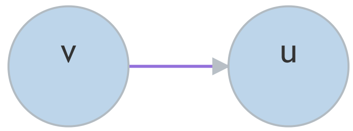
Graph theory

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



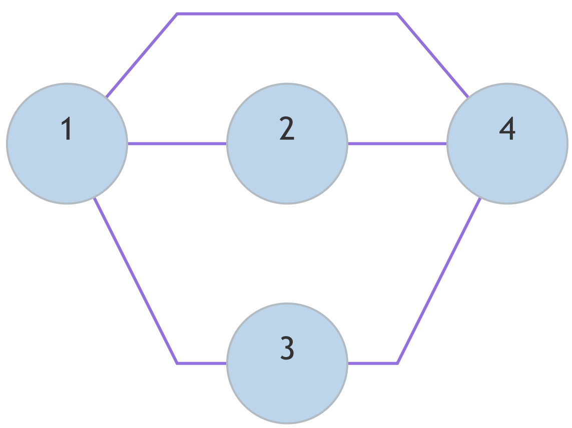
- **deg$_{\text{in}}$(v)**: number of incoming edges into v
- **deg$_{\text{out}}$(v)**: number of outgoing edges into v

Graph Representation

Adjacency matrix:

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

Graph:

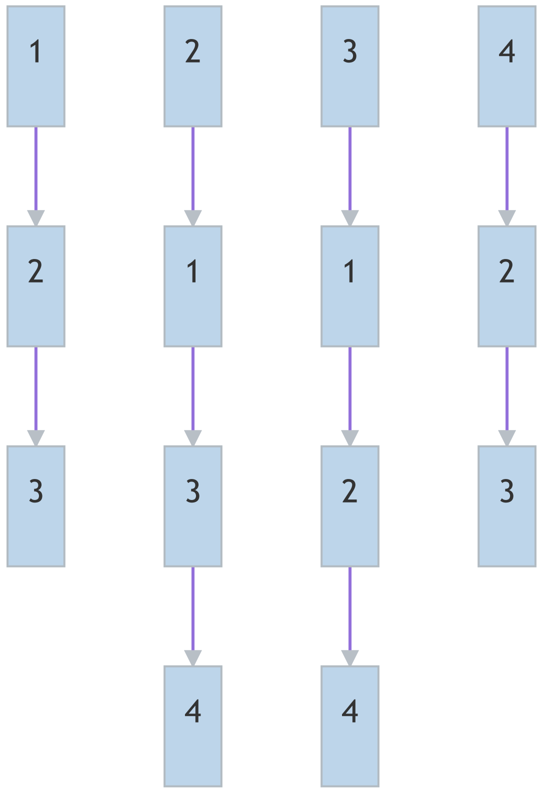


Matrix:

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

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	$\mathcal{O}(\|V\| * \|E\|)$	$\mathcal{O}(\|V\| + \|E\|)$

