# Graph theory

Glossary

- **Graph G (V, E)**:
  - **V**: vertices set
  - **E**: edges set
- **Degree**: number of vertices

- **Walk**: series of connected vertices

- **Path**: walk without repeated vertices

- **Closed walk**: walk where $v_0 = v_n$

- **Cycle**: closed walk without repeating vertices

- **Euler path**: visit each edge exactly once

- **Hamilton path**: visit each vertex exactly once

- **Directed graph**: edges are ordered pairs

- **Ancestor**: v, **Successor**: u in
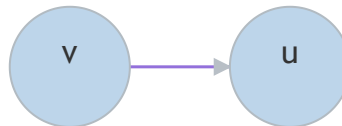
- **deg$_{in}$(v)**: number of incoming edges into v

- **deg$_{out}$(v)**: number of outgoing edges into v

Graph Representation

Adjacency matrix:

matrix where $A_{uv} = \begin{cases} 1 & \text{if}(u, v) \in E \\ 0 & \text{otherwise} \end{cases}$

Graph:
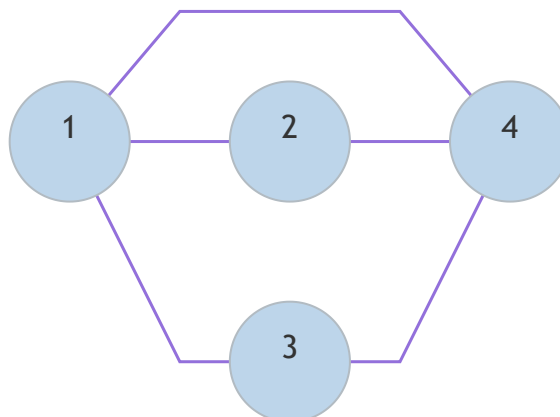
Matrix:

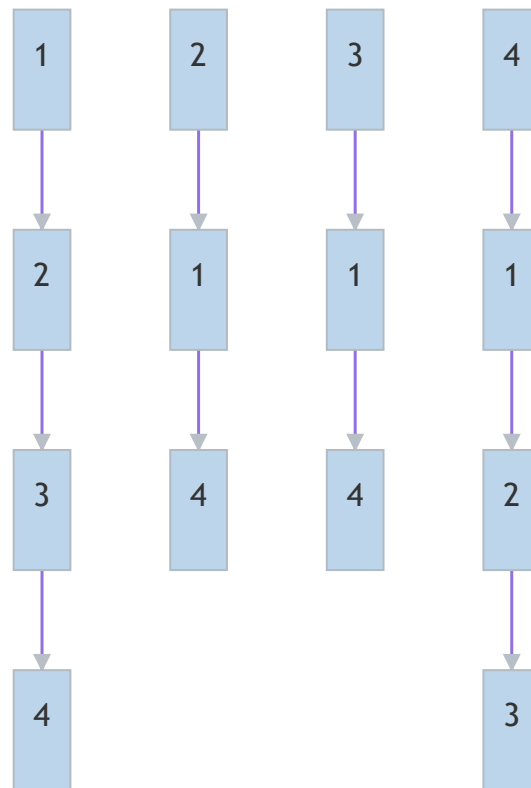$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Adjacency list

Array of linked lists, where Adj[u] contains a list containing all the neighbors of u.
Graph: Same as above
List:

1	2	3	4
↓	↓	↓	↓
2	1	1	1
↓	↓	↓	↓
3	4	4	2
↓			↓
4			3

Runtimes

	Matrix	List
Find all neighbors of v	$\mathcal{O}(n)$	$\mathcal{O}(deg_{out}(v))$
Find $v \in V$ without neighbors	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Check if $(v, u) \in E$	$\mathcal{O}(1)$	$\mathcal{O}(1 + deg_{out}(v))$
Insert edge	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Remove edge (v, u)	$\mathcal{O}(1)$	$\mathcal{O}(deg_{out}(v))$
Check whether an Eulerian path exists or not	$\mathcal{O}(|V| * |E|)$	$\mathcal{O}(|V| + |E|)$

Algorithms

Depth-First Search (DFS)

Used mainly to check whether a Graph can be topological sorted or not (⇔ has a cycle). A
topological sorting of a graph it's a sequence of all its nodes with the property that a node u
comes after a node v **if and only if** either a walk from v to u exists or u cannot be reached
starting from v.

