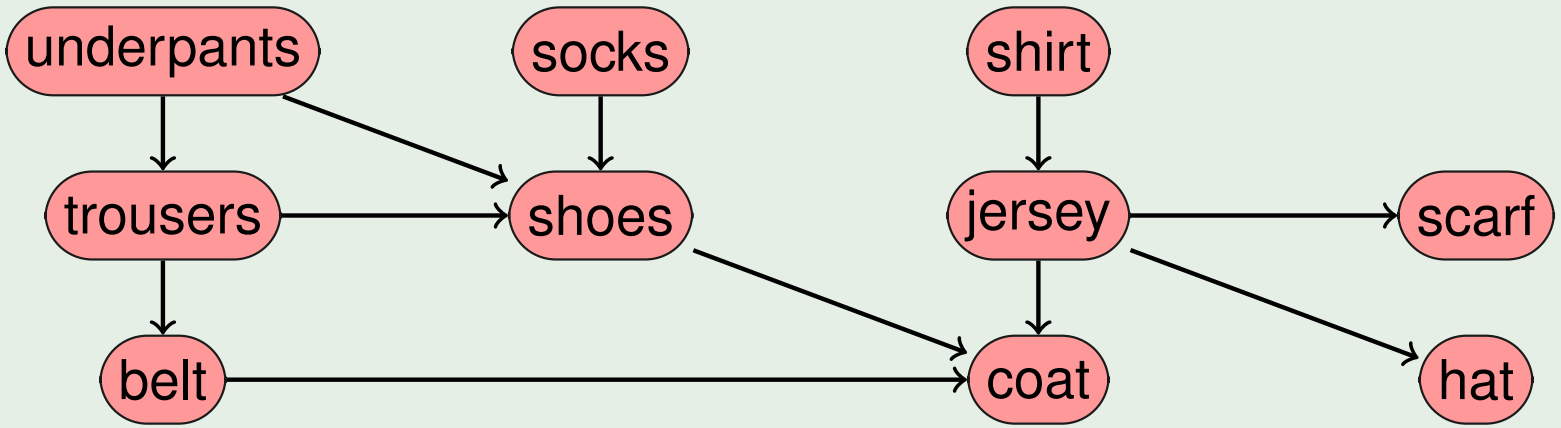
Exercises

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

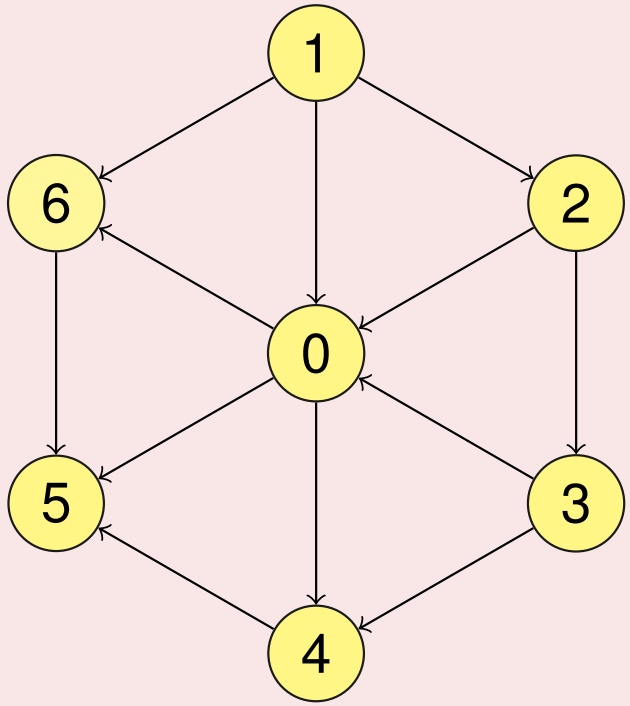
- each vertex once
- each edge once

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**.



Breadth-First Search

Breadth-First Search (BFS) advances locally from an initial vertex s, visiting all vertices at distance $k+1$ from s after having visited all vertices at distance k from s.

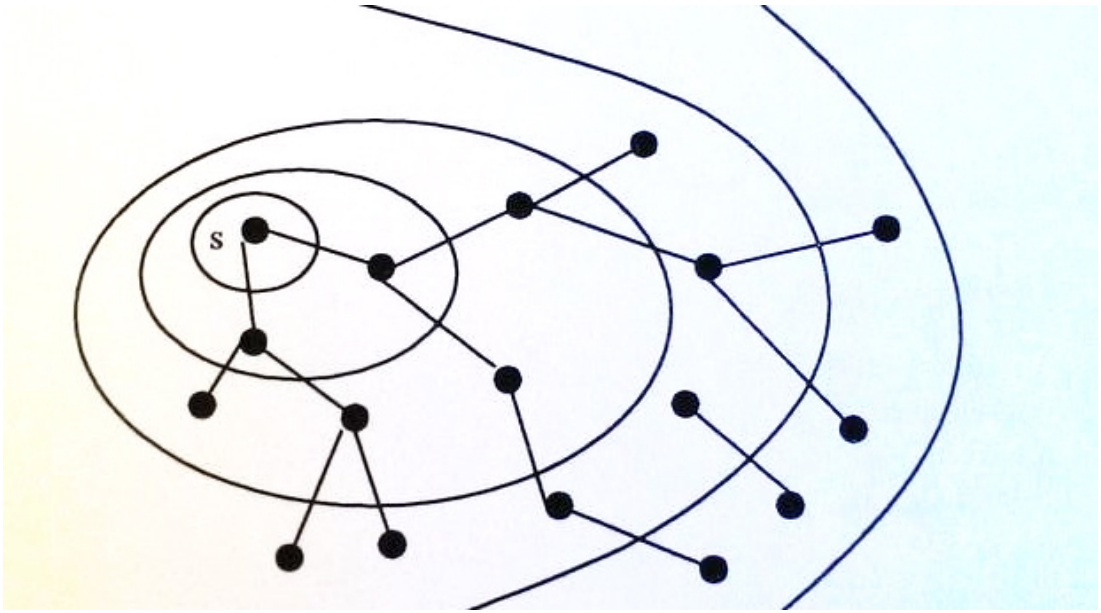


Figura: Breadth-First Search

Breadth-First Search

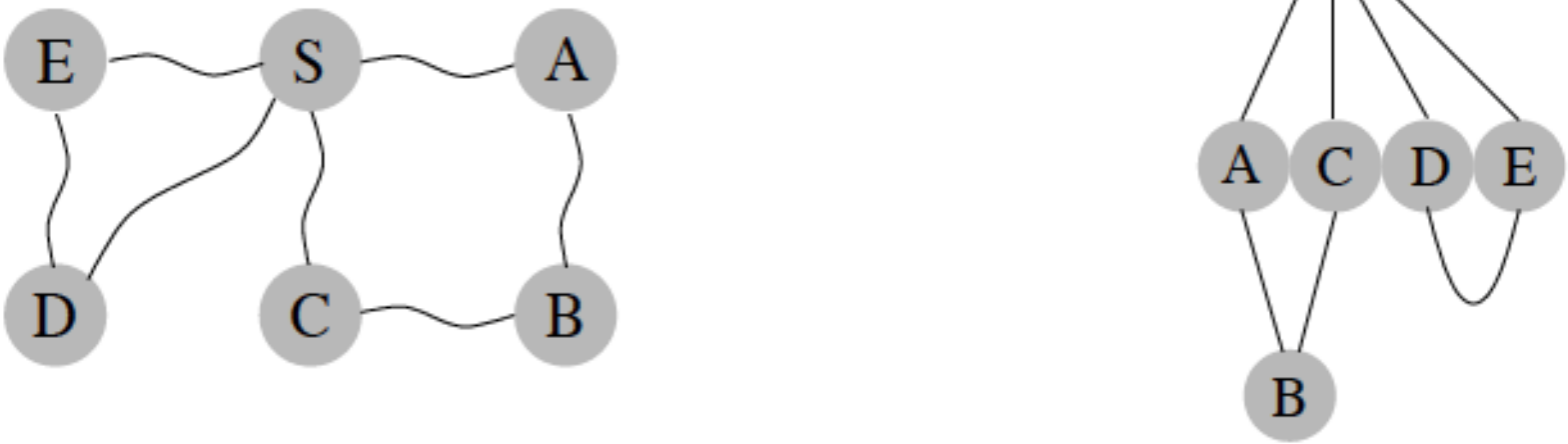


Figura: Physical model of a graph. When the graph hangs from S, the edges in the shortest paths get tight. The edge $\{D, E\}$ does no belong to the shortest paths. *Algorithms*, Dasgupta et al.

Breadth-First Search

- BFS, starting from a vertex s, computes
  - a breadth-first traversal starting from s
  - the minimum distances from s to all other vertices
  - the shortest paths from s to all other vertices
- BFS works in graphs that are
  - directed or undirected
  - unweighted (no weights on the edges)
- It is one of the easiest graph algorithms and is a model for others:
  - Dijkstra's algorithm for shortest paths in weighted graphs
  - Prim's algorithm for finding a minimum spanning tree

Breadth-First Search

- BFS, starting from a vertex s, computes
  - a breadth-first traversal starting from s
  - the minimum distances from s to all other vertices
  - the shortest paths from s to all other vertices
- BFS works in graphs that are
  - directed or undirected
  - unweighted (no weights on the edges)
- It is one of the easiest graph algorithms and is a model for others:
  - Dijkstra's algorithm for shortest paths in weighted graphs
  - Prim's algorithm for finding a minimum spanning tree

Breadth-First Search

- BFS, starting from a vertex *s*, computes
  - a breadth-first traversal starting from *s*
  - the minimum distances from *s* to all other vertices
  - the shortest paths from *s* to all other vertices
- BFS works in graphs that are
  - directed or undirected
  - unweighted (no weights on the edges)
- It is one of the easiest graph algorithms and is a model for others:
  - Dijkstra's algorithm for shortest paths in weighted graphs
  - Prim's algorithm for finding a minimum spanning tree

Breadth-First Search

Input: directed or undirected graph $G = (V, E)$ and vertex $s \in V$
Output: for all vertices u reachable from s, dist(u) is the distance from s to u

```
BFS(G, s)
    for all u ∈ V
        dist(u) = ∞
    dist(s) = 0
    Q = [s] (queue containing only s)
    while Q is not empty
        u = dequeue(Q)
        for all edge (u, v) ∈ E
            if dist(v) = ∞ then
                enqueue(Q, v)
                dist(v) = dist(u) + 1
```

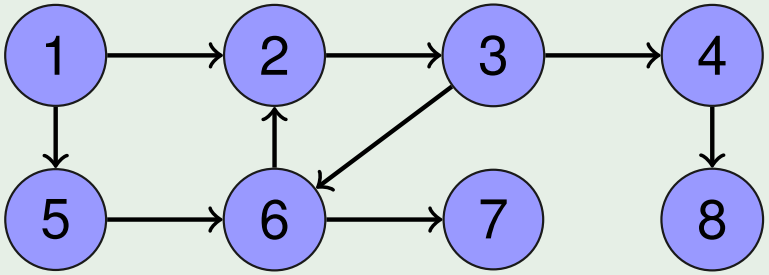
Breadth-First Search

For each $d = 0, 1, 2, \ldots$ there is a moment when:

- vertices at distance $\leq d$ from s have the correct distance
- vertices at distance $> d$ from s have distance ∞
- the queue contains all nodes at distance d

Breadth-First Search

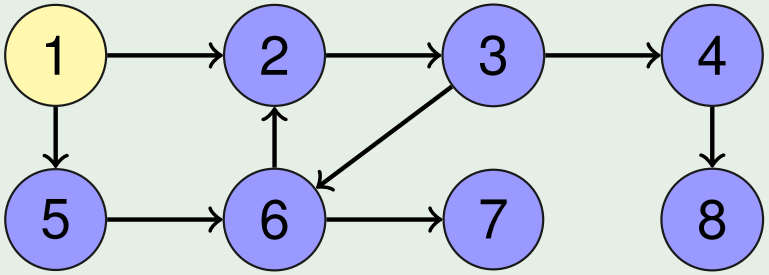
Example 1



order of visit	distance from 1	queue after processing vertex
		[1]
1	0	[2 5]
2	1	[5 3]
5	1	[3 6]
3	2	[6 4]
6	2	[4 7]
4	3	[7 8]
7	3	[8]
8	4	[]

Breadth-First Search

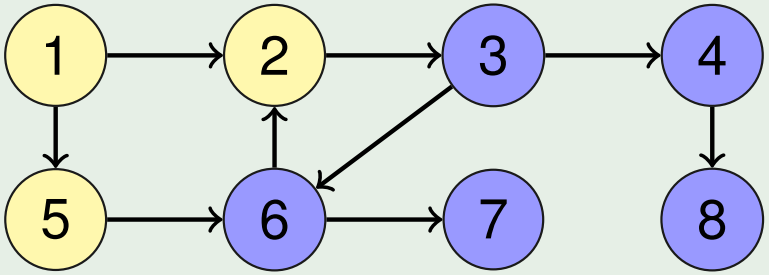
Example 1



order of visit	distance from 1	queue after processing vertex
		[1]
1	0	[2 5]
2	1	[5 3]
5	1	[3 6]
3	2	[6 4]
6	2	[4 7]
4	3	[7 8]
7	3	[8]
8	4	[]