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

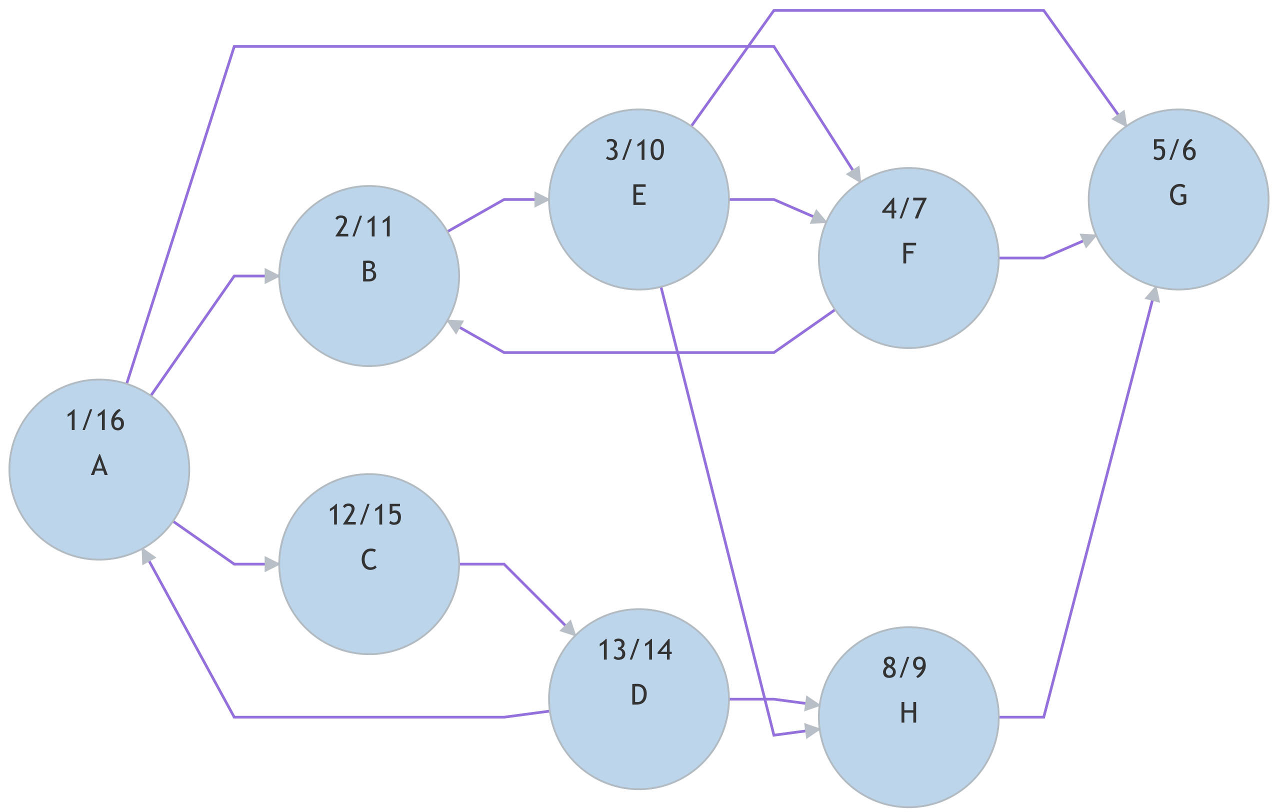
```
DFS-Visit(v):
    t = 0
    pre[v] = t++
    marked[v] = true
    for ((u, v) in E not marked)
        DFS_Visit(u)
    post[u] = t++
```

Runtime

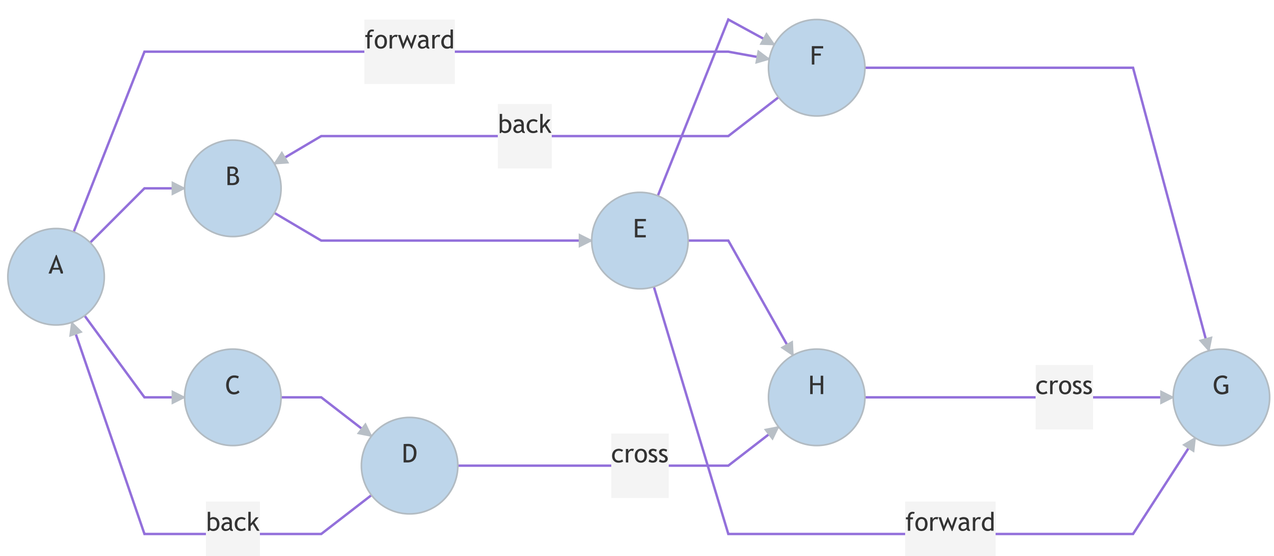
Operations	$T(n) \in \Theta(\lvert E\rvert + \lvert V\rvert)$
Memory	$T(n) \in \Theta(\lvert V\rvert)$

