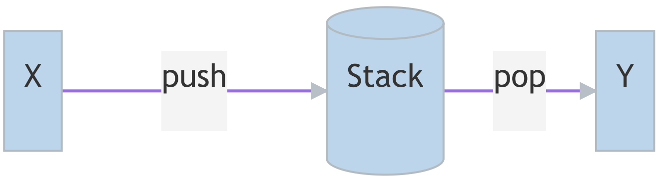
Abstract Data Types (ADTs)

Stack

Methods

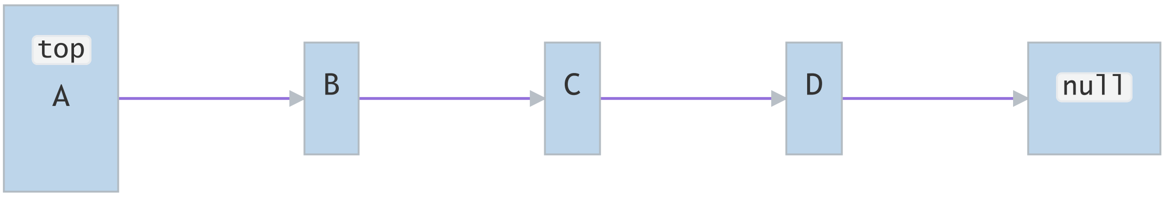
- `push(x, S)`: Puts `x` onto the stack `S`
- `pop(S)`: Remove (and returns) the top element of the stack `S`
- `top(S)`: Returns the top element of the stack `S`

Visualization



Structure

Linked List:



Runtime

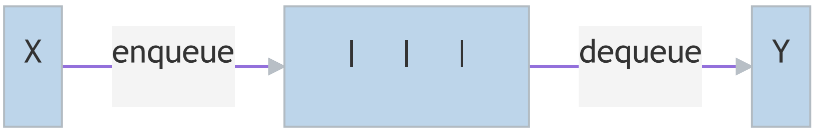
- `push(x, S)` $\in \mathcal{O}(1)$
- `pop(S)` $\in \mathcal{O}(1)$
- `top(S)` $\in \mathcal{O}(1)$

Queue

Methods

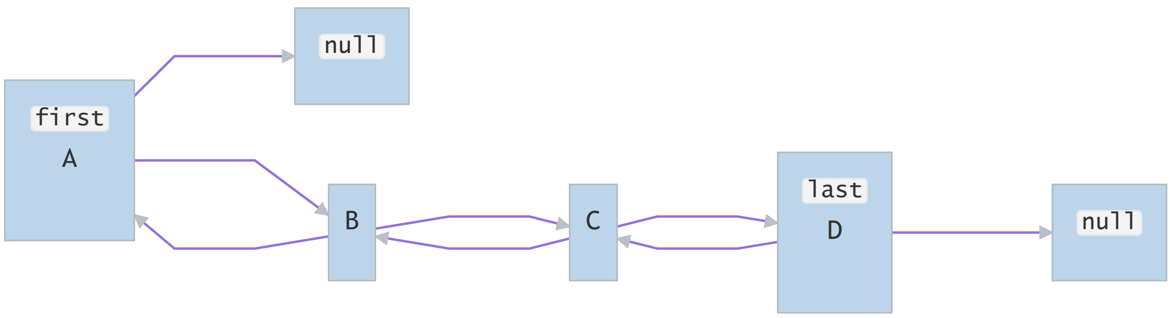
- `enqueue(x, S)` : Add `x` to the queue `S`
- `dequeue(S)` : Remove the first element of the queue `S`

Visualization



Structure

Doubly Linked List:



Runtime

- `enqueue(x, S)` : $\in \mathcal{O}(1)$
- `dequeue(S)` : $\in \mathcal{O}(1)$

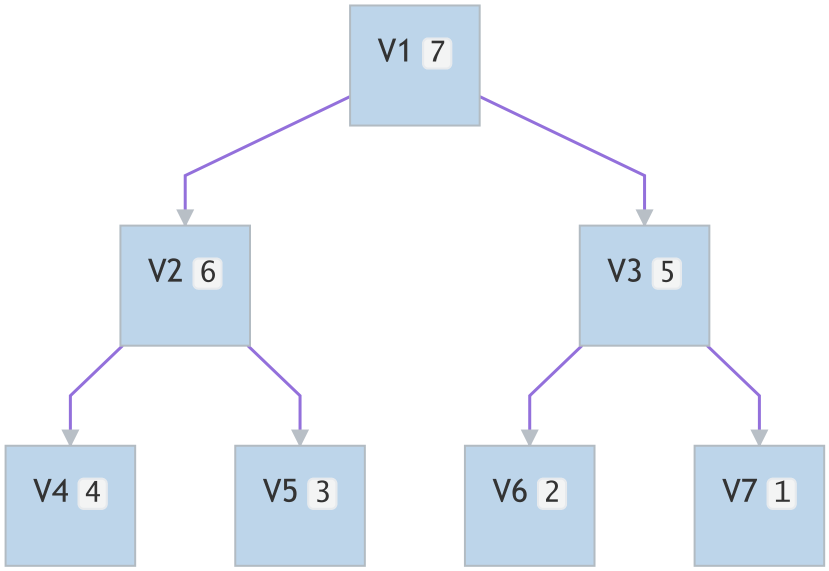
Priority Queue

Methods

- `insert(x, p, P)` : Insert `x` with priority `p` into the queue `P`
- `extractMax(P)` : Extracts the elements with maximal priority from the queue `P`

Structure

Max-Heap:



Runtime

- `insert(x, p, P)`: $\in \mathcal{O}(log(n))$
- `extractMax(P)`: $\in \mathcal{O}(log(n))$

