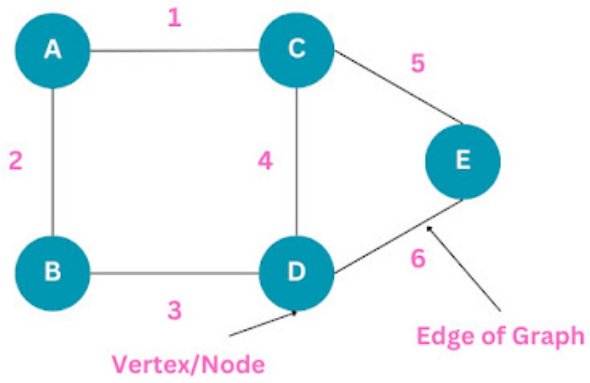
Graphs

What is graph?

A graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. The graph is denoted by **G (V, E)**.



Adjacent Vertex: Vertices with common edge (E)

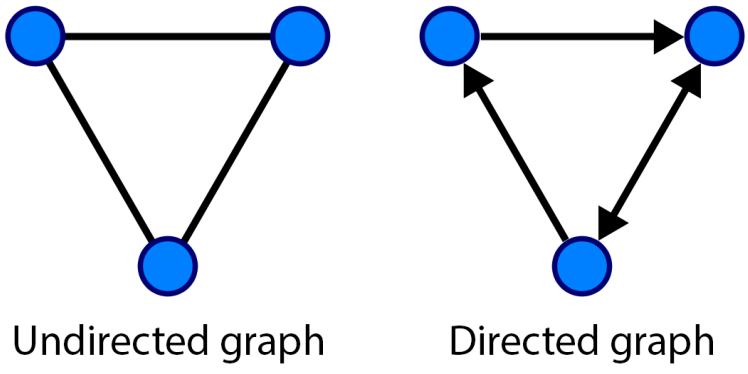
Adjacent Edges: Edges with common vertex (V)

Order of Graph: |V| = number of vertices

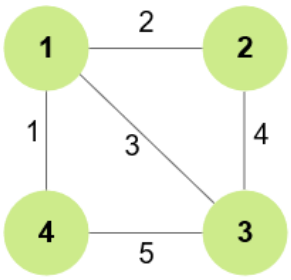
Size of Graph: |E| = number of edges

Major Types:

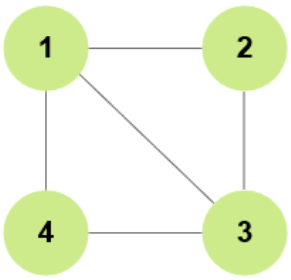
1. Directed Graph and Un-directed Graphs