Breadth-First Search

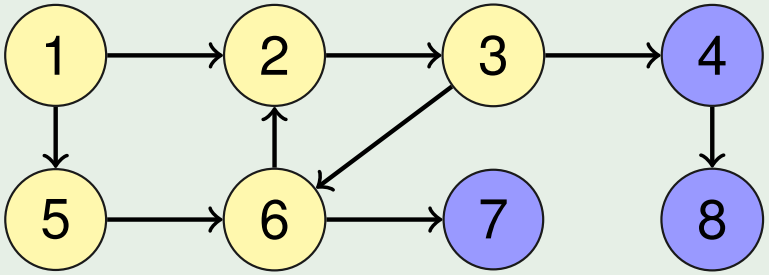
Example 1



order of visit	distance from 1	queue after processing vertex
		[1]
1	0	[2 5]
2	1	[5 3]
5	1	[3 6]
3	2	[6 4]
6	2	[4 7]
4	3	[7 8]
7	3	[8]
8	4	[]

Breadth-First Search

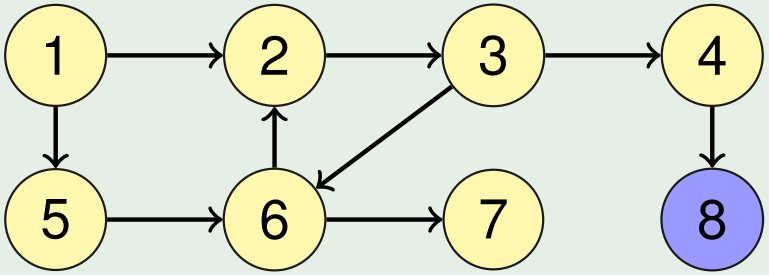
Example 1



order of visit	distance from 1	queue after processing vertex
		[1]
1	0	[2 5]
2	1	[5 3]
5	1	[3 6]
3	2	[6 4]
6	2	[4 7]
4	3	[7 8]
7	3	[8]
8	4	[]

Breadth-First Search

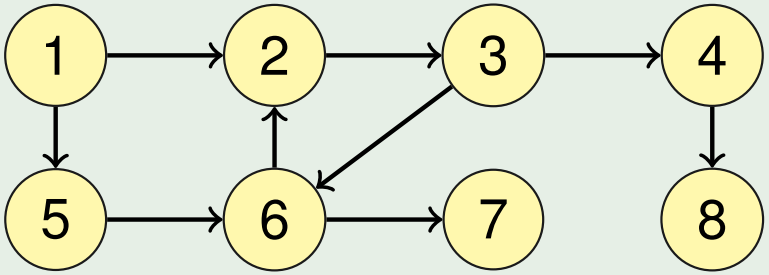
Example 1



order of visit	distance from 1	queue after processing vertex
		[1]
1	0	[2 5]
2	1	[5 3]
5	1	[3 6]
3	2	[6 4]
6	2	[4 7]
4	3	[7 8]
7	3	[8]
8	4	[]

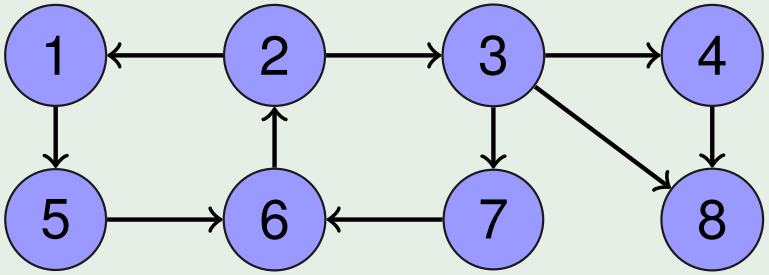
Breadth-First Search

Example 1



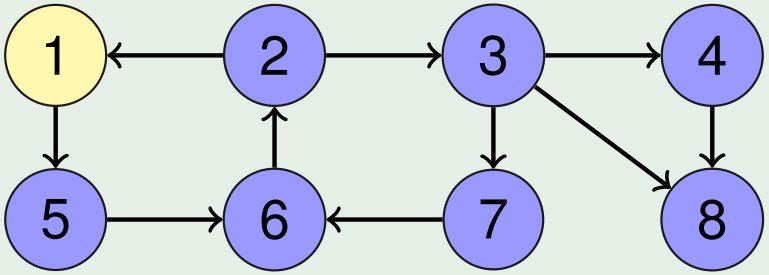
order of visit	distance from 1	queue after processing vertex
		[1]
1	0	[2 5]
2	1	[5 3]
5	1	[3 6]
3	2	[6 4]
6	2	[4 7]
4	3	[7 8]
7	3	[8]
8	4	[]

Example 2



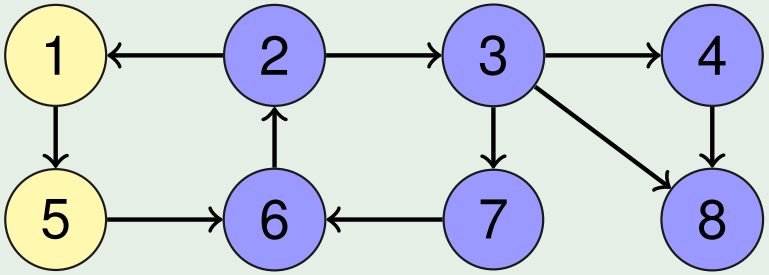
order of visit	distance from 1	queue after processing vertex
		[1]
1	0	[5]
5	1	[6]
6	2	[2]
2	3	[3]
3	4	[4 7 8]
4	5	[7 8]
7	5	[8]
8	5	[]

Example 2



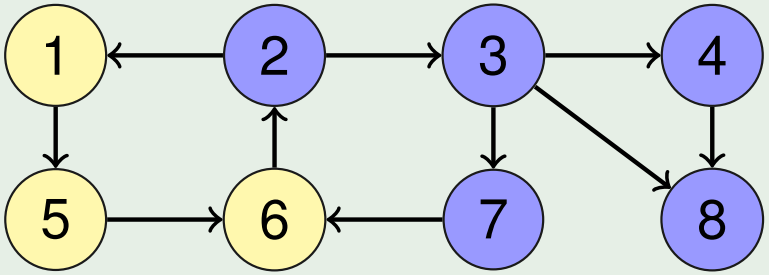
order of visit	distance from 1	queue after processing vertex
		[1]
1	0	[5]
5	1	[6]
6	2	[2]
2	3	[3]
3	4	[4 7 8]
4	5	[7 8]
7	5	[8]
8	5	[]

Example 2



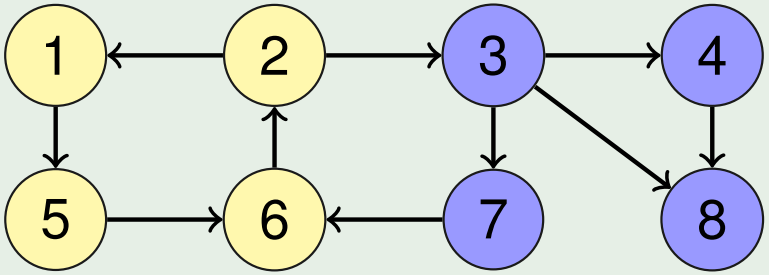
order of visit	distance from 1	queue after processing vertex
		[1]
1	0	[5]
5	1	[6]
6	2	[2]
2	3	[3]
3	4	[4 7 8]
4	5	[7 8]
7	5	[8]
8	5	[]

Example 2



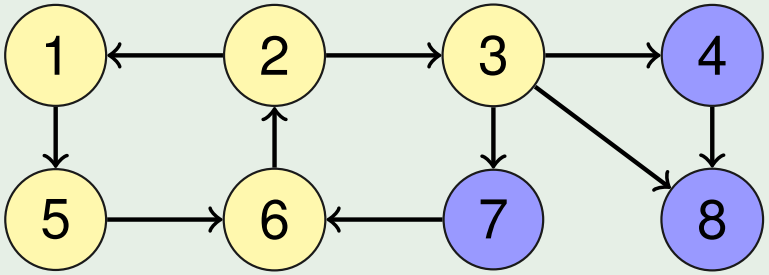
order of visit	distance from 1	queue after processing vertex
		[1]
1	0	[5]
5	1	[6]
6	2	[2]
2	3	[3]
3	4	[4 7 8]
4	5	[7 8]
7	5	[8]
8	5	[]

Example 2



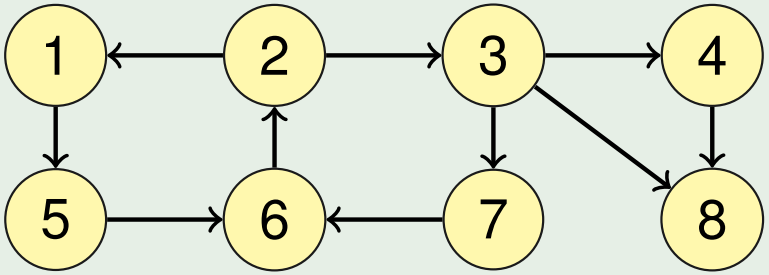
order of visit	distance from 1	queue after processing vertex
		[1]
1	0	[5]
5	1	[6]
6	2	[2]
2	3	[3]
3	4	[4 7 8]
4	5	[7 8]
7	5	[8]
8	5	[]

Example 2



order of visit	distance from 1	queue after processing vertex
		[1]
1	0	[5]
5	1	[6]
6	2	[2]
2	3	[3]
3	4	[4 7 8]
4	5	[7 8]
7	5	[8]
8	5	[]

Example 2



order of visit	distance from 1	queue after processing vertex
		[1]
1	0	[5]
5	1	[6]
6	2	[2]
2	3	[3]
3	4	[4 7 8]
4	5	[7 8]
7	5	[8]
8	5	[]

Breadth-First Search

Breadth-First Search

```
BFS(G, s)
    for all u ∈ V
        dist(u) = ∞
    dist(s) = 0
    Q = [s] (queue containing only s)
    while Q is not empty
        u = dequeue(Q)
        for all edge (u, v) ∈ E
            if dist(v) = ∞ then
                enqueue(Q, v)
                dist(v) = dist(u) + 1
```

Cost on a graph with n vertices and m edges:

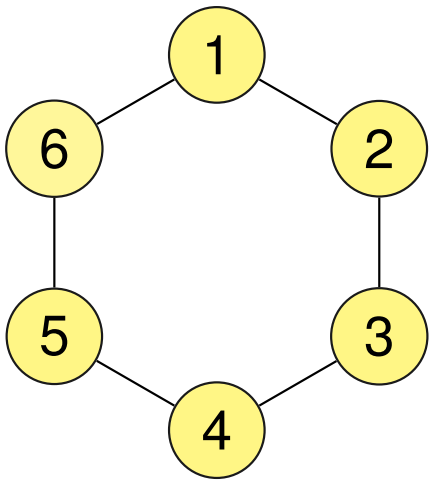
- Each vertex is added once to the queue: $\Theta(n)$ queue operations
- Each edge is visited once (directed) or twice (undirected): $\Theta(m)$

Total cost: $\Theta(n+m)$ (with adjacency lists).

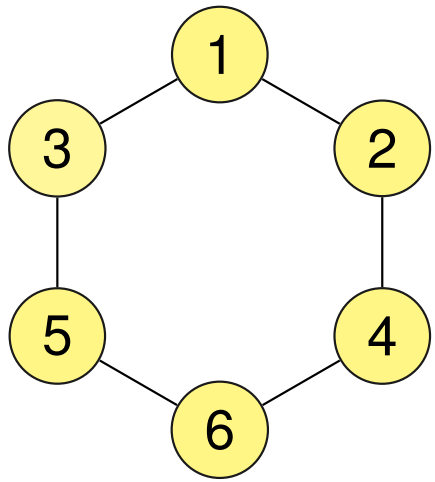
Breadth-First Search

We can now compare DFS with BFS:

- In DFS, the search returns when there are no more vertices to visit (a very long path may be traversed to visit a neighbor)
- In BFS, vertices are visited in increasing order of distance



DFS

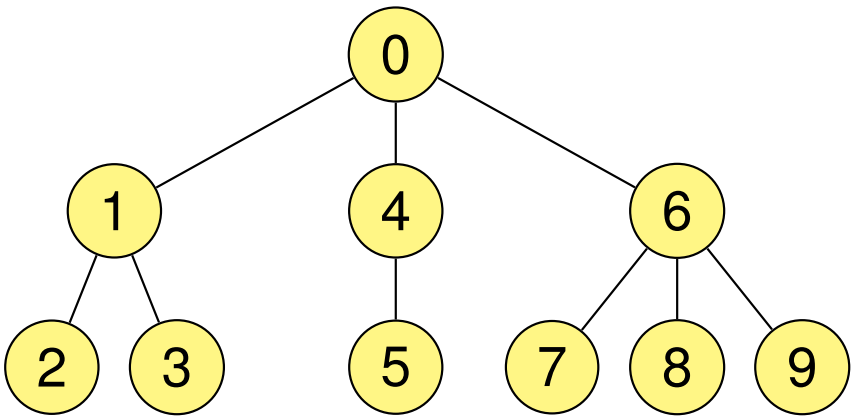


BFS

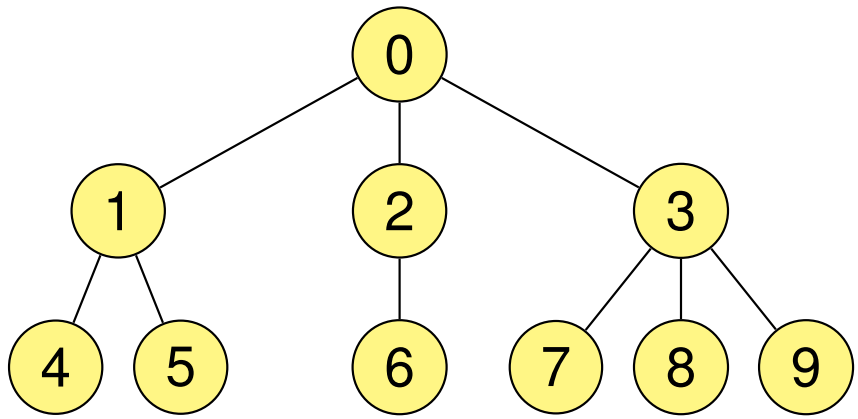
Breadth-First Search

In trees,

- DFS corresponds to preorder traversal and is the base of *backtracking*
- BFS traverses the vertices by levels



DFS



BFS

Breadth-First Search

DFS and BFS are very similar.

