SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 OF THE UGC ACT, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

Course

CSE318 - Algorithm Design Strategies & Analysis

Topic

Unit-3

Shri B. Srinivasan

Assistant Professor-III, School of Computing

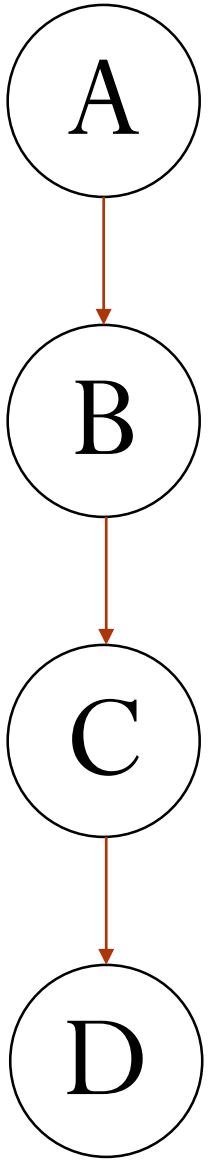
SASTRA Deemed To Be University

Topics

- ✓ Graph – Basic Terminologies
- ✓ Traversal Algorithms
  - ✓ Breadth First Search (BFS)
  - ✓ Depth First Search (DFS)
  - ✓ Topological Sort
- ✓ Minimum Spanning Tree
  - ✓ Prim's Algorithm
  - ✓ Kruskal's Algorithm
- ✓ Shortest Path Algorithms
  - ✓ Bellman-Ford Algorithm
  - ✓ Dijkstra's Algorithm
  - ✓ Floyd-Wharshall Algorithm
- ✓ Flow Network Problem
  - ✓ Ford-Fulkerson Algorithm