2. Weighted and Un-weighted Graphs

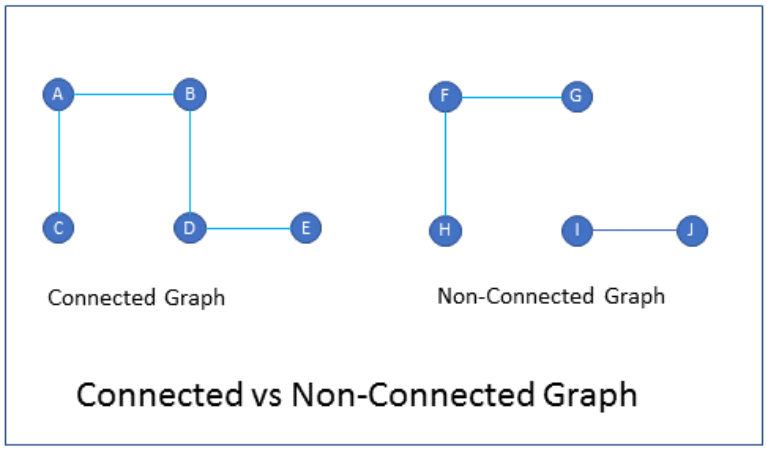


Weighted Graph

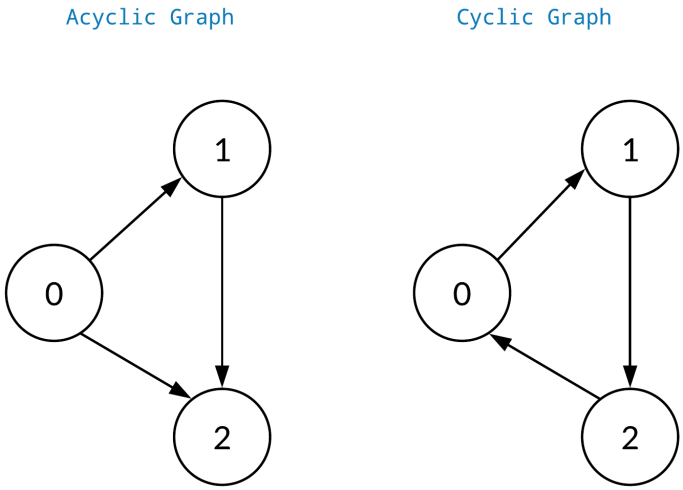


Unweighted Graph

3. Connected and Non-connected Graphs

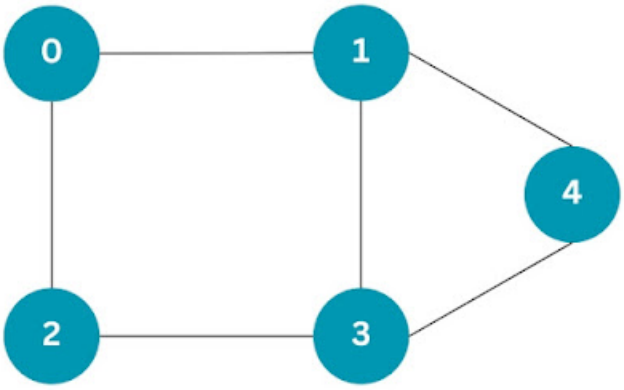


4. Cyclic and Acyclic Graphs



Representation of Graph:

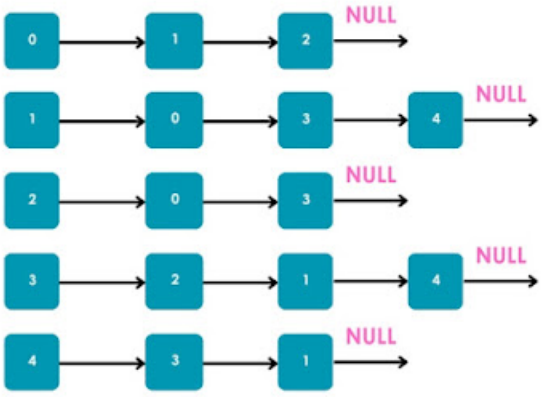
Representation of Graph:



1. Using Adjancey Matrix

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

2. Using Adjancey List

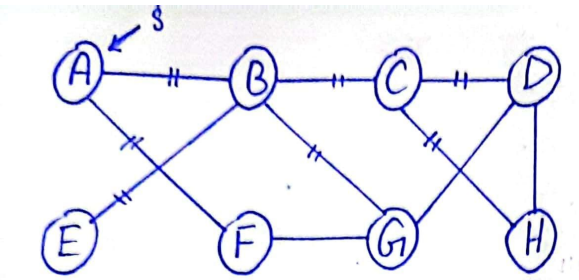


Traversals in Graph:

1. Breadth-First Search: (BFS)

```
BFS(G, s)
 1 for each vertex u ∈ G.V − {s}
 2     u.color = WHITE
 3     u.d = ∞
 4     u.π = NIL
```

```
 5 s.color = GRAY
 6 s.d = 0
 7 s.π = NIL
 8 Q = Ø
 9 ENQUEUE(Q, s)
10 while Q ≠ Ø
11     u = DEQUEUE(Q)
12     for each vertex v in G.Adj[u]      // search the neighbors of u
13         if v.color == WHITE            // is v being discovered now?
14             v.color = GRAY
15             v.d = u.d + 1
16             v.π = u
17             ENQUEUEd(Q, v)             // v is now on the frontier
18     u.color = BLACK                    // u is now behind the frontier
```