- They key difference is the use
  - of a stack in DFS
  - of a queue in BFS
- As in DFS, in BFS we only explore the connected component of the initial vertex. In order to explore the rest of the graph, we add an outer loop that restarts the search from other vertices.

Breadth-First Search

DFS and BFS are very similar.

- They key difference is the use
  - of a stack in DFS
  - of a queue in BFS
- As in DFS, in BFS we only explore the connected component of the initial vertex. In order to explore the rest of the graph, we add an outer loop that restarts the search from other vertices.

Breadth-First Search

We use the adjacency lists representation. No distance computation here.

```
list<int> bfs (const graph& G) {
    int n = G.size();
    list<int> L;
    queue<int> Q;
    vector<bool> enc(n, false);
    for (int u = 0; u < n; ++u) {
        if (not enc[u]) {
            Q.push(u); enc[u] = true;
            while (not Q.empty()) {
                int v = Q.front(); Q.pop();
                L.push_back(v);
                for (w : G[v])
                    if (not enc[w])
                        Q.push(w); enc[w] = true;
            }
        }
    return L; }
```

Cost of Breadth-First Search (as in DFS)

If graph G has

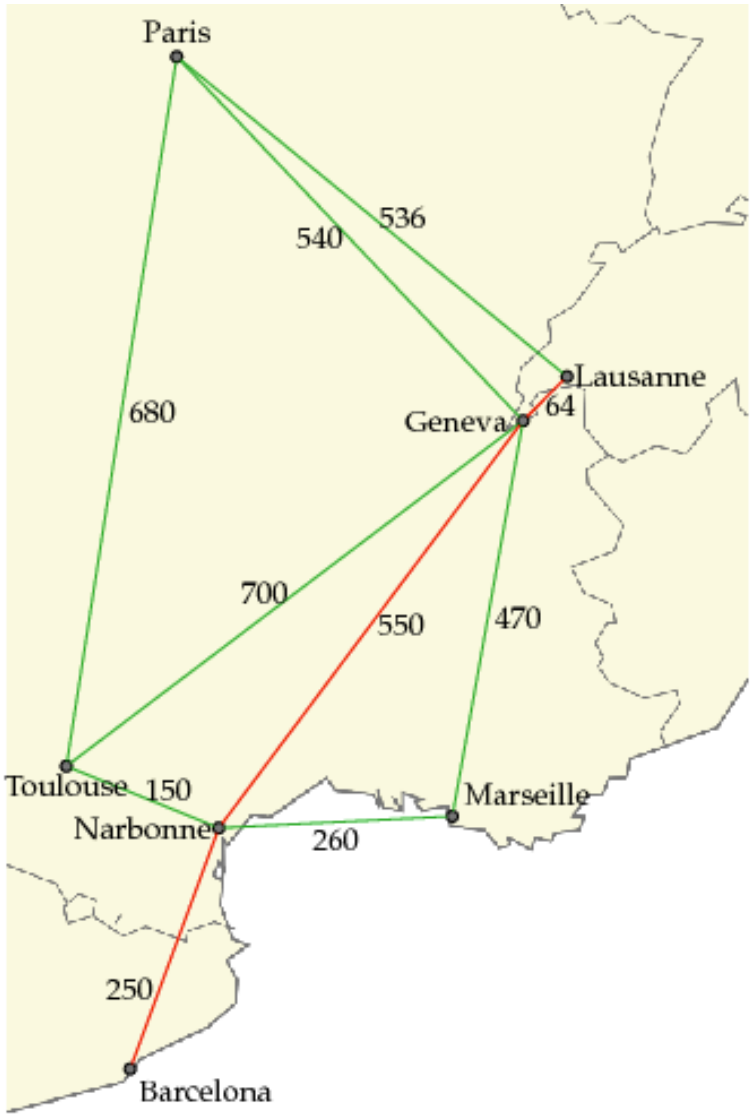
- k connected components (c.c.)
- $n = \sum_{i=1}^{k} n_i$ vertices (c.c. i has n_i vertices)
- $m = \sum_{i=1}^{k} m_i$ edge (c.c. i has m_i edges)

then the cost is

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

Dijkstra's Algorithm

Breadth-First Search treats all edges as if they had the same length, something not useful in applications that require shortest paths.

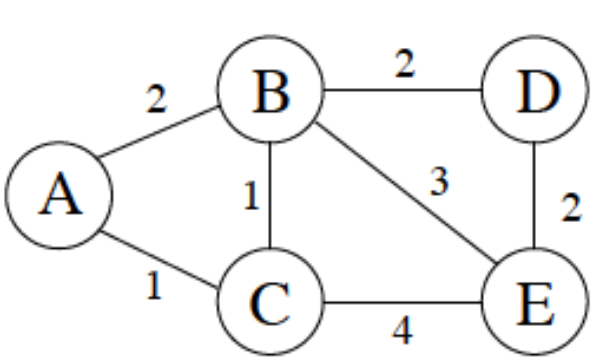


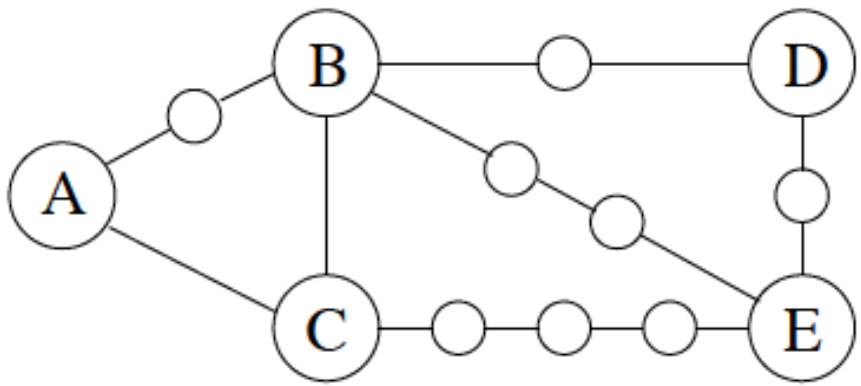
Dijkstra's Algorithm

In order to adapt Breadth-First Search to graphs with natural "distances" on the edges, we could transform the graph.

Trick to use BFS in graphs with distances

Break the edges into edges of length one, introducing additional vertices.

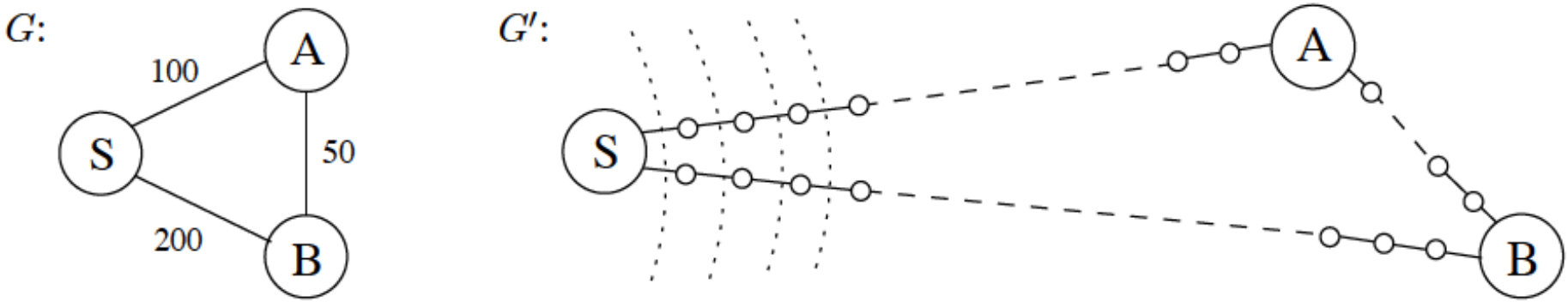




For each edge $e = (u, v)$ with distance $d(e)$, replace it with $d(e)$ edges with distance 1 by adding $d(e) - 1$ new vertices between u and v.

Dijkstra's Algorithm

The previous idea is not efficient for large distances. But we can imagine that we add alarms that alert us when we go from one original vertex to another one.



For example,

- We add an alarm for A and $T = 100$ and another one for B and $T = 200$
- When we reach A, the time of B is modified to $T = 150$

Dijkstra's Algorithm

"Alarms" algorithm

The following algorithm simulates BFS on G' without adding additional vertices.

Given graph G with distances in d and an initial vertex S:

1. Set the alarm of vertex S to 0
2. **Repeat** until no alarms are left:
   - Find and remove next alarm: vertex u and time T
   - **For each** neighbor v of u in G
     if v has no alarm **or** it is $> T + d(u, v)$, **then**
     readjust the alarm to $T + d(u, v)$

Excursion: Greedy Algorithms

The previous algorithm is an example of greedy algorithm.

Greedy approach

A greedy algorithm solves a problem in stages making, at each step, the choice that looks better. Usually, they consist of a simple strategy that is iterated.

Excursion: Greedy Algorithms

Example: Give change using minimum number of coins

To give change, we use coins of 2 and 1 euros, and then, coins of 50, 20, 10, 5, 2, and 1 cents.

Strategy: **While the amount to be paid is not exceeded, choose the coin with the largest value.**

Greedy algorithms are efficient, but do not always work.

Counterexample

If we add a 12 cents coin and we have to give 15 cents, we get:

- Greedy algorithm's solution: $12 + 2 + 1$ (3 coins).
- Optimal solution: $10 + 5$ (2 coins).

Excursion: Greedy Algorithms

Example: Give change using minimum number of coins

To give change, we use coins of 2 and 1 euros, and then, coins of 50, 20, 10, 5, 2, and 1 cents.

Strategy: **While the amount to be paid is not exceeded, choose the coin with the largest value.**

Greedy algorithms are efficient, but do not always work.

Counterexample

If we add a 12 cents coin and we have to give 15 cents, we get:

- Greedy algorithm's solution: $12 + 2 + 1$ (3 coins).
- Optimal solution: $10 + 5$ (2 coins).

Dijkstra's Algorithm

The alarms algorithm computes distances in any graph with positive distances on the edges.

In order to implement the alarms system, we choose a priority queue with the following operations:

- **create-queue**. Construct a priority queue with the available elements and keys.
- **insert**. Insert a new element to the queue.
- **remove-min**. Return the element with the smallest key and remove it.
- **decrement-key**. Update the decrement in the key of an element.
- **empty**. Returns a boolean indicating whether the queue is empty or not.

Exercise

How can we get a cost of $\Theta(\log n)$ for **decrement-key**?