Pseudocode

```
1  DFS(G):
2      t = 1
3      for (v in V not marked):
4          DFS-Visit(v)
```
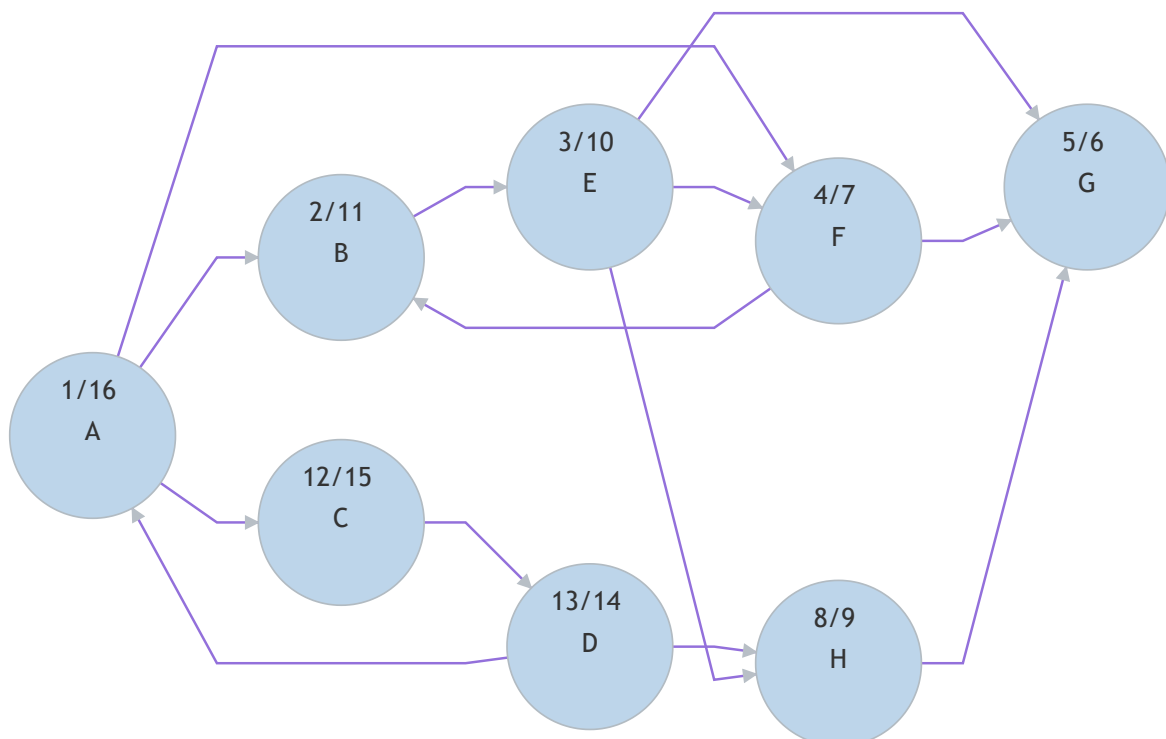
```
1  DFS-Visit(v):
2      pre[v] = t++
3      marked[v] = true
4      for ((v, u) in E, u not marked)
5          DFS_Visit(u)
6      post[v] = t++
```

Runtime

Operations	$T(n) \in \Theta(|E| + |V|)$
Memory	$T(n) \in \Theta(|V|)$

Edge classification (post and pre numbers)

Example: `DFS(A)` got called

This graph generate the following tree (rotated of 90 degree to save space):
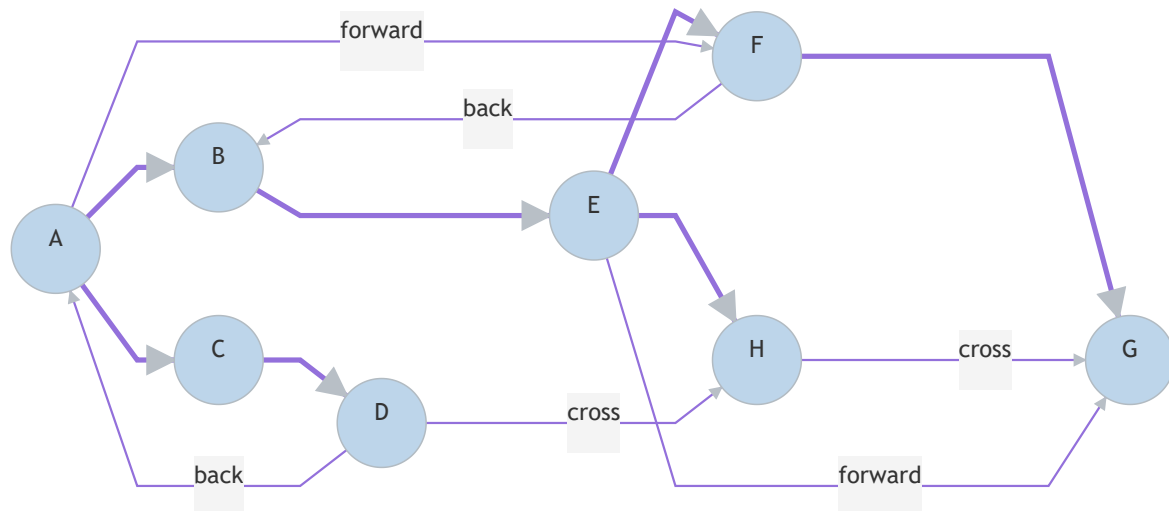
Pre and post number	Name of the edge $(v, u) \in E$
$pre(u) < pre(v)$ and $post(u) < post(v)$	Not possible
$pre(u) < pre(v)$ and $post(u) > post(v)$	**Tree edge**
$pre(u) < pre(v)$ and $post(u) < post(v)$ but $(u, v) \notin E$	**Forward edge**
$pre(u) > pre(v)$ and $post(u) > post(v)$	**Back edge**
$pre(u) > pre(v)$ and $post(u) > post(v)$	**Cross edge**
$pre(u) < pre(v)$ and $post(u) < post(v)$	Not possible

Remark: $\nexists$ back edge $\Leftrightarrow \nexists$ closed walk (cycle)

Breadth-First Search (BFS)

Instead of searching through the depth of a graph, one can also go first through all the successor of the root with the BFS algorithm.

Pseudocode

```
1  BFS(G):
2      for (v in V not marked):
3          BFS-Visit(v)
```

```
1   BFS-VIsit(v):
2       Q = new Queue()
3       active[v] = true //used to check whether a vertex is in the queue or not
4       enqueue(v, Q)
5       while (!isEmpty(Q)):
6           w = dequeue(Q)
7           visited[w] = true
8           for ((w, x) in E):
9               if(!active[x] && !visited[x]):
10                  active[x] = true
11                  enqueue(x, Q)
```

Runtime

Operations	$T(n) \in \Theta(|E| + |V|)$
Memory	$T(n) \in \Theta(|V|)$

Find shortest path in DAG (Directed Acyclic Graph)

We can compute a recurrence following the topological sorting of the graph.