	π	d	Color
A	ϕ	0	~~Black~~
B	~~A~~	~~ϕ~~ 1	~~Black~~
C	~~B~~	~~ϕ~~ 2	~~Black~~
D	~~C~~	~~ϕ~~ 3	~~Black~~
E	~~B~~	~~ϕ~~ 2	~~Black~~
F	~~A~~	~~ϕ~~ 1	~~Black~~
G	~~B~~	~~ϕ~~ 2	~~Black~~
H	~~C~~	~~ϕ~~ 3	~~Black~~

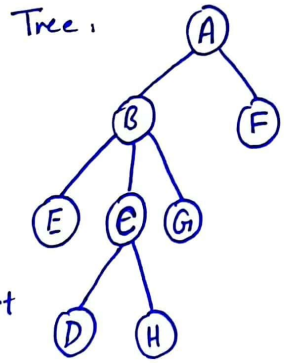
Queue:

~~A~~ ~~B~~ ~~F~~ ~~C~~ ~~G~~ ~~E~~ ~~D~~ ~~H~~

u: ~~A~~ ~~B~~ ~~F~~ ~~C~~ ~~G~~ ~~E~~ ~~D~~ ~~H~~

v: ~~B~~ ~~F~~ ~~C~~ ~~G~~ ~~E~~ ~~A~~ ~~G~~ ~~D~~ ~~H~~ ~~B~~ ~~D~~ ~~G~~ ~~F~~

Tree:



RAM Model:

Individual Cost	Repitition	Total Cost
C_1	$\|V\|-1+1$	$\|V\|$
C_2	$\|V\|-1$	$\|V\|$
C_3	$\|V\|-1$	$\|V\|$
C_4	$\|V\|-1$	$\|V\|$
C_5	1	C_5
C_6	1	C_6
C_7	1	C_7
C_8	1	C_8
C_9	$O(\log V)$	$O(\log V)$
C_{10}	$\|V\|+1$	$\|V\|+1$
C_{11}	$V\ O(\log V)$	$VO(\log V)$
C_{12}	$O(V+E)$	$O(V+E)$
C_{13}	$O(V+E)$	$O(V+E)$
C_{14}	$O(V+E)$	$O(V+E)$
C_{15}	$O(V+E)$	$O(V+E)$
C_{16}	$O(V+E)$	$O(V+E)$
C_{17}	$O(V+E)$	$O(V+E)$
C_{18}	V	$\|V\|$