Dijkstra's Algorithm

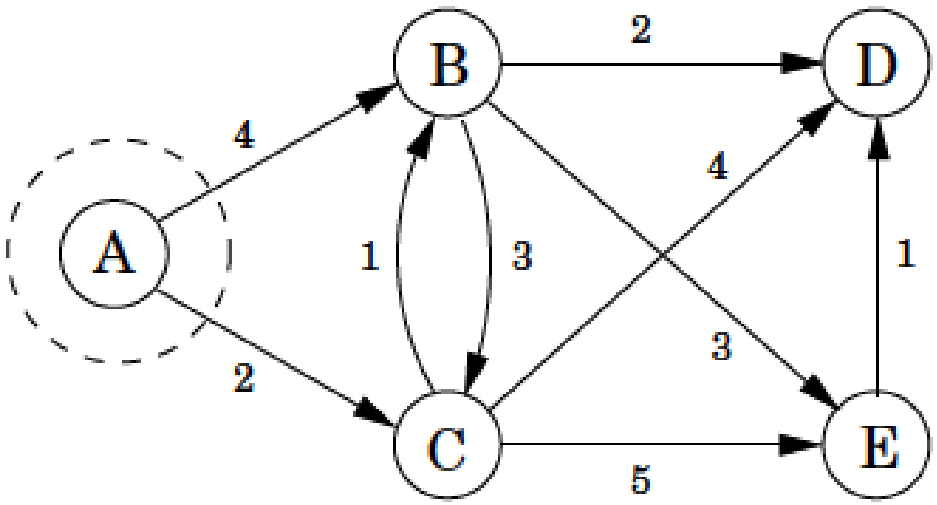
Dijkstra's Algorithm for shorts paths

Input: Graph $G = (V, E)$, directed or not; positive distances on the edges $\{d(u, v) \mid u, v \in V\}$; initial vertex s;
Output: For any vertex u reachable from s, $dist(u) =$ distance from s to u.

```
Dijkstra (G,d,s)
    for all vertex u ∈ V
        dist(u) = ∞
        prev(u) = nil
    dist(s) = 0
    H =create-queue(V)     (using dist as keys)
    while not empty(H)
        u =remove-min(H)
        for all edge (u, v) ∈ E
            if dist(v) > dist(u) + d(u, v)
                dist(v) = dist(u) + d(u, v)
                prev(v) = u
                decrement-key(H, v)
```

Dijkstra's Algorithm

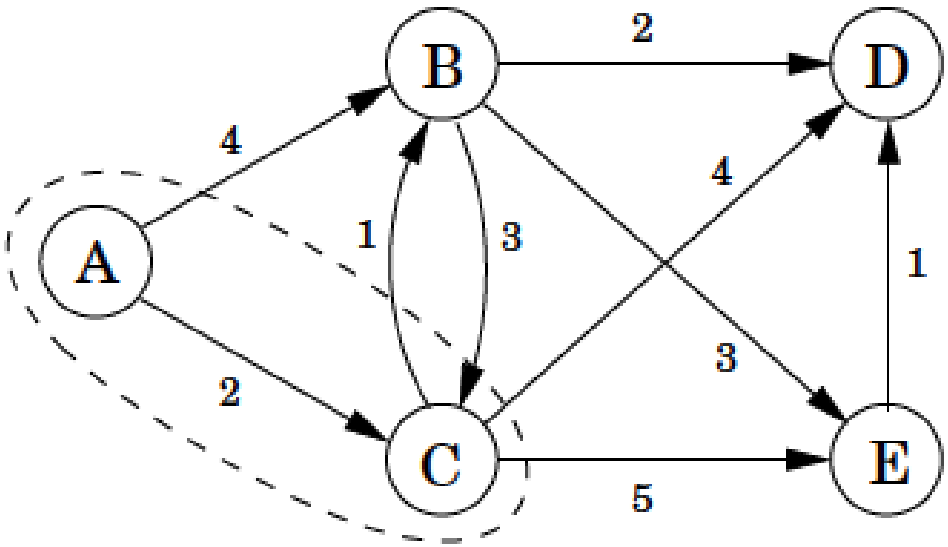
Example (*Algorithms*, Dasgupta et al.)



A: 0	D: ∞
B: 4	E: ∞
C: 2	

Dijkstra's Algorithm

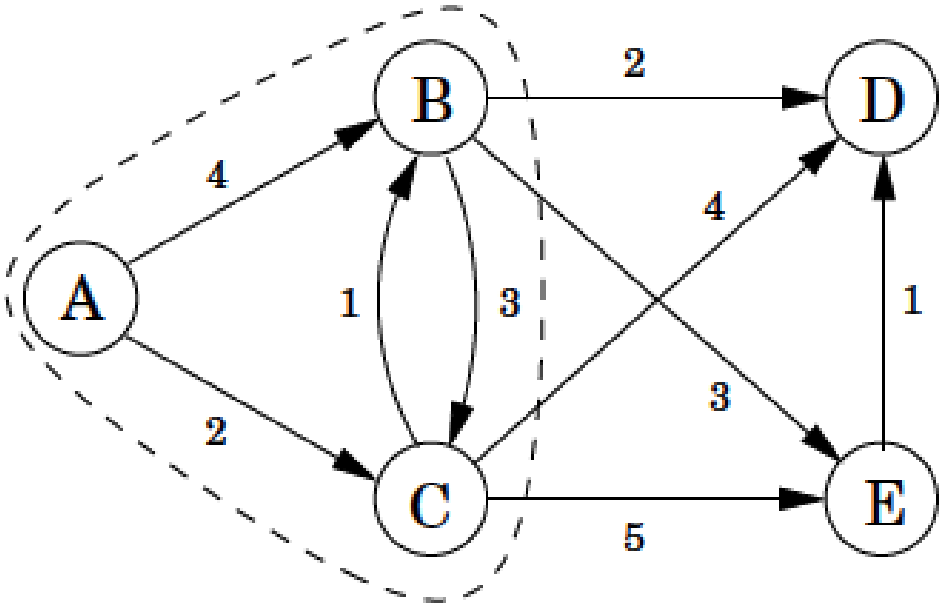
Example (*Algorithms*, Dasgupta et al.)



A: 0	D: 6
B: 3	E: 7
C: 2	

Dijkstra's Algorithm

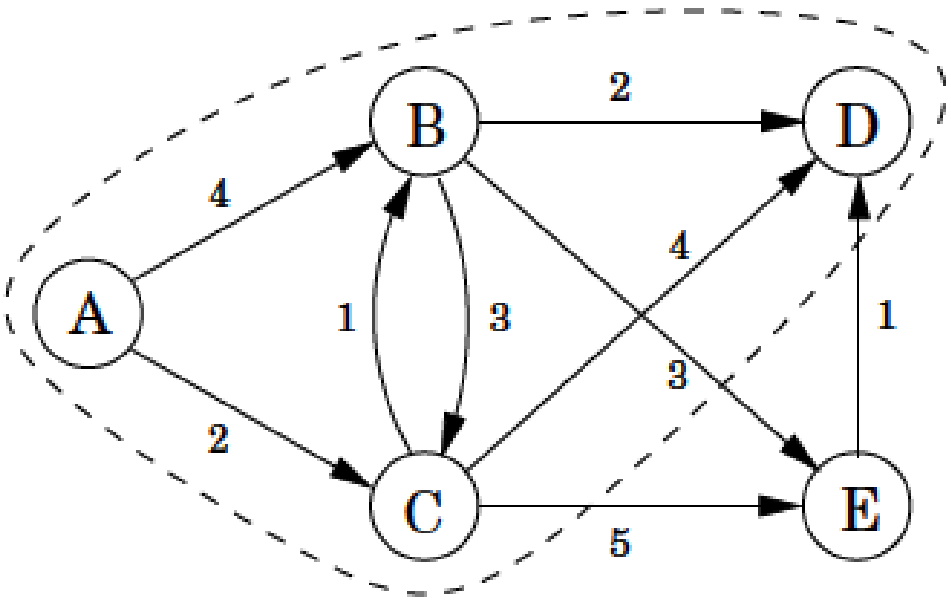
Example (*Algorithms*, Dasgupta et al.)



A: 0	D: 5
B: 3	E: 6
C: 2	

Dijkstra's Algorithm

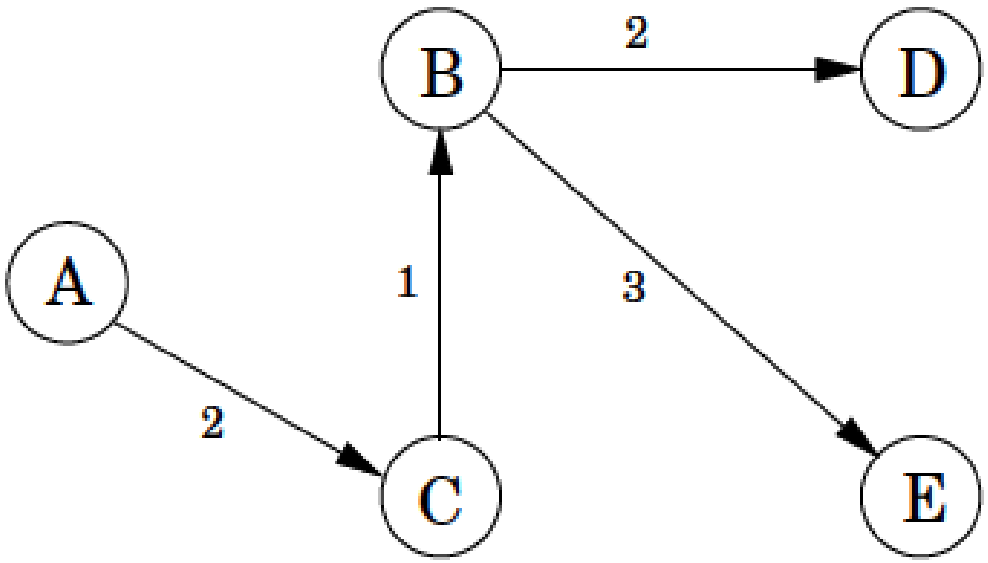
Example (*Algorithms*, Dasgupta et al.)



A: 0	D: 5
B: 3	E: 6
C: 2	

Dijkstra's Algorithm

Example (*Algorithms*, Dasgupta et al.)



Dijkstra's Algorithm

Cost of Dijkstra's algorithm

The structure is the same as in BFS, but we have to consider the costs of the priority queue operations. With a binary heap, the cost for a graph with n vertices and m edges is: $\Theta((n+m)\log n)$.

```
Dijkstra (G,d,s)
    for all vertex u ∈ V
        dist(u) = ∞
        prev(u) = nil
    dist(s) = 0
    H =create-queue(V)     (using dist as keys)
    while not empty(H)
        u =remove-min(H)
        for all edge (u, v) ∈ E
            if dist(v) > dist(u) + d(u, v)
                dist(v) = dist(u) + d(u, v)
                prev(v) = u
                decrement-key(H, v)
```

Dijkstra's Algorithm

Which priority queue is better?

Implementation	**remove-min**	**insert/ decrement-key**	$n\times$ **remove-min** $+(n+m)\times$ **insert**
vector	$O(n)$	$O(1)$	$O(n^2)$
binary heap	$O(\log n)$	$O(\log n)$	$O((n+m)\log n)$
d-ary heap *	$O(\frac{d \log n}{\log d})$	$O(\frac{\log n}{\log d})$	$O((nd+m)\frac{\log n}{\log d})$
Fibonacci heap	$O(\log n)$	$O(1)^+$	$O(n \log n + m)$

* Optimal choice for d: $d = m/n$.

+ *Amortized* cost ($O(1)$ on average throughout the whole algorithm).

Dijkstra's Algorithm: implementation

Dijkstra's Algorithm

Instead of decrementing a key, all decrements are stored in the priority queue, but a vector **S** will avoid considering a vertex twice.

```
typedef pair<double, int> WArc;          // weighted arc
typedef vector< vector<WArc> > WGraph;   // weighted graph

void dijkstra (const WGraph& G, int s, vector<int>& dist,
                        vector<int>& prev) {
    int n = G.size();
    dist = vector<int>(n, infinit); dist[s] = 0;
    prev = vector<int>(n, -1);
    vector<bool> S(n, false);
    priority_queue<WArc, vector<WArc>, greater<WArc> > Q;
    Q.push(WArc(0, s));
```

Dijkstra's Algorithm: implementation

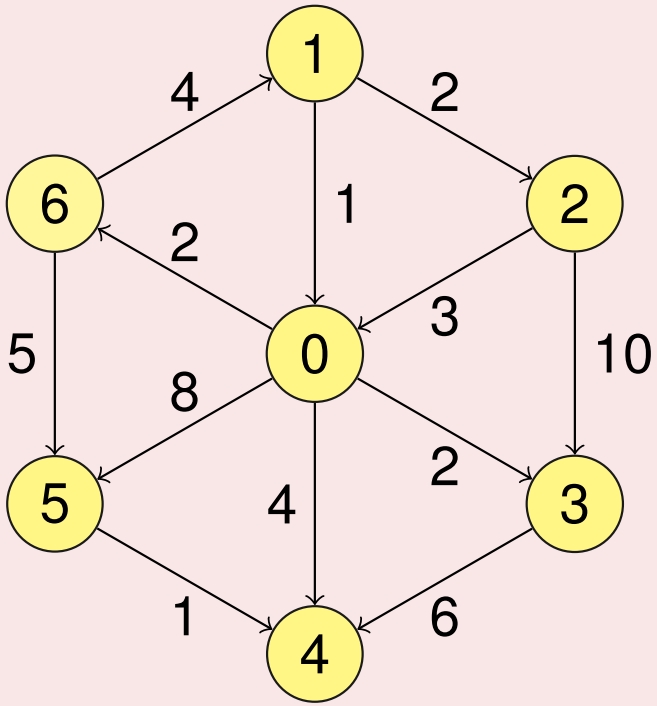
Dijkstra's Algorithm

```
    while (not Q.empty()) {
        int u = Q.top().second; Q.pop();
        if (not S[u]) {
            S[u] = true;
            for (Warc a : G[u]) {
                int v = a.second;
                int c = a.first;
                if (dist[v] > dist[u] + c) {
                    dist[v] = dist[u] + c;
                    prev[v] = u;
                    Q.push(WArc(dist[v], v));
}   }   }   }   }
```

Dijkstra's Algorithm

Exercise

Apply Dijkstra's algorithm to find the shortest paths starting from vertex 1 in the following graph.