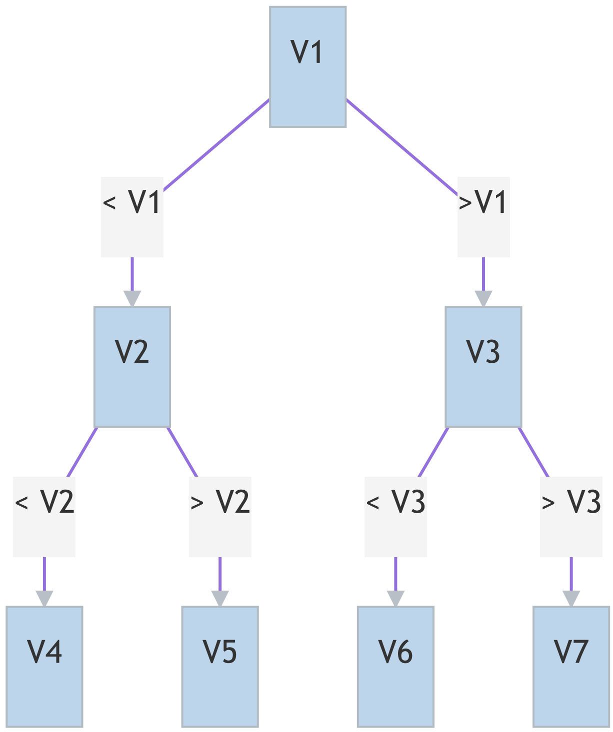
Dictionary

Methods

- `search(x, W)`: Finds `w` in dictionary `W`
- `insert(x, W)`: Insert `x` in dictionary `W`
- `remove(x, W)`: Remove `x` from the dictionary `W`

Structure

Search Tree:



Union-Find

Data structure used to compare ZHKs of a given graph.

Methods

- `make(V)`: Create a data structure for $F = \emptyset$
- `same(u, v)`: Test whether u, v are in the same ZHK of F
- `union(u, v)`: Merge ZHKs where u and v are

Structure

List `rep[]` which stores the identifiers of all the vertices. `rep[u]` = `rep[v]` if and only if THK(v) = ZHK(u).

Implementation

```
make(V):
    for (v in V):
        rep[v] = v

same(u, v):
    return rep[u] == rep[v]

// members[rep[u]] is a list containing all the nodes in ZHK(u)
union(u, v):
    for (x in memebers[rep[u]]):
        rep[x] = rep[v]
        members[rep[v]].add(x)
```

Runtime

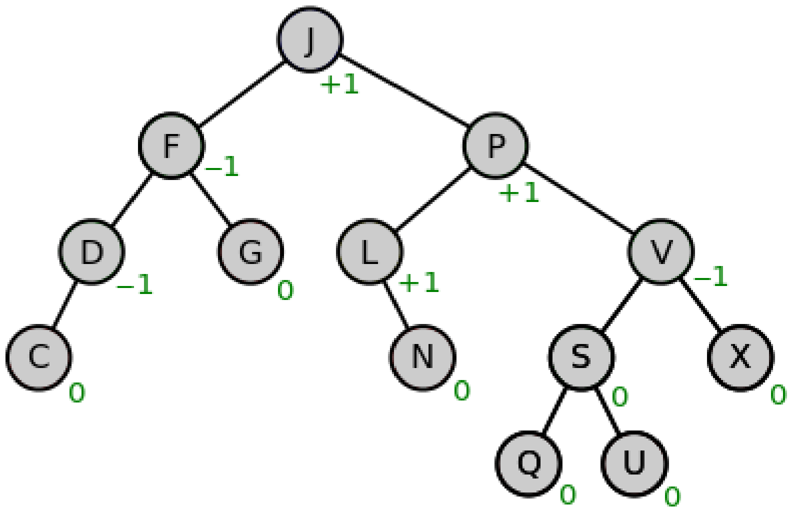
- `make(v)` $\in \mathcal{O}(|V|)$
- `same(u, v)` $\in \mathcal{O}(1)$
- `union(u, v)` $\in \mathcal{O}(|ZHK(u)|)$

AVL Trees

Description

Most of the BST operations (e.g., `search`, `max`, `min`, `insert`, `delete`,...) take $\mathcal{O}(h)$ time where h is the height of the BST. The cost of these operations may become $\mathcal{O}(n)$ for a skewed Binary tree. If we make sure that height of the tree remains $\mathcal{O}(log(n))$ after every insertion and deletion, then we can guarantee an upper bound of $\mathcal{O}(log(n))$ for all these operations. The height of an AVL tree is always $\mathcal{O}(log(n))$ where n is the number of nodes in the tree

We define the balance of a vertex v, $bal(v) = h(T_r(v)) - h(T_l(v))$. For a Search Tree to fulfill the AVL-condition, we need $\forall v \; bal(v) \in \{-1, 0, 1\}$



An AVL Tree with every balance value written below the corresponding node

We distinguish three states of a node p before inserting a node:

- $bal(p) = -1$: not possible
- $bal(p) = 0$
- $bal(p) = 1$

Insertion

Left and right rotation

```
T1, T2 and T3 are subtrees of the tree rooted with y (on the left side) or x (on the right side)

                                     y                                x
                                    / \     Right Rotation          /  \
                                   x   T3   - - - - - - - - >      T1   y
                                  / \       < - - - - - - - -          / \
                                 T1  T2     Left Rotation            T2  T3

Keys in both of the above trees follow the following order:
                               keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
So BST property is not violated anywhere.
```

Insertion and rotations

Steps to follow for insertion

Let the newly inserted node be w

- Perform standard BST insert for w.
- Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.
- Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:
  - y is left child of z and x is left child of y (**Left Left Case**)
  - y is left child of z and x is right child of y (**Left Right Case**)
  - y is right child of z and x is right child of y (**Right Right Case**)
  - y is right child of z and x is left child of y (**Right Left Case**)

a) Left Left Case

```
T1, T2, T3 and T4 are subtrees.
         z                                      y
        / \                                   /   \
       y   T4      Right Rotate (z)          x      z
      / \          - - - - - - - - ->      /  \    /  \
     x   T3                               T1  T2  T3  T4
    / \
  T1   T2
```

b) Left Right Case

```
     z                               z                           x
    / \                            /   \                        /  \
   y   T4  Left Rotate (y)        x    T4  Right Rotate(z)    y      z
  / \      - - - - - - - - ->    /  \      - - - - - - - ->  / \    / \
T1   x                          y    T3                    T1  T2 T3  T4
    / \                        / \
  T2   T3                    T1   T2
```

c) Right Right Case

```
  z                                y
 /  \                          /   \
T1   y      Left Rotate(z)      z      x
    /  \    - - - - - - - ->   / \    / \
   T2   x                     T1  T2 T3  T4
       / \
     T3  T4
```

