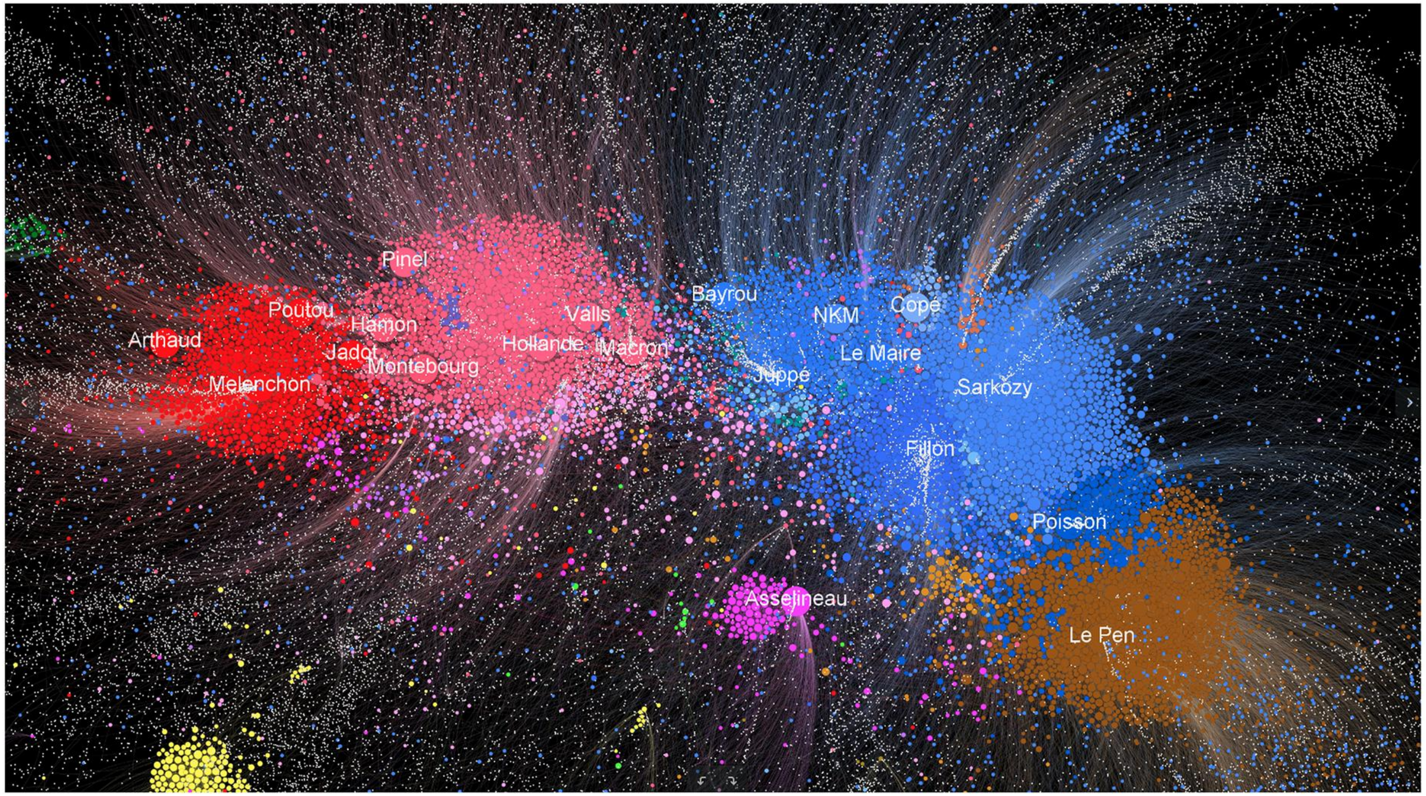
Introduction to Graph Theory

Bartosz Pankratz

Introduction

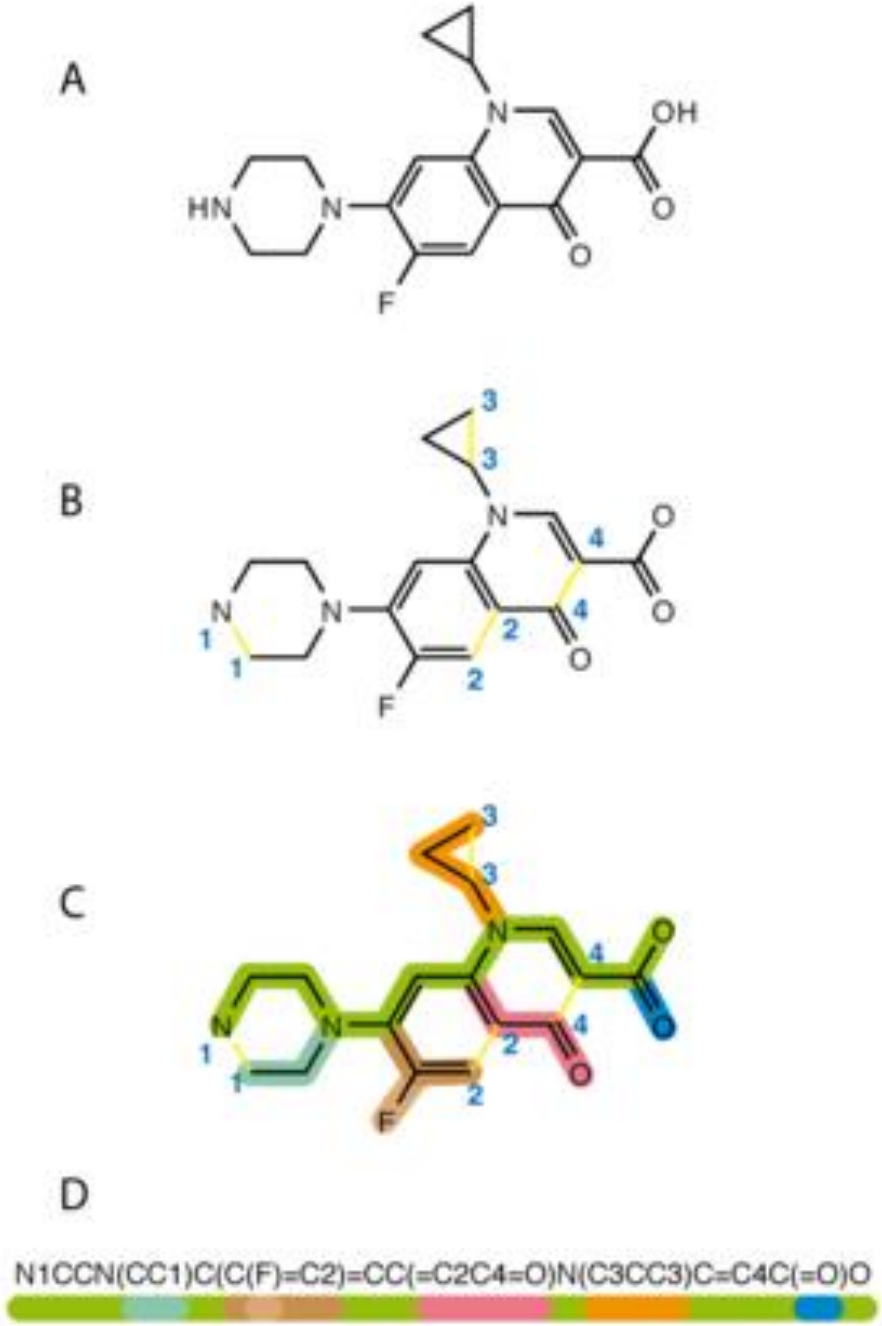
- Graphs are one of the most widely used mathematical structures.
- They allow us to efficiently represent **relational data** and as a result better understand plenty of different phenomena, ranging from social interactions to molecules.
- Let us briefly discuss some examples:

Social networks:

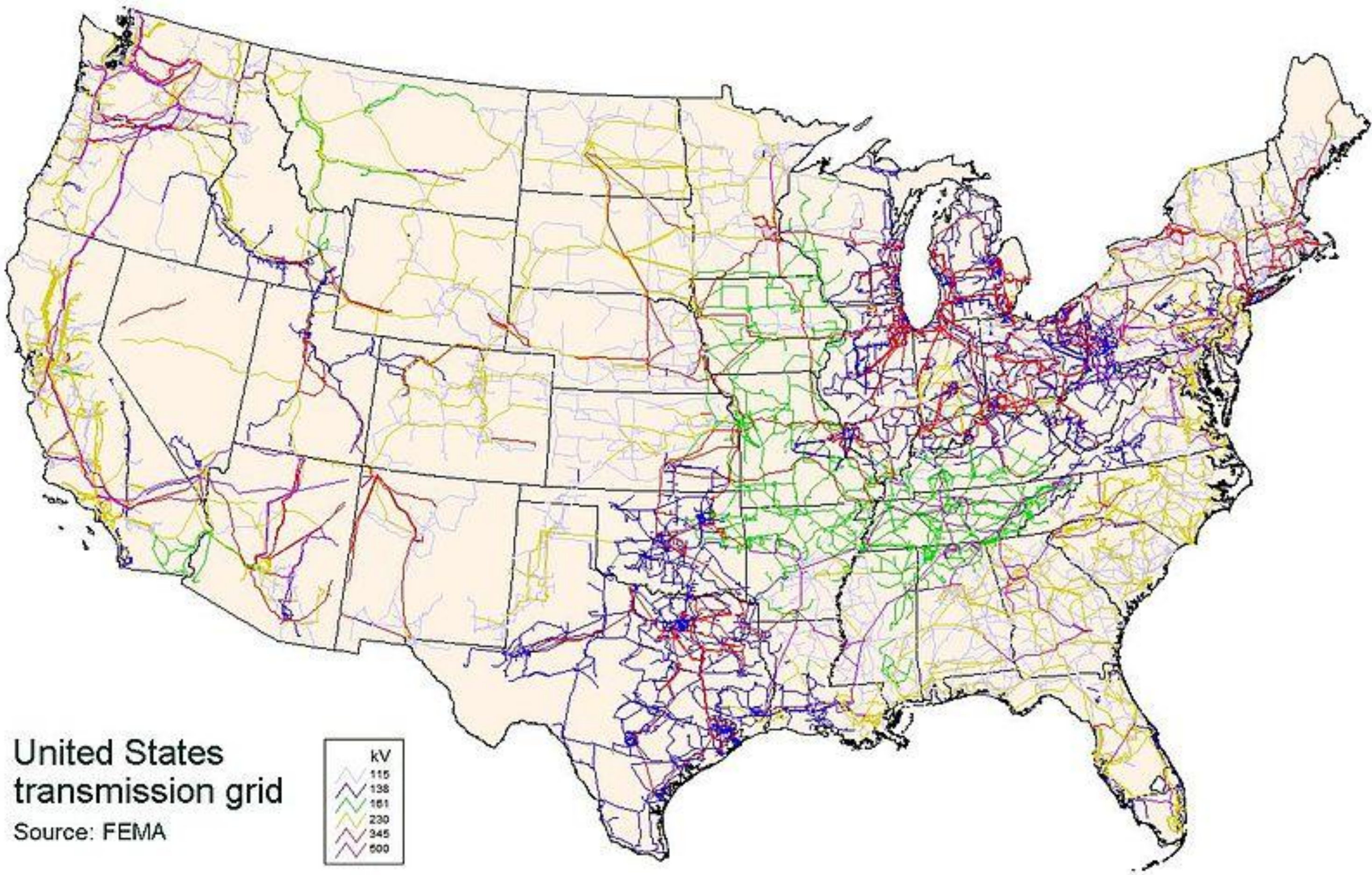


Source: N. Gaumont, M. Panachi and D. Chavalarias, Reconstruction of the socio-semantic dynamics of political activist Twitter networks—Method and application to the 2017 French presidential election, PLOS ONE 13(9) (2018).