Minimum Spanning Trees

Let $G = (V, E)$ be a nondirected connected graph with weights $\omega : E \to \mathbb{R}$.

Definition

A spanning tree of G is a subgraph $T = (V, A)$ of G, with $A \subseteq E$, that is connected and acyclic.

Observe that T is a tree and contains all the vertices of G

Definició

A minimum spanning tree (MST) of G is a spanning tree $T = (V, A)$ of G whose total weight

$$\omega(T) = \sum_{e \in A} \omega(e)$$

is minimum among all spanning trees of G.

Minimum Spanning Trees

Let $G = (V, E)$ be a nondirected connected graph with weights $\omega : E \to \mathbb{R}$.

Definition

A spanning tree of G is a subgraph $T = (V, A)$ of G, with $A \subseteq E$, that is connected and acyclic.

Observe that T is a tree and contains all the vertices of G

Definició

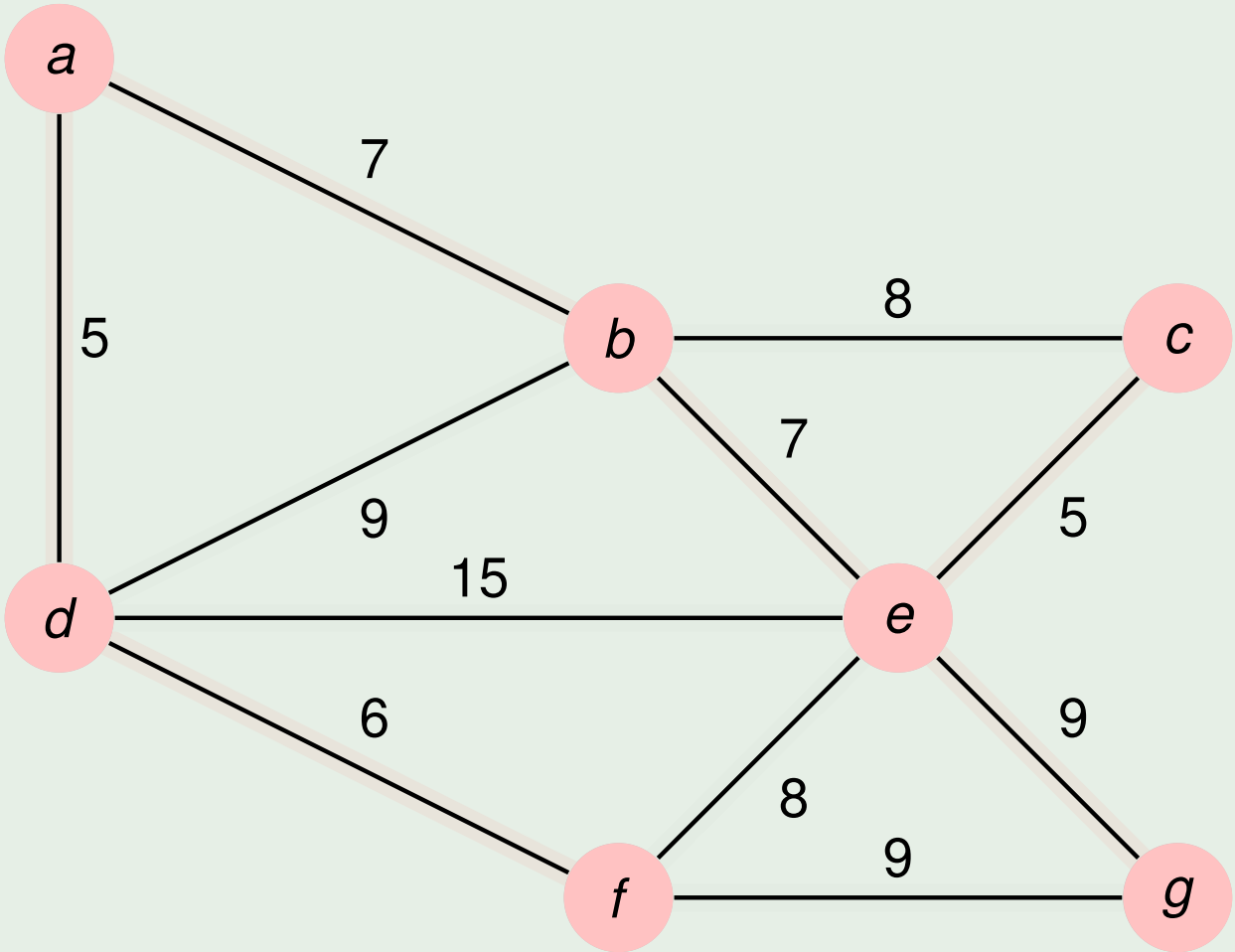
A minimum spanning tree (MST) of G is a spanning tree $T = (V, A)$ of G whose total weight

$$\omega(T) = \sum_{e \in A} \omega(e)$$

is minimum among all spanning trees of G.

Minimum Spanning Trees

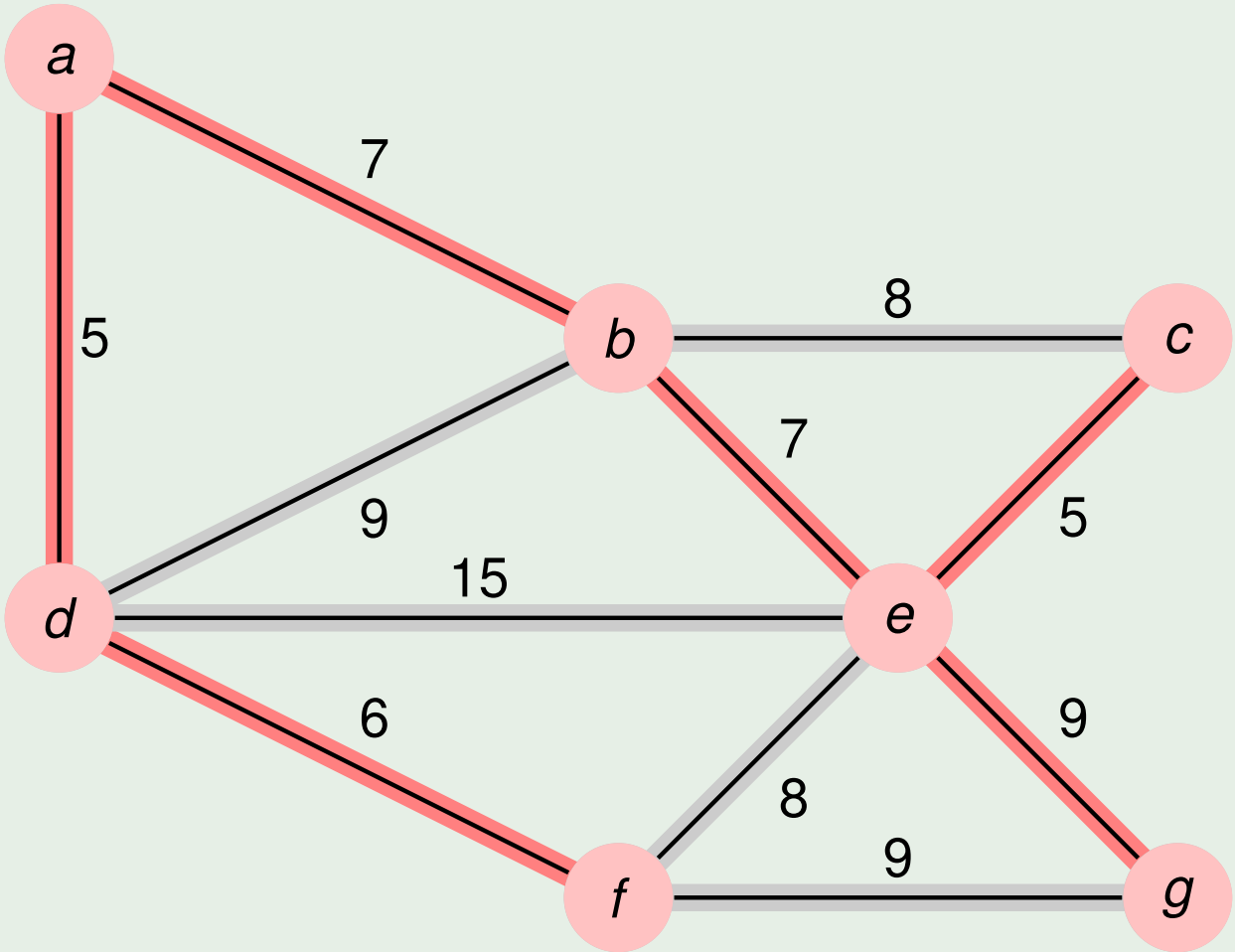
Two minimum spanning trees



Font: `www.texample.net/tikz/examples/author/kjell-magne-fauske`

Minimum Spanning Trees

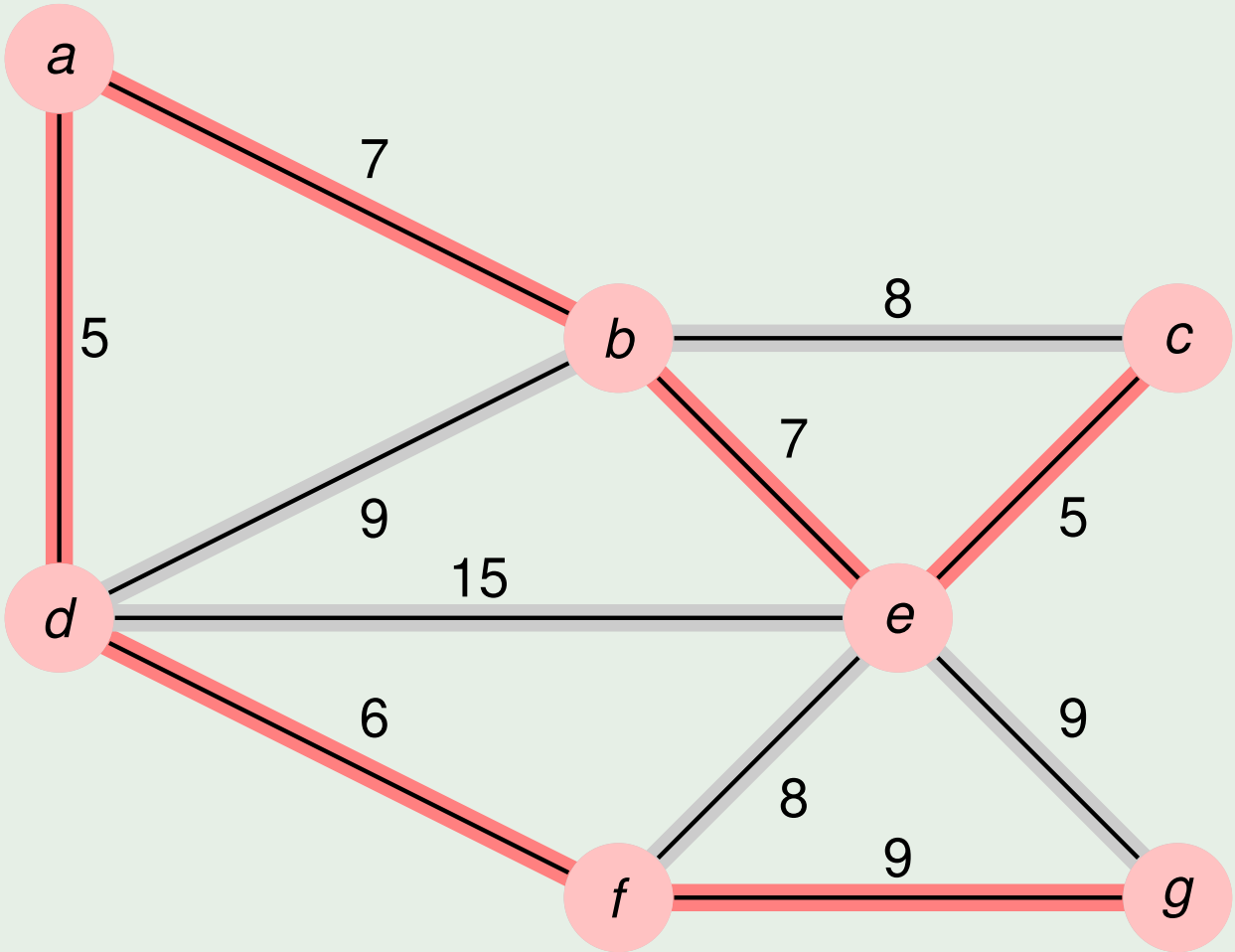
Two minimum spanning trees



Font: `www.texample.net/tikz/examples/author/kjell-magne-fauske`

Minimum Spanning Trees

Two minimum spanning trees



Font: `www.texample.net/tikz/examples/author/kjell-magne-fauske`

Minimum Spanning Trees

There are many different algorithms to calculate MSTs. All of them follow a greedy approach, that is, they repeat an action that looks optimal at each step.

```
A = ∅;
Cand = E;
while (|A| ≠ |V| − 1) {
   choose e ∈ Cand s.t. T = (V, A ∪ {e}) has no cycles;
   A = A ∪ {e};
   Cand = Cand − {e}
}
```

The following concepts lead to a greedy approach to choose an edge from a set of candidates.