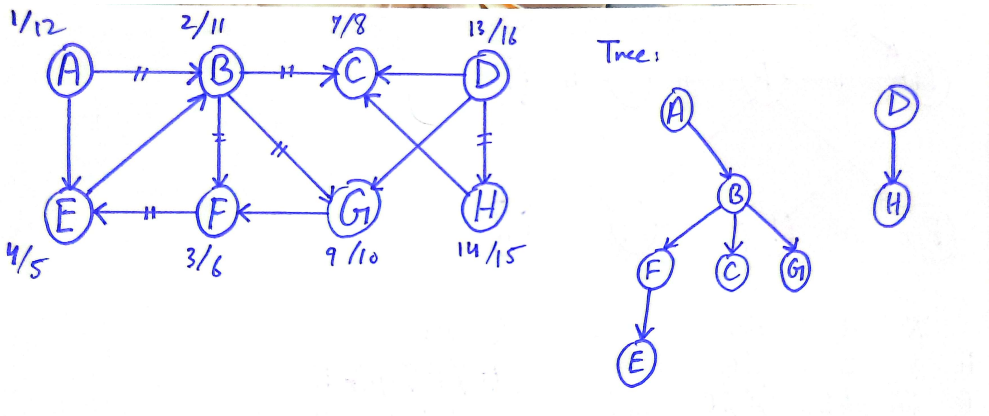
$T(n) = O(V+E)$

2. Depth-First Search: (DFS)

```
DFS(G)
 1 for each vertex u ∈ G.V
 2     u.color = WHITE
 3     u.π = NIL
 4 time = 0
 5 for each vertex u ∈ G.V
 6     if u.color == WHITE
 7         DFS-VISIT(G, u)

DFS-VISIT(G, u)
 1 time = time + 1                  // white vertex u has just been discovered
 2 u.d = time
 3 u.color = GRAY
 4 for each vertex v in G.Adj[u]// explore each edge (u, v)
 5     if v.color == WHITE
 6         v.π = u
```

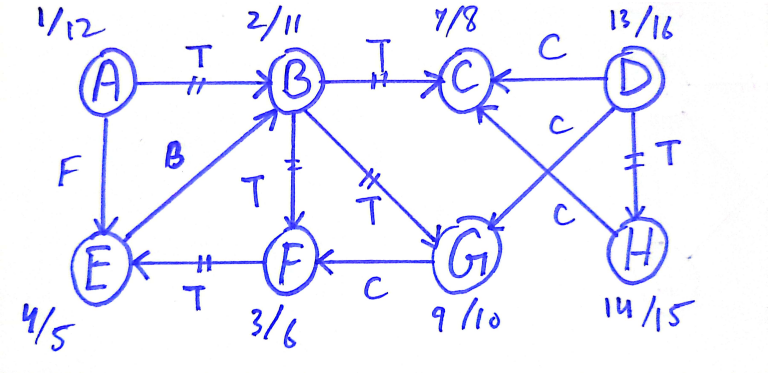
```
 7         DFS-VISIT(G, v)
 8 time = time + 1
 9 u.f = time
10 u.color = BLACK                  // blacken u; it is finished
```



Edges Categorization:

1. Tree
2. Forward
3. Backward

4. Cross

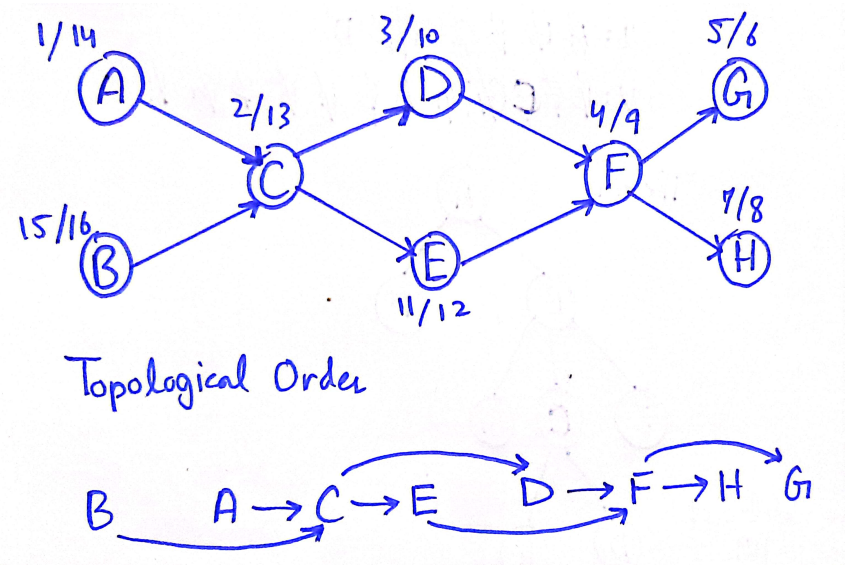


Directed Acyclic Graph (DAG)

A **Directed Acyclic Graph (DAG)** is a graph that satisfies the following properties:

- **Directed**: Each edge has a direction, meaning it goes from one vertex to another in a specific way.
- **Acyclic**: There are no cycles, meaning it is impossible to start from a vertex and return to the same vertex by following the directed edges.