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 $post(u) > post(v)$	**forward** or simply no name, since it is part of the tree
$pre(u) < pre(w)$ 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):
        DFS-Visit(v)
```

```
DFS-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	$T(n) \in \Theta(\lvert E\rvert + \lvert V\rvert)$
Memory	$T(n) \in \Theta(\lvert V\rvert)$

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

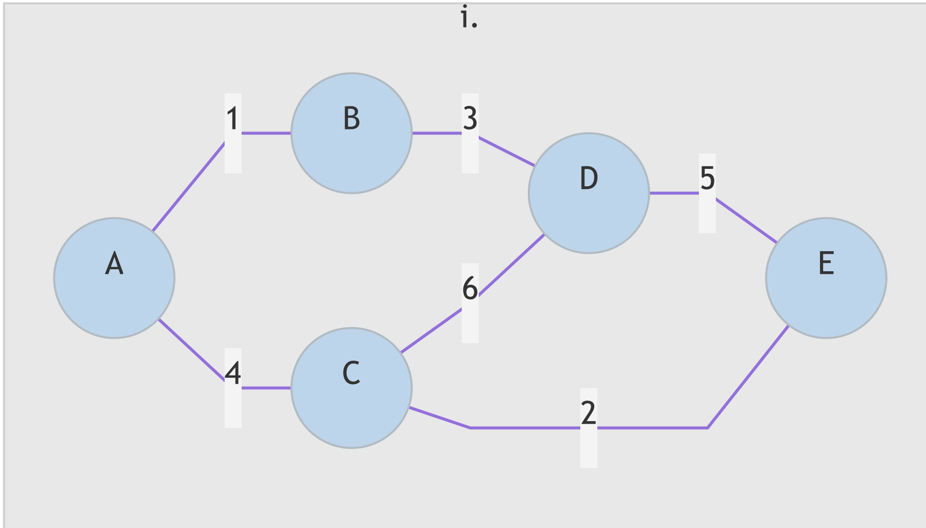
```
Boruvka(G):
    F = new Set() // Initialize a new forest with every vertex being a tree and 0 edges
    while (F not SpanningTree): // check that ZHKs of F > 1
        ZHKs of F = (S1, ..., Sk)
        minEdges of S1, ..., Sk = (e1, ..., ek)
        F = F U (e1, ..., ek)
    return F
```

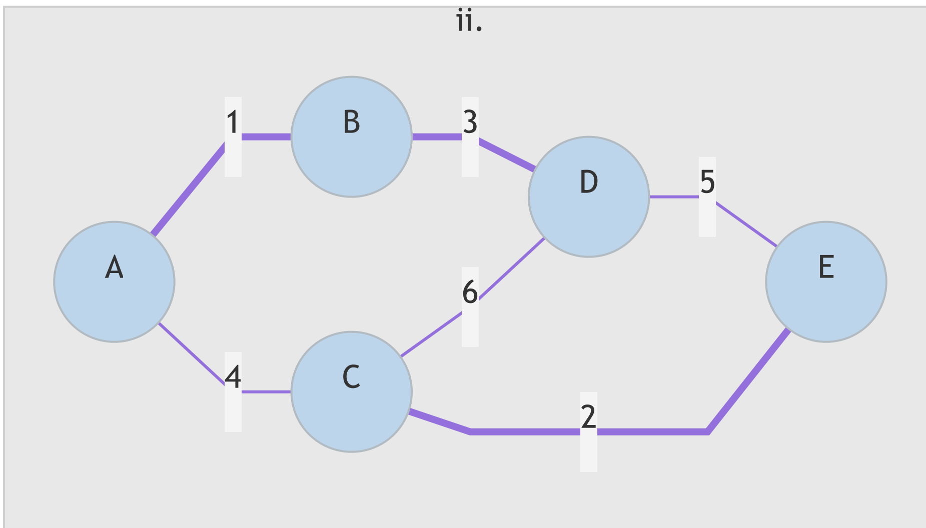
First choose the minimal edge for every vertex and add them to the new graph. Then repeat for every ZHK (vertices connected with edges) until you have a MST (until there is only 1 ZHK).