Pseudocode

```
1   ShortestPath(G, s):
2       d[s] = 0, d[v] = inf
3       for (v in V \ {s}, following topological sorting):
4           for ((u, v) in E):
5               d[v] = min(d[v], d[u] + c(u,v))
```

Runtime

$T(n) \in \mathcal{O}(|E| * |V|)$ if adjacency list is given

Djikstra

Used to find the shortest (cheapest) path between two nodes in a graph.
Remark: The graph must **not** have negative weights

Pseudocode

```
1   Dijkstra(G, s):
2       for (v in V):
3           d[v] = infinity
4           parent[v] = null
5           insert(Q, v, d[v])
6       d[s] = 0
7       Q = new Queue()
8       decreaseKey(Q, s, 0) // decrease the priority of s to 0 (min)
9       while(!Q.isEmpty()):
10          v* = Q.extractMin() // extract from Q the node with minimum priority
11          for ((v*, v) in E):
12              dist = d[v*] + w(v*, v)
13              if (dist < d[v]):
14                  d[v] = dist
```

```
15                    parent[v] = v*
16                    decreaseKey(Q, v, d[v])
```

Runtime

If implemented with a Heap: $T(n) \in \mathcal{O}((|E| + |V|) * log(|V|))$
If implemented with a **Fibonacci-Heap**: $T(n) \in \mathcal{O}((|E| + |V| * log(|V|)))$

Bellman-Ford

Used for graph with general weight (**positive and negative!**)

Pseudocode

```
1   BellmanFord(G, s):
2       for (v in V):
3           distance[v] = infinity
4           parent[v] = null
5       distance[s] = 0
6       for (i = 1, 2, ..., |V| - 1):
7           for ((u, v) in E):
8               if(distance[v] > distance[u] + w(u, v)):
9                   distance[v] = distance[u] + w(u, v)
10                  parent[v] = u
11      for ((u, v) in E):
12          if (distance[u] + w(u, v) < distance[v]):
13              return "negative cyrcle!"
```

Runtime

$T(n) \in \mathcal{O}(|E| * |V|)$

Boruvka

Used to find a MST in a given graph G

Minimum Spanning Trees (MSTs)

A minimum spanning tree is a subgraph $H = (V, E^*)$ of a graph $G = (V, E)$ with $E^* \subseteq E$, such that every vertex $v \in V$ is connected and that **the sum of all edges' weight is minimal**.

Pseudocode

```
1   Boruvka(G):
2       F = new Set() // Initialize a new forest with every vertex being a tree
    and 0 edges
3       while (F not SpanningTree): // check that ZHKs of F > 1 or number of
    edges < |V| - 1
4           ZHKs of F = (S1, ..., Sk)
5           minEdges of S1, ..., Sk = (e1, ..., ek)
6           F = F U (e1, ..., ek)
7       return F
```
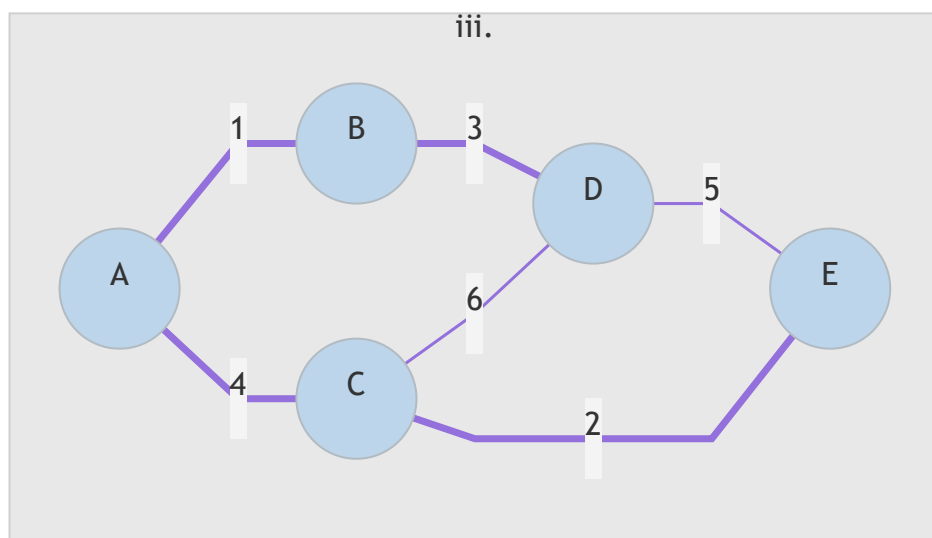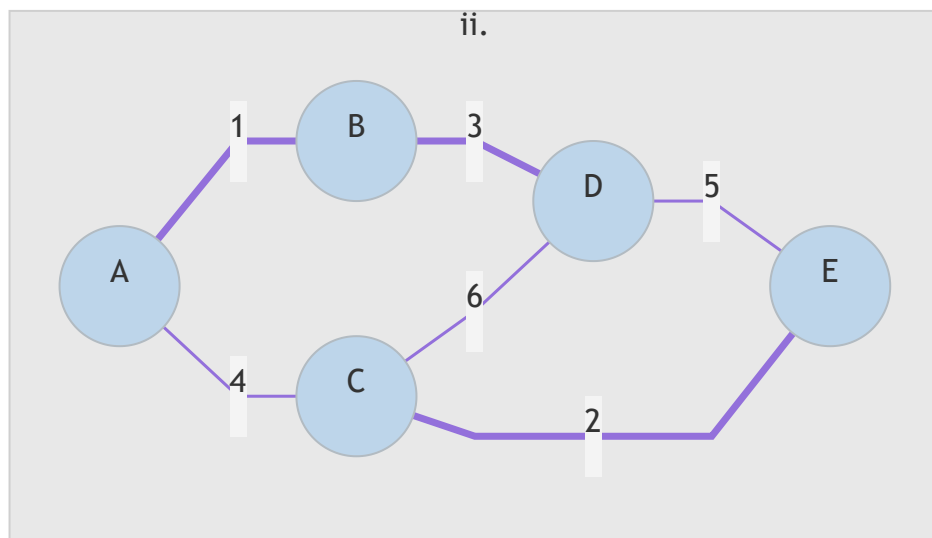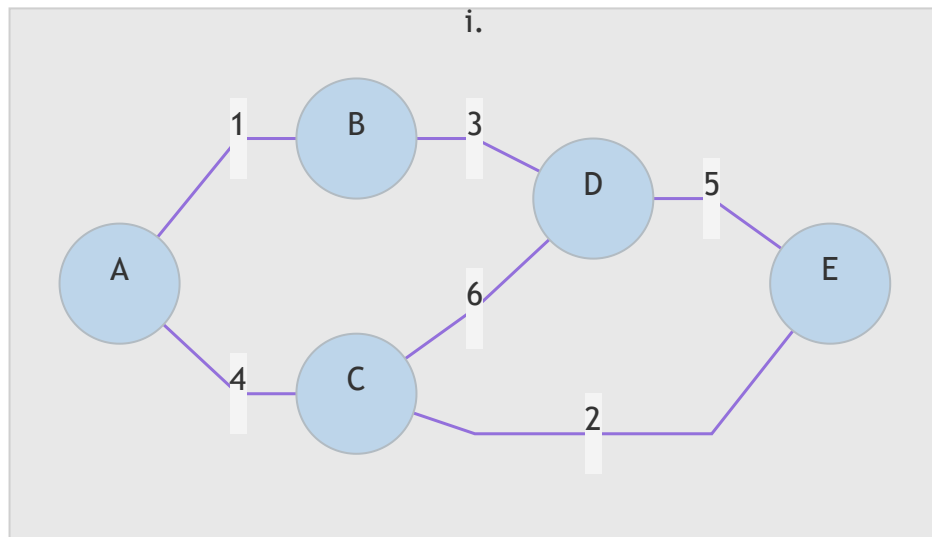
Runtime