TOPOLOGICAL-SORT(G)
1 call DFS(G) to compute finish times $v.f$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices



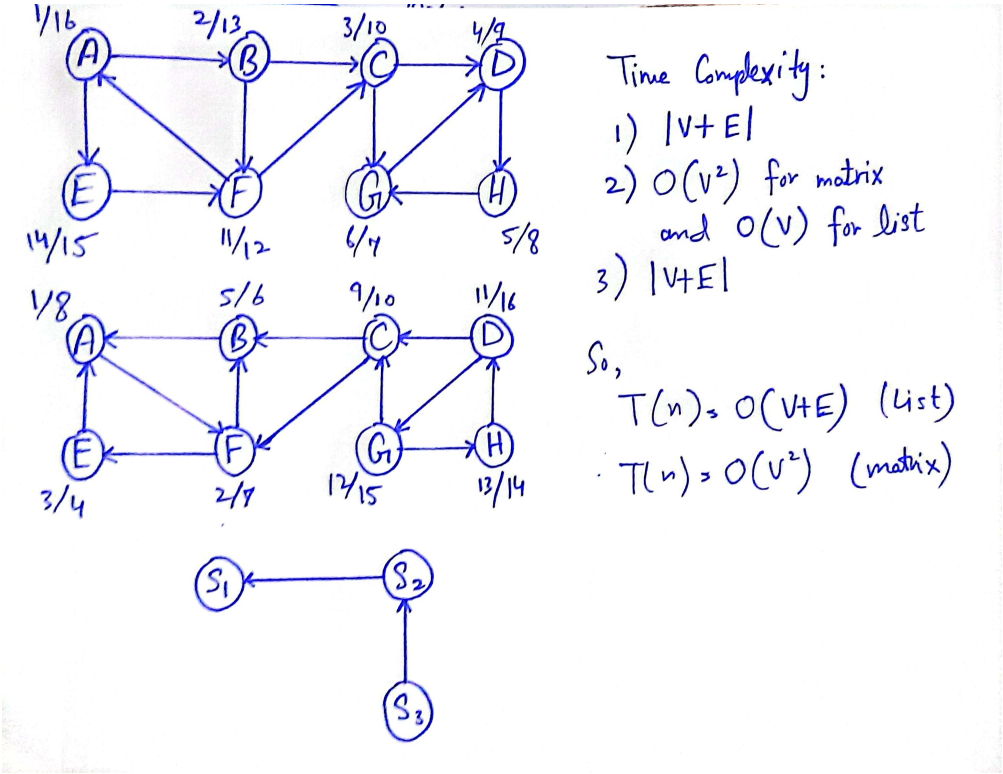
Key Characteristics of a DAG

1. **No Cycles**: Since the graph is acyclic, no vertex can be visited twice in a path that returns to the starting vertex.
2. **Topological Ordering**: Sort by finish time. A DAG allows a **topological sort**, which is a linear ordering of its vertices such that for every directed edge , vertex comes before in the ordering.

Strongly Connected Components (Kosaraju's Algorithm)

A **Strongly Connected Component (SCC)** of a directed graph G = (V, E) is a **maximal subgraph** in which any two vertices are reachable from each other.

STRONGLY-CONNECTED-COMPONENTS(G)
1 call DFS(G) to compute finish times $u.f$ for each vertex u
2 create G^{T}
3 call DFS(G^{T}), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component



Minimum Spanning Tree

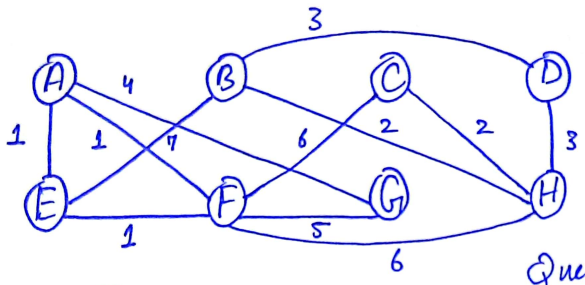
A **minimum spanning tree (MST)** is defined as a **spanning tree** that has the minimum weight among all the possible spanning trees.