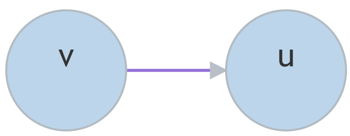
d) Right Left Case

```
   z                            z                            x
  / \                          / \                          /  \
T1   y   Right Rotate (y)    T1   x      Left Rotate(z)   z      y
    / \  - - - - - - - - ->     /  \   - - - - - - - ->  / \    / \
   x   T4                      T2   y                  T1  T2  T3  T4
  / \                              /  \
T2   T3                           T3   T4
```

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



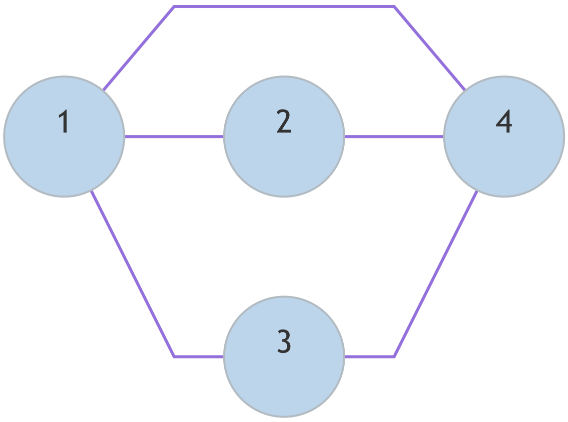
- **deg_{in}(v)**: number of incoming edges into v
- **deg_{out}(v)**: number of outgoing edges into v

Graph Representation

Adjacency matrix:

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

Graph:



Matrix:

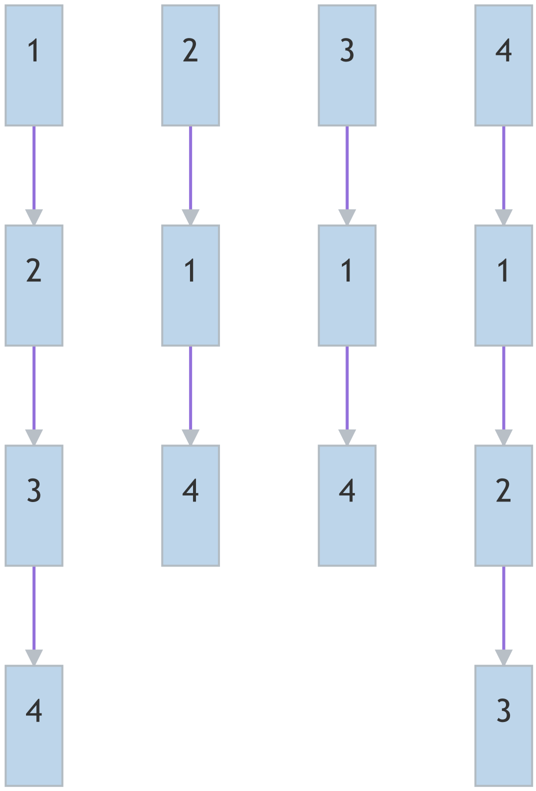
$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Adjacency list

Array of linked lists, where Adj[u] contains a list containing all the neighbors of u.

Graph: Same as above

List:



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 $(v, u) \in E$	$\mathcal{O}(1)$	$\mathcal{O}(1 + min(deg_{out}(v), deg_{out}(u)))$
Insert edge	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Remove edge v	$\mathcal{O}(1)$	$\mathcal{O}(deg_{out}(v))$
Check whether an Eulerian path exists or not	$\mathcal{O}(\lvert V \rvert * \lvert E \rvert)$	$\mathcal{O}(\lvert V \rvert + \lvert E \rvert)$

Algorithms

Depth-First Search (DFS)

Used mainly to check whether a Graph can be topological sorted or not ($\Leftrightarrow$ 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

```
DFS(G):
    t = 1
    for (v in V not marked):
        DFS-Visit(v)
```

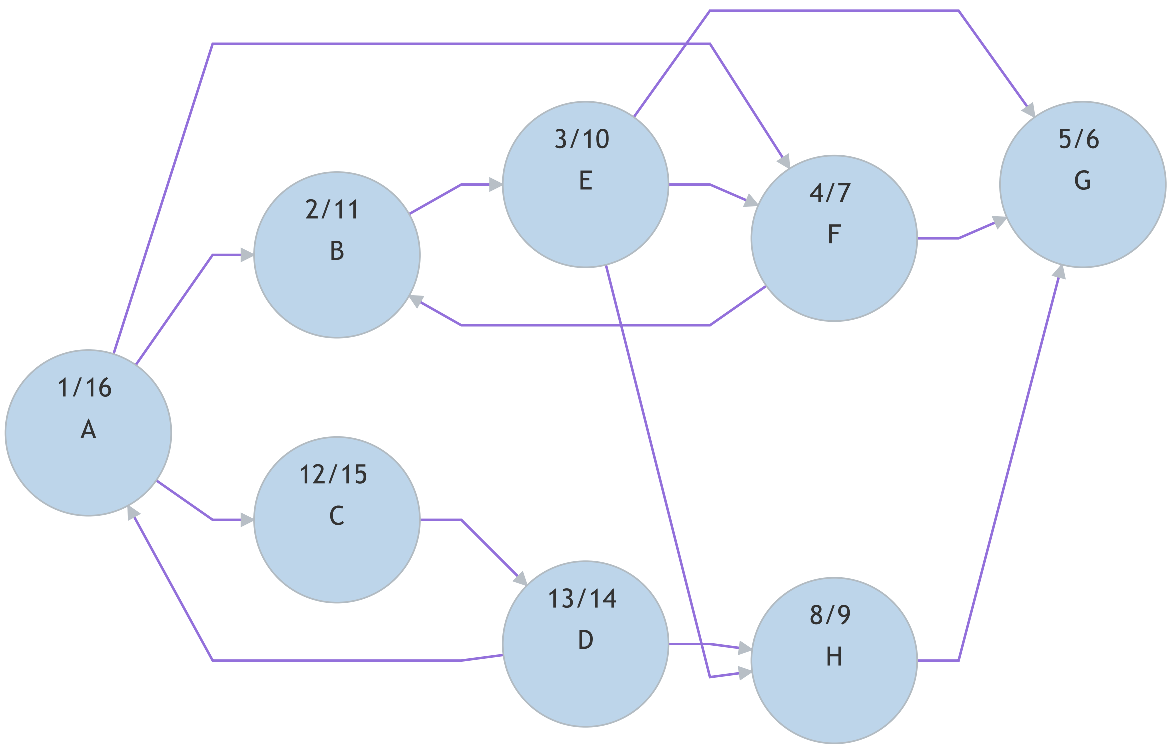
```
DFS-Visit(v):
    pre[v] = t++
    marked[v] = true
    for ((u, v) in E not marked)
        DFS_Visit(u)
    post[u] = t++
```