Graph – Basic Terminologies

List



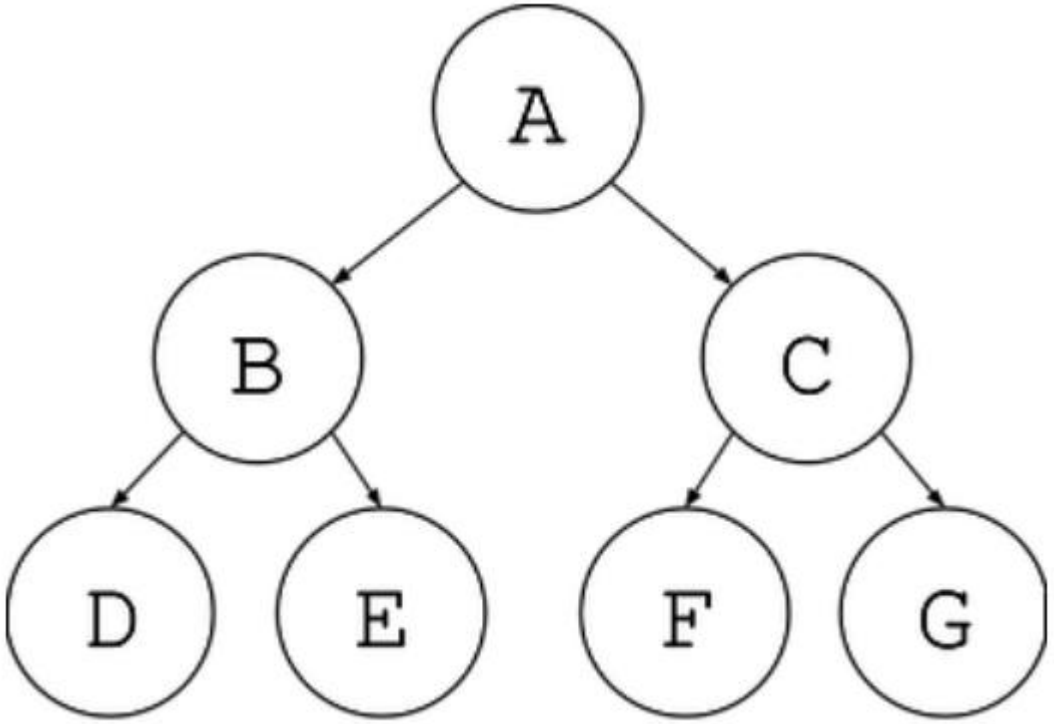
Node May Have

Single Predecessor

&

Single Successor

Tree



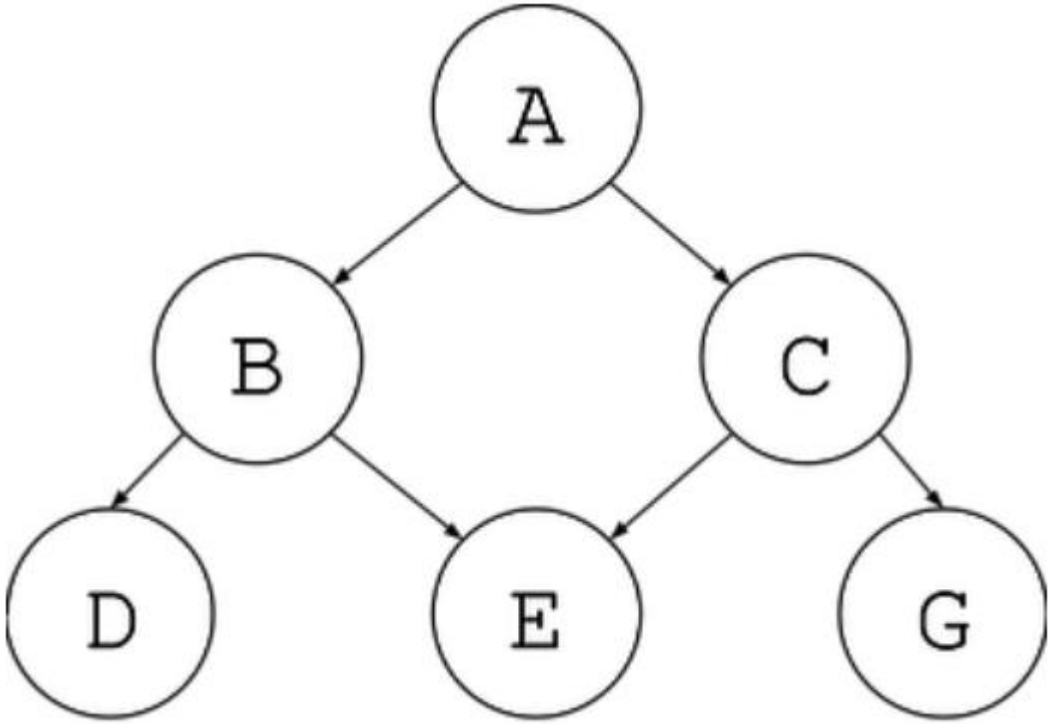
Node May Have

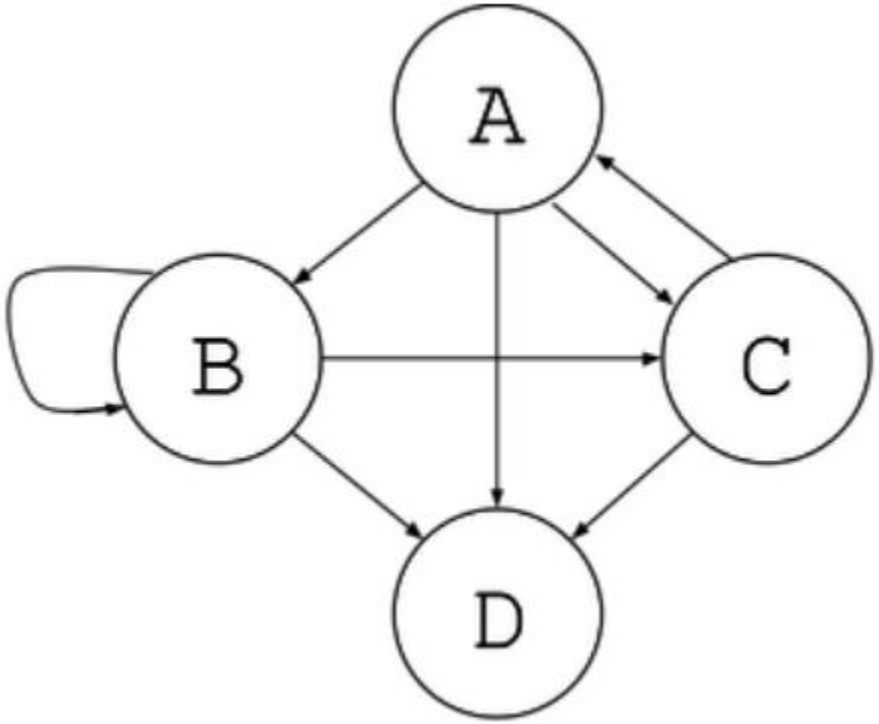
Single Predecessor

&

Multiple Successor

Graph





Node May Have

Multiple Predecessor

&

Multiple Successor

Graph - Definition

A graph is simply a collection of Vertices plus Edges.

A graph G is a pair (V, E) where

- ✓ V is a set of **vertices** (singular "Vertex") or **nodes**
- ✓ E is a set of **edges** that connect vertices
- ✓ A node in a graph – Vertex
- ✓ A line that connect 2 vertices – Edge
- ✓ Each edge is represented as a pair (v1, v2), where v1, v2 are vertices in V

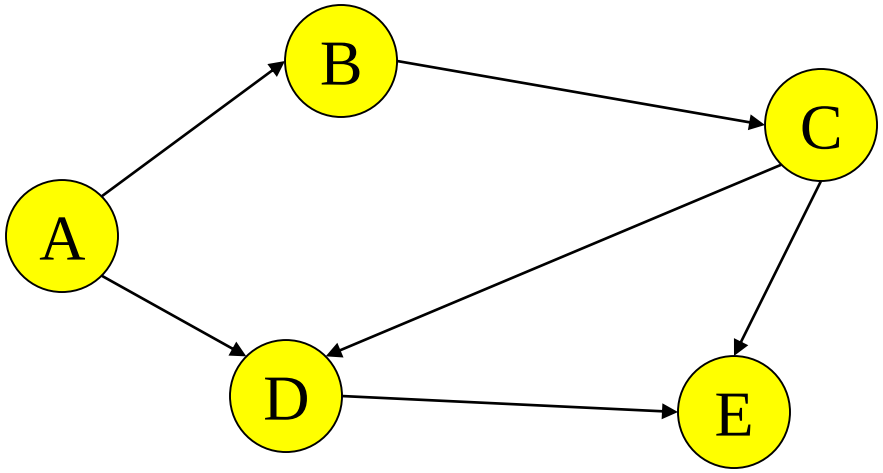
Note: Linked lists and Tree are special cases of graphs

Graph - Example

Here is a graph G = (V, E)

V = {A, B, C, D, E, F}

E = {(A,B), (A,D), (B,C), (C,D), (C,E), (D,E)}

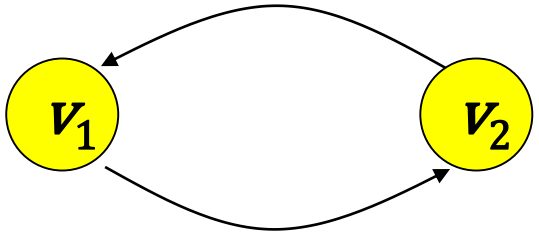


Graph - Directed Graph vs Undirected Graph

Directed Graph (also called as "digraph"):

- ✓ Each edge has a Direction. The edge in digraph is usually called as "ARC".
- ✓ If the order of edge pairs (v1, v2) matters, the graph is directed (also called a digraph):

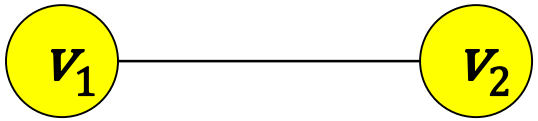
(v1, v2) ≠ (v2, v1)



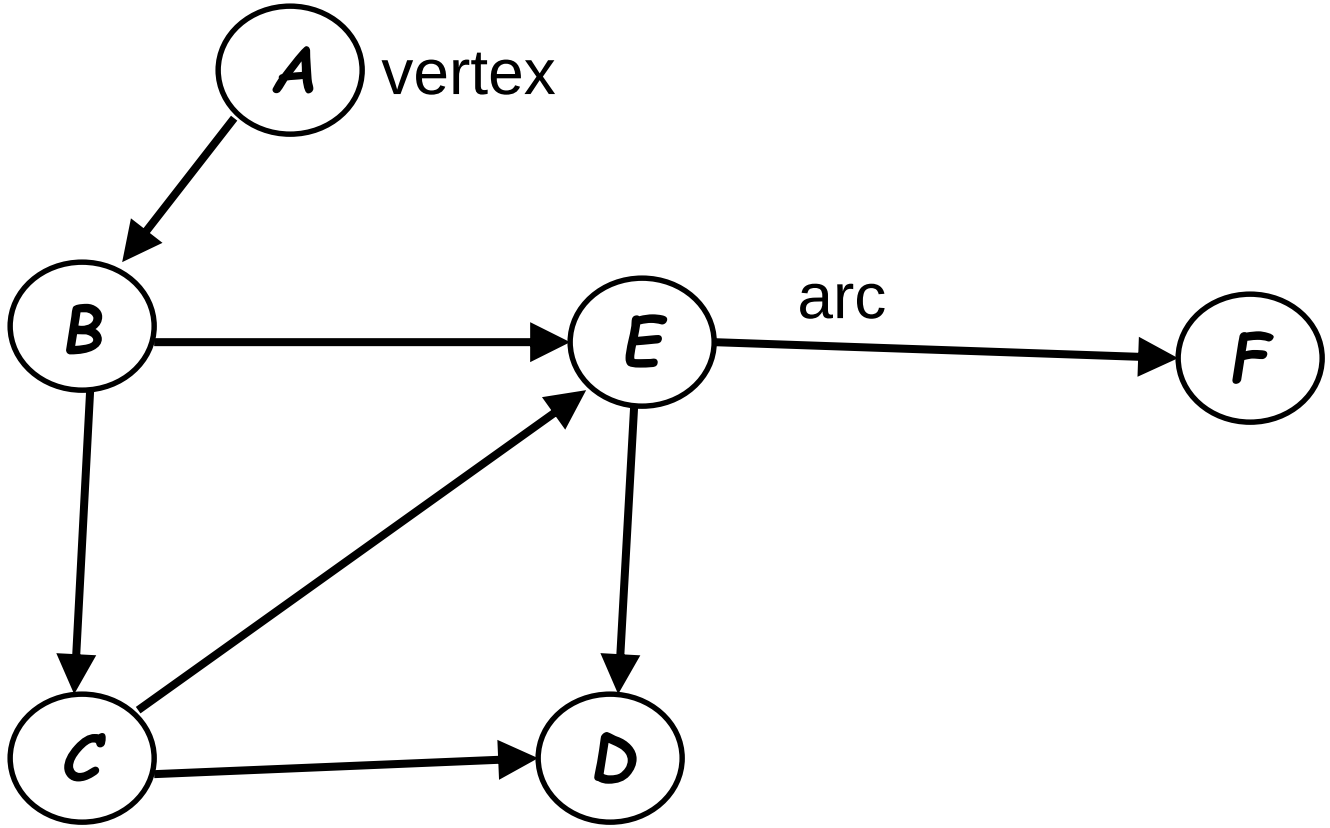
Undirected Graph:

- ✓ The edges does not have any direction.
- ✓ If the order of edge pairs (v1, v2) does not matter, the graph is called an undirected graph:

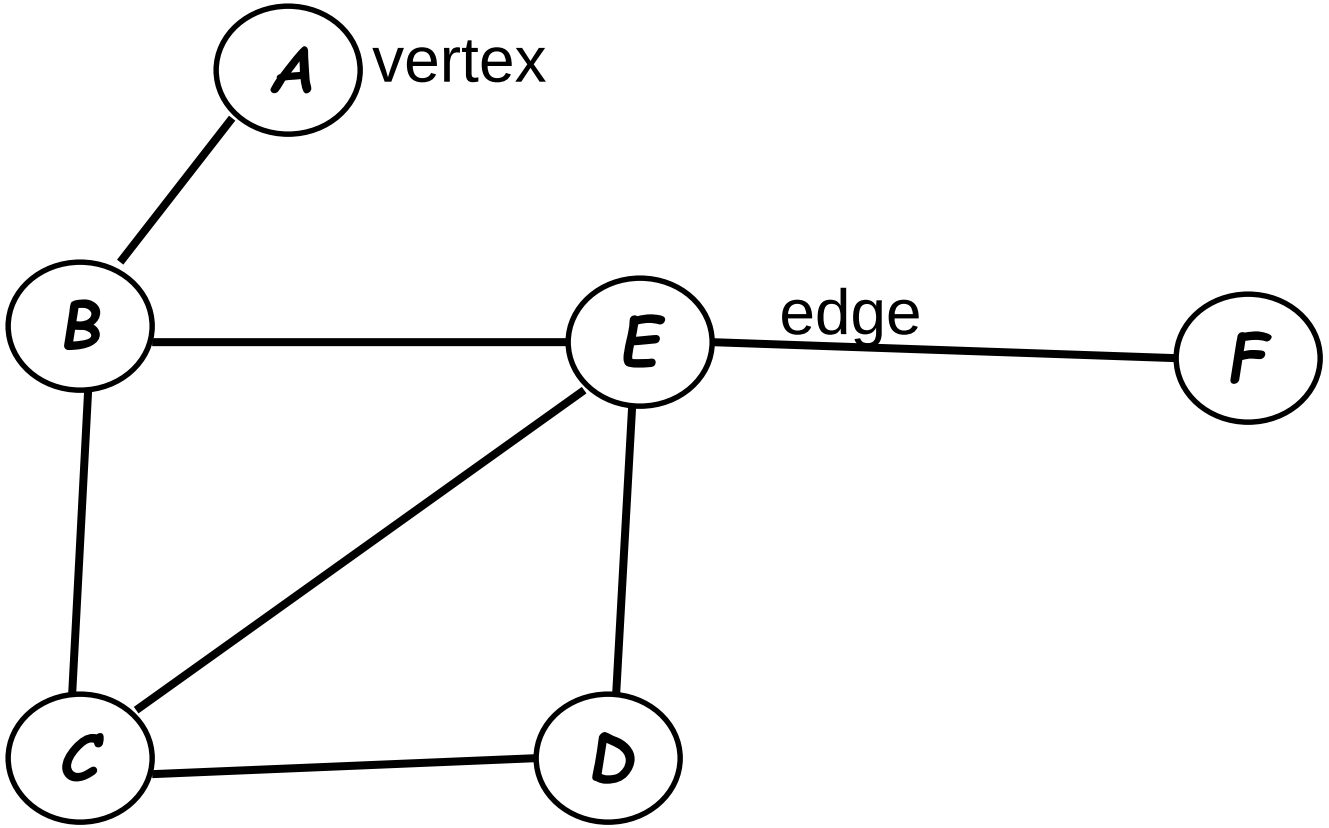
(v1, v2) = (v2, v1)



Digraph - Example



Undirected Graph - Example



Graph – Basic Terminology

Adjacent Vertices (neighbors)

In an undirected graph G:

- ✓ Two vertices u and v are said to be adjacent, if there exist an edge, (u, v) in the graph G.

In a digraph G:

- ✓ Vertex u is said to be adjacent to vertex v AND Vertex v is said to be adjacent from vertex u, if there exist an edge, (u, v) in the graph G.
- ✓ Here, u is initial vertex and v is terminal vertex.

Graph – Basic Terminology

Degree of Vertex

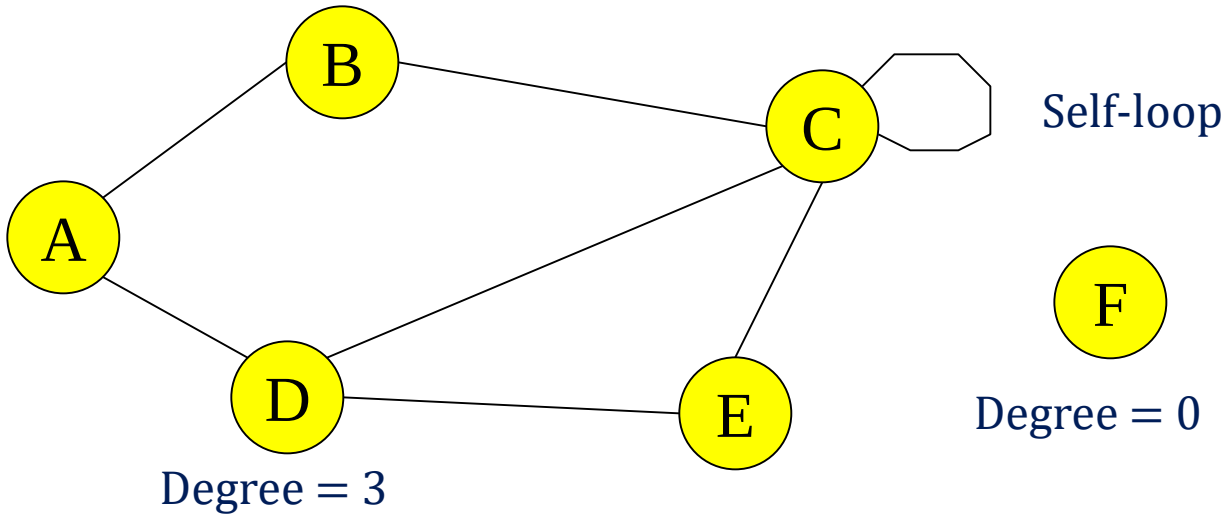
In an undirected graph G:

- ✓ The degree of the vertex is number of edges associated with it.