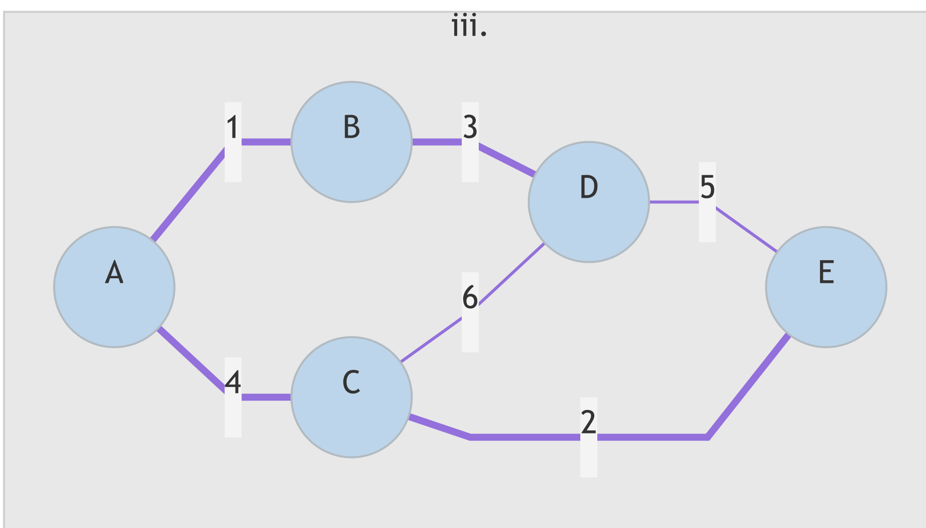
Runtime

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

Example


i.
A
B
C
D
E
1
3
4
6
5
2


ii.
A
B
C
D
E
1
3
4
6
5
2


iii.
A
B
C
D
E
1
3
4
6
5
2

Prim

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

Pseudocode

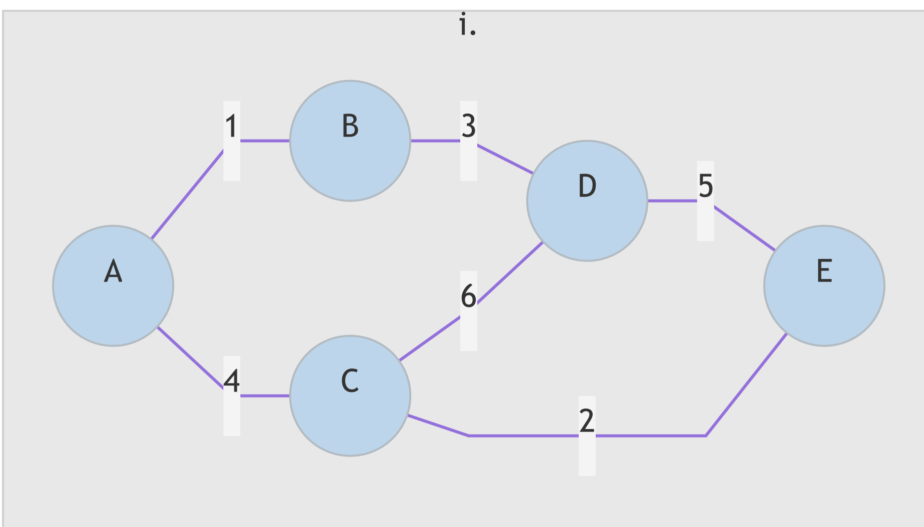
```
Prim(G, s):
    MST = new Set()
    H = new Heap(V, infinity)
    for (v in V):
        d[v] = infinity
    d[s] = 0
    decreaseKey(H, s, 0)
    while (!H.isEmpty()):
        v = extractMin(H)
        MST.add(v)
        for ((v, u) in E && v != s)