Runtime

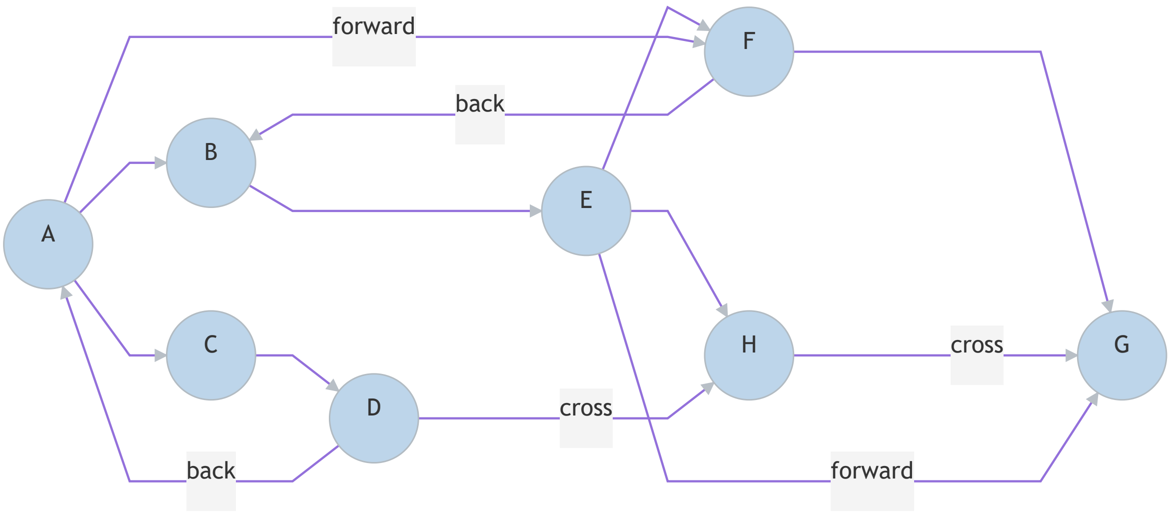
Operations	$T(n) \in \Theta(\lvert E\rvert + \lvert V\rvert)$
Memory	$T(n) \in \Theta(\lvert V\rvert)$

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**
$pre(u) < pre(w)$ 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

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

```
BFS-VIsit(v):
    Q = new Queue()
    active[v] = true //used to check whether a vertex is in the queue or not
    enqueue(v, Q)
    while (!isEmpty(Q)):
        w = dequeue(Q)
        visited[w] = true
        for ((w, x) in E):
            if(!active[x] && !visited[x]):
                active[x] = true
                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

```
ShortestPath(V):
    d[s] = 0, d[v] = inf
    for (v in V \ {s}, following topological sorting):
        for (u, v, s.t. (u, v) in E):
            d[v] = min(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

```
DijkstraG, s):
    for (v in V):
        distance[v] = infinity
        parent[v] = null
    distance[s] = 0
    Q = new Queue()
    insert(Q, s, 0) // insert s into the queue Q, with priority 0 (min)
    while(!Q.isEmpty()):
        v* = Q.extractMin() // extract from Q the node with minimum distance
        for ((v*, v) in E):
            if (parent[v] == null):
                distance[v] = distance[v*] + w(v*, v)
                parent[v] = v*
            else if (distance[v*] + w(v*, v) < distance[v]):
```

```
                distance[v] = distance[v*] + w(v*, v)
                parent[v] = v*
                decreaseKey(Q, v, distance[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

```
BellmanFord(G, s):
    for (v in V):
        distance[v] = infinity
        parent[v] = null
    distance[s] = 0
    for (i = 1, 2, ..., |V| - 1):
        for ((u, v) in E):
            if(distance[v] > distance[u] + w(u, v)):
                distance[v] = distance[u] + w(u, v)
                parent[v] = u
    for ((u, v) in E):
        if (distance[u] + w(u, v) < distance [v]):
            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

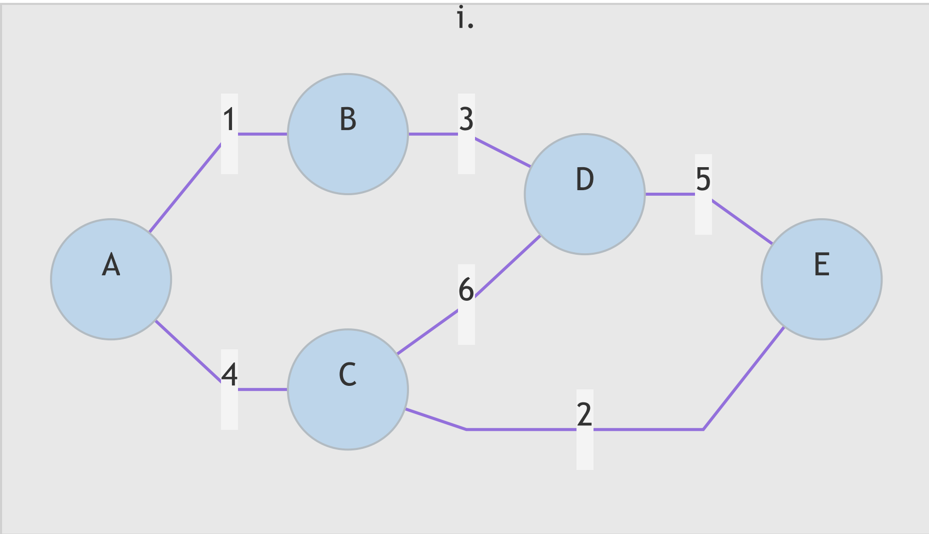
```
Boruvka(G):
    F = new Set() // Initialize a new forest with every vertex being a tree and 0 edges
    while (F not SpanningTree): // check that ZHKs of F > 1
        ZHKs of F = (S1, ..., Sk)
        minEdges of S1, ..., Sk = (e1, ..., ek)
        F = F U (e1, ..., ek)
    return F
```