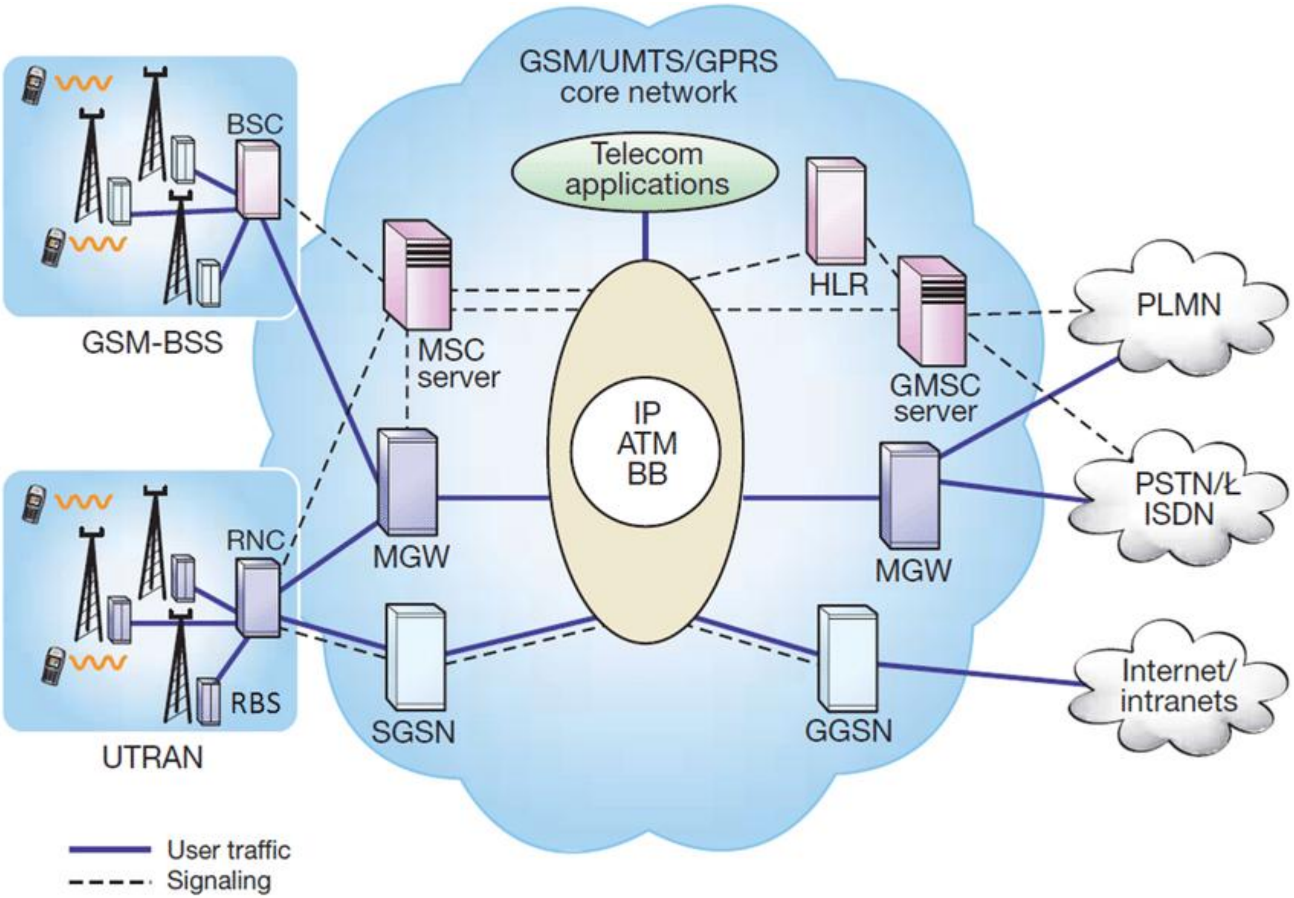
Molecules:



Electrical grids:



Telecommunication networks:



Source:
https://www.researchgate.net/figure/Telecommunication-Network-Structure-4_fig1_220841033/

Road networks:

Source: http://www.marcinkossakowski.com/finding-shortest-path-using-dijkstras-algorithm/

Social networks again:

Quora Search Quora

I like girl A. We are very good friends. A's best friend B likes me. Also, A likes a guy C who is my best bro and C likes another girl D who is in a relationship with another guy. What should we do?

Answer · Follow · 145 · Request · 7

All related (100+) · Sort · Recommended

Follow
Studied at Indian Institute of Technology, Delhi · Updated 8y

Originally Answered: I like girl A. We are very good friends, but her best friend B likes me. Also, A likes guy C, who is my best bro. C likes another girl D, who is in a relationship with another guy. What should we do?

You can apply Bipartite Matching algorithm of Graph Theory. All you have to do is:

1. Put all girls as vertices (a's) in Part A, and boys as vertices (b's) in Part B.
2. Draw an edge between vertex a to vertex b if a likes b.
3. It will be a bipartite graph (there are no edges within vertices of Part A, and same for Part B) unless your friends are not straight.
4. Then find maximum matching for this bipartite graph. (refer this link Maximum Bipartite Matching - GeeksforGeeks)
5. You will get maximum matching as output and the matched edges will be the pairs who should be together.
6. You can apply weighted bipartite matching algorithm if you know how much they love/like each other. In that case, assign weights according to their amount of love. and repeat step 4 and 5.

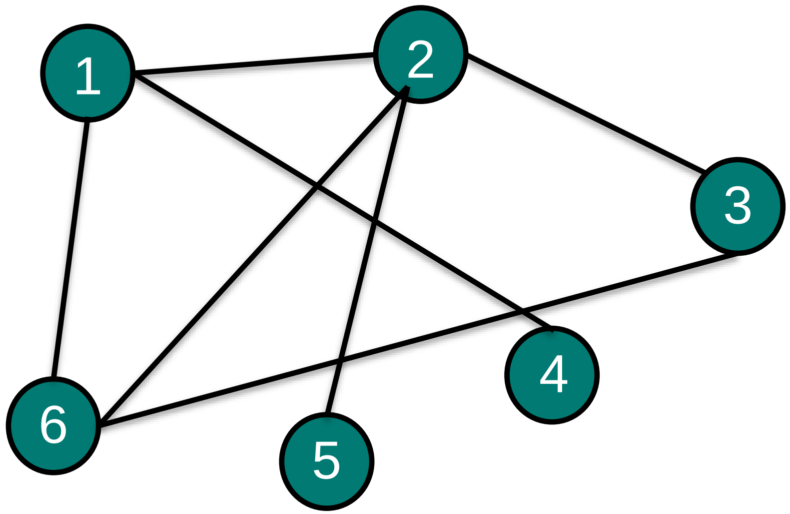
Thanks for asking this question. Finally I have found a real life problem which can be solved by what Prof Panda has been teaching us for 2-3 years.

P.S. I can provide you code for both weighted and unweighted maximum bipartite matching which I made for my project.

Formal definition

- Graph is a tuple (ordered pair) $G = (V, E)$, where:
  - V is a set of **vertices** (**nodes**).
  - $E \subseteq \{ \{x, y\} | \ x, y \in V \ \wedge x \neq y\}$ is a set of **edges** - pairs of vertices.
  - We assume that graph is **simple**; there are no **loops** (edges from x to x) nor **parallel edges** (more than one edge from x to y)
- Two nodes v_i and v_j, connected by edge e_{ij} are **adjacent** (**neighbors**).

Undirected graph:

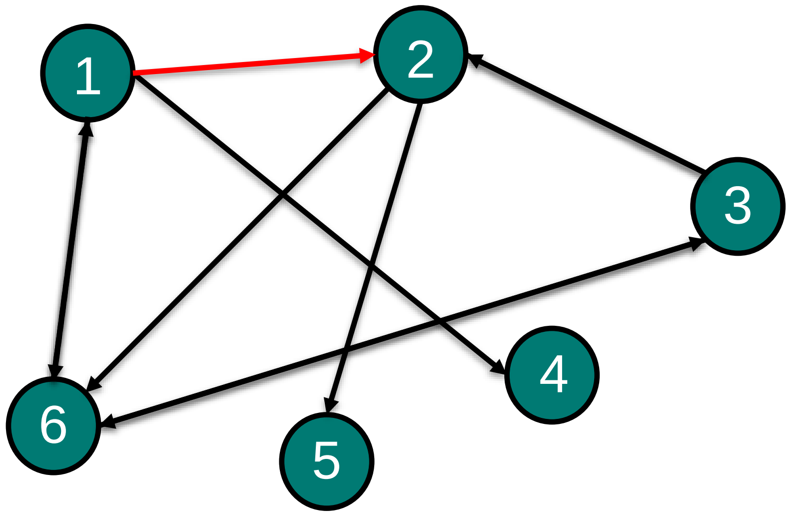


Formal definition

- Directed graph is a tuple (ordered pair) $\mathrm{D} = (V, E)$, where:
  - V is a set of **vertices** (**nodes**).
  - $E \subseteq \{(x, y) | \, (x, y) \in V^2 \wedge x \neq y\}$ is a set of **edges** - pairs of vertices.
  - This time order of nodes in edge matters, $(x, y) \neq (y, x)$. Formally directed graph is an incidence function mapping every edge to an **ordered pair** of vertices.

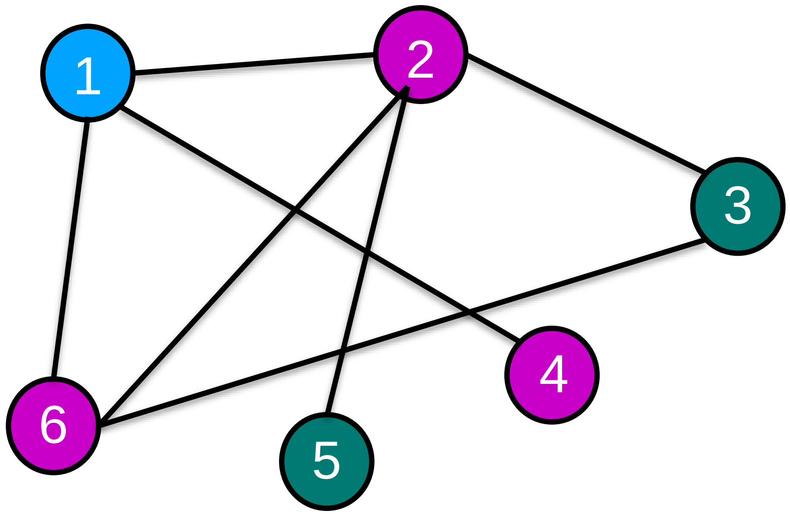
Directed graph:

- Node v_1 is a **predecessor** of node v_2.
- Node v_2 is a **successor** of node v_1 .
- The **source** of the edge e_{12} is node v_1, and the **target** of the edge is node v_2.



Degree – number of neighbors of a given node:

Degree of v_1: $\deg(v_1) = 3$



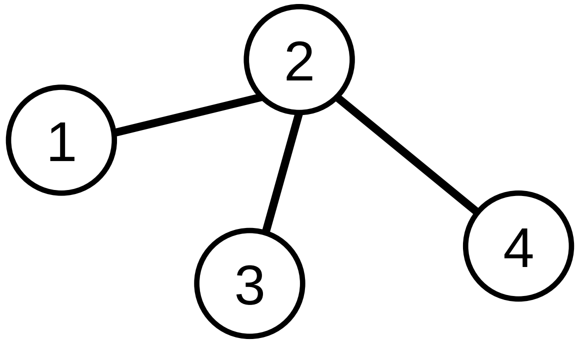
Formal definition

- Graphs might be represented as matrices (or arrays):
  - **Adjacency matrices**
  - **Adjacency lists**
  - **Degree matrices**
  - **Laplacian matrices**

Formal definition

- For a given graph $G = (V, E)$ (directed or undirected), where $|V| = n$ **adjacency matrix** $\mathrm{A} = \big(\mathrm{a}(\mathrm{x}, \mathrm{y})\big)_{x,y \in V}$ is a square matrix of size $n \times n$.
- $\mathrm{a}(\mathrm{x}, \mathrm{y}) = 1$ when there is an edge from x to y, $\mathrm{a}(\mathrm{x}, \mathrm{y}) = 0$ otherwise.
- Note that for undirected graph matrix A is symmetric.

