Graph theory

Introduction and basic algorithms

Lorenzo Ferrari

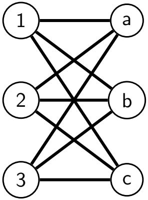
Campus Bornholm

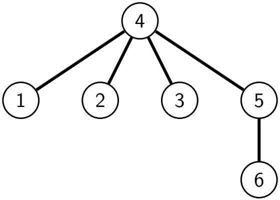
November 19, 2021

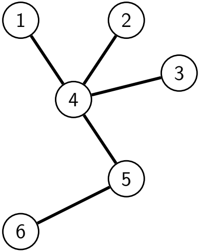
Table of contents

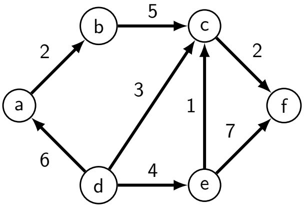
Table of contents

Examples









Why graph theory?

- A huge number of problems can be converted in a graph problem
- Graph theory is fun `\(^-^)/`

We will study problems in abstract form. Their application can be found in the most diverse areas.

Typical graphs problems

- Given the description of a city, find the shortest path between locations A and B or determine that it is impossible to reach B from A.
- On an electronic board, choose a set of connections whose sum in length is minimal and which allows to pass through all points of interest.
- Given dependency relationships, find a suitable order to install some packages (or attend some classes) or determine that it is impossible.

Definition: Graph

Graph $G = (V, E)$

A graph is defined as a pair of sets:

- V is a set of vertexes/nodes
- E is a set of edges

Definition: Vertexes and Edges

Vertex

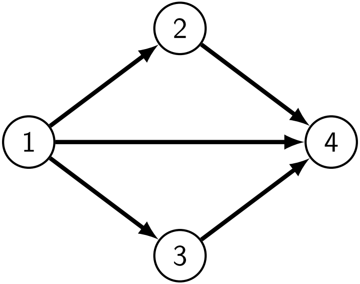
- Vertexes are also called *nodes*
- Vertexes are denoted with labels

Edge

- Each edge is defined by a pair of vertexes
- An edge *connects* the vertexes that define it
- In some cases, the vertexes can be the same

Example

- $G = (V, E)$
- $V = \{1, 2, 3, 4\}$
- $E = \{(1,2), (1,3), (1,4), (2,4), (3,4)\}$



Directed and Undirected graphs: definition

Directed graph $G = (V, E)$

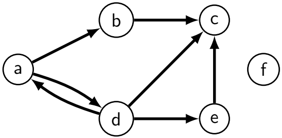
- E is a set of *ordered* pairs (u, v) of nodes

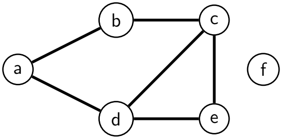
Undirected graph $G = (V, E)$

- E is a set of *unordered* pairs $[u, v]$ of nodes

```
V = { a,b,c,d,e,f }
E = { (a,b),(a,d),(b,c),
      (d,a),(d,c),(d,e),
      (e,c) }
```

```
V = { a,b,c,d,e,f }
E = { [a,b],[a,d],[b,c],
      [c,d],[c,e],[e,d] }
```

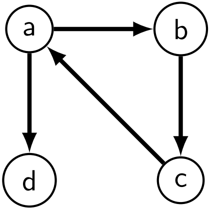




Cyclic and Acyclic graphs: definition

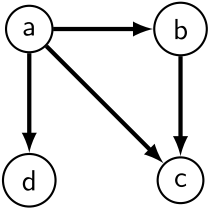
Cyclic graph

- Contains cycles



Acyclic graph

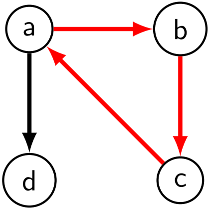
- Contains no cycles



Cyclic and Acyclic graphs: definition

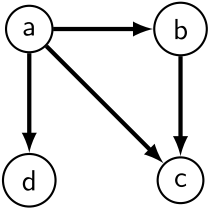
Cyclic graph

- Contains cycles



Acyclic graph

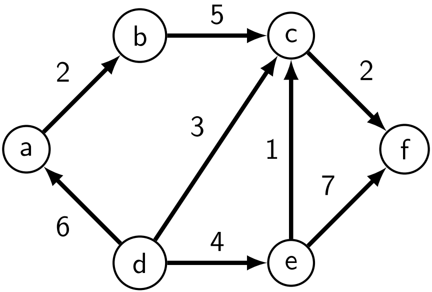
- Contains no cycles



Weighted graphs: definition

Weighted graph

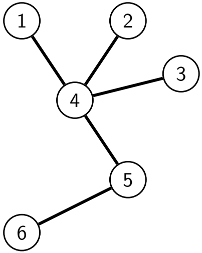
- Each edge is assigned a *wheight*
- Weigth typically shows cost of traversing



Trees: definition

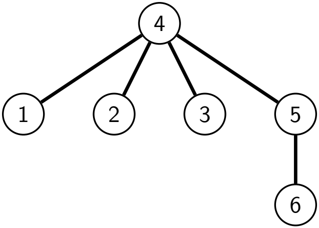
Tree

- *Connected* graph with $m = n - 1$



Rooted tree

- *Connected* graph with $m = n - 1$ in which some special node is designed as root