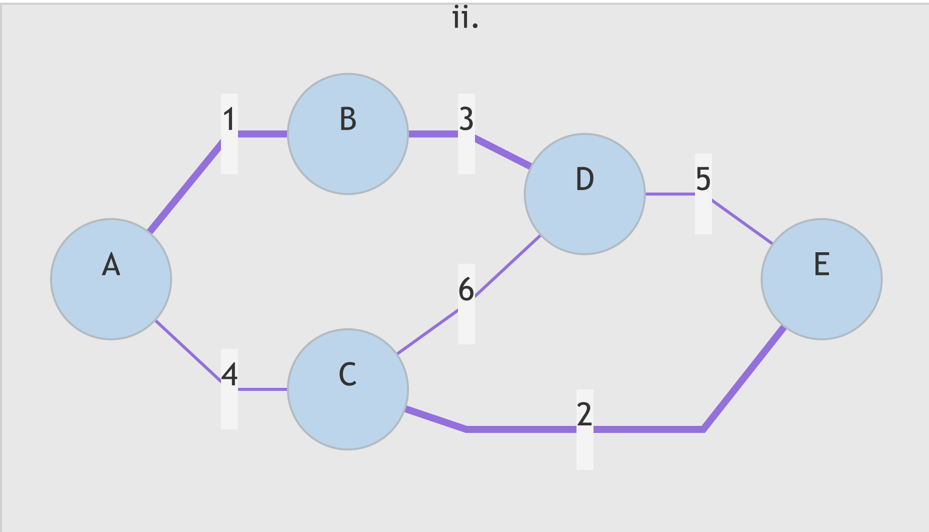
Runtime

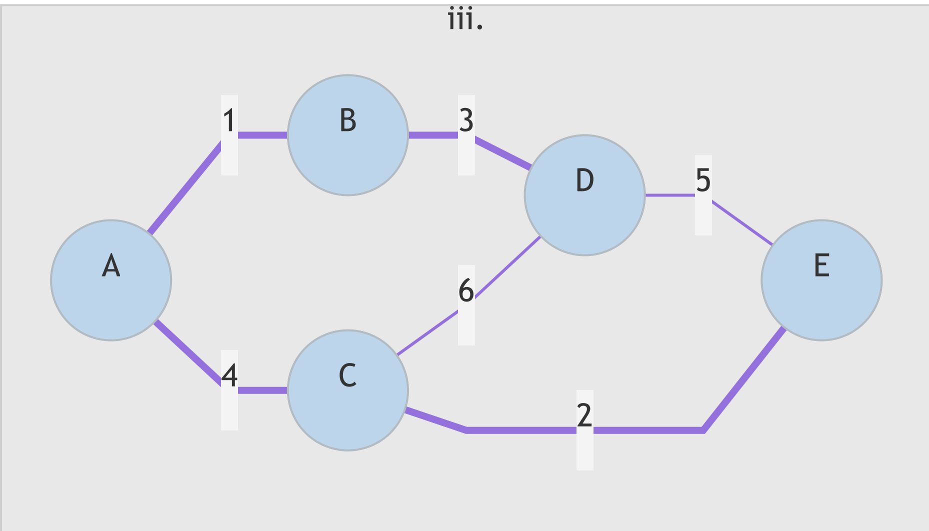
$T(n) \in \mathcal{O}((|E| + |V|) * log(|V|))$

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).







Prim

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

Pseudocode

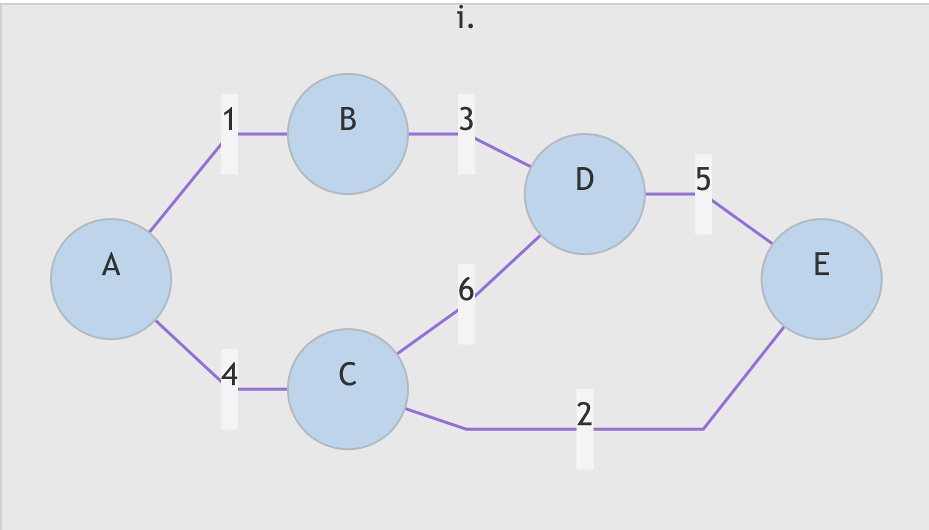
```
Prim(G, s):
    MST = new Set()
    H = new Heap(V, infinity)
    for (v in V):
        d[v] = infinity
    d[s] = 0
    decreaseKey(H, s, 0)
    while (!H.isEmpty()):
        v = extractMin(H)
        MST.add(v)
        for ((v, u) in E && v != s)
            d[v] = min(d[v], w(v, u))
            decreaseKey(H, v, d[v])
```