In a digraph G:

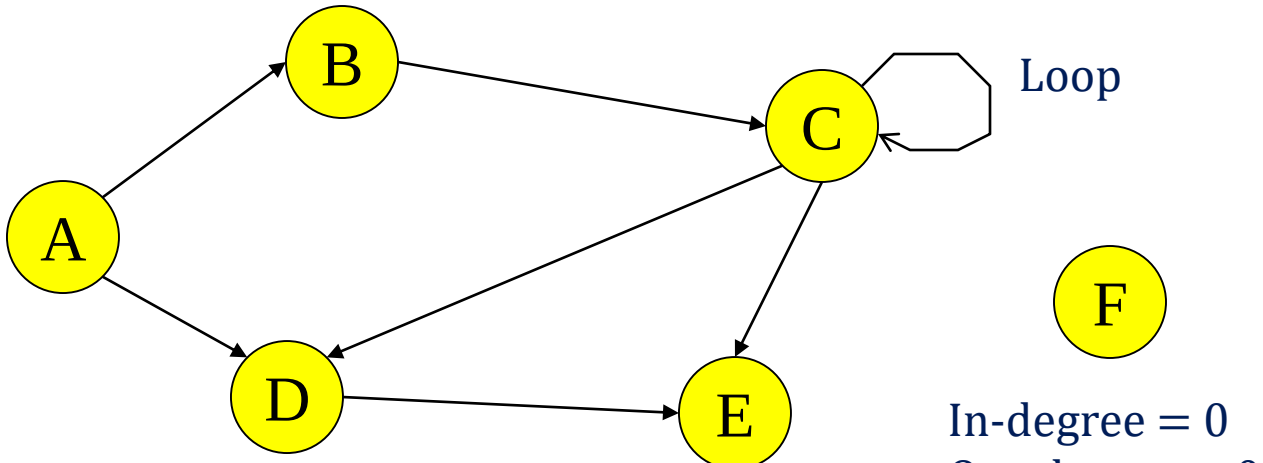
- ✓ The in-degree is the number of edges with the vertex as the terminal vertex.
- ✓ The out-degree is the number of edges with the vertex as the initial vertex

Graph – Basic Terminology



Since an edge (A,B) is there, A and B are said to be adjacent to each other.
But, the vertices A and C are not adjacent vertices.

Graph – Basic Terminology



Since an edge (A,B) is there, A adjacent to B and B adjacent from A.

Graph – Basic Terminology

Path

A path is a sequence of vertices with the property that each vertex in the sequence is adjacent to the vertex next to it.

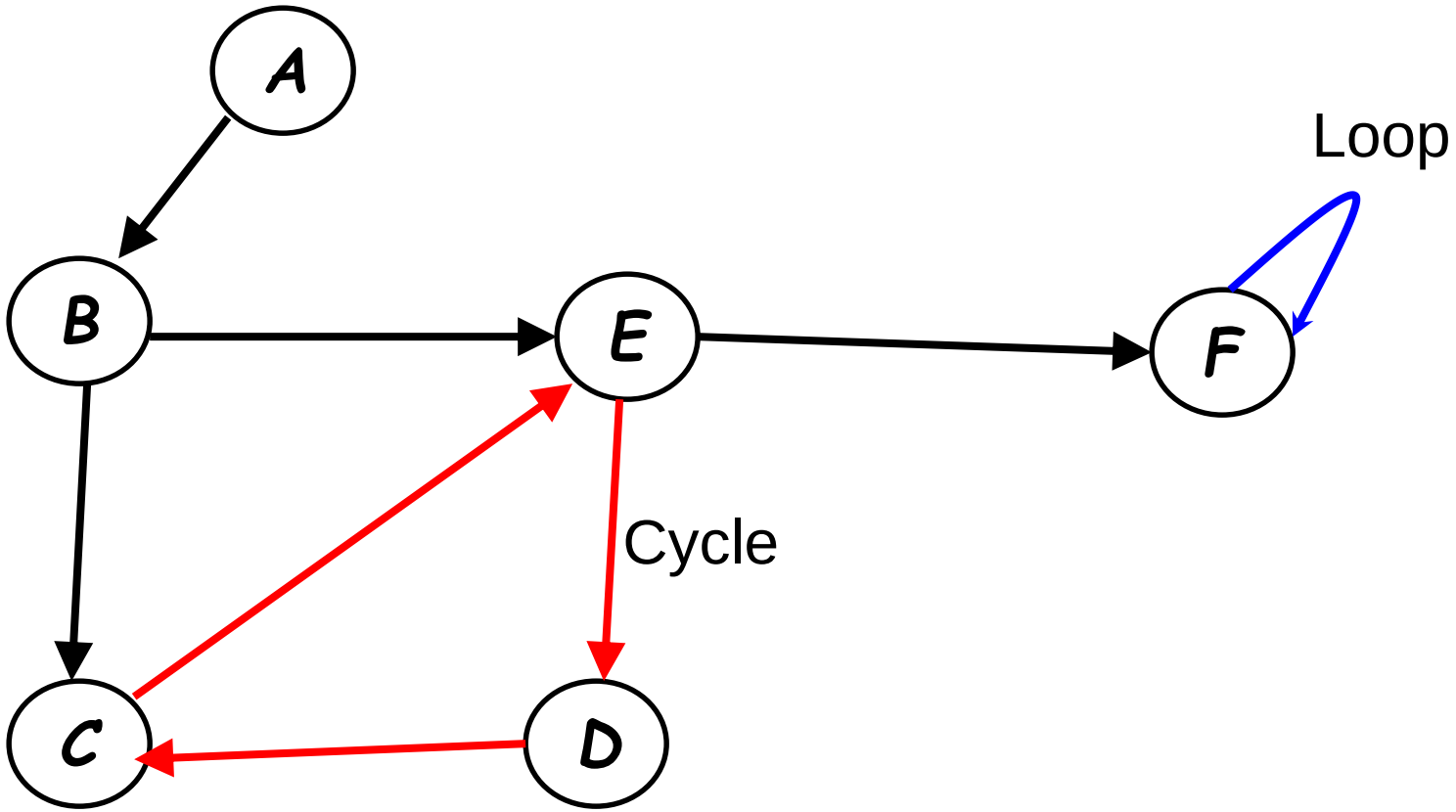
Cycle

A path consisting of at least three vertices that starts and ends with the same vertex. Must follow the proper direction in a digraph

Loop

A special case of a cycle in which a single arc begins and ends with the same vertex

Graph – Basic Terminology



Examples:

Path: ABEDC, ABEF, DCEF, EDC, ABCED, etc.,

Cycle: EDCE

Loop: FF

Graph – Basic Terminology

Connected Vertices

Two vertices are said to be connected if there exist a path between them

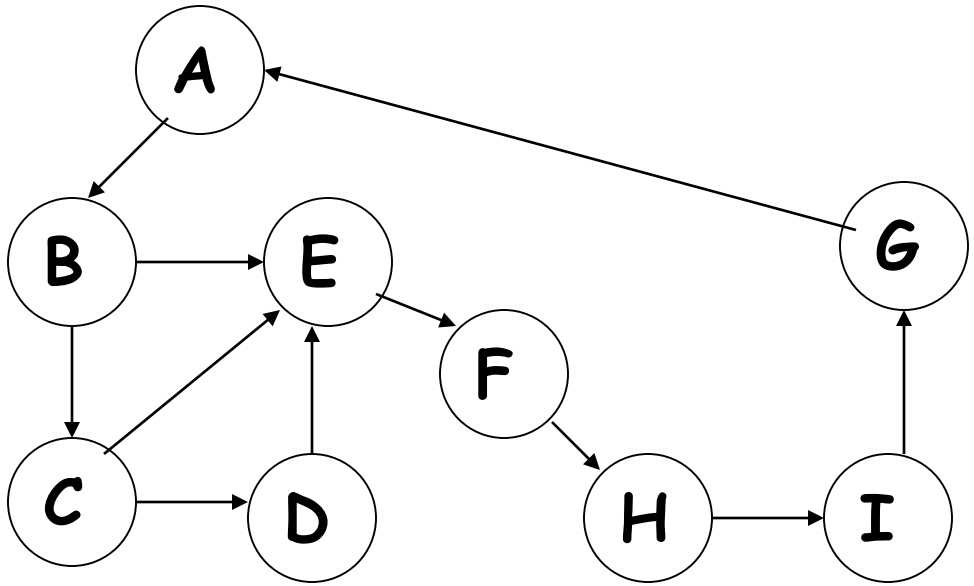
Strongly Connected Graph

A Directed Graph is said to be strongly connected if there is a path from every vertex to every other vertex in the digraph

Weakly Connected Graph

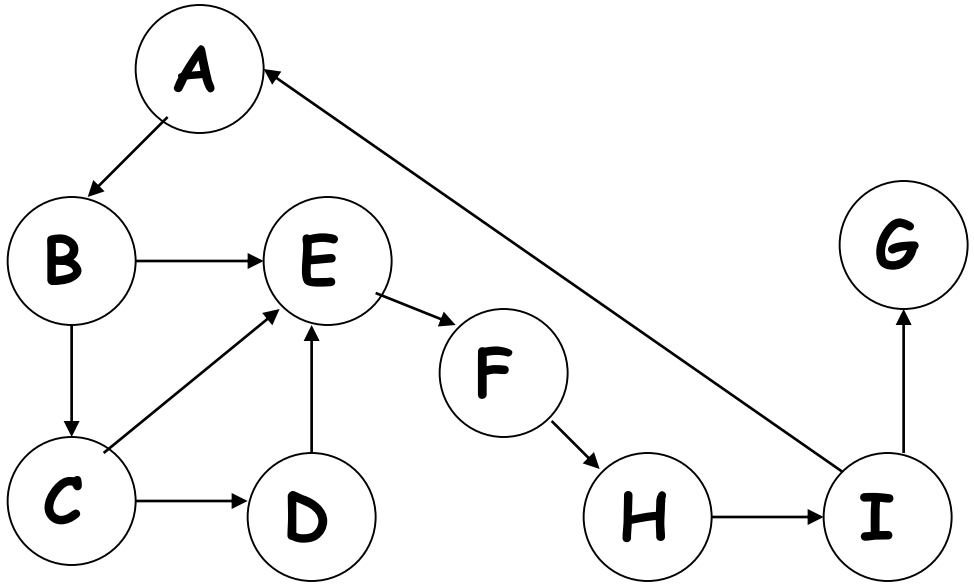
A Directed Graph is said to be weakly connected when there are at least two vertices that are not connected

Graph – Basic Terminology



Strongly Connected – For each and every pair of vertices, there exist a path.

Graph – Basic Terminology

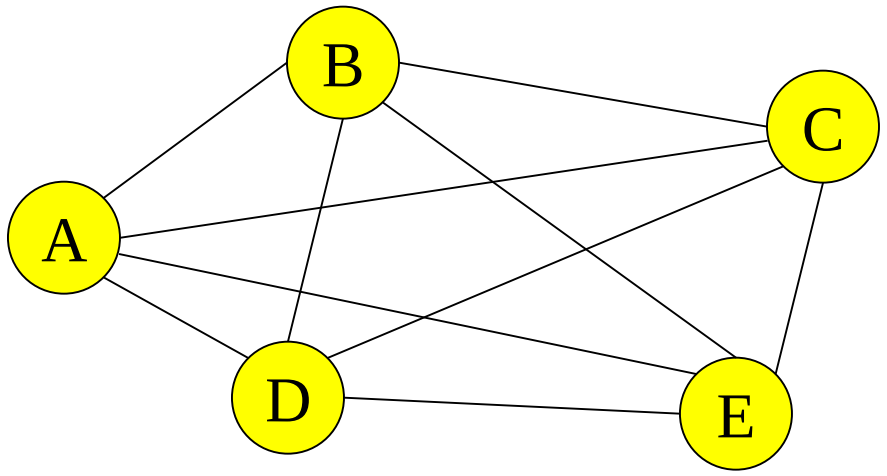


Weakly Connected – No path exist from G to all other vertices.

Graph – Basic Terminology

Complete Graph

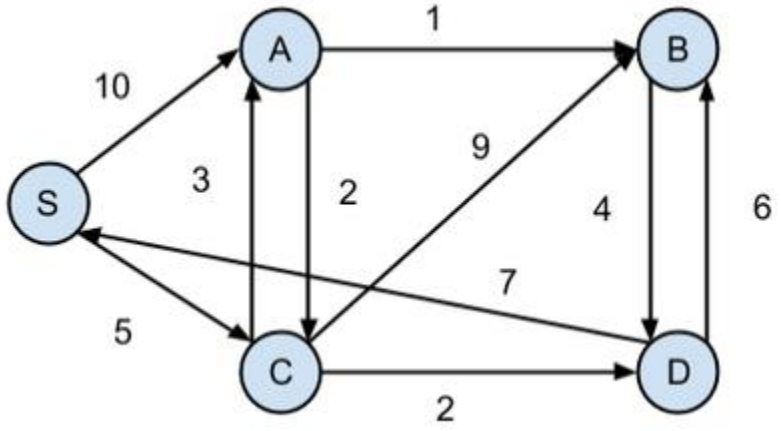
A complete graph is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge.