Minimum Spanning Trees

There are many different algorithms to calculate MSTs. All of them follow a greedy approach, that is, they repeat an action that looks optimal at each step.

```
A = ∅;
Cand = E;
while (|A| ≠ |V| − 1) {
   choose e ∈ Cand s.t. T = (V, A ∪ {e}) has no cycles;
   A = A ∪ {e};
   Cand = Cand − {e}
}
```

The following concepts lead to a greedy approach to choose an edge from a set of candidates.

Minimum Spanning Trees

Definition

A subset of edges $A \subseteq E$ is promising if A is a subset of the edges of an MST of G.

Definition

A cut in a graph $G = (V, E)$ is a partition of V, that is, a pair (C, C') such that

- $C \cup C' = V$
- $C \cap C' = \emptyset$

Definition

An edge e respects a cut (C, C') if both endpoints of e belong to C or both belong to C'; otherwise, we say that e crosses the cut.

A set of edges A respects a cut if all edges in A respect it.

Minimum Spanning Trees

Definition

A subset of edges $A \subseteq E$ is promising if A is a subset of the edges of an MST of G.

Definition

A cut in a graph $G = (V, E)$ is a partition of V, that is, a pair (C, C') such that

- $C \cup C' = V$
- $C \cap C' = \emptyset$

Definition

An edge e respects a cut (C, C') if both endpoints of e belong to C or both belong to C'; otherwise, we say that e crosses the cut.

A set of edges A respects a cut if all edges in A respect it.

Minimum Spanning Trees

Definition

A subset of edges $A \subseteq E$ is promising if A is a subset of the edges of an MST of G.

Definition

A cut in a graph $G = (V, E)$ is a partition of V, that is, a pair (C, C') such that

- $C \cup C' = V$
- $C \cap C' = \emptyset$

Definition

An edge e respects a cut (C, C') if both endpoints of e belong to C or both belong to C'; otherwise, we say that e crosses the cut.

A set of edges A respects a cut if all edges in A respect it.

Minimum Spanning Trees

Theorem

Let A be a promising set of edges that respects a cut (C, C') of G.
Let e be an edge with minimum weight among those crossing the cut (C, C').
Therefore, $A \cup \{e\}$ is promising.

Minimum Spanning Trees

Theorem

Let A be a promising set of edges that respects a cut (C, C') of G.
Let e be an edge with minimum weight among those crossing the cut (C, C').
Therefore, $A \cup \{e\}$ is promising.

This theorem gives us a recipee to design algorithms for an MST:

1. start with an empty set of edges A
2. define which is the initial cut of G
3. while the cut is not trivial
   - choose an edge e with minimum weight among those crossing the cut
   - add e to A and move to C the endpoint of e belonging to C'
     (since A respects the cut, this cannot create a cycle)

Minimum Spanning Trees

Theorem

Let A be a promising set of edges that respects a cut (C, C') of G.
Let e be an edge with minimum weight among those crossing the cut (C, C').
Therefore, $A \cup \{e\}$ is promising.

Demostració

Let A' be the set of edges of an MST T such that $A \subseteq A'$.
(T exists because we assume that A is promising)

Consider two cases:

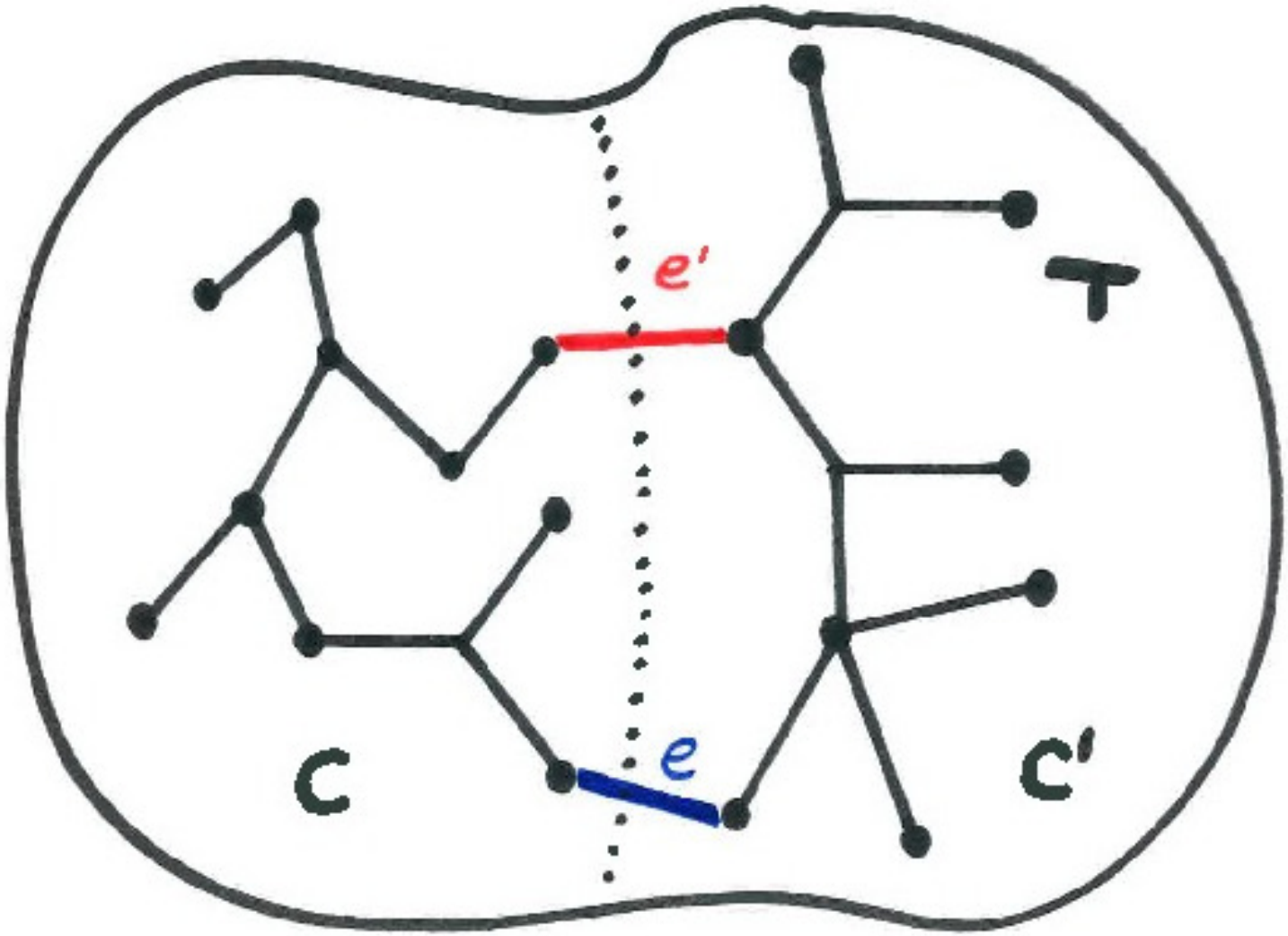
1. $e \in A'$

   Then, $A \cup \{e\}$ is promising.

2. $e \notin A'$

   Since A respects the cut, there is an edge $e' \in A' - A$ crossing the cut (otherwise, T would not be connected).

Minimum Spanning Trees



Minimum Spanning Trees

Theorem

Let A be a promising set of edges that respects a cut (C, C') of G.
Let e be an edge with minimum weight among those crossing the cut (C, C').
Therefore, $A \cup \{e\}$ is promising.

Proof

2. The subgraph $T' = (V, A' - \{e'\} \cup \{e\})$ is a spanning tree and, then,

$$\omega(T) \leq \omega(T') = \omega(T) - \omega(e') + \omega(e).$$

Thus, $\omega(e') \leq \omega(e)$.

But since e has minimum weight, we have $\omega(e) \leq \omega(e')$.

Therefore, $\omega(T) = \omega(T')$ and T' is an MST.

Then, $A \cup \{e\}$ is promising.

Minimum Spanning Trees

Theorem

Let A be a promising set of edges that respects a cut (C, C') of G.
Let e be an edge with minimum weight among those crossing the cut (C, C').
Therefore, $A \cup \{e\}$ is promising.

Proof

2. The subgraph $T' = (V, A' - \{e'\} \cup \{e\})$ is a spanning tree and, then,

$$\omega(T) \leq \omega(T') = \omega(T) - \omega(e') + \omega(e).$$

Thus, $\omega(e') \leq \omega(e)$.

But since e has minimum weight, we have $\omega(e) \leq \omega(e')$.

Therefore, $\omega(T) = \omega(T')$ and T' is an MST.

Then, $A \cup \{e\}$ is promising.

Minimum Spanning Trees

Theorem

Let A be a promising set of edges that respects a cut (C, C') of G.
Let e be an edge with minimum weight among those crossing the cut (C, C').
Therefore, $A \cup \{e\}$ is promising.

Proof

2. The subgraph $T' = (V, A' - \{e'\} \cup \{e\})$ is a spanning tree and, then,

$$\omega(T) \leq \omega(T') = \omega(T) - \omega(e') + \omega(e).$$

Thus, $\omega(e') \leq \omega(e)$.

But since e has minimum weight, we have $\omega(e) \leq \omega(e')$.

Therefore, $\omega(T) = \omega(T')$ and T' is an MST.

Then, $A \cup \{e\}$ is promising.

Minimum Spanning Trees

Theorem

Let A be a promising set of edges that respects a cut (C, C') of G.
Let e be an edge with minimum weight among those crossing the cut (C, C').
Therefore, $A \cup \{e\}$ is promising.

Proof

2. The subgraph $T' = (V, A' - \{e'\} \cup \{e\})$ is a spanning tree and, then,

$$\omega(T) \leq \omega(T') = \omega(T) - \omega(e') + \omega(e).$$

Thus, $\omega(e') \leq \omega(e)$.

But since e has minimum weight, we have $\omega(e) \leq \omega(e')$.

Therefore, $\omega(T) = \omega(T')$ and T' is an MST.

Then, $A \cup \{e\}$ is promising.

Prim's Algorithm

In Prim's algorithm (also known as Prim-Jarník's algorithm), we keep a subset of visited vertices.

- The set of vertices is then divided between visited and unvisited
- Each iteration of the algorithm chooses the edge of minimum weight between the ones that link visited and unvisited vertices
- By the theorem, the algorithm is correct

Prim's Algorithm

In Prim's algorithm (also known as Prim-Jarník's algorithm), we keep a subset of visited vertices.

- The set of vertices is then divided between visited and unvisited
- Each iteration of the algorithm chooses the edge of minimum weight between the ones that link visited and unvisited vertices
- By the theorem, the algorithm is correct

Prim's Algorithm

In Prim's algorithm (also known as Prim-Jarník's algorithm), we keep a subset of visited vertices.

- The set of vertices is then divided between visited and unvisited
- Each iteration of the algorithm chooses the edge of minimum weight between the ones that link visited and unvisited vertices
- By the theorem, the algorithm is correct

Prim's Algorithm

In Prim's algorithm (also known as Prim-Jarník's algorithm), we keep a subset of visited vertices.