Formal definition



- Adjacency Matrix

```
nx.adjacency_matrix(g)
[[ 0.  1.  0.  0.]
 [ 1.  0.  1.  1.]
 [ 0.  1.  0.  0.]
 [ 0.  1.  0.  0.]]
```

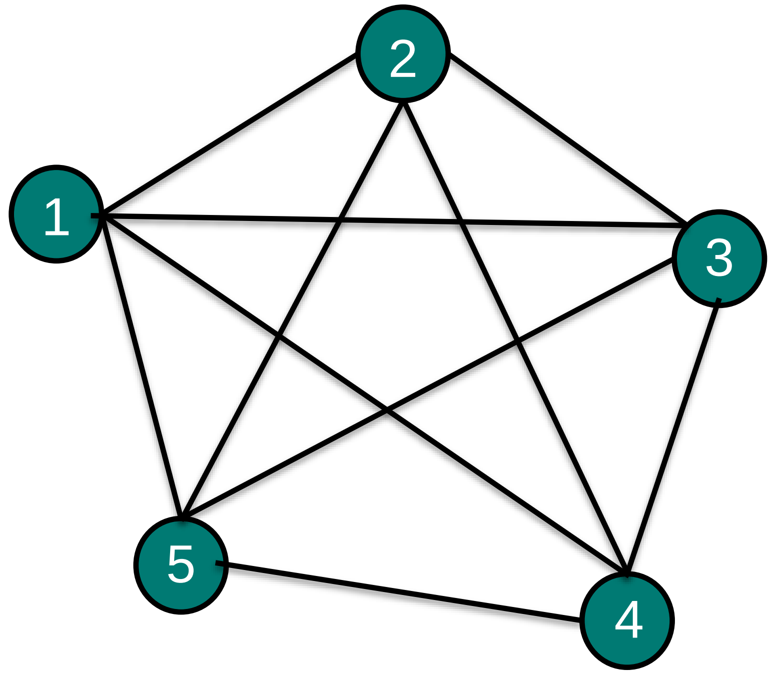
- Adjacency List

```
g.adjacency_list()
[[2], [1, 3, 4], [2],
[2]]
```

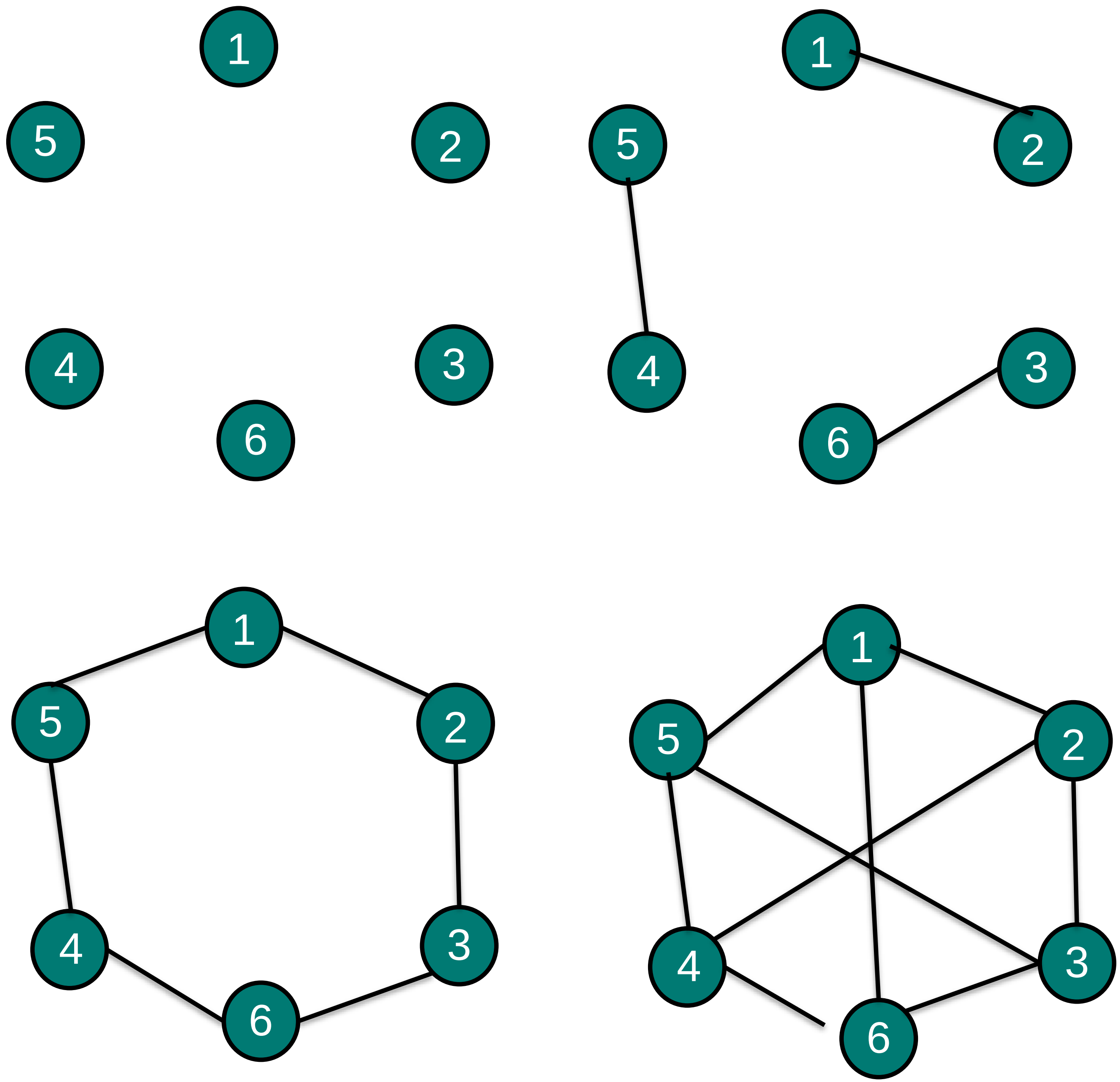
Formal definition

- In **Unweighted Graphs** edge between two nodes is indicated by 1: $a(x, y) = 1$.
- In **Weighted Graphs** edges can take any arbitrary value (however, we assume that $a(x, y) = 0$ indicates no edge between nodes).
- Road network is a perfect example of weighted graph.

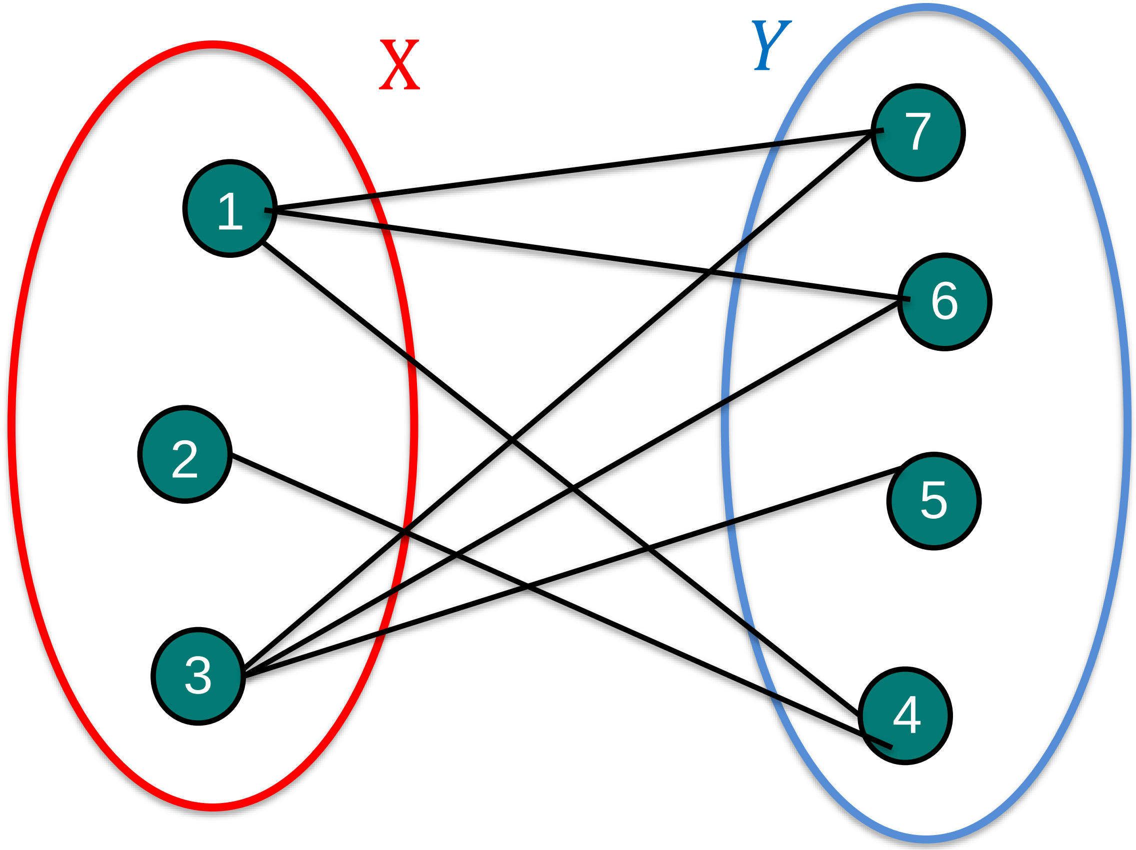
Complete graph is a graph (or subgraph – then it is called a **clique**) where all nodes are connected to each other:



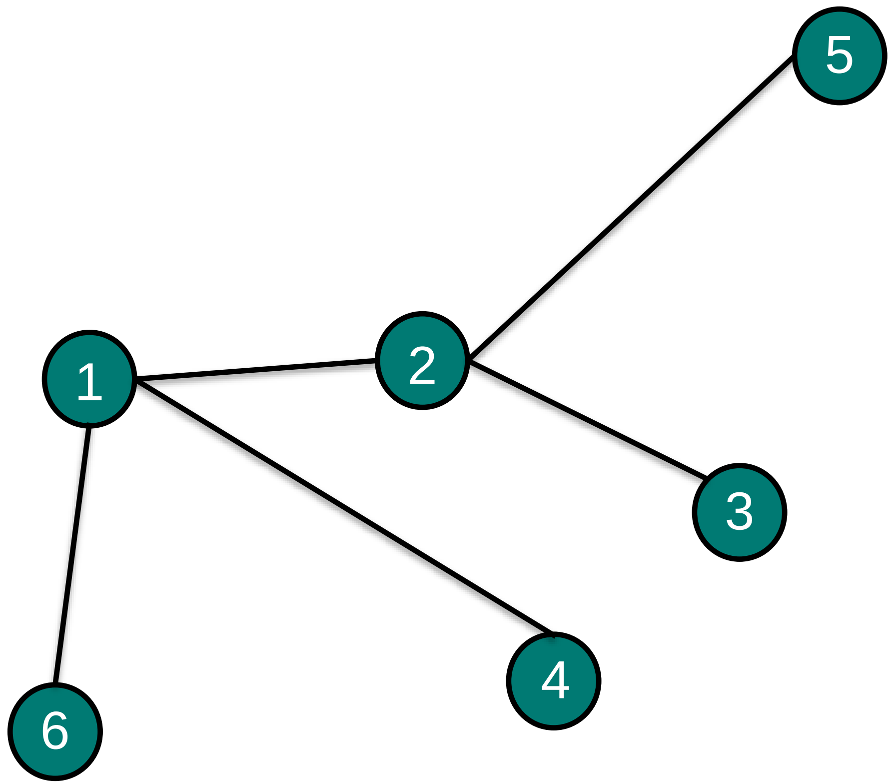
d-regular graph is a graph $G = (V, E)$, where $|V| = n \geq d + 1$ and $\deg(v_i) = d \; \forall v_i \in V$.