            d[v] = min(d[v], w(v, u))
         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

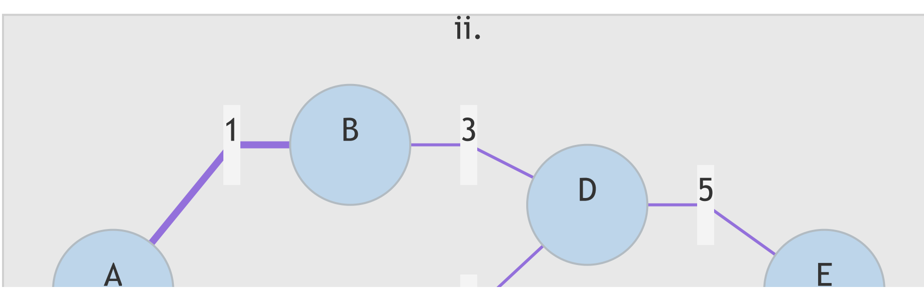
Add the minimal edge adjacent to s. Then take the newly created ZHK and add to it its minimal outgoing edge. Proceed like that until you have a spanning tree (all the vertices are connected).

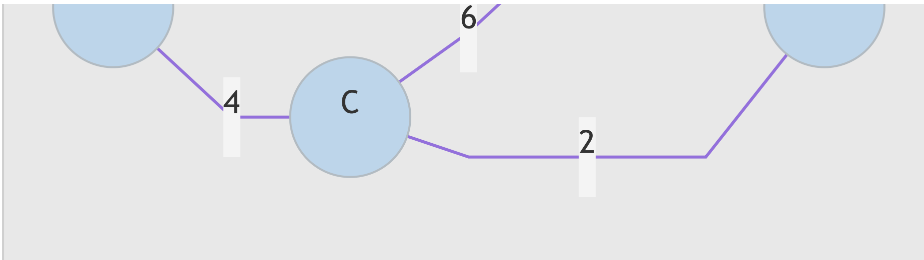
Runtime

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

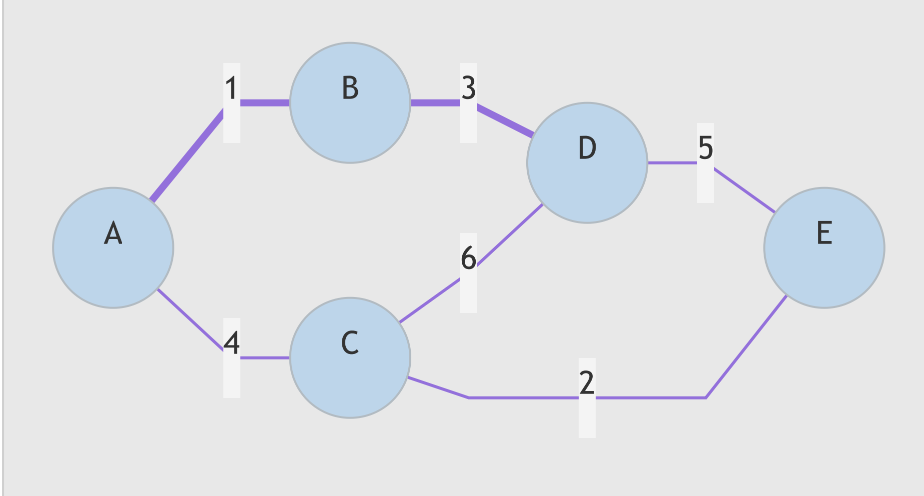
Example



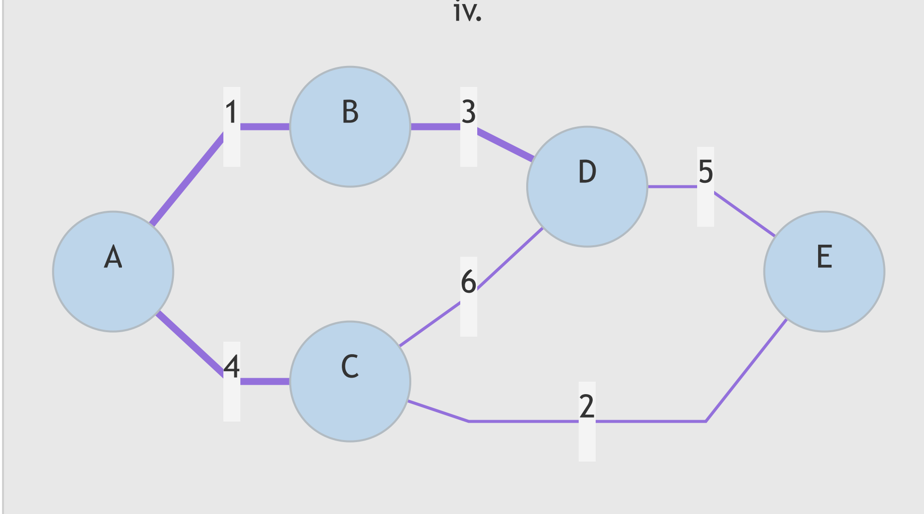




iii.



iv.



v.