- The set of vertices is then divided between visited and unvisited
- Each iteration of the algorithm chooses the edge of minimum weight between the ones that link visited and unvisited vertices
- By the theorem, the algorithm is correct

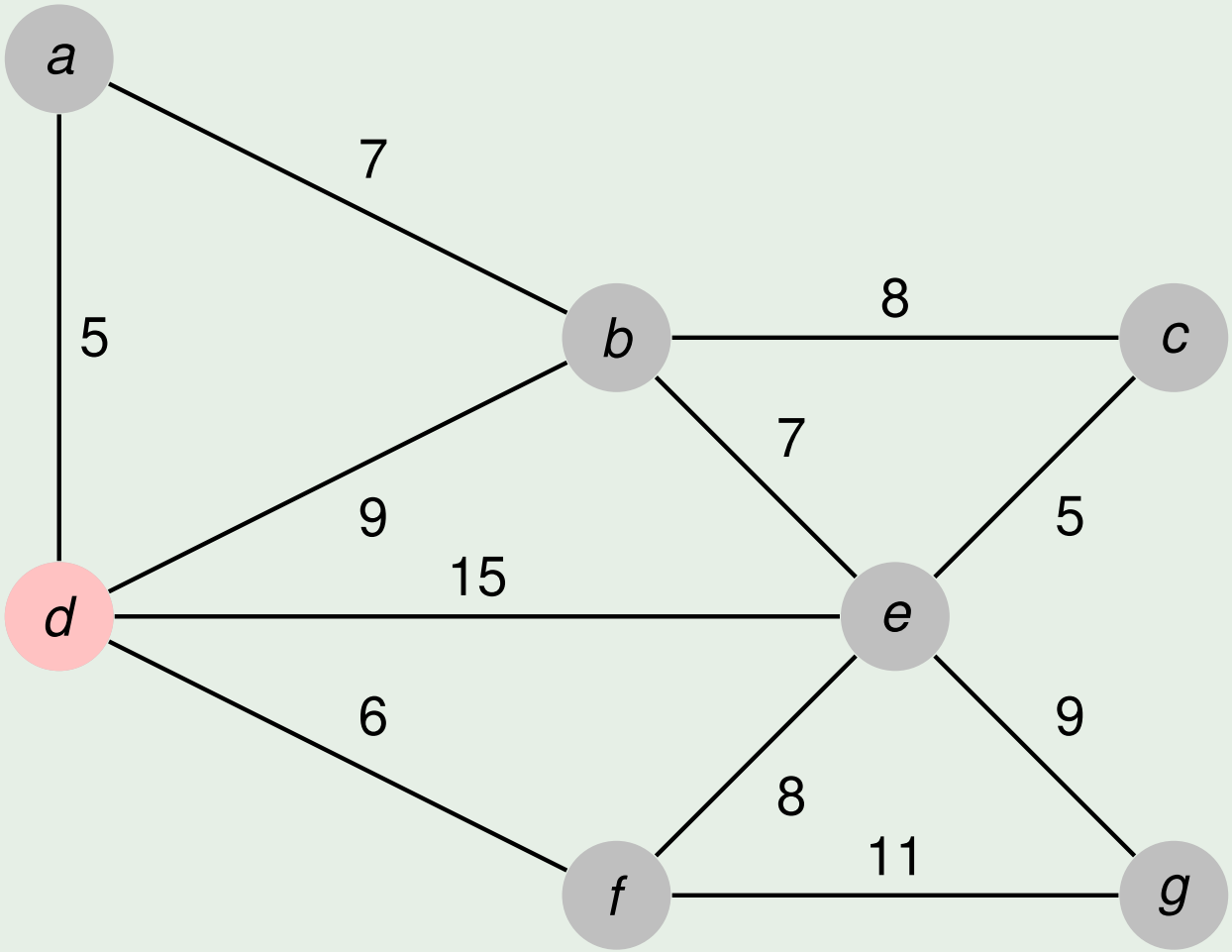
Prim's Algorithm

The graph is represented by adjacency lists. The pairs are (cost, vertex).

```
typedef pair<int,int> P;
int mst(const vector<vector<P>>& g) { // Ret. cost of an MST
  vector<bool> vis(n, false);
  vis[0] = true;
  priority_queue<P, vector<P>, greater<P> > pq;
  for (P x : g[0]) pq.push(x);
  int sz = 1;
  int sum = 0;
  while (sz < n) {
    int c = pq.top().first;
    int x = pq.top().second;
    pq.pop();
    if (not vis[x]) {
      vis[x] = true;
      for (P y : g[x]) pq.push(y);
      sum += c;
      ++sz; } }
  return sum; }
```