Bipartite graph is a graph $G = (V, E)$, where set of nodes V can be divided into two subsets X and Y (called **partite sets**) such that $\mathrm{X} \cup Y = V$ and $\mathrm{X} \cap Y = \emptyset$. Then, every edge $(\mathrm{x}, \mathrm{y}) \in E$ satisfies $\mathrm{x} \in \mathrm{X}$ and $y \in Y$



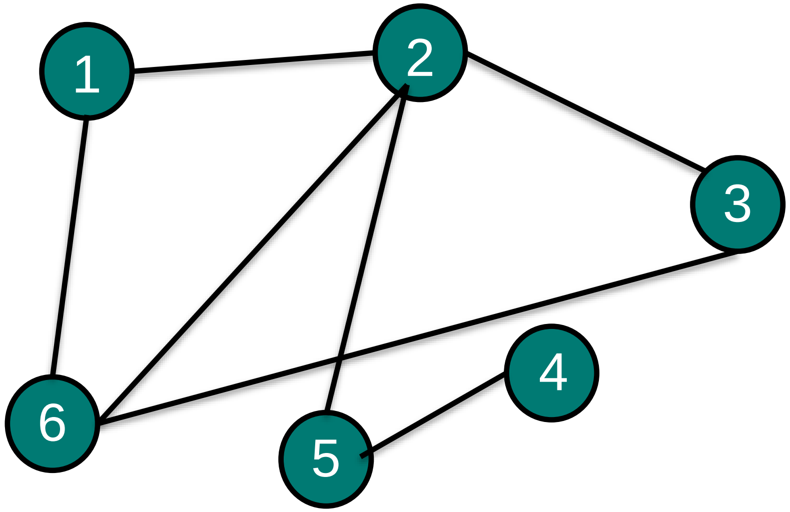
Tree is an acyclic connected graph.

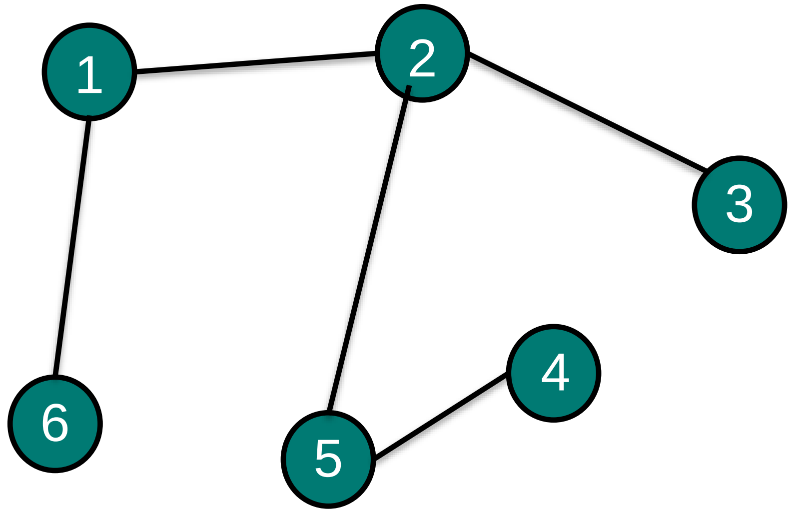


Spanning trees

Spanning trees

- **Spanning tree** T of an undirected graph G is defined as a subgraph that is a **tree** which includes all of the vertices of G, with **minimum** possible number of edges.





Spanning trees

- **Spanning tree** T of an undirected graph G is defined as a subgraph that is a **tree** which includes all of the vertices of G, with **minimum** possible number of edges.
- In weighted graphs we can also define the **minimum spanning tree (MST)**, which is the spanning tree with the lowest possible sum of edge weights.

Spanning trees

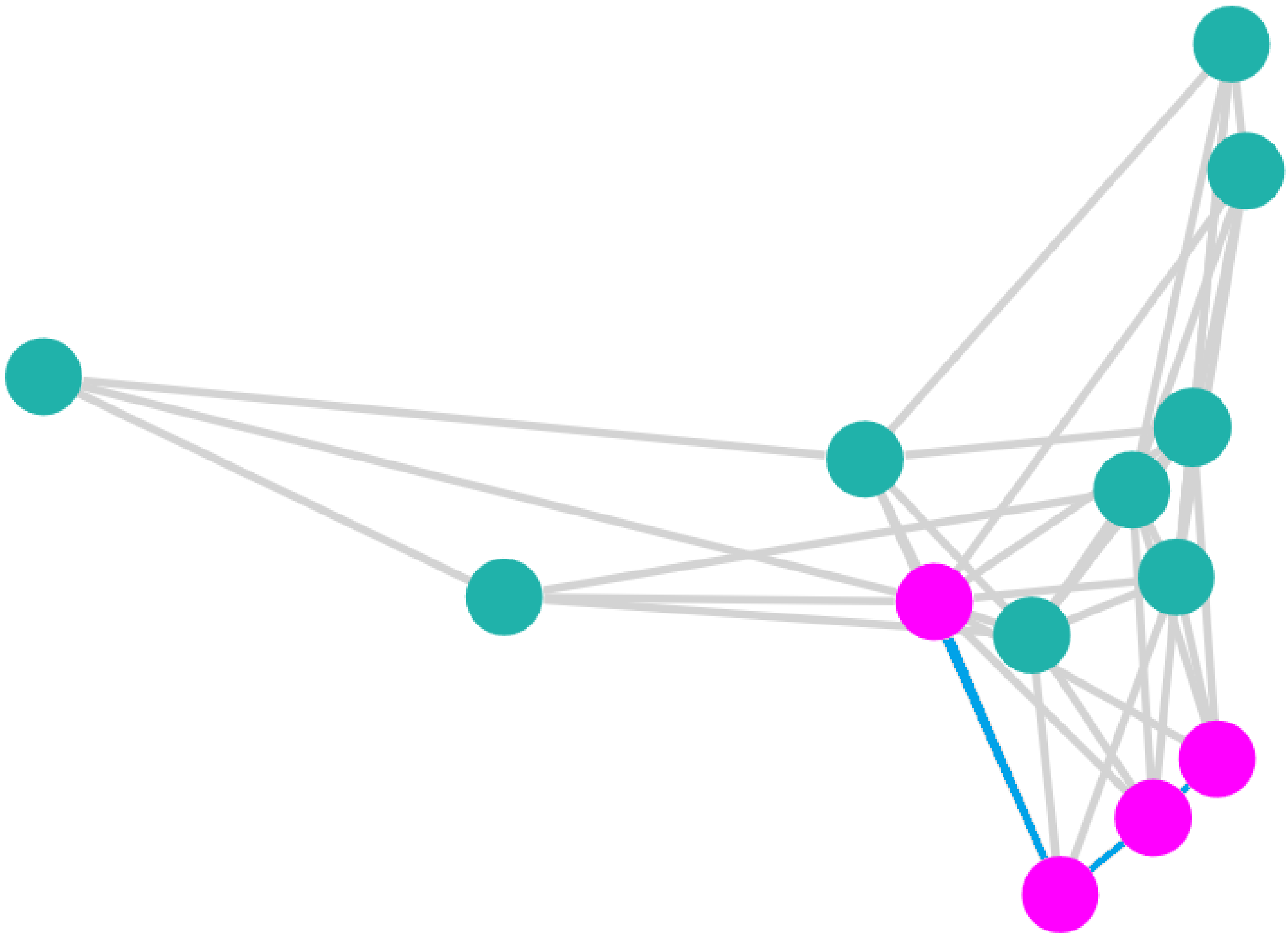
- A concept of the **Minimum spanning tree (MST)** have direct applications in the design of networks (e.g. electrical grids) and testing their robustness.
- Algorithms for finding the MST:
  - **Borůvka algorithm.**
  - **Kruskal algorithm.**
  - **Prim algorithm.**

Paths

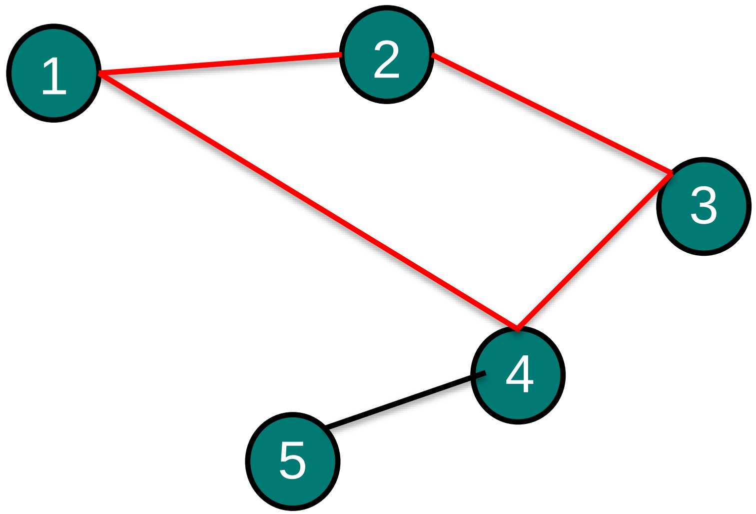
Path – a path from v_1 to v_n is a list of nodes $P = (v_1, v_2, \ldots, v_n)$ such that $(v_{k-1}, v_k) \in E$ for all $k = 2, \ldots, n$.

The length of the path is just a sum of weights of edges: $\sum_{k=2}^{n} a((v_{k-1}, v_k))$.

Note that for unweighted graph it just simplifies to $|P|$ (number of edges).



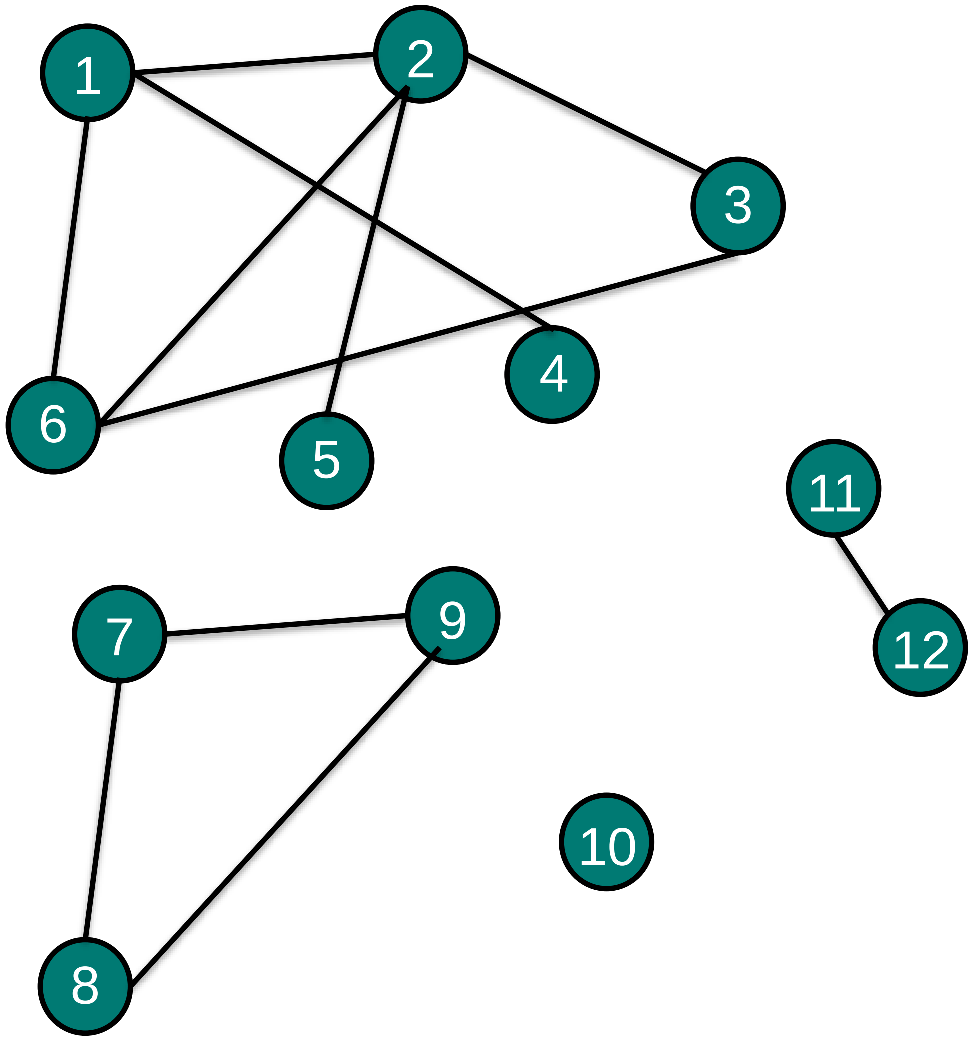
Cycle is a sequence $C = (v_1, v_2, \dots, v_n)$ such that $(v_{k-1}, v_k) \in E$ for all $k = 2, \dots, n$ and $(v_n, v_1) \in E$.