A **spanning tree** is defined as a tree-like subgraph of a connected, undirected graph that includes all the vertices of the graph. Or, to say in Layman's words, it is a subset of the edges of the graph that forms a tree (**acyclic**) where every node of the graph is a part of the tree.

The minimum spanning tree has all the properties of a spanning tree with an added constraint of having the minimum possible weights among all possible spanning trees. Like a spanning tree, there can also be many possible MSTs for a graph.

1. PRIMS Algorithm:

```
MST-PRIM(G, w, r)
 1 for each vertex u ∈ G.V
 2     u.key = ∞
 3     u.π = NIL
 4 r.key = 0
 5 Q = Ø
 6 for each vertex u ∈ G.V
 7     INSERT(Q, u)
 8 while Q ≠ Ø
 9     u = EXTRACT-MIN(Q)  // add u to the tree
10     for each vertex v in G.Adj[u]  // update keys of u's non-tree neighbors
11         if v ∈ Q and w(u, v) < v.key
12             v.π = u
13             v.key = w(u, v)
14             DECREASE-KEY(Q, v, w(u, v))
```



	π	key
A	ϕ	0
B	ϕ E H	ϕ 5 2
C	ϕ F	ϕ 6
D	ϕ H	ϕ 3
E	ϕ A	ϕ 1
F	ϕ A	ϕ 1
G	ϕ A	ϕ 4
H	ϕ F C	ϕ 5 2

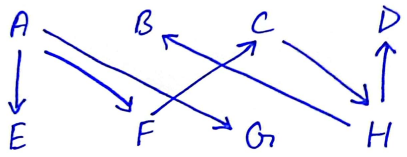
Queue :

A B C D E F G H

u : A E F G C H B D

v : E F G A B F E A C G H F A F H B C D E H D B H

MST:



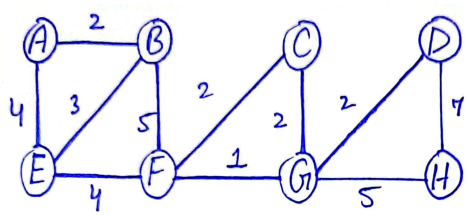
	Cost
1.	$\|V\|+1$
2.	$\|V\|$
3.	$\|V\|$
4.	1
5.	1
6.	$\|V\|+1$
7.	$\|V\| \log V$
8.	$\|V\|+1$
9.	$\|V\| \log V$
10.	$\|E\|$
11.	$\|E\|$
12.	$\|E\|$
13.	$\|E\|$
14.	$\|E\| \log V$

$T(n) = O(V \log V + E \log V)$

$= O(E \log V)$

2. KRUSKAL Algorithm:

```
MST-KRUSKAL(G, w)
 1 A = Ø
 2 for each vertex v ∈ G.V
 3     MAKE-SET(v)
 4 create a single list of the edges in G.E
 5 sort the list of edges into monotonically increasing order by weight
   w
 6 for each edge (u, v) taken from the sorted list in order
 7     if FIND-SET(u) ≠ FIND-SET(v)
 8         A = A ∪ {(u, v)}
 9         UNION(u, v)
10 return A
```