Prim's Algorithm

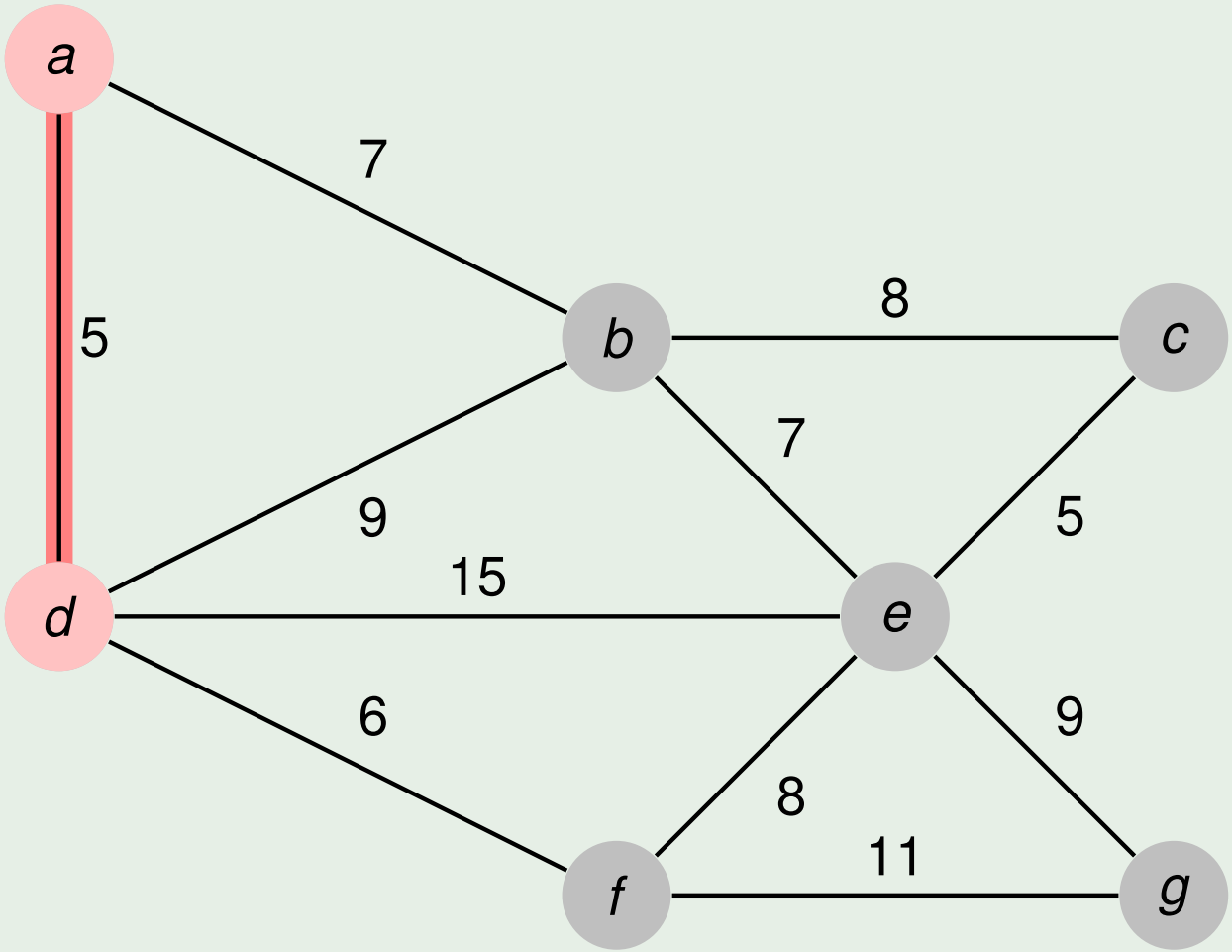
Example



Source: `www.texample.net/tikz/examples/author/kjell-magne-fauske`

Prim's Algorithm

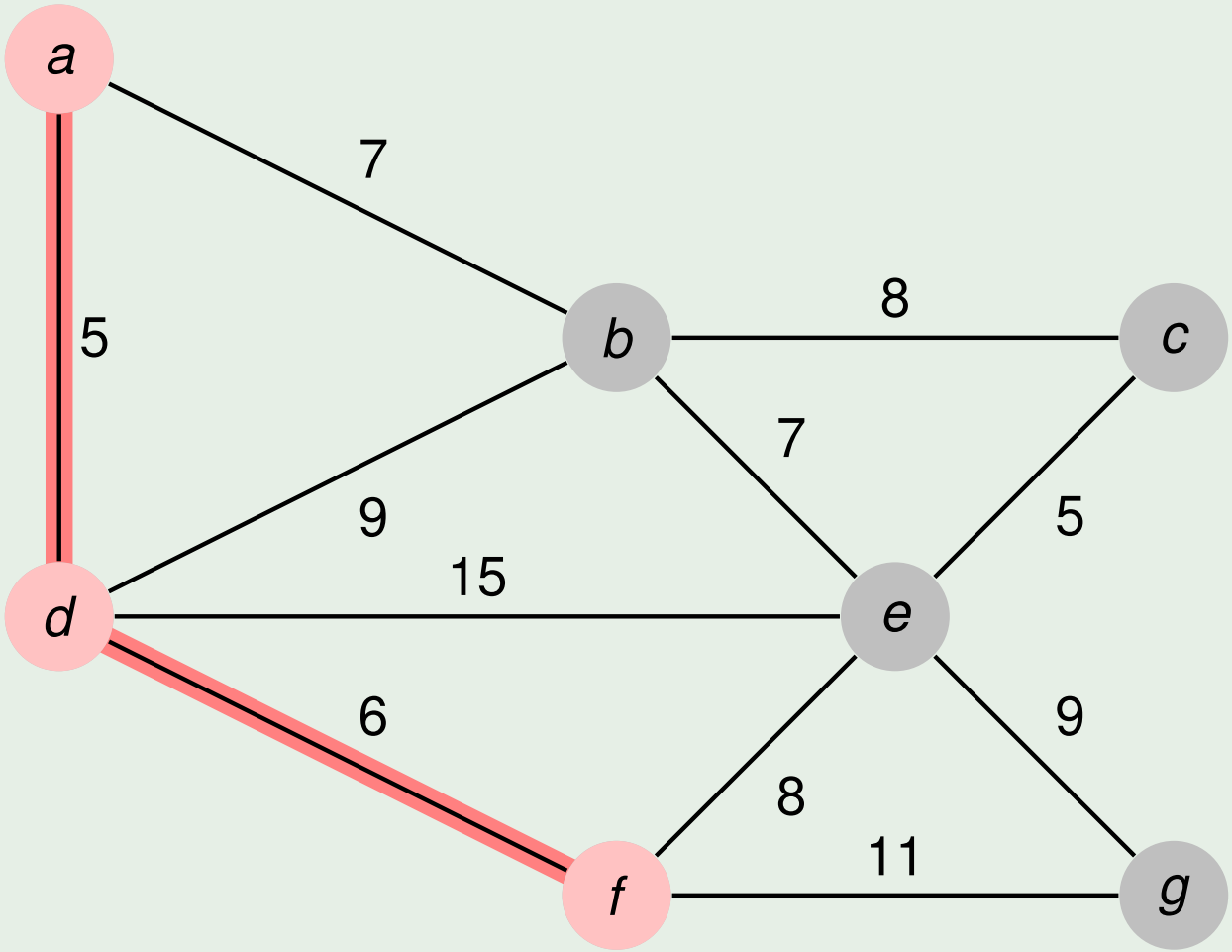
Example



Source: `www.texample.net/tikz/examples/author/kjell-magne-fauske`

Prim's Algorithm

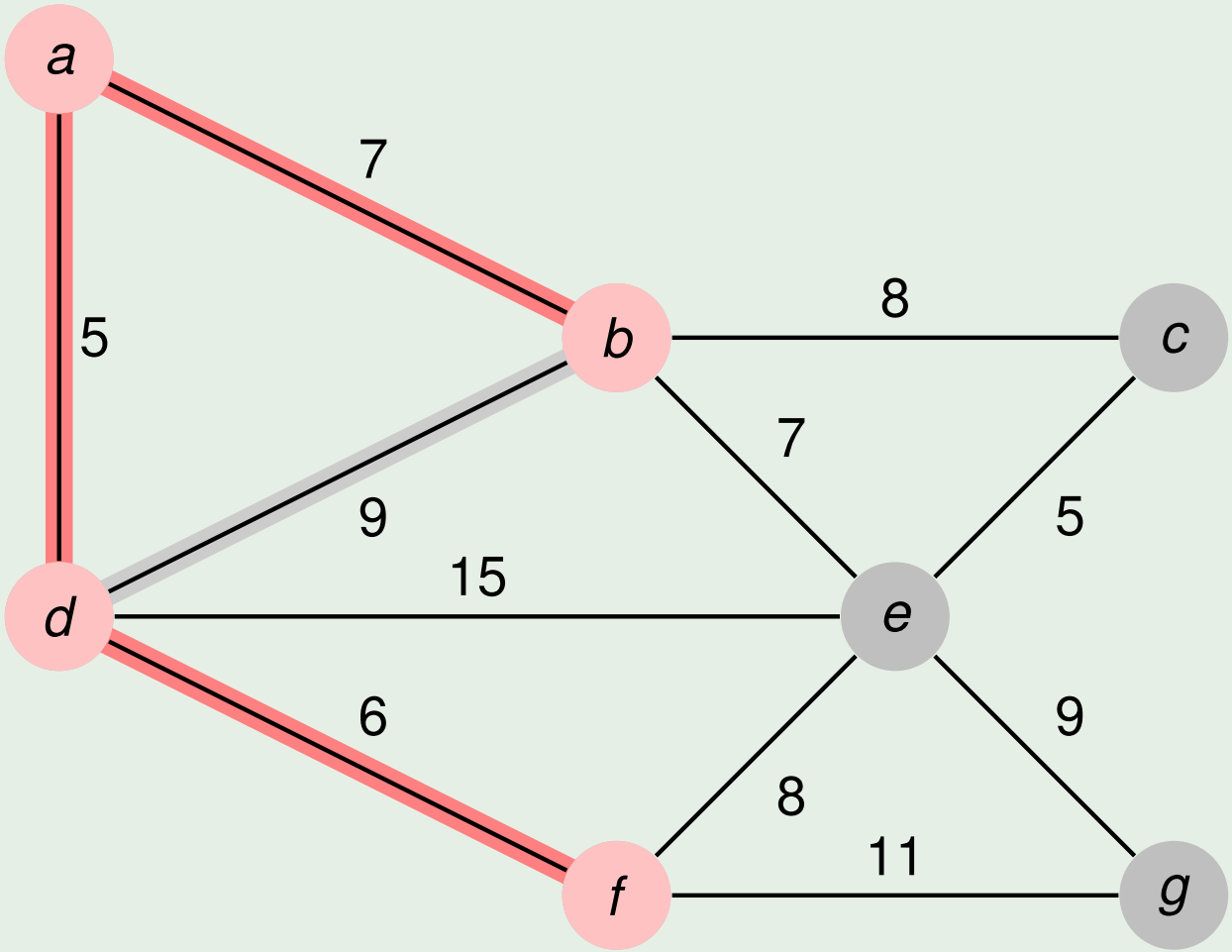
Example



Source: `www.texample.net/tikz/examples/author/kjell-magne-fauske`

Prim's Algorithm

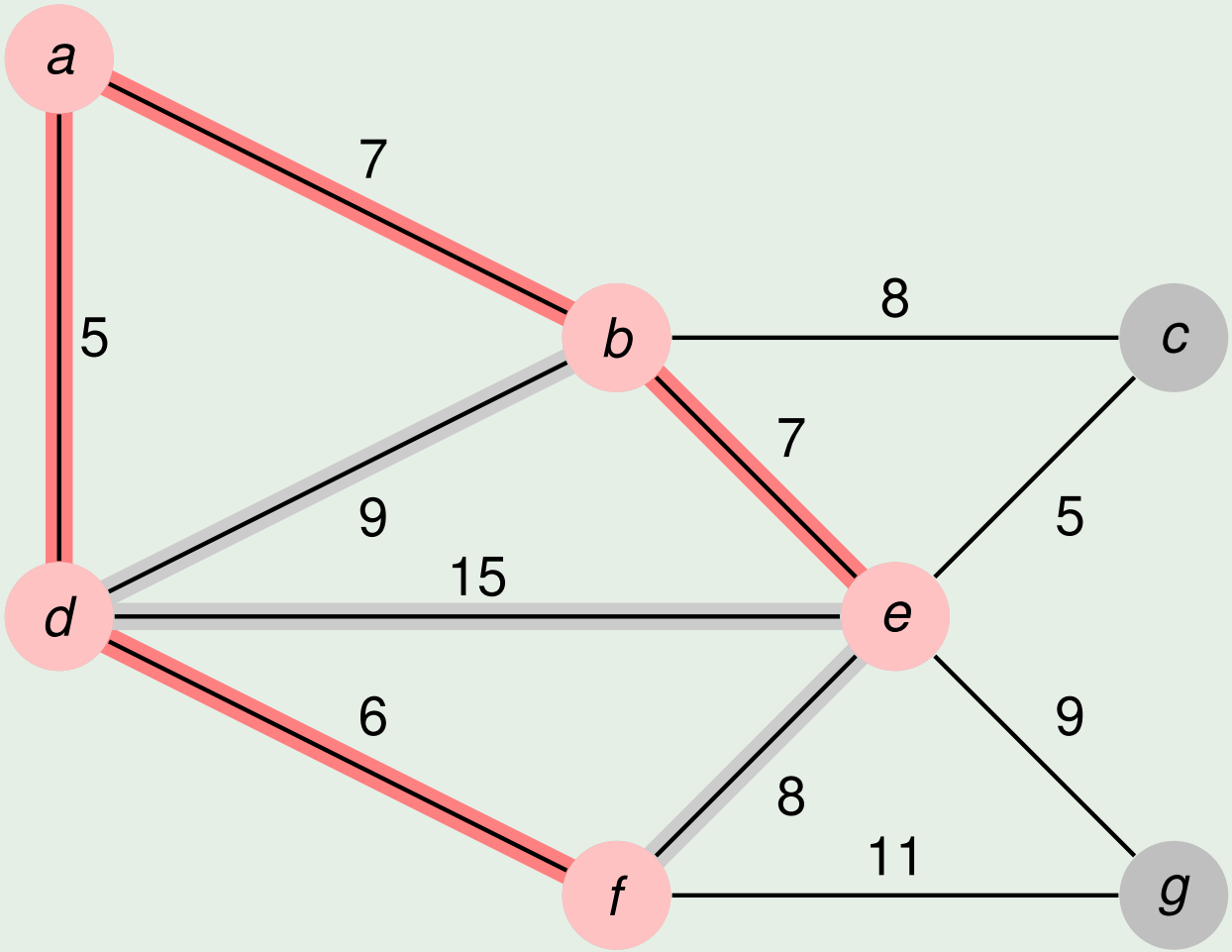
Example



Source: `www.texample.net/tikz/examples/author/kjell-magne-fauske`

Prim's Algorithm

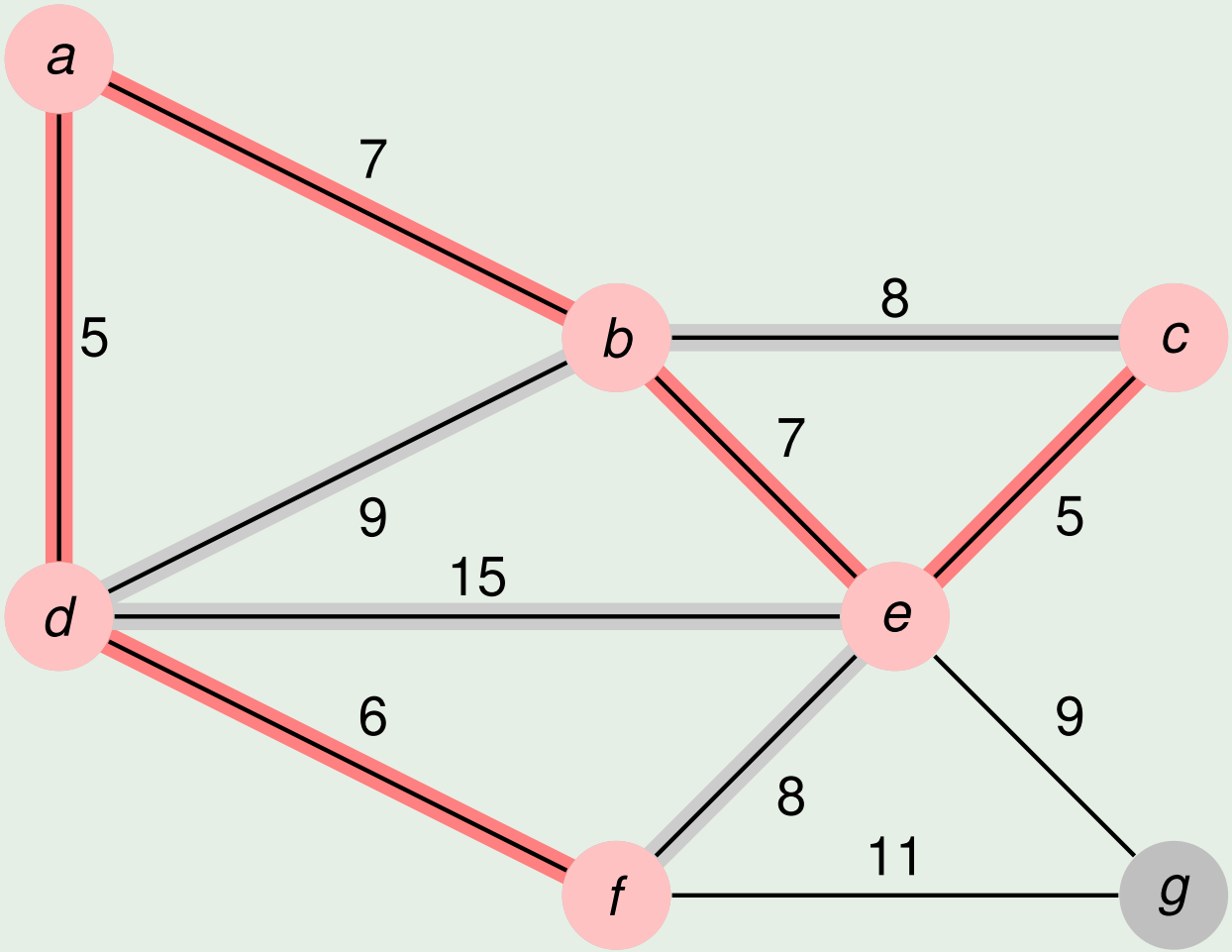
Example



Source: `www.texample.net/tikz/examples/author/kjell-magne-fauske`

Prim's Algorithm

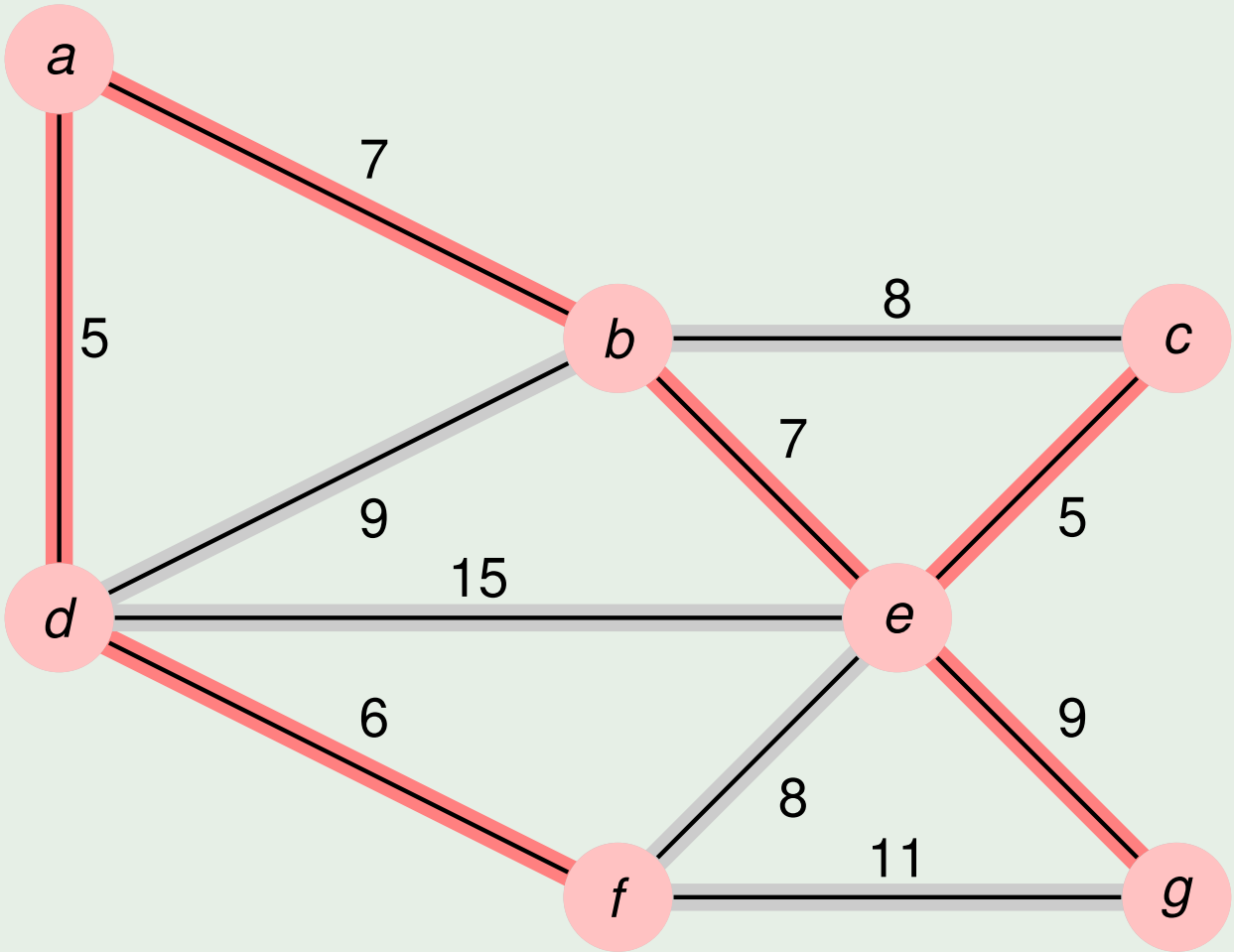
Example



Source: `www.texample.net/tikz/examples/author/kjell-magne-fauske`

Prim's Algorithm

Example



Source: `www.texample.net/tikz/examples/author/kjell-magne-fauske`

Algorisme de Prim

Cost analysis

The cost is dominated by the cost of the loop, with $O(m)$ iterations.

We count the selection of the edge and the visit of new candidates in each iteration separately:

- **Selection of the edge**. The cost is $O(\log m)$.
- **Visit of new candidates**. Each vertex is marked exactly once. When a vertex x is visited, adding new candidates costs $O(\deg(x) \log m)$.

Total cost is then,

$$O(m \log m) + \sum_{x \in V} O(\deg(x) \log m) = O(m \log m) = O(m \log n).$$

In fact, it can be proved that the cost in the worst case is $\Theta(m \log n)$.