Connected component – in an undirected graph G is the largest subgraph in which there exists a path between any two nodes.

If G has exactly one component, then G is connected, otherwise G is disconnected.

For a directed graph definition is less trivial, there are plenty of generalizations of the connectivity concept.



Distance

- **Distance** $\mathrm{dist}(\mathrm{x}, \mathrm{y})$ is defined as a shortest path between x and y:

$$\mathrm{dist}(\mathrm{x}, \mathrm{y}) = \min_{(x=v_1, v_2, \ldots, v_n=y)} \sum_{k=2}^{n} a((v_{k-1}, v_k)$$

- **Diameter** $\mathrm{diam}(G)$ is the farthest distance between any two of its vertices:

$$\mathrm{diam}(G) = \max_{x,y \in V} \mathrm{dist}(\mathrm{x}, \mathrm{y})$$

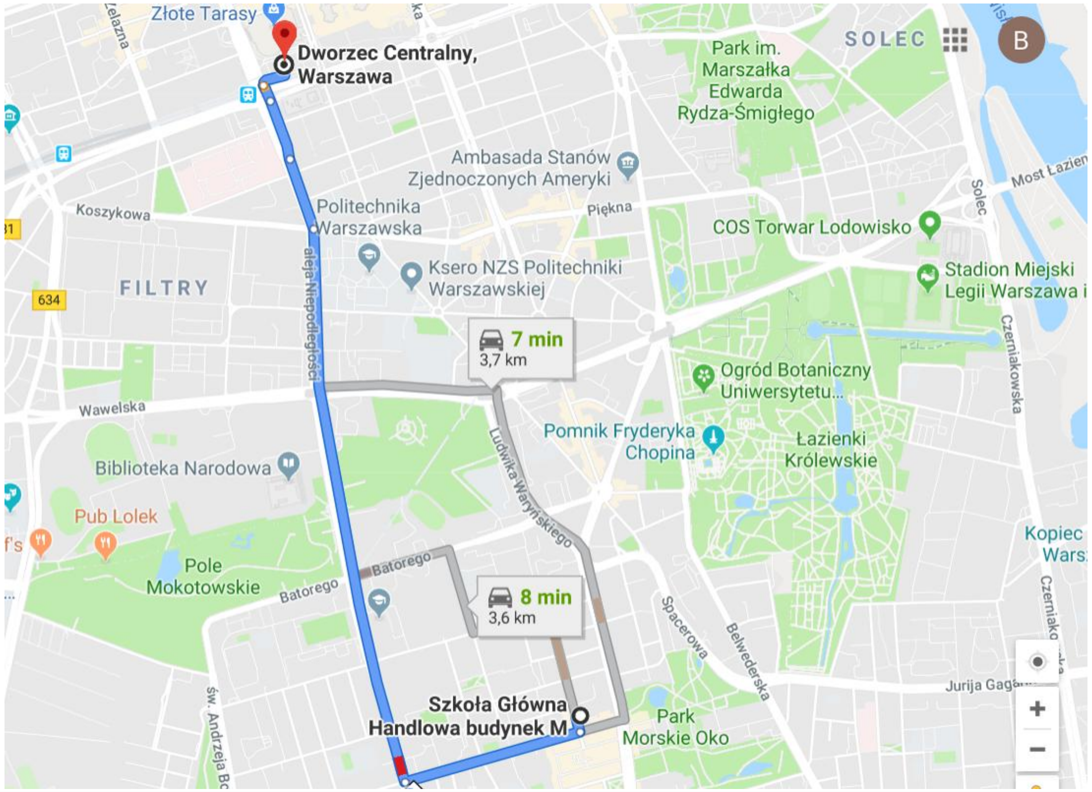
Shortest path problem

Shortest path problem

- One of the key concepts in graphs analysis is finding an optimal (minimal) path between two vertices in a graph.
- Finding directions between two locations on a map is a great example of a shortest path problem application**:**

Shortest path problem

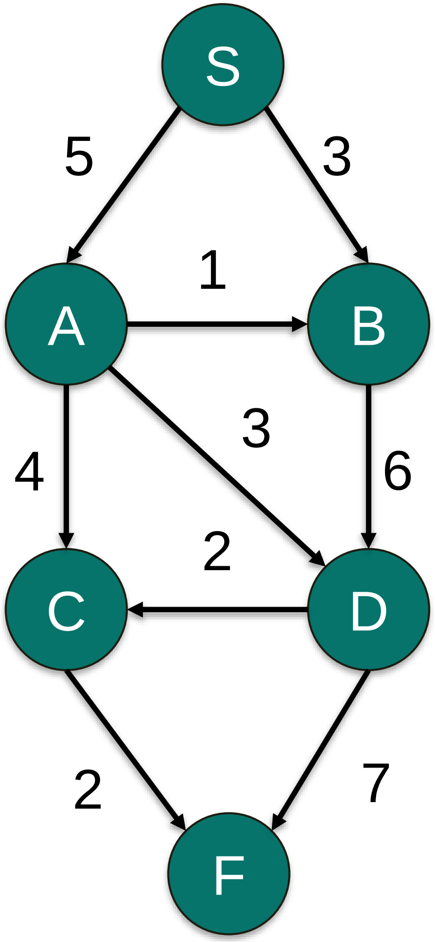
- One of the key concepts in graphs analysis is finding an optimal (minimal) path between two vertices in a graph.
- Finding directions between two locations on a map is a great example of a shortest path problem application**:**



Shortest path problem

- Commonly used algorithms:
  - **Dijkstra's algorithm**
  - **A* search algorithm**
  - **Bellman–Ford algorithm**
  - **Floyd–Warshall algorithm**

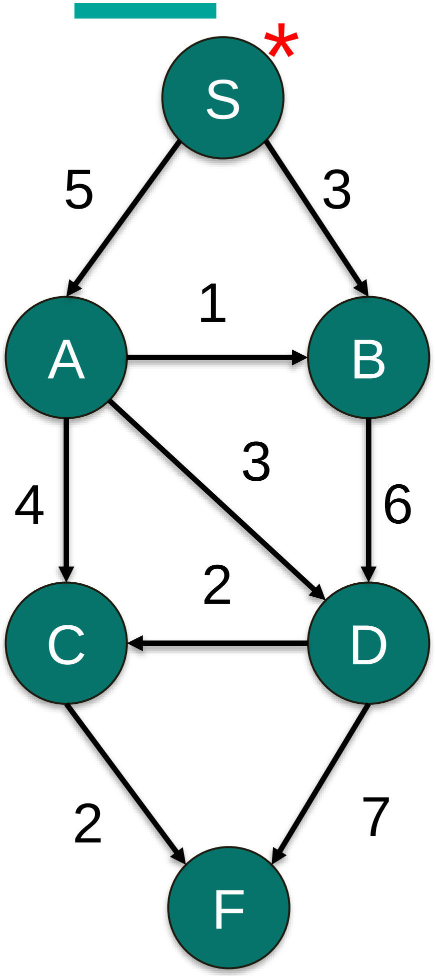
Dijkstra Algorithm



$$Visited = []$$
$$Unvisited = [S, A, B, C, D, F]$$

Node	Shortest path from **S**	Previous node
S	0	-
A	∞	
B	∞	
C	∞	
D	∞	
F	∞	

Dijkstra Algorithm



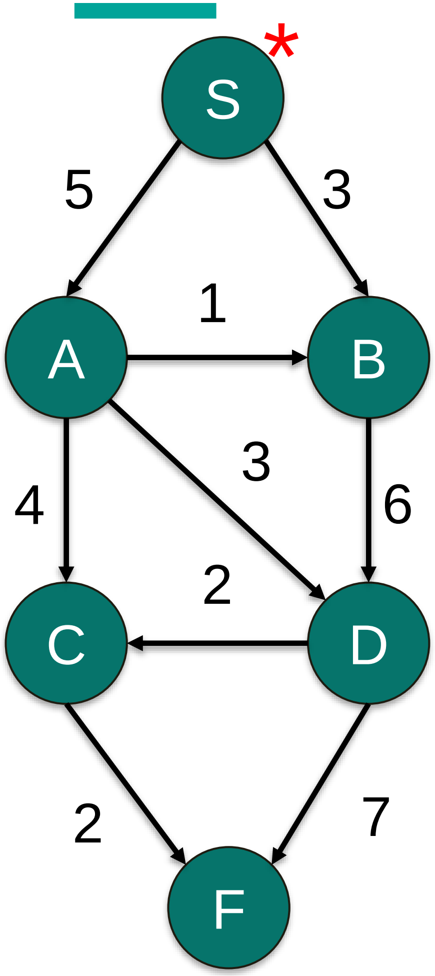
$$Visited = []$$
$$Unvisited = [S, A, B, C, D, F]$$

Node	Shortest path from **S**	Previous node
S	0	-
A	∞	
B	∞	
C	∞	
D	∞	
F	∞	

In each step:

- Select unvisted node with the **shortest path from v_o.**

Dijkstra Algorithm



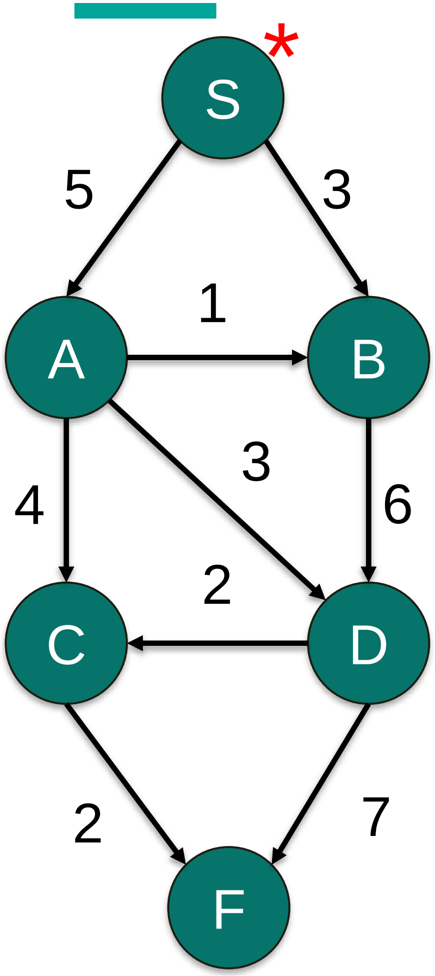
$$Visited = [S]$$
$$Unvisited = [A, B, C, D, F]$$

Node	Shortest path from **S**	Previous node
S	0	-
A	∞	
B	∞	
C	∞	
D	∞	
F	∞	

In each step:

- Select unvisted node with the **shortest path from v_o.**
- Mark it as visited.