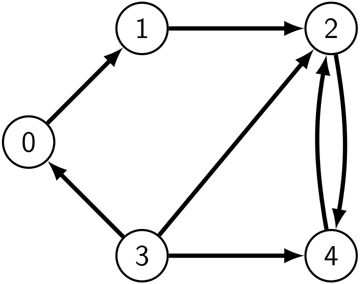
Representations

Two possible "classic" implementations

- Adjacency matrix
- Adjacency list

Adjacency matrix

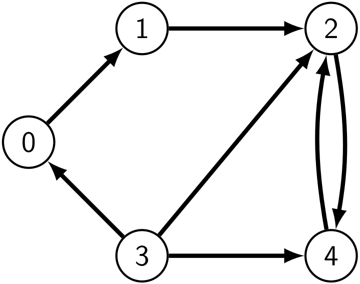
$$m_{uv} = \begin{cases} 1, & \text{if } (u, v) \in E \\ 0, & \text{if } (u, v) \notin E \end{cases}$$



	0	**1**	**2**	**3**	**4**
0	0	1	0	0	0
1	0	0	1	0	0
2	0	0	0	0	4
3	1	0	1	0	1
4	0	0	1	0	0

Adjacency list

$$G.adj(u) = \{v \mid (u, v) \in E\}$$



$\boxed{0} \rightarrow \boxed{1}$

$\boxed{1} \rightarrow \boxed{2}$

$\boxed{2} \rightarrow \boxed{4}$

$\boxed{3} \rightarrow \boxed{0}\boxed{2}\boxed{4}$

$\boxed{4} \rightarrow \boxed{2}$

Table of contents

Breadth-first search

Problem definition

Given a graph $G = (V, E)$ and a vertex $r \in V$ (root), visit exactly once all the vertexes of the graph that can be reached from r.

Breadth-first search (BFS)

Traverse the graph by visiting the nodes by levels: first nodes at distance 1 from the source, then distance 2, etc.

- Application: shortest distances

Breadth-first search

```python
def bfs(G, r):
    Q = deque()
    Q.append(r)
    visited = {r}
    while len(Q) > 0:
        u = Q.popleft()
        for v in G.adj(u):
            if not v in visited:
                visited.add(v)
                Q.append(v)
```

Depth-first search

Problem definition

Given a graph $G = (V, E)$ and a vertex $r \in V$ (root), visit exactly once all the vertexes of the graph that can be reached from r.

Depth-first search (DFS)

Traverse the graph by visiting all the nodes that can be reached by a node, and all the nodes that can be reached by those nodes, etc.

- Application: topological sort
- Application: cycle detection
- Application: connected components
- Application: strongly connected components

Depth-first search: iterative

```
def dfs(G, r):
  stack = [r]
  visited = {r}
  while len(st) > 0:
    u = stack.pop()
    for v in G.adj(u):
      if not v in visited:
        visited.add(v)
        stack.append(v)
```

Depth-first search: recursive

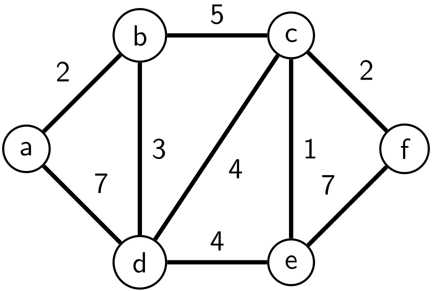
```
def dfs(G, u, visited):
  visited.add(r)
  for v in G.adj(u):
    if not v in visited
      dfs(G, v, visited)
```

Table of contents

Minimum Spanning Tree (MST)

Problem definition

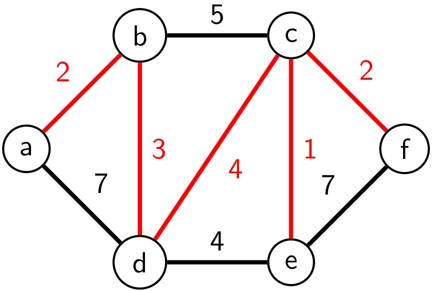
Given a connected weighted graph $G = (V, E)$, find $S \subseteq E$ such that (V, S) is still connected and S has the minimum total edge weight.



Minimum Spanning Tree (MST)

Problem definition

Given a connected weighted graph $G = (V, E)$, find $S \subseteq E$ such that (V, S) is still connected and S has the minimum total edge weight.



Kruskal algorithm for MST

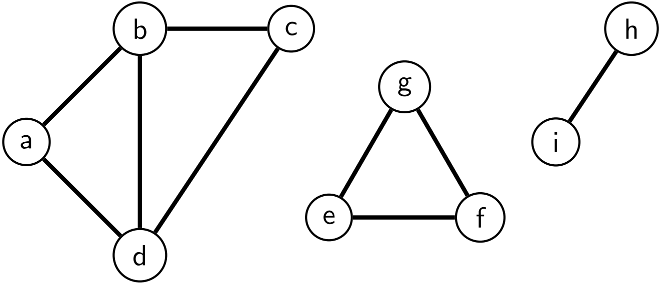
```
def mst(E):
  E.sort()  # sort for increasing weight
  ans = []
  for edge in E:
    if not Cycle(ans, edge):
      ans.append(edge)

  return ans
```

Connected Components

Problem definition

Given an undirected graph $G = (V, E)$, find the number of connected components.



Connected Components

```
def connectedComponents(G):
    visited = {}
    count = 0
    for u in G.V:
        if not u in visited:
            count += 1
            dfs(G, u, visited)

    return count
```