Graph – Basic Terminology

Weighted Graph

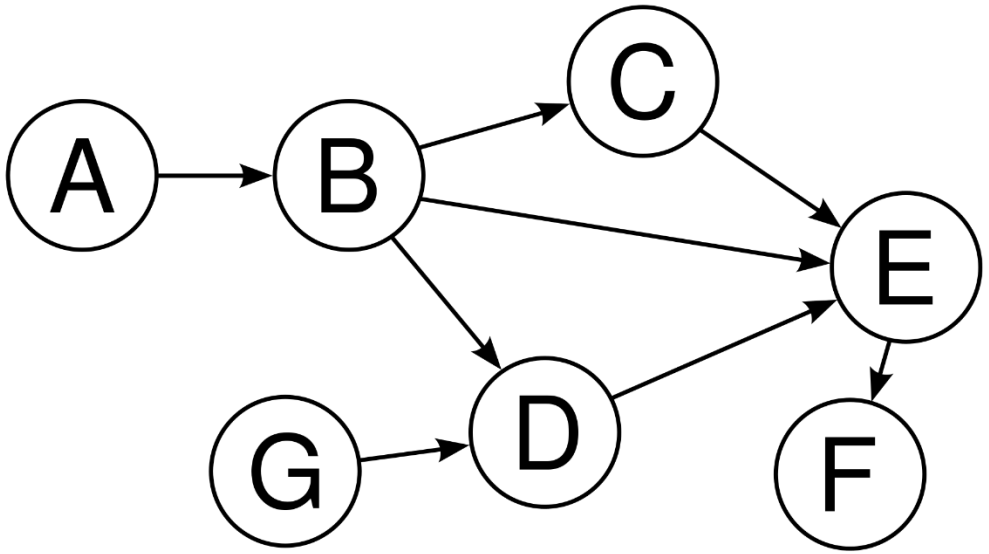
The weighted graph is either directed or undirected. But every edge will have a numerical COST or WEIGHT associated with it. The cost of the edge is not specified, then it will be 1 by default.



Graph – Basic Terminology

Directed Acyclic Graph (DAG)

A directed graph is said to be directed acyclic graph if there is no cycle in the graph.



Graph – Representation

Space and time complexity of graph algorithms are analyzed in terms of:

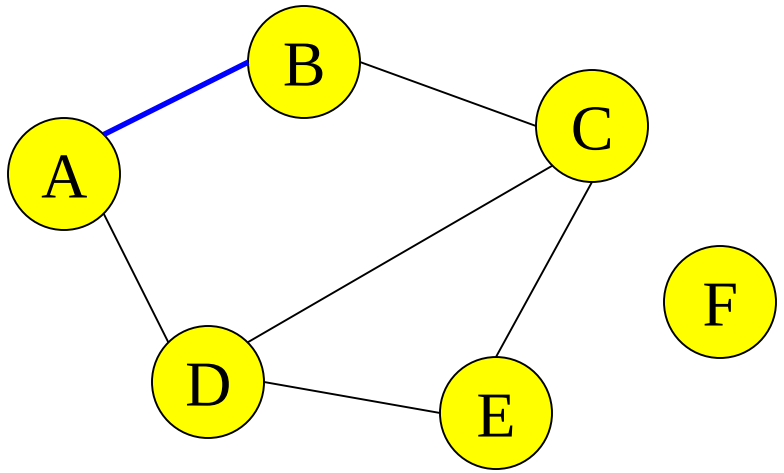
- ✓ Number of vertices = |V| and
- ✓ Number of edges = |E|

There are TWO basic representations for graph.

1. Adjacency Matrix
2. Adjacency List

Graph – Representation

Adjacency Matrix – Undirected Graph



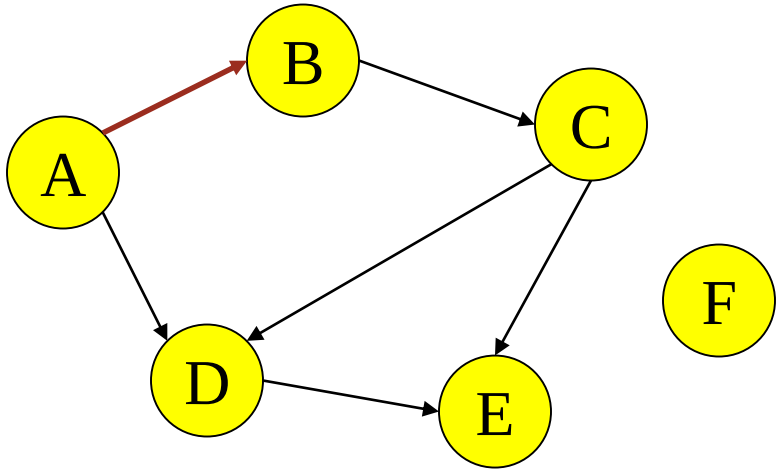
$$M(v, w) = \begin{cases} 1 \text{ if } [v, w] \text{ is in } E \\ 0 \text{ otherwise} \end{cases}$$

	A	B	C	D	E	F
A	0	1	0	1	0	0
B	1	0	1	0	0	0
C	0	1	0	1	1	0
D	1	0	1	0	1	0
E	0	0	1	1	0	0
F	0	0	0	0	0	0

Space = $|V|^2$

Graph – Representation

Adjacency Matrix – Digraph



$$M(v, w) = \begin{cases} 1 \text{ if } [v, w] \text{ is in } E \\ 0 \text{ otherwise} \end{cases}$$

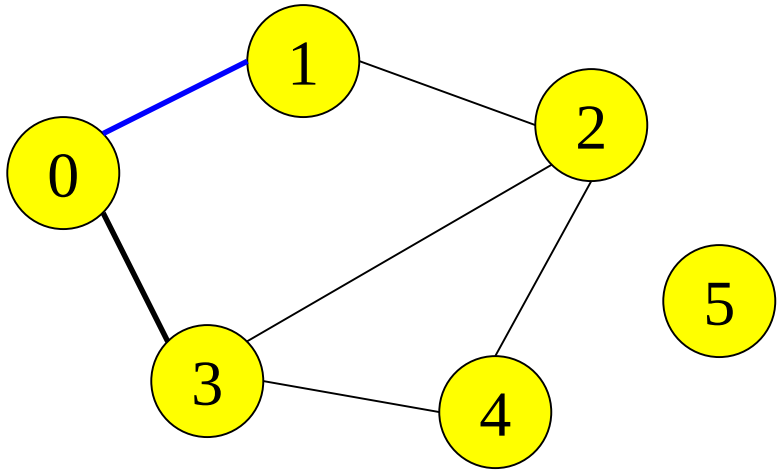
	A	B	C	D	E	F
A	0	1	0	1	0	0
B	0	0	1	0	0	0
C	0	0	0	1	1	0
D	0	0	0	0	1	0
E	0	0	0	0	0	0
F	0	0	0	0	0	0