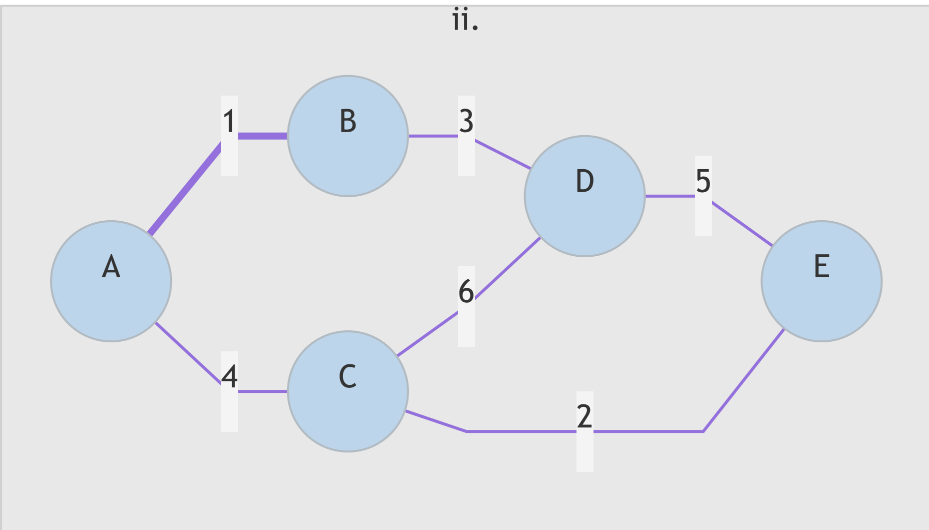
Runtime

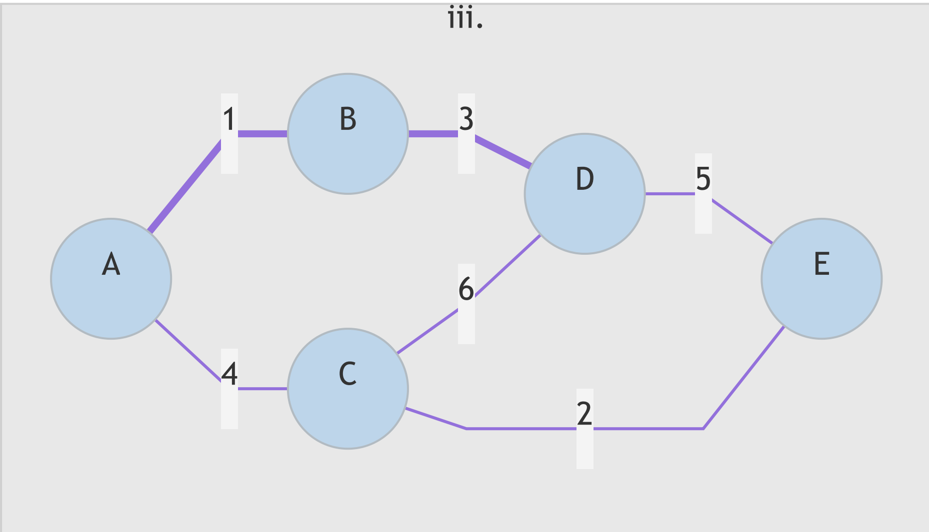
$T(n) \in \mathcal{O}((|E| + |V|) * log(|V|))$

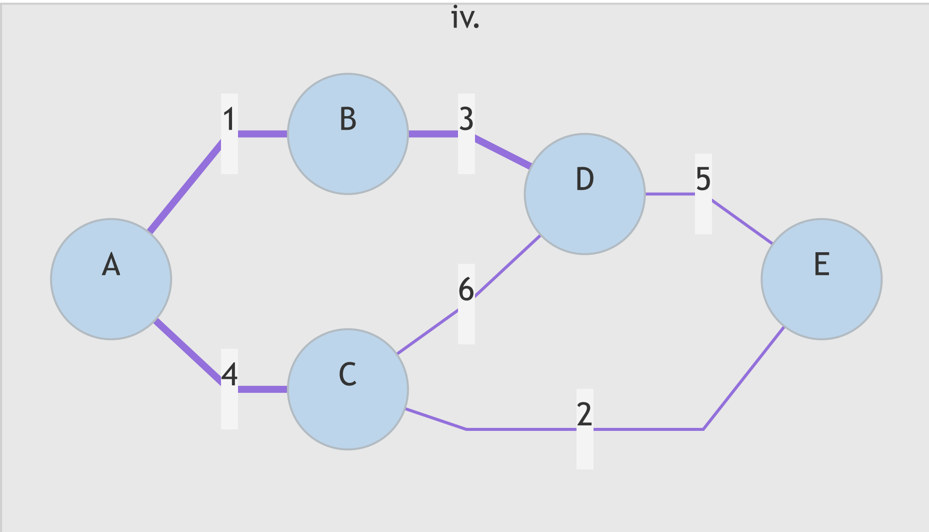
Example

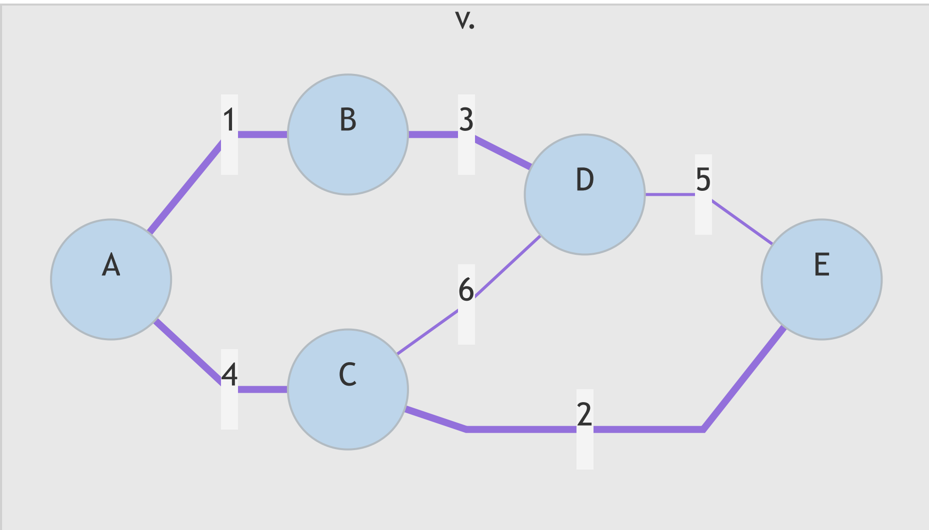
Add the minimal edge adjacent to s. Then take the newly created ZHK and add to it its minimal outgoing edge. Proceed like that until you have a spanning tree (all the vertices are connected).