$$T(n) \in \mathcal{O}((|E| + |V|) * log(|V|))$$

Example

First choose the minimal edge for every vertex and add them to the new graph. Then repeat for every ZHK (vertices connected with edges) until you have a MST (until there is only 1 ZHK).

Runtime

$$T(n) \in \mathcal{O}((|E| + |V|) * log(|V|))$$

Example

i.

ii.

iii.

Prim

Alternative to Kruskal, it needs a starting vertex as input.

Pseudocode

```
1  Prim(G, s):
2      MST = new Set()
3      H = new minHeap(V, infinity)
4      decreaseKey(H, s, 0)
5      while (!H.isEmpty()):
6          v = extractMin(H)
7          MST.add(v)
8          for ((v, u) in E && u not in MST)
9              decreaseKey(H, u, w(v, u))
```
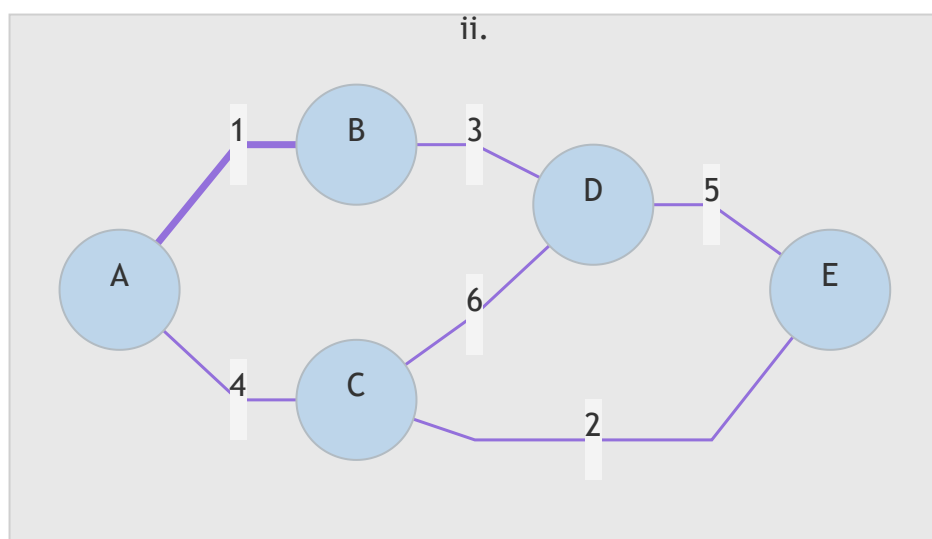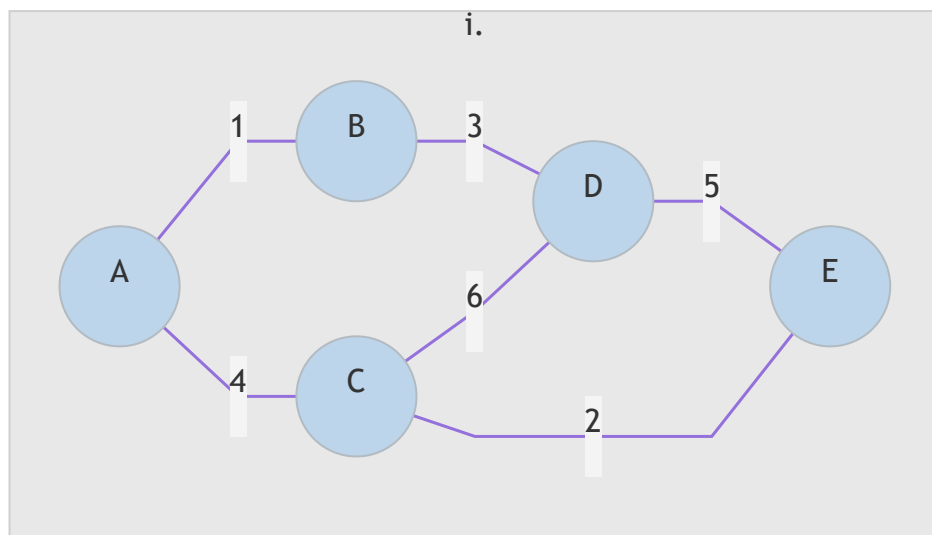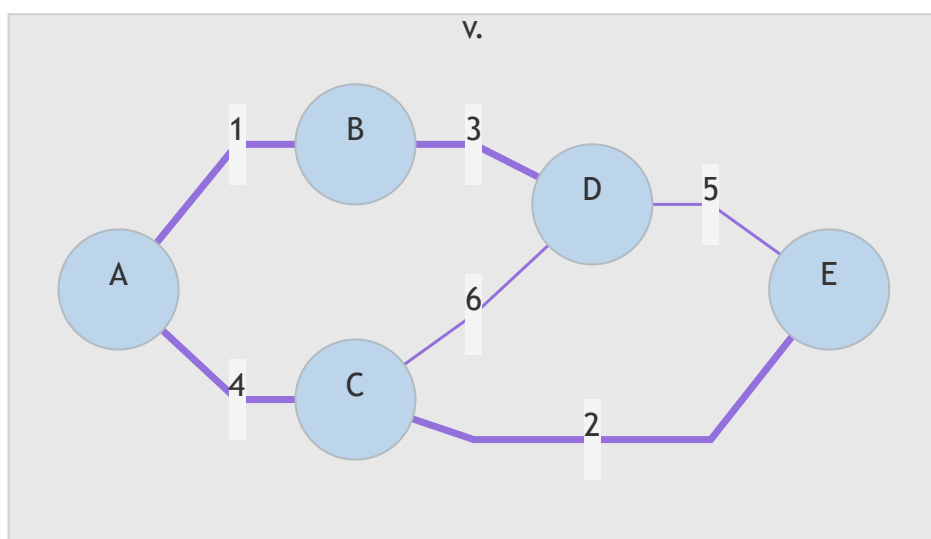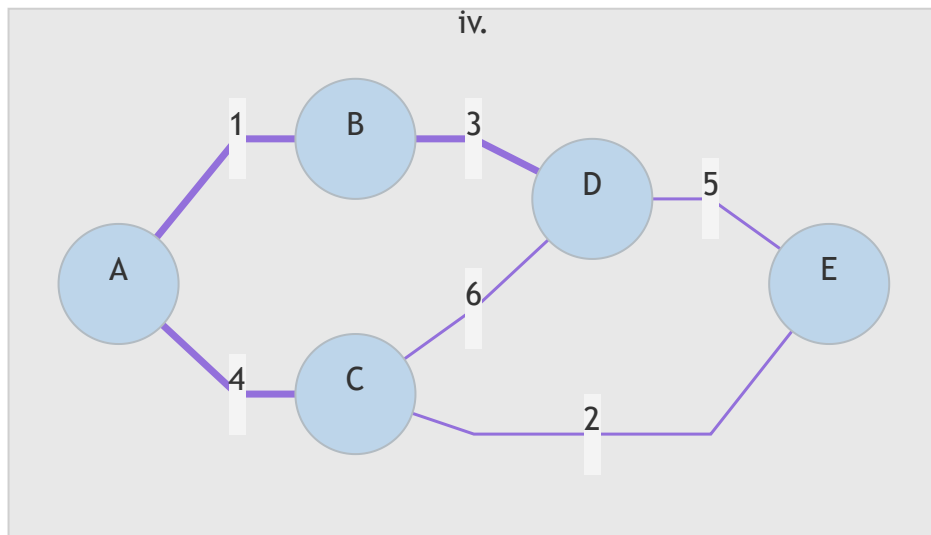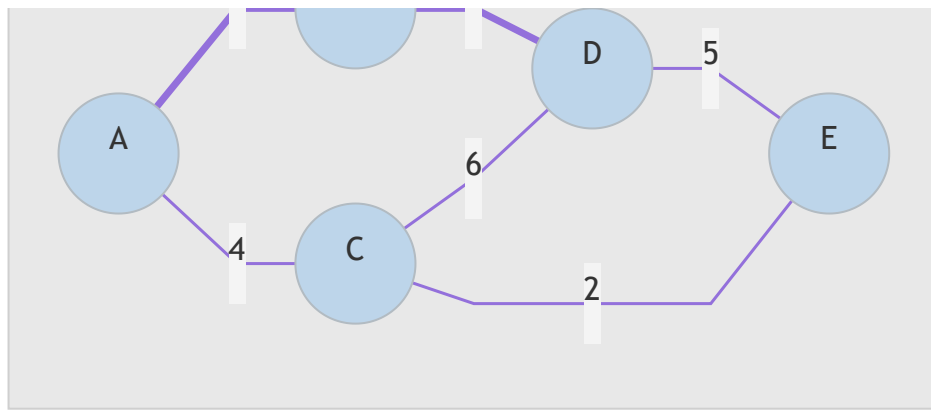
Runtime

$$T(n) \in \mathcal{O}((|E| + |V|) * log(|V|))$$

Example

Add the minimal edge adjacent to s. Then take the newly created ZHK and add to it its minimal outgoing edge. Proceed like that until you have a spanning tree (all the vertices are connected).

i.

ii.

iii.

iv.

v.