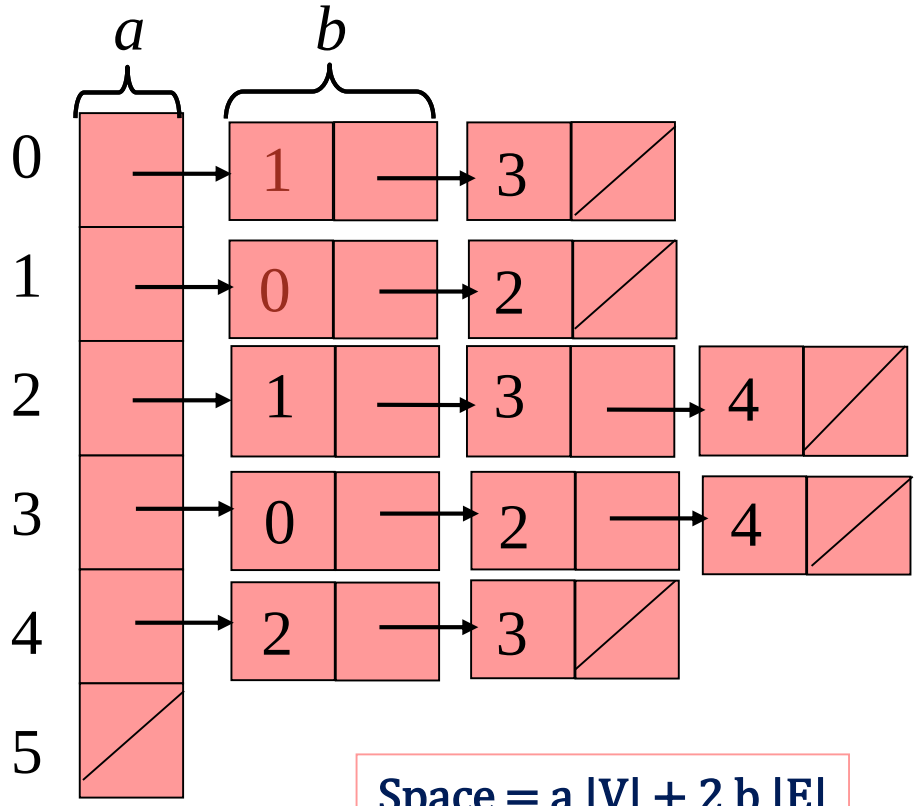
Space = $|V|^2$

Graph – Representation

Adjacency List – Undirected Graph

For each v in V, L(v) = list of w such that [v, w] is in E



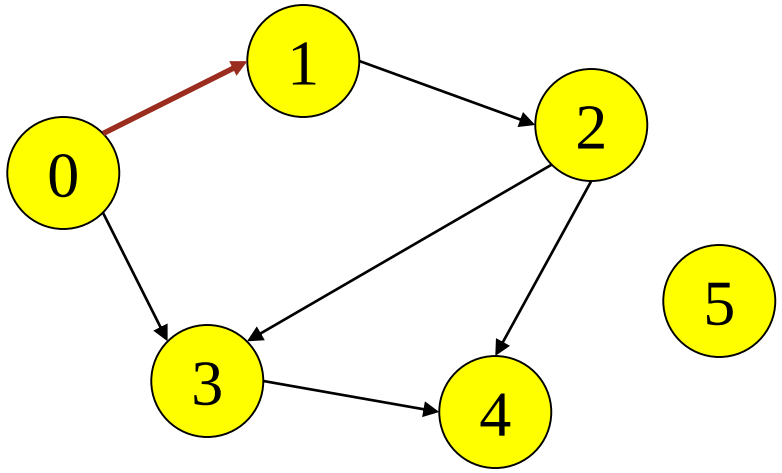


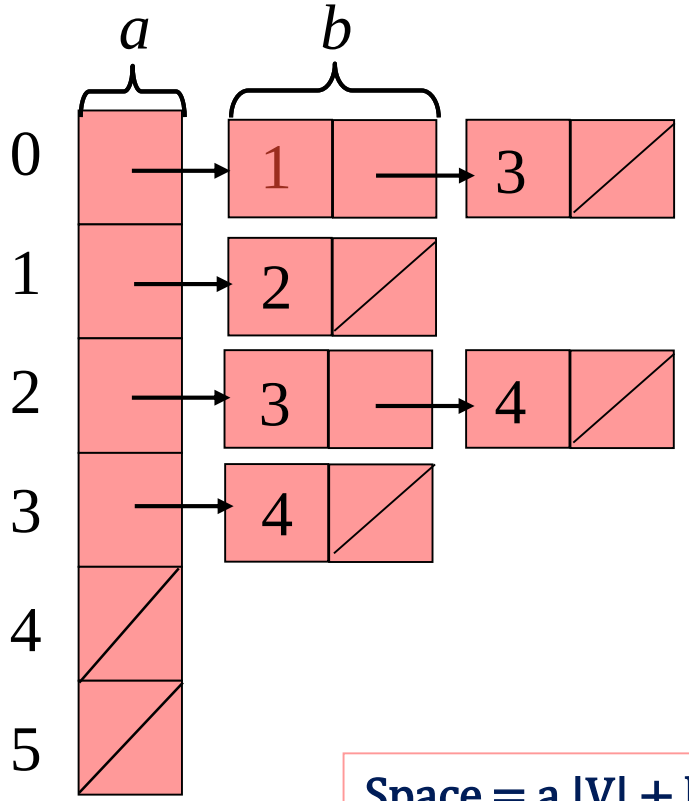
Space = a |V| + 2 b |E|

Graph – Representation

Adjacency List – Digraph

For each v in V, L(v) = list of w such that (v, w) is in E





0 is a source
4 is a sink
5 is disconnected from the rest

Space = a |V| + b |E|

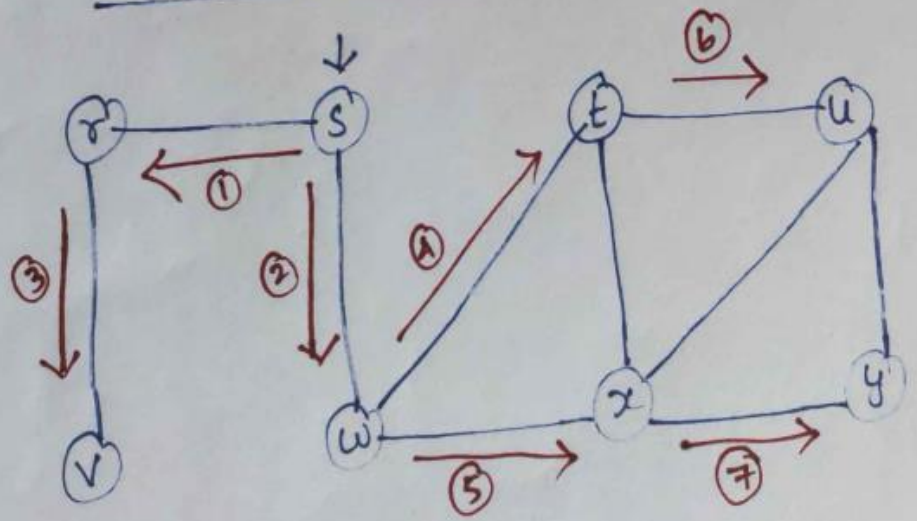
Traversal Algorithms – Breadth First Search

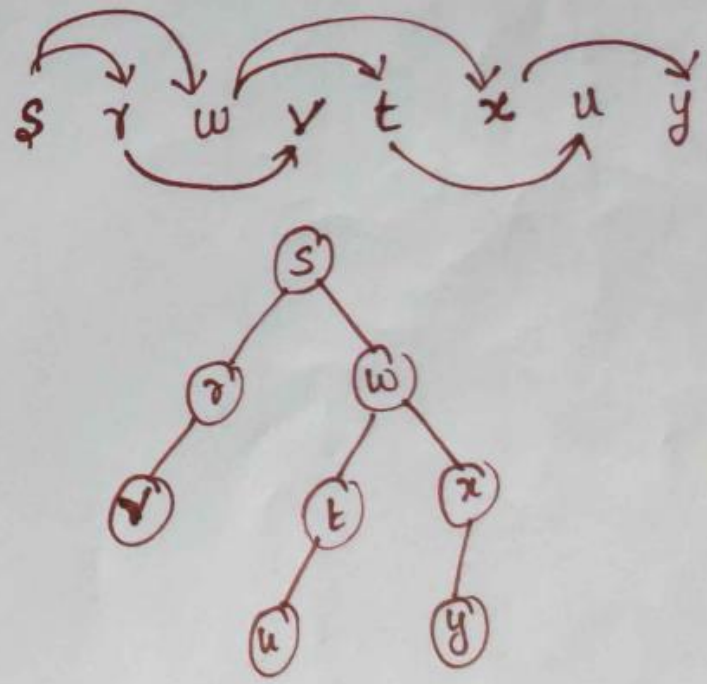
Traversal in Graph:

→ Order of visiting the vertices in Graph.