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

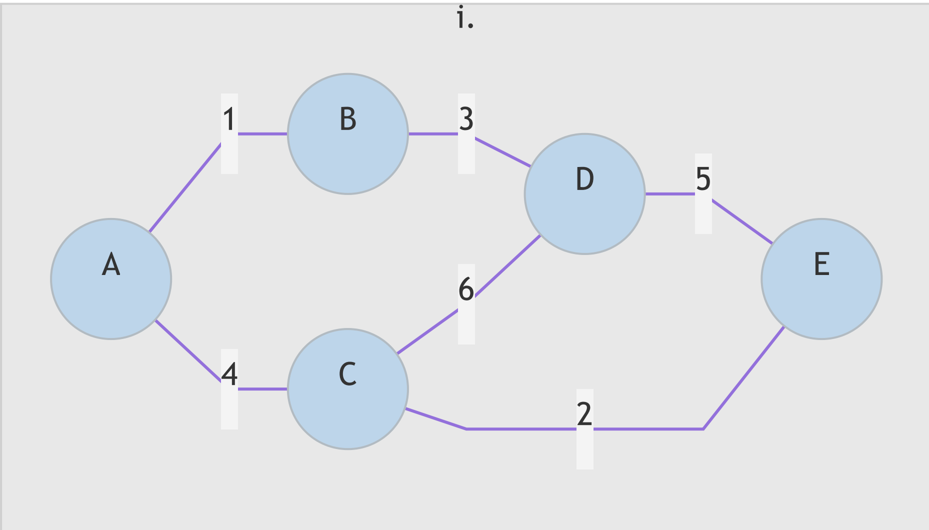
```
Kruskal(G):
    MST = new Set()
    E.sort() // sort all edges by weight
    for ((u, v) in E):
        if (u and v in 2 different ZHKs of MST):
            MST.add(e)
```

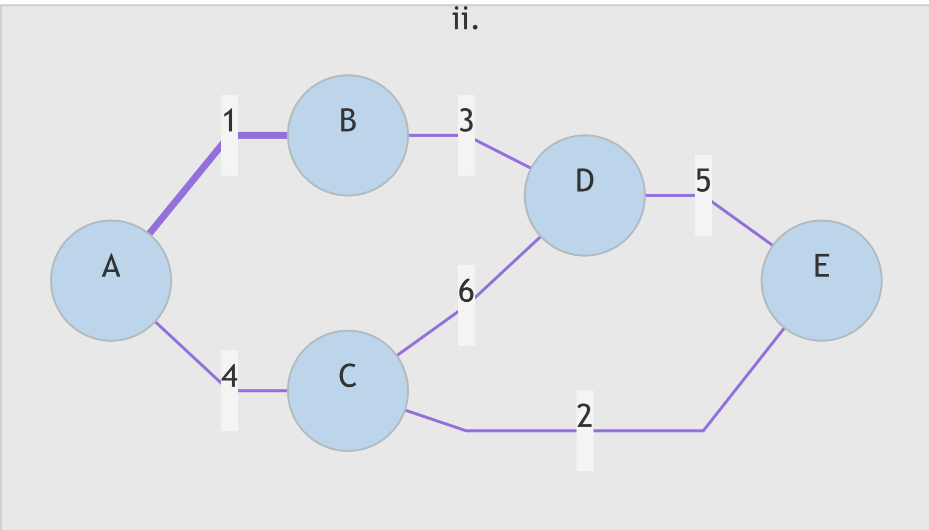
Runtime

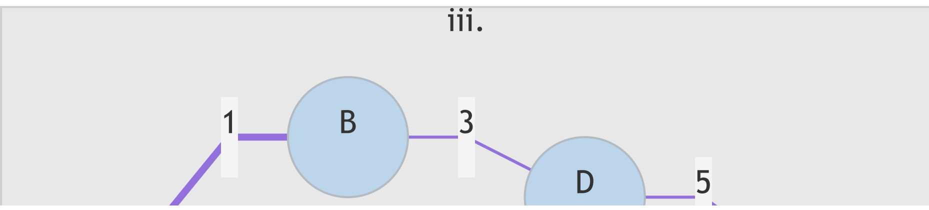
If implemented normally: $T(n) \in \mathcal{O}(|E| * |V| + |E| * log(|E|))$ (second part to sort)
If implemented with an improved union-find DS: $T(n) \in \mathcal{O}(|V| * log(|V|) + |E| * log(|E|))$ (second part to sort)

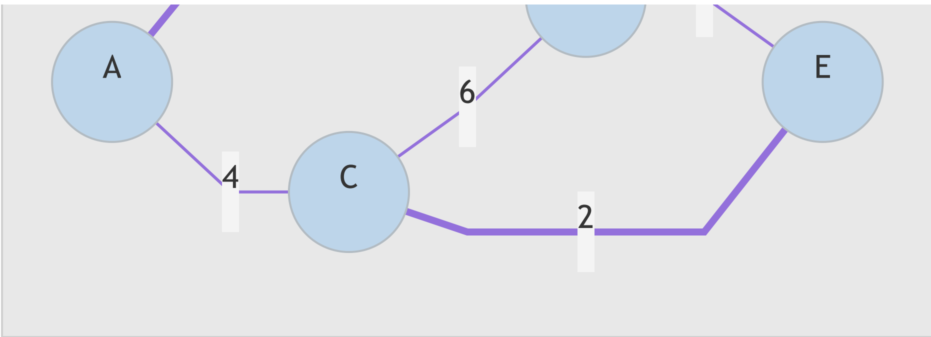
Example

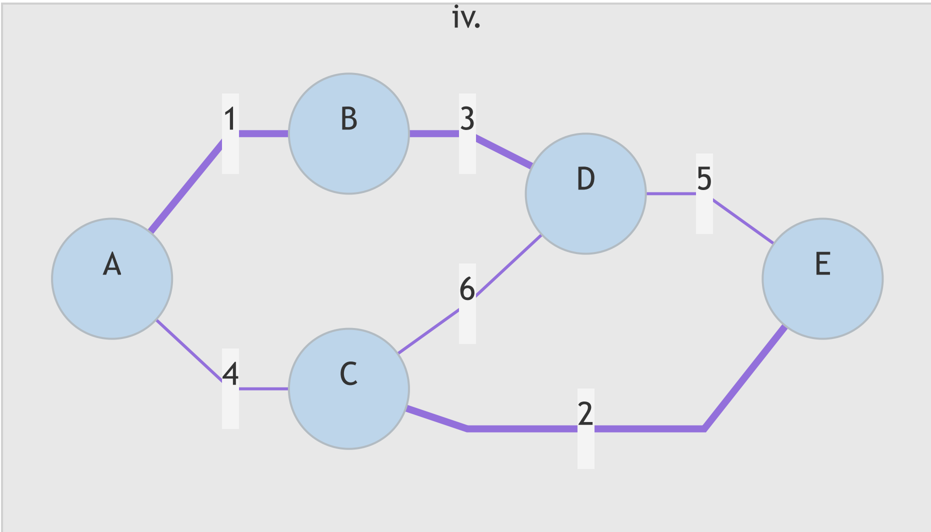
Add edges one by one following weight-order. If adding an edge would form a cycle, skip it.