Kruskal

Another algorithm to find a MST in a given graph. It sorts edges by weight and adds them one by one, **unless adding an edge would form a cycle**.

Pseudocode

```
1  Kruskal(G):
2      MST = new Set()
3      E.sort() // sort all edges by weight
4      for ((u, v) in E):
5          if (u and v in 2 different ZHKs of MST):
6              MST.add((u, v))
```
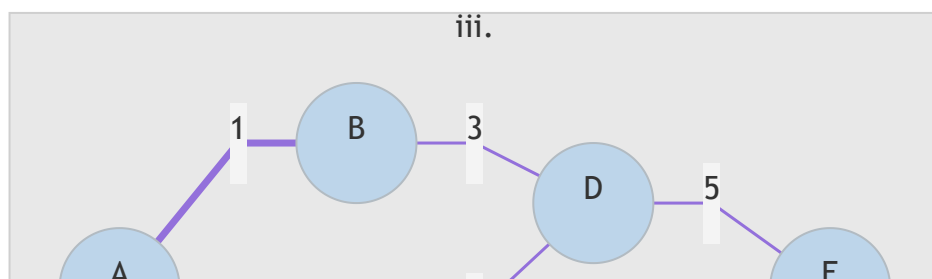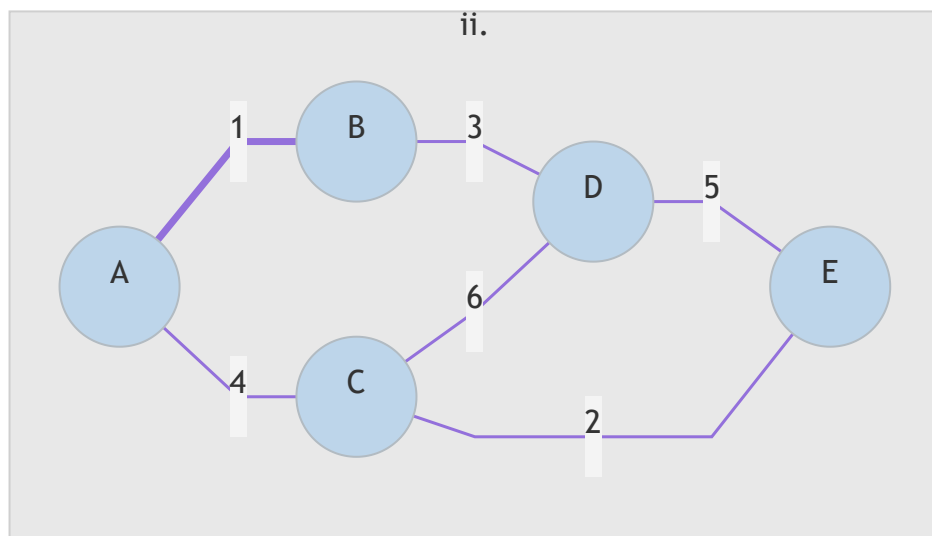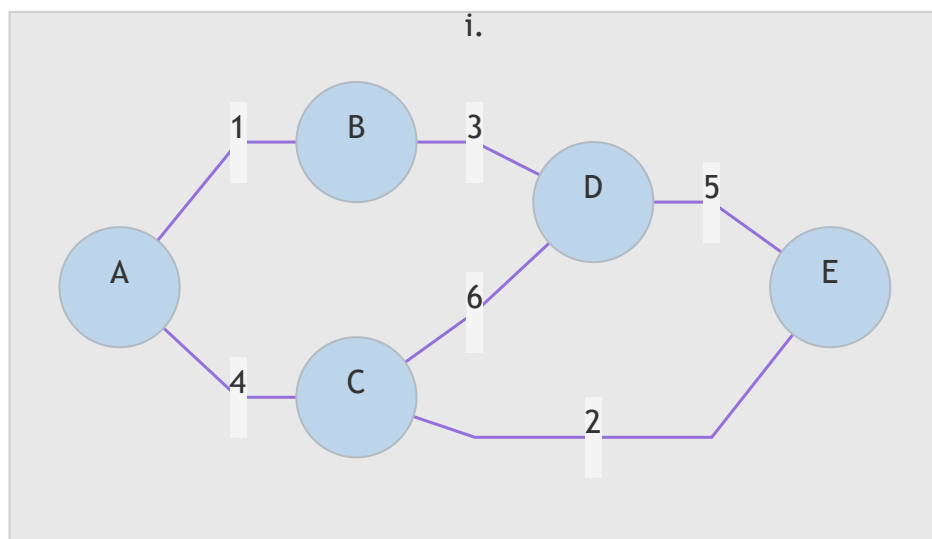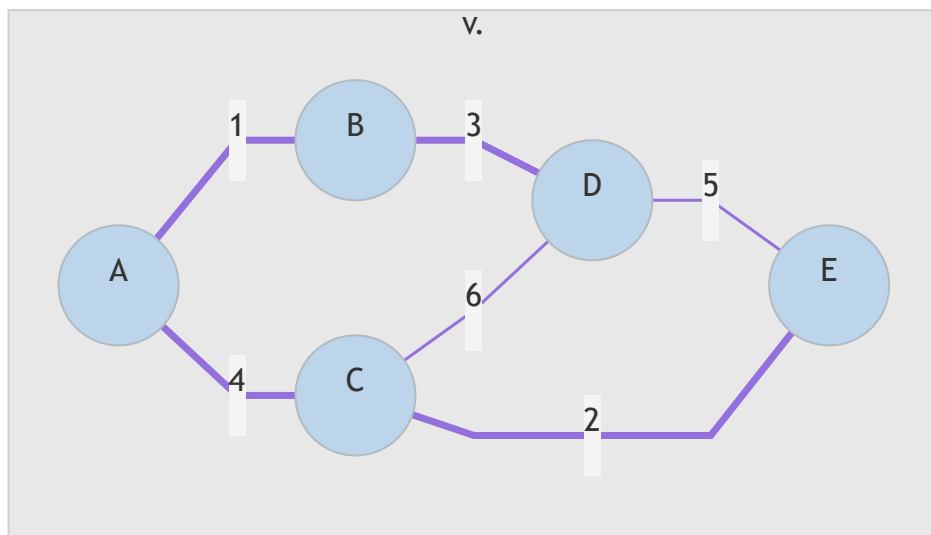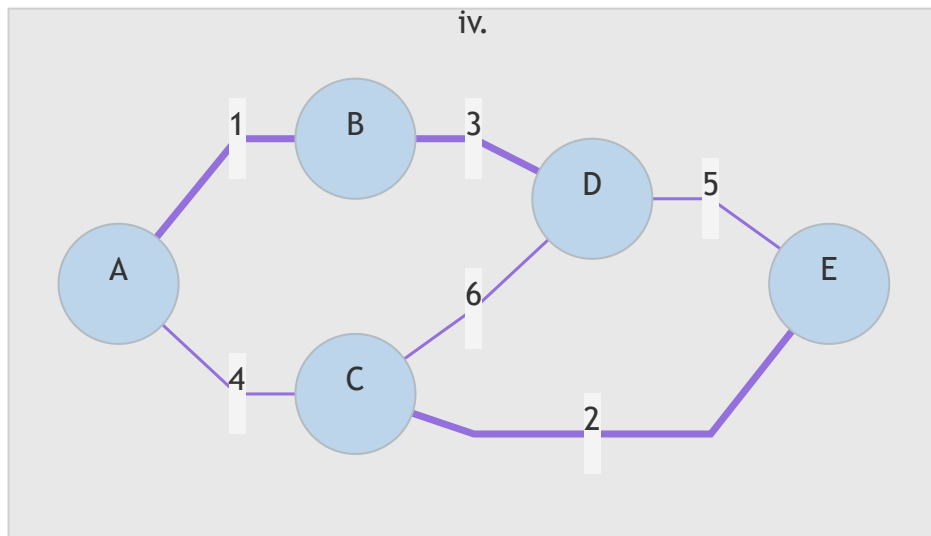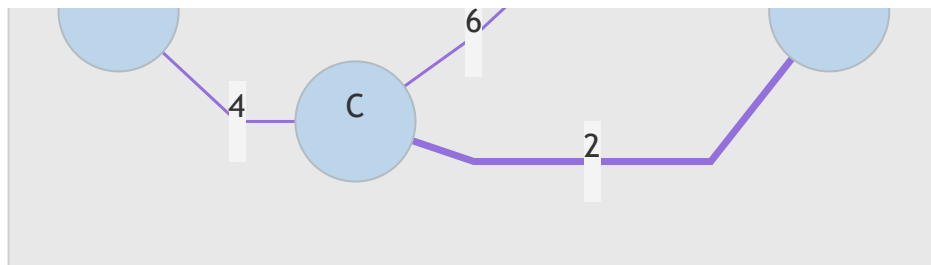
Runtime