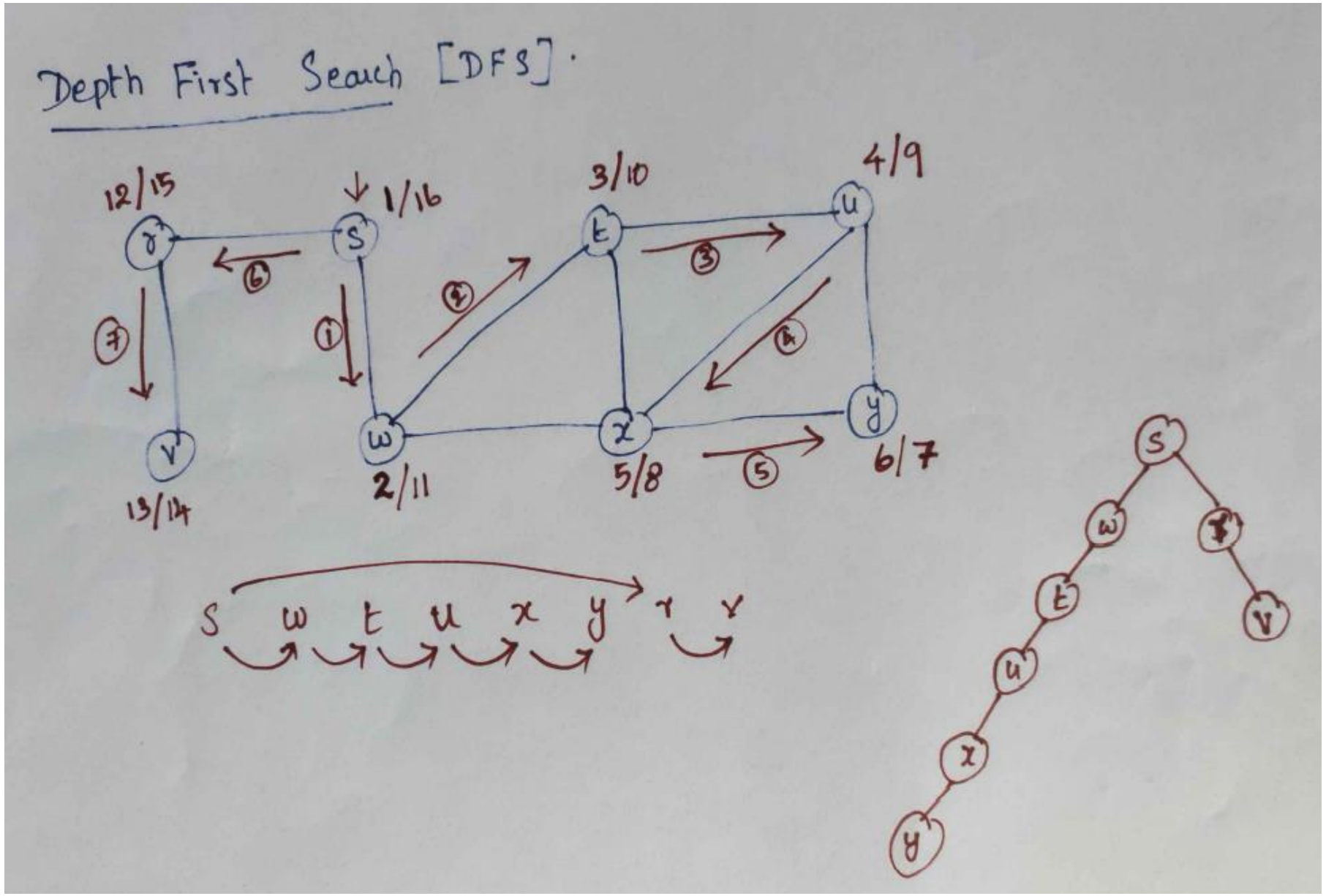
1. Breadth First Seach [BFS]
2. Depth First Search [DFS]

Breadth First Seach: [BFS]





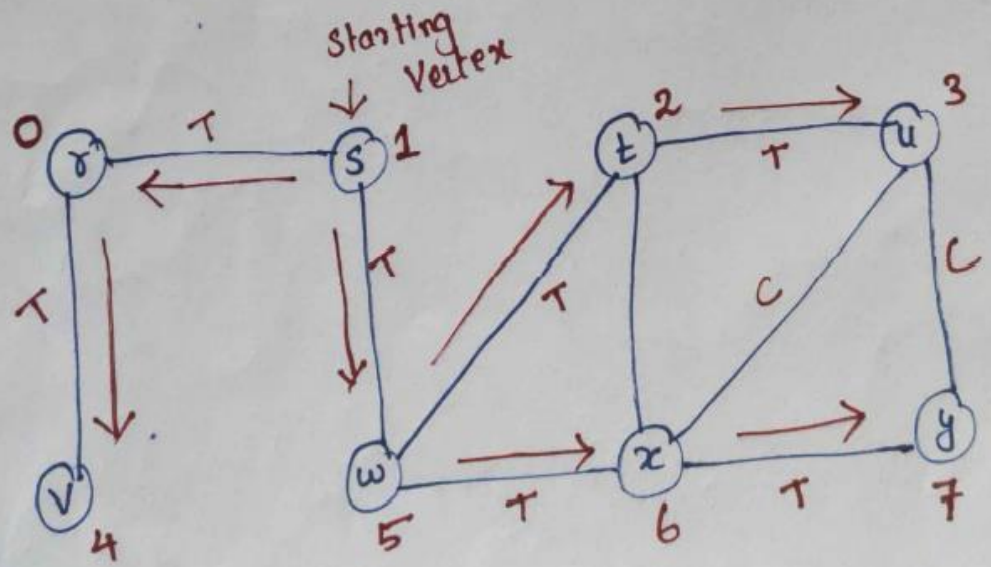
Depth First Seach [DFS].

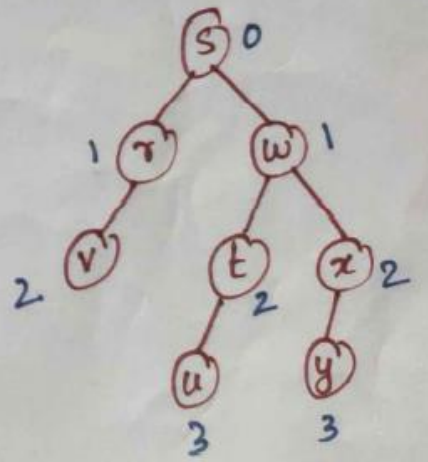