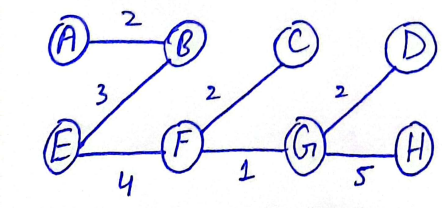


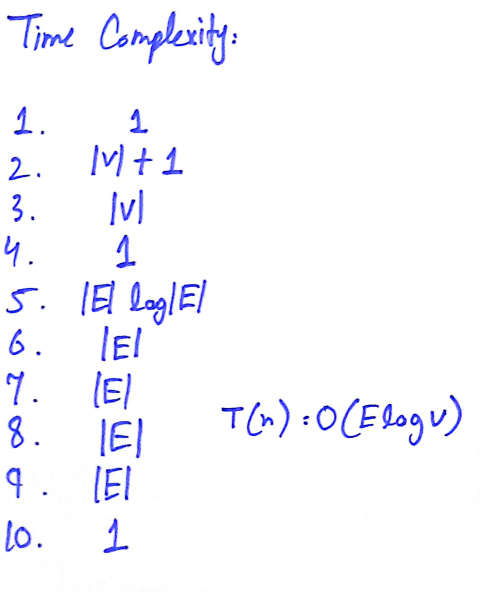
{A} {A B} {A B E}
{B} {A B C D E F G}
{C} {C F G} {C D F G}
{D}
{E}
{F} {F G}
{G}
{H}

{A B C D E F G H}

Final Set

FG — 1 F ≠ G ✓
FC — 2 F ≠ C ✓
GD — 2 G ≠ D ✓
CG — 2 C = G ✗
AB — 2 A ≠ B ✓
EB — 3 E ≠ B ✓
AE — 4 A = E ✗
EF — 4 A ≠ C ✓
GH — 5 A ≠ H ✓
BF — 5 A = A ✗
DH — 7 A = A ✗





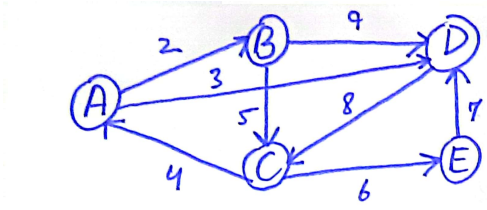
Single Source Shortest Path Finding Algorithms

1. DIJKSTRA Algorithm:

```
INITIALIZE-SINGLE-SOURCE(G, s)
1 for each vertex v ∈ G.V
2     v.d = ∞
3     v.π = NIL
4 s.d = 0
```

```
RELAX(u, v, w)
1 if v.d > u.d + w(u, v)
2     v.d = u.d + w(u, v)
3     v.π = u
```

```
DIJKSTRA(G, w, s)
 1 INITIALIZE-SINGLE-SOURCE(G, s)
 2 S = Ø
 3 Q = Ø
 4 for each vertex u ∈ G.V
 5     INSERT(Q, u)
 6 while Q ≠ Ø
 7     u = EXTRACT-MIN(Q)
 8     S = S ∪ {u}
 9     for each vertex v in G.Adj[u]
10         RELAX(u, v, w)
11         if the call of RELAX decreased v.d
12             DECREASE-KEY(Q, v, v.d)
```