If implemented normally: $T(n) \in \mathcal{O}(|E| * |V| + |E| * log(|E|))$ (second part to sort)
If implemented with an improved union-find DS: $T(n) \in \mathcal{O}(|V| * log(|V|) + |E| * log(|E|))$
(second part to sort)

Example

Add edges one by one following weight-order. If adding an edge would form a cycle, skip it.

i.

ii.

iii.

6

4   C   2

iv.

1   B   3

A   D   5

6   E

4   C

2

v.

1   B   3

A   D   5

6   E

4   C

2

Floyd-Warshall

Used to solve the **all-pair shortest path** problem, i.e., to find the shortest distance between **any** two vertices of a given graph G.
It makes use of a 3-Dimensional DP table.

Pseudocode

`d[i][u][v]` represents the shortest path from u to v passing through $\leq i$ vertices.

```
1   FloydWarshall(G):
2       for (v in V):
3           d[0][v][v] = 0 // layer 0, row v, column v
4       for ((v, u) in E):
5           d[0][v][u] = w(v, u)
6       else: // if u, v isn't in E
7           d[0][v][u] = infinity
8       for (i = 1, ..., |V|):
9           for (u = 1, ..., |V|):
10              for (v = 1, ..., |V|):
11                  d[i][u][v] = min(d[i-1][u][v], d[i-1][u][i] + d[i-1][i][v])
12      return d
```

Remarks:

- This algorithm can be implemented **inplace**, it just suffice to leave the indices away.
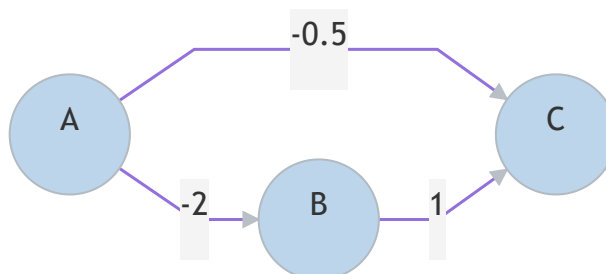- The algorithm does **not** work if negative cycles are present.