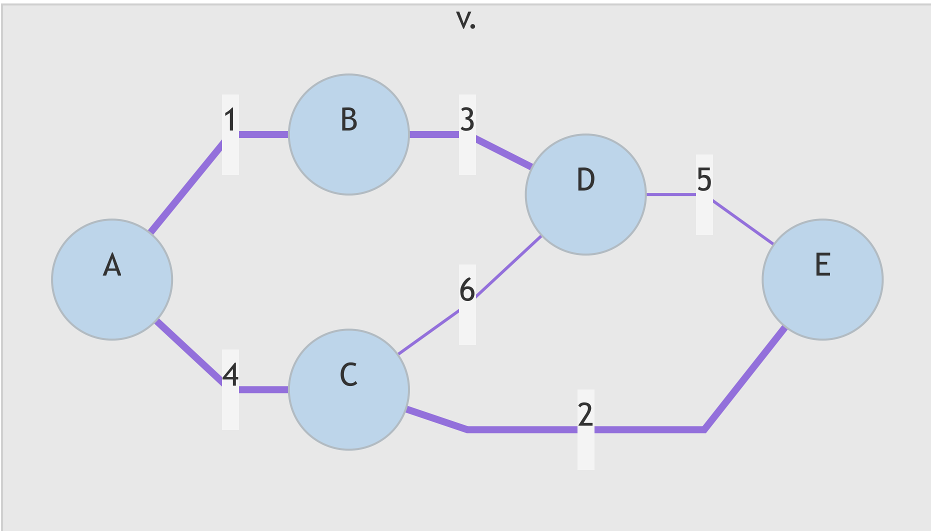












Floyd-Warshall

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.

```
FloydWarshall(G):
    for (v in V):
        d[0][v][v] = 0 // layer 0, row v, column v
    for ((v, u) in E):
        d[0][v][u] = w(v, u)
    else: // if u, v isn't in E
        d[0][v][u] = infinity
    for (i = 1, ..., |V|):
        for (u = 1, ..., |V|):
            for (v = 1, ..., |V|):
                d[i][u][v] = min(d[i-1][u][v], d[i-1][u][i] + d[i-1][i][v])
    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.

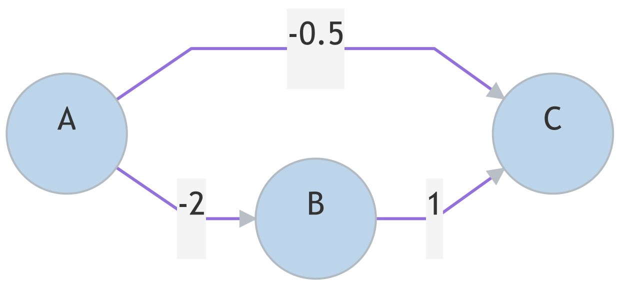
Runtime

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

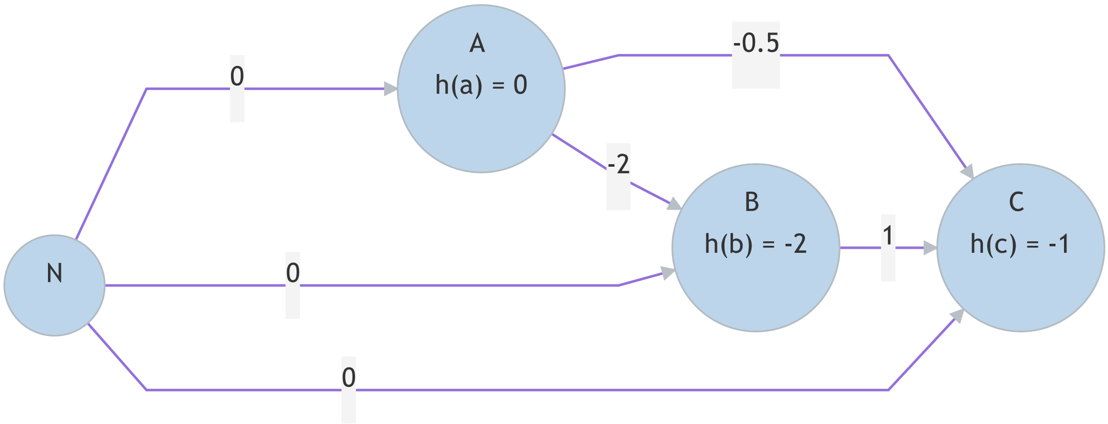
Johnson

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.

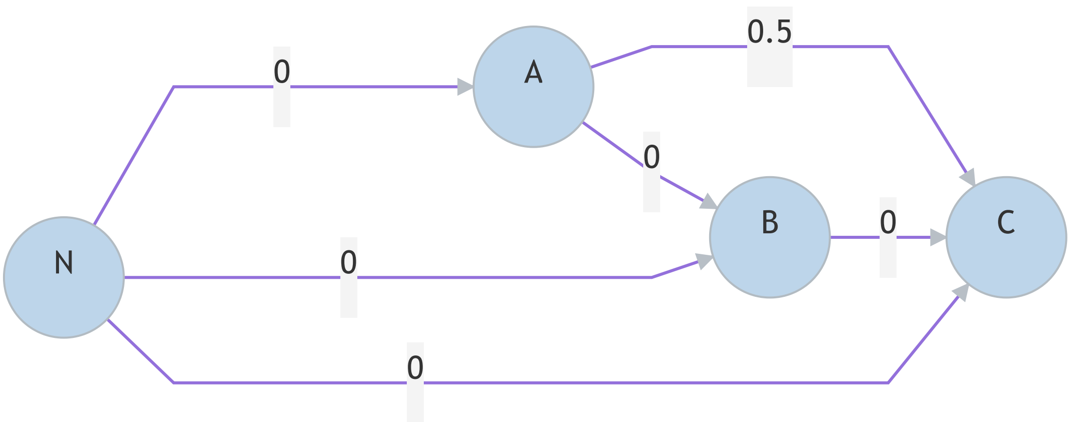
Example



- 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 $w^*(u, v) = w(u, v) + (h(u) - h())$



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