Dijkstra Algorithm



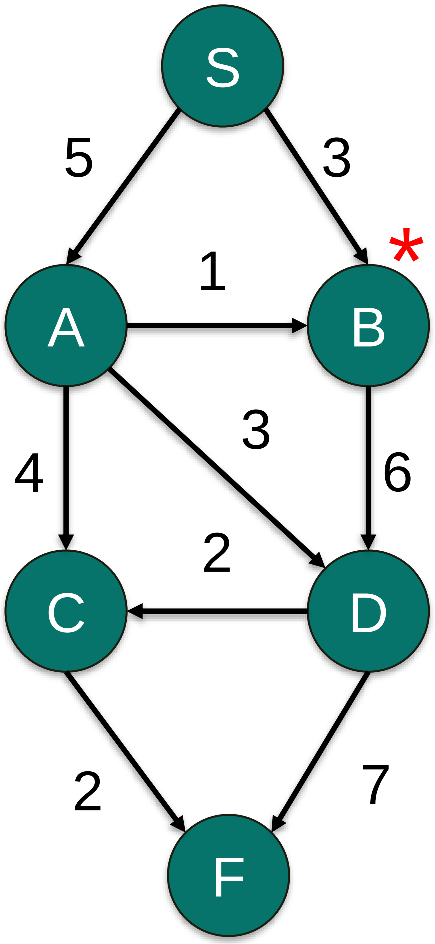
$$Visited = [S]$$
$$Unvisited = [A, B, C, D, F]$$

Node	Shortest path from **S**	Previous node
S	0	-
A	$\cancel{\infty}$ 5	S
B	$\cancel{\infty}$ 3	S
C	∞	
D	∞	
F	∞	

In each step:

- Select unvisted node with the **shortest path from v_o.**
- Mark it as visited.
- Calculate the distance to all its successors.

Dijkstra Algorithm



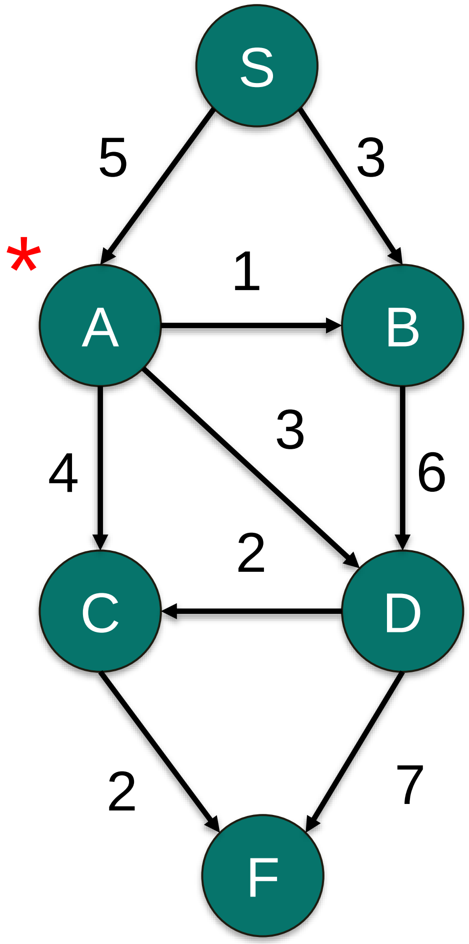
$$Visited = [S, B]$$
$$Unvisited = [A, C, D, F]$$

Node	Shortest path from **S**	Previous node
S	0	-
A	5	S
B	3	S
C	∞	
D	~~∞~~ 9	B
F	∞	

In each step:

- Select unvisted node with the **shortest path from v_o.**
- Mark its as visited.
- Calculate the distance to all its successors.
- Select next node with shortest path from v_o and repeat the procedure.

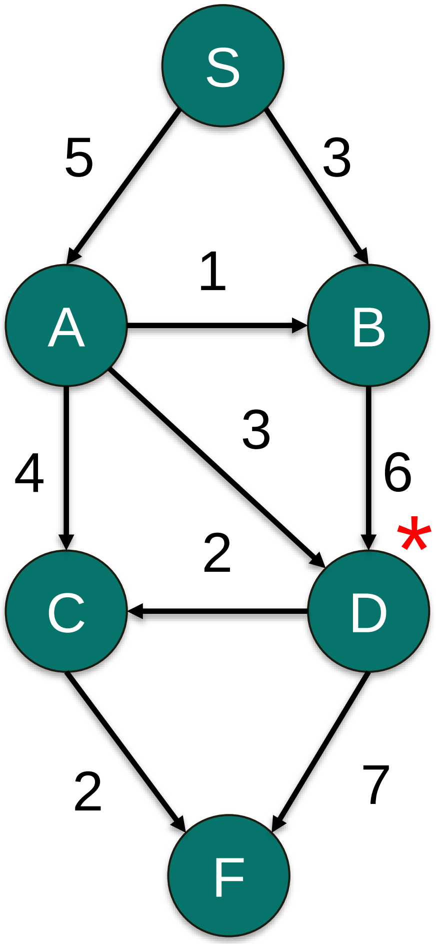
Dijkstra Algorithm



$$Visited = [S, B, A]$$
$$Unvisited = [C, D, F]$$

Node	Shortest path from **S**	Previous node
S	0	-
A	5	S
B	3	S
C	$\cancel{\infty}$ 9	A
D	$\cancel{9}$ 8	~~B~~ A
F	∞	

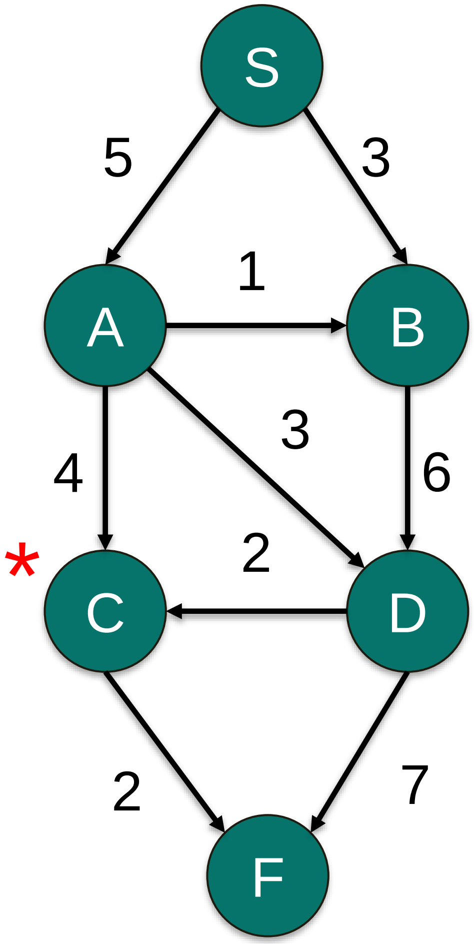
Dijkstra Algorithm



$$Visited = [S, B, A, D]$$
$$Unvisited = [C, F]$$

Node	Shortest path from **S**	Previous node
S	0	-
A	5	S
B	3	S
C	9	A
D	8	A
F	~~∞~~ 15	D

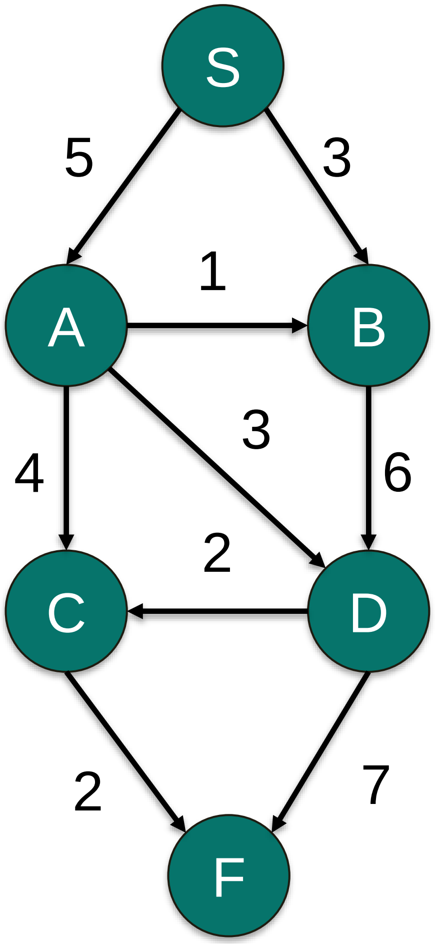
Dijkstra Algorithm



$$Visited = [S, B, A, D, C]$$
$$Unvisited = [F]$$

Node	Shortest path from **S**	Previous node
S	0	-
A	5	S
B	3	S
C	9	A
D	8	A
F	~~15~~ 11	~~D~~ C

Dijkstra Algorithm

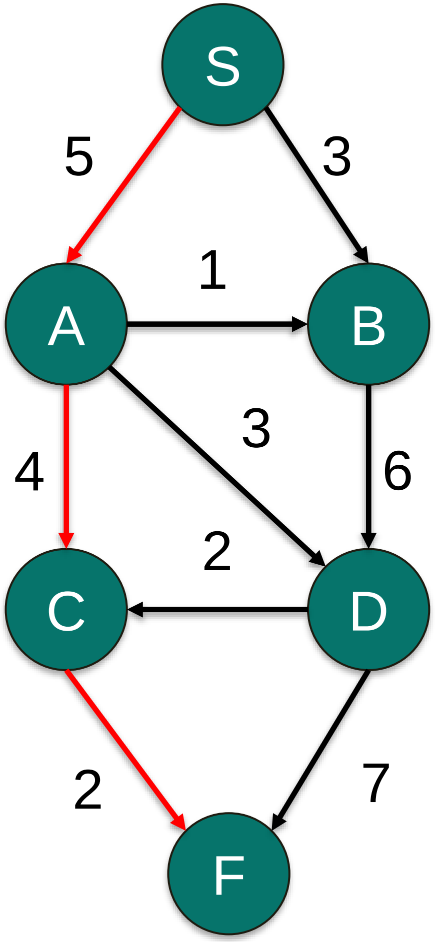


$$Visited = [S, B, A, D, C, F]$$
$$Unvisited = []$$

Node	Shortest path from **S**	Previous node
S	0	-
A	5	S
B	3	S
C	9	A
D	8	A
F	11	C

In the final step, you need to find the optimal path based on the collected information.