Runtime
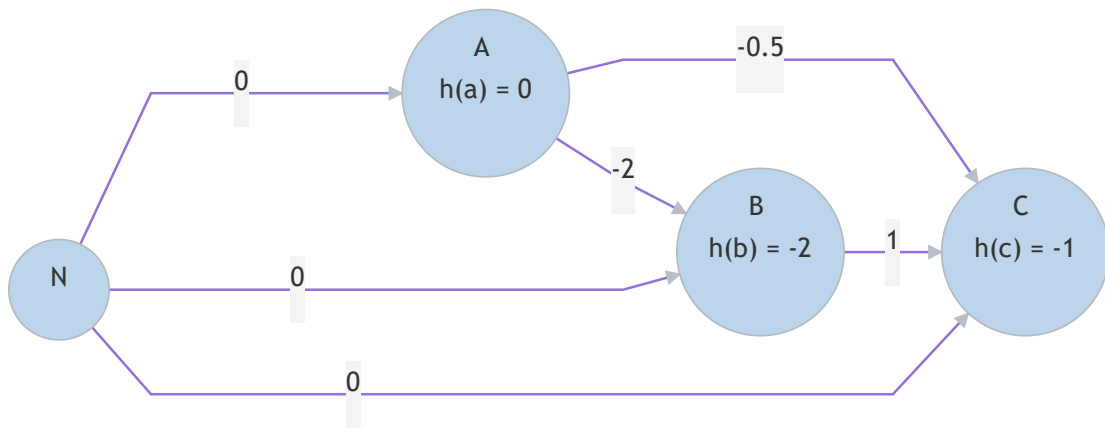
$T(n) \in \mathcal{O}(|V|^3)$

Johnson

Used to solve the all-pair shortest path problem. First one has to make every weight positive, by adding an "external" vertex, and then proceed by using Dijkstra $|V|$ times.
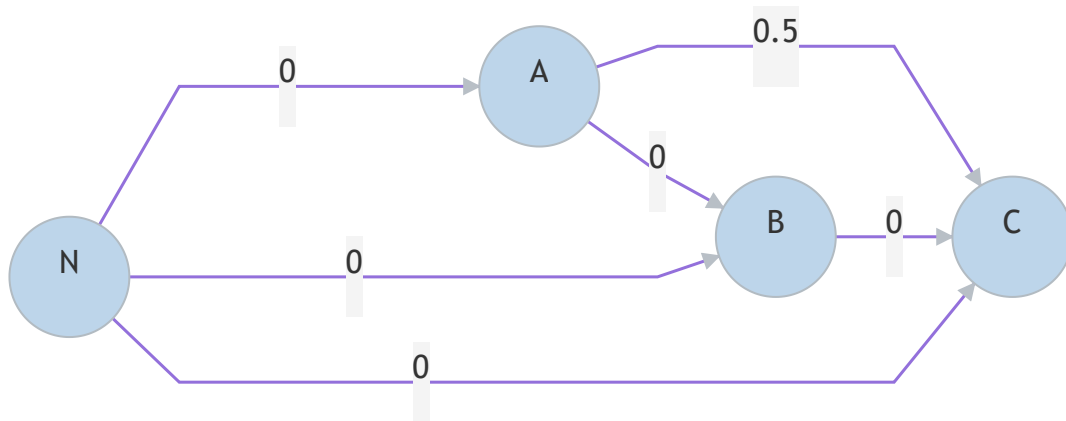
Example

- First, add the new vertex, and connect it to every other vertex with weight 0.
  $h(n)$ is the "height" of the node n, equals to **the shortest path from N to n**, found by applying n times Dijkstra.

- We now can modify each weight $w(u, v)$ of each edge into a new weight $w^*(u, v) = w(u, v) + (h(u) - h(v))$

Runtime

- Create new node and add new edges: $\mathcal{O}(|V|)$
- Assign h-values: Bellman-Ford, $\mathcal{O}(|V| * |E|)$
- $|V|$ times Dijkstra: $\mathcal{O}(|V| * |E| + |V|^2 * log(|V|))$

All Pair-Shortest Path

All the algorithms we know to solve the APSP problem can be compared in the following way (**top**: less general, **bottom**: more general):

Graph	Algorithm	Runtime
$G = (V, E)$	$\lvert V \rvert * BFS$	$\mathcal{O}(\lvert V \rvert * \lvert E \rvert + \lvert V \rvert^2)$
$G = (V, E, w)$ $w : E \to \mathbb{R}^+$	$\lvert V \rvert * Dijkstra$	$\mathcal{O}(\lvert V \rvert * \lvert E \rvert + \lvert V \rvert^2 * log(\lvert V \rvert))$
$G = (V, E, w)$ $w : E \to \mathbb{R}$	$\lvert V \rvert * Bellman - Ford$ $Floyd - Warshall$ $Johnson$	$\mathcal{O}(\lvert V \rvert^2 * \lvert E \rvert)$ $\mathcal{O}(\lvert V \rvert^3)$ $\mathcal{O}(\lvert V \rvert * \lvert E \rvert + \lvert V \rvert^2 * log(\lvert V \rvert))$