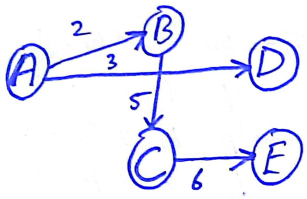


	π	d
A	ϕ	0
B	~~A~~	~~∞~~ 2
C	~~B~~	~~∞~~ 7
D	~~A~~	~~∞~~ 3
E	~~C~~	~~∞~~ 13

Queue : ~~A~~ ~~B~~ ~~C~~ ~~D~~ ~~E~~

u : ~~A~~ ~~B~~ ~~C~~ ~~D~~ ~~E~~

v : ~~B~~ ~~D~~ ~~C~~ ~~D~~ ~~A~~ ~~E~~ ~~C~~ ~~D~~

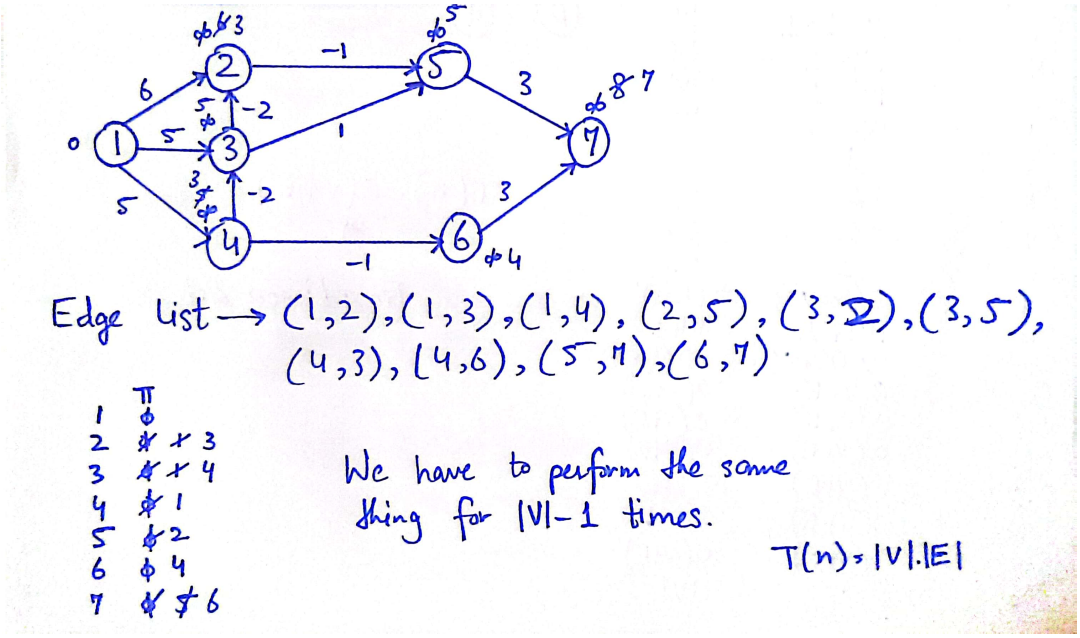


Time Complexity:

$O((|V|+|E|)\log(|V|))$

2. BELLMAN-FORD Algorithm

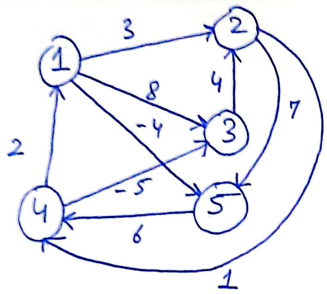
```
BELLMAN-FORD(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 for i = 1 to |G.V| − 1
3     for each edge (u, v) ∈ G.E
4         RELAX(u, v, w)
5 for each edge (u, v) = G.E
6     if v.d > u.d + w(u, v)
7         return FALSE
8 return TRUE
```



All Pairs Shortest Path Finding Algorithm

1. FLOYD-WARSHALL Algorithm:

```
FLOYD-WARSHALL(W, n)
1  D^(0) = W
2  for k = 1 to n
3      let D^(k) = (d_ij^(k)) be a new n × n matrix
4      for i = 1 to n
5          for j = 1 to n
6              d_ij^(k) = min{d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1)}
7  return D^(n)
```



$$D^{(0)} = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \qquad \Pi^{(0)} = \begin{bmatrix} \phi & 1 & 1 & \phi & 1 \\ \phi & \phi & \phi & 2 & 2 \\ \phi & 3 & \phi & \phi & \phi \\ 4 & \phi & 4 & \phi & \phi \\ \phi & \phi & \phi & 5 & \phi \end{bmatrix}$$

$$D^{(1)} = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \qquad \Pi^{(1)} = \begin{bmatrix} \phi & 1 & 1 & \phi & 1 \\ \phi & \phi & \phi & 2 & 2 \\ \phi & 3 & \phi & \phi & \phi \\ 4 & 1 & 4 & \phi & 1 \\ \phi & \phi & \phi & 5 & \phi \end{bmatrix}$$

$i = 2, j = 1$

$\min(d_{ij}, d_{ik} + d_{kj})$

$d_{21} = \min(\infty, \infty + 0) = \infty$

We will do these steps for all vertices.

Time Complexity: $O(|V|^3)$