Dijkstra Algorithm



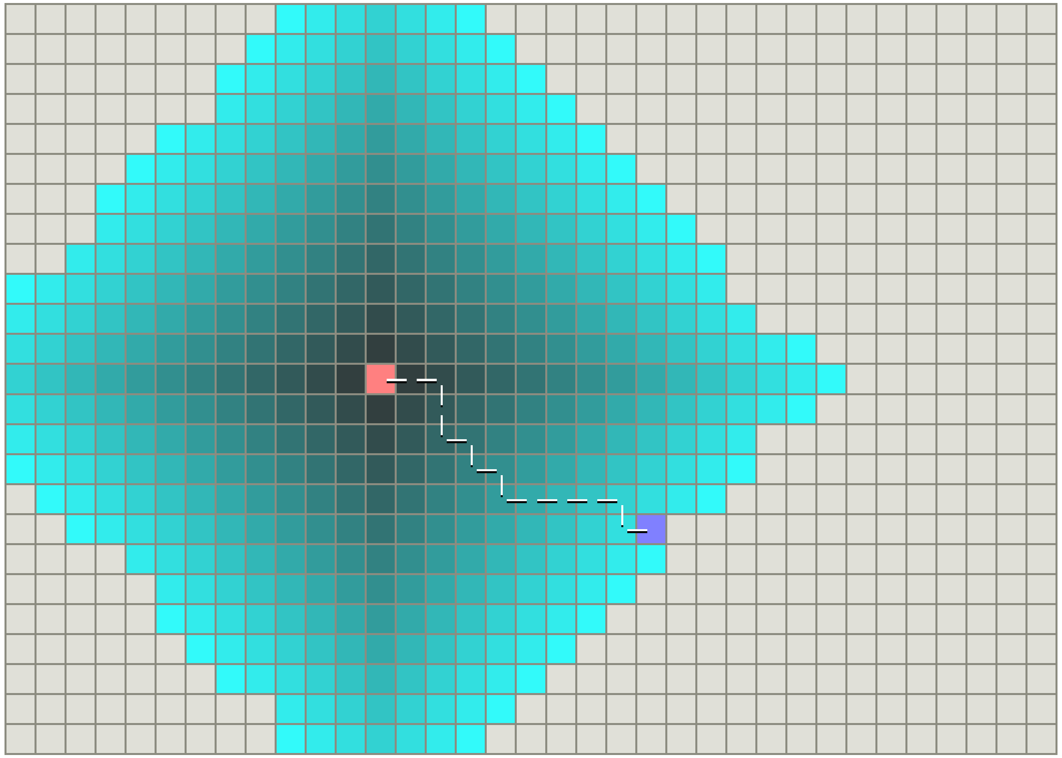
$$Visited = [S, B, A, D, C, F]$$
$$Unvisited = []$$

Node	Shortest path from **S**	Previous node
S	0	-
A	5	S
B	3	S
C	9	A
D	8	A
F	11	C

Dijkstra Algorithm

- Time complexity: $O(|E| + |V| \log|V|)$.
- Can be used only for non-negative edge weights.
- However, the Dijkstra algorithm has one major drawback:

When the Dijkstra algorithm is searching for the shortest path it examines vertices in all possible directions, until it reaches the goal.

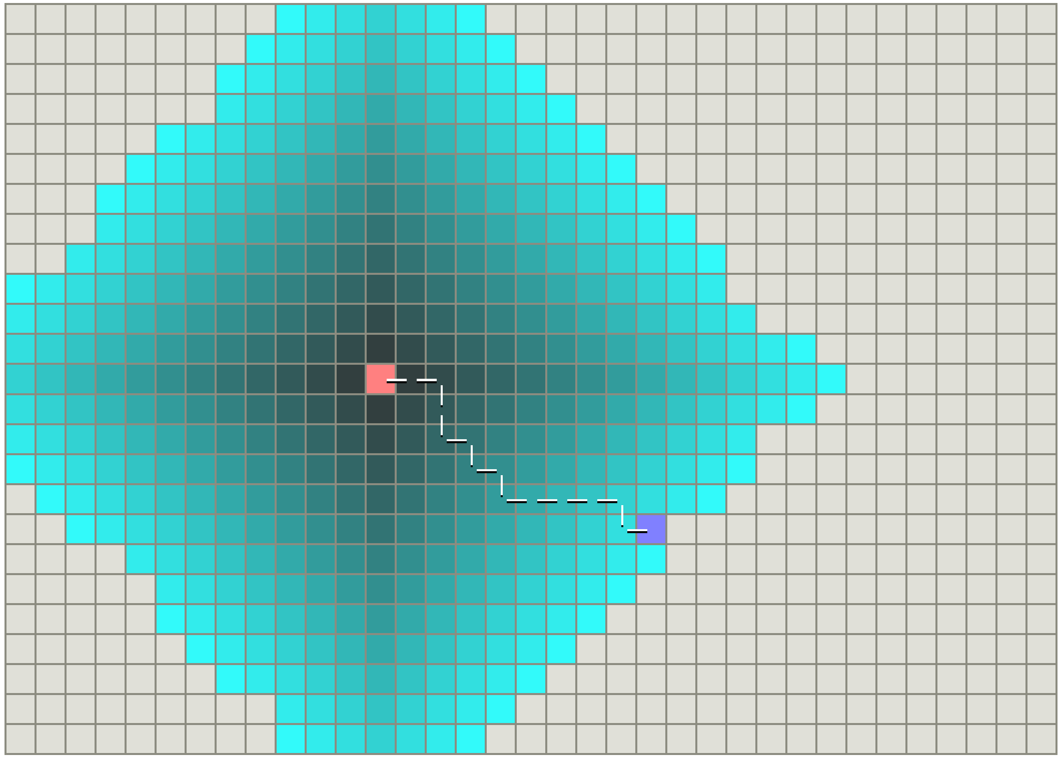


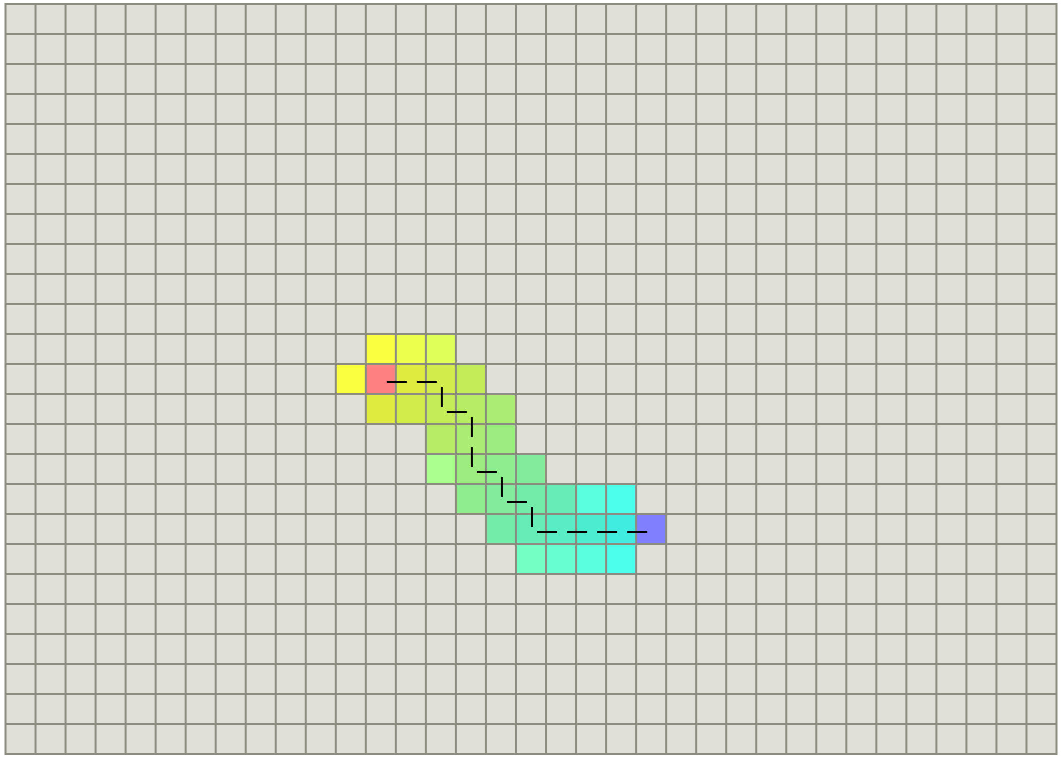
Source: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

When the Dijkstra algorithm is searching for the shortest path it examines vertices in all possible directions, until it reaches the goal.

A* search algorithm is a significantly more effective approach. It utilizes a heuristic function to prioritize paths that seem to be leading closer to a goal.

Source: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html





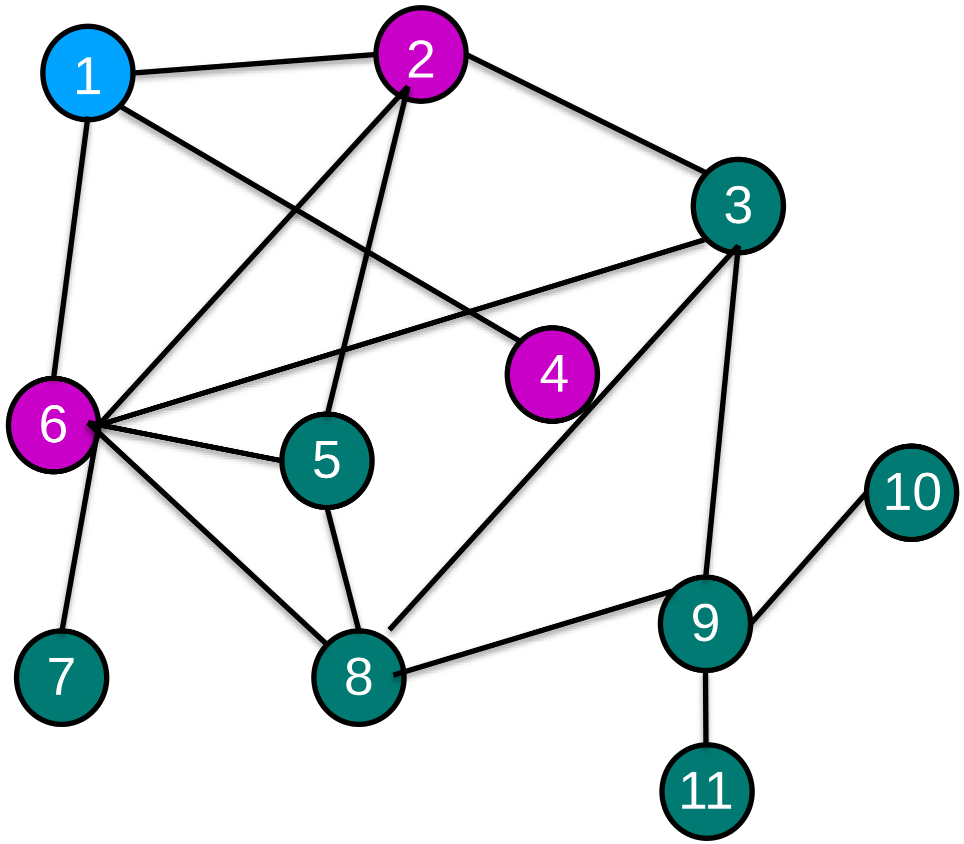
Degree distribution

Degree – number of neighbors of a given node:

Degree of v_1: $\deg(v_1) = 3$

In an undirected graph G **neighborhood** of node v_i is defined as:

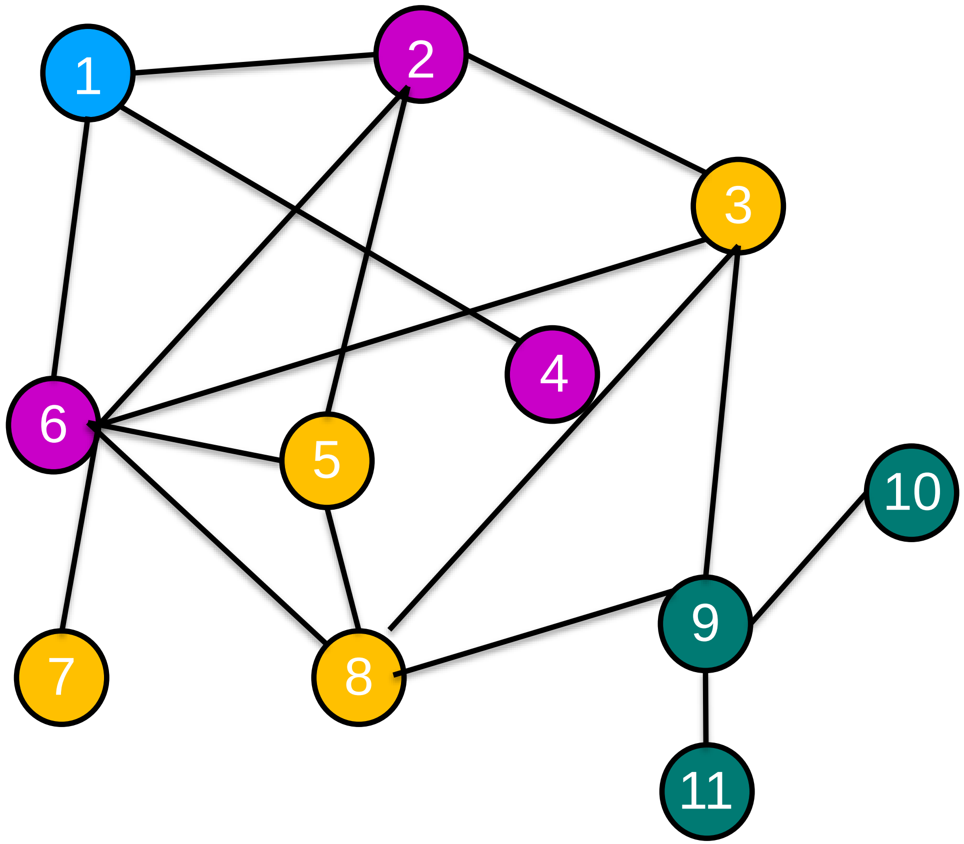
$\mathrm{Ne}(\mathrm{v_i}) = \{v_j \in \mathrm{V}: \{v_i, v_j\} \in E\}$



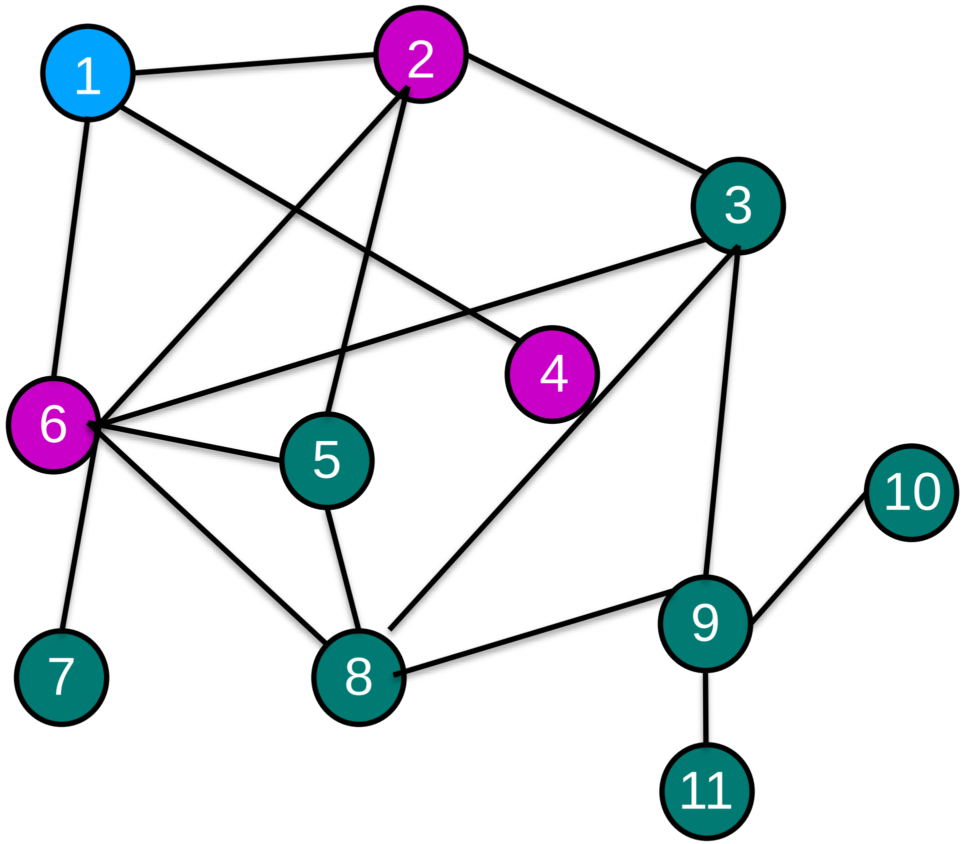
Definition of **neighborhood** can be generalized to include nodes at distance at most ℓ:

$$\mathrm{Ne}_{\ell}(\mathrm{v_i}) = \{v_j \in \mathrm{V} : \mathrm{dist}(v_i, v_j) \leq \ell\}$$

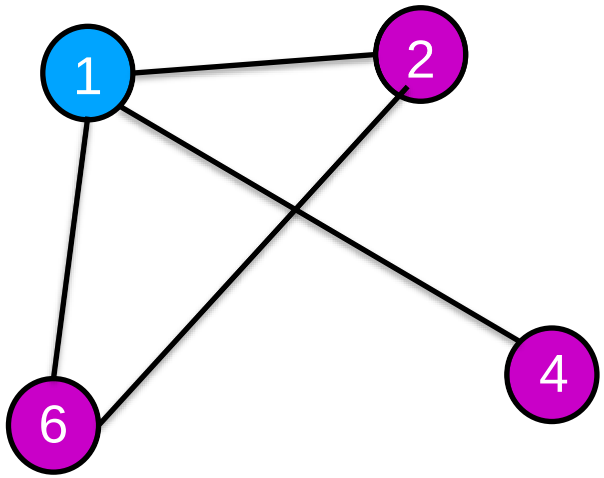
Example of $\mathrm{Ne}_2(v_1)$:



Ego Network –
subgraph where only its adjacent neighbors and their mutual links are included**:**



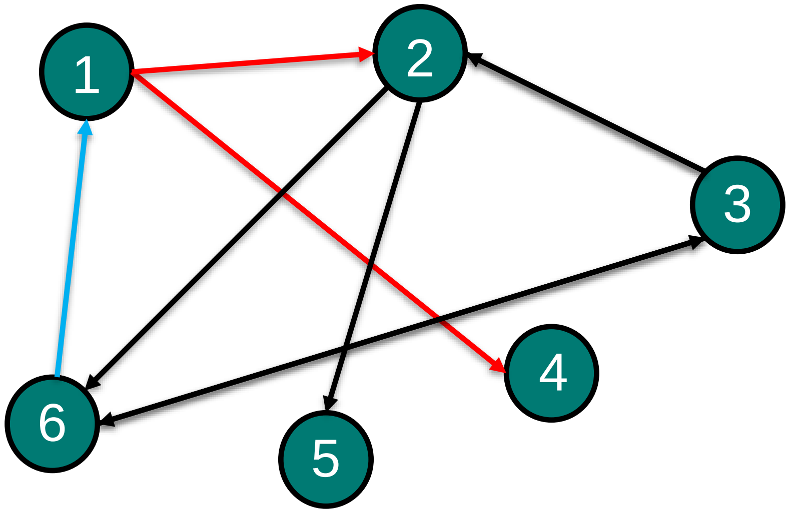
Ego Network – subgraph where only its adjacent neighbors and their mutual links are included**:**



Neighborhoods

In a directed graph D **neighborhood** of node v_i generalize into two measures :

$$\mathrm{Ne}^{in}(\mathrm{v_i}) = \{v_j \in \mathrm{V}: \{v_j, v_i\} \in E\}$$
$$\mathrm{Ne}^{out}(\mathrm{v_i}) = \{v_j \in \mathrm{V}: \{v_i, v_j\} \in E\}$$



Degree sequence

- The **degree sequence** is non-decreasing sequence $\mathbf{d} = \big(\deg(v_1), \deg(v_2), \ldots, \deg(v_n)\big)$

- The **degree distribution** d_ℓ is the fraction of nodes with degree ℓ, namely, $d_\ell = \frac{n_\ell}{n}$.
- The **average degree** G in the unweighted graph is equal to:

Degree sequence

- The **average degree** G in the unweighted graph is equal to:

$$\langle k \rangle = \frac{1}{n} \sum_{v \in V} \deg(v) = \sum_{\ell \in \mathbb{N}} \ell \, d_\ell = \frac{2m}{n}$$

- In general, for $s \in \mathbb{N}$, sth moment of the degree sequence of G is defined as:

$$\langle k^s \rangle = \frac{1}{n} \sum_{v \in V} \deg(v)^s = \sum_{\ell \in \mathbb{N}} \ell^s \, d_\ell$$

Degree sequence

- For a given set of nodes $S \subseteq V$, the **volume** of set S is defined as

$$\mathrm{vol}(S) = \sum_{v \in S} \deg(v)$$

- In particular, the volume of an undirected graph G with $m = |E|$ edges is equal to

$$\mathrm{vol}(V) = \sum_{v \in V} \deg(v) = 2m$$

- The minimum degree:

$$\delta(G) = \min_{v \in V} \deg(v)$$

- The maximum degree:

$$\Delta(G) = \max_{v \in V} \deg(v)$$

Degree sequence

- Note that, not all sequences of non-negative integers $\mathbf{d} = (d_1, d_2, \ldots, d_n)$ can be a degree sequence of some simple graph.
- A sequence of numbers is said to be a **graphic sequence** if one can construct a graph having the sequence as its degree sequence.

Clustering Coefficient

Clustering Coefficient

- In the majority of real-world networks nodes tend to cluster together (we call this property a **homophily**).
- Clustering coefficient is a graph parameter that tries to measure how heavily nodes cluster together.
- Clustering coefficient can be defined **globally** (for entire network) or **locally** (for each node).

Clustering Coefficient

- Let $G = (V, E)$ be an unweighted graph.
- For every node $v_i \in V$ with $\deg(v_i) \geq 2$ **local clustering coefficient** is defined as:

$$c(v_i) = \frac{\left|\left\{v_j, v_k\right\} \in E: v_j, v_k \in Ne(v_i)\right|}{\binom{\deg(v_i)}{2}}$$

- Local clustering coefficient is a ratio of number of triangles containing v_i and all the triplets centered around v_i
- Clearly $0 \leq c(v_i) \leq 1$. When $c(v_i)$ tends to 1, then G is getting closer to a clique.

Clustering Coefficient

- Local clustering coefficient is a noisy measure (there are always outliers in graphs!), thus usually global measures are preferred.
- **The local clustering coefficient averaged over the nodes of degree $\boldsymbol{d}$** is an example of such a measure:

$$C(d) = \frac{\sum_{v \in V : \mathrm{deg(v)} = d} c(v)}{|\{v \in V : \mathrm{deg(v)} = d\}|}$$

assuming that $\{v \in V : \mathrm{deg(v)} = d\} \neq \emptyset$

Clustering Coefficient

- Local clustering coefficient is a noisy measure (there are always outliers in graphs!), thus usually global measures are preferred.
- Alternatively, one might use the **average local clustering coefficient**:

$$C_{\text{loc}}(G) = \frac{1}{n} \sum_{v \in V} c(v)$$

where $c(v) = 0$ when $\deg(\mathrm{v}) = 0$ or $\deg(\mathrm{v}) = 1$ (alternatively, $C_{\text{loc}}(G)$ can be defined only for nodes with degree larger than 2).

Clustering Coefficient

- Finally, the **global clustering coefficient** is defined as

$$C_{\mathrm{glob}}(G) = \frac{3 \times \#\ of\ triangles\ in\ G}{\#\ of\ triplets\ in\ G}$$

- The interpretation of $C_{\mathrm{glob}}(G)$ is straightforward; $C_{\mathrm{glob}}(G)$ is a probability that a random triplet of nodes forms a triangle.
- In some cases, $C_{\mathrm{glob}}(G)$ and $C_{\mathrm{loc}}(G)$ are equal (or very close). However, they can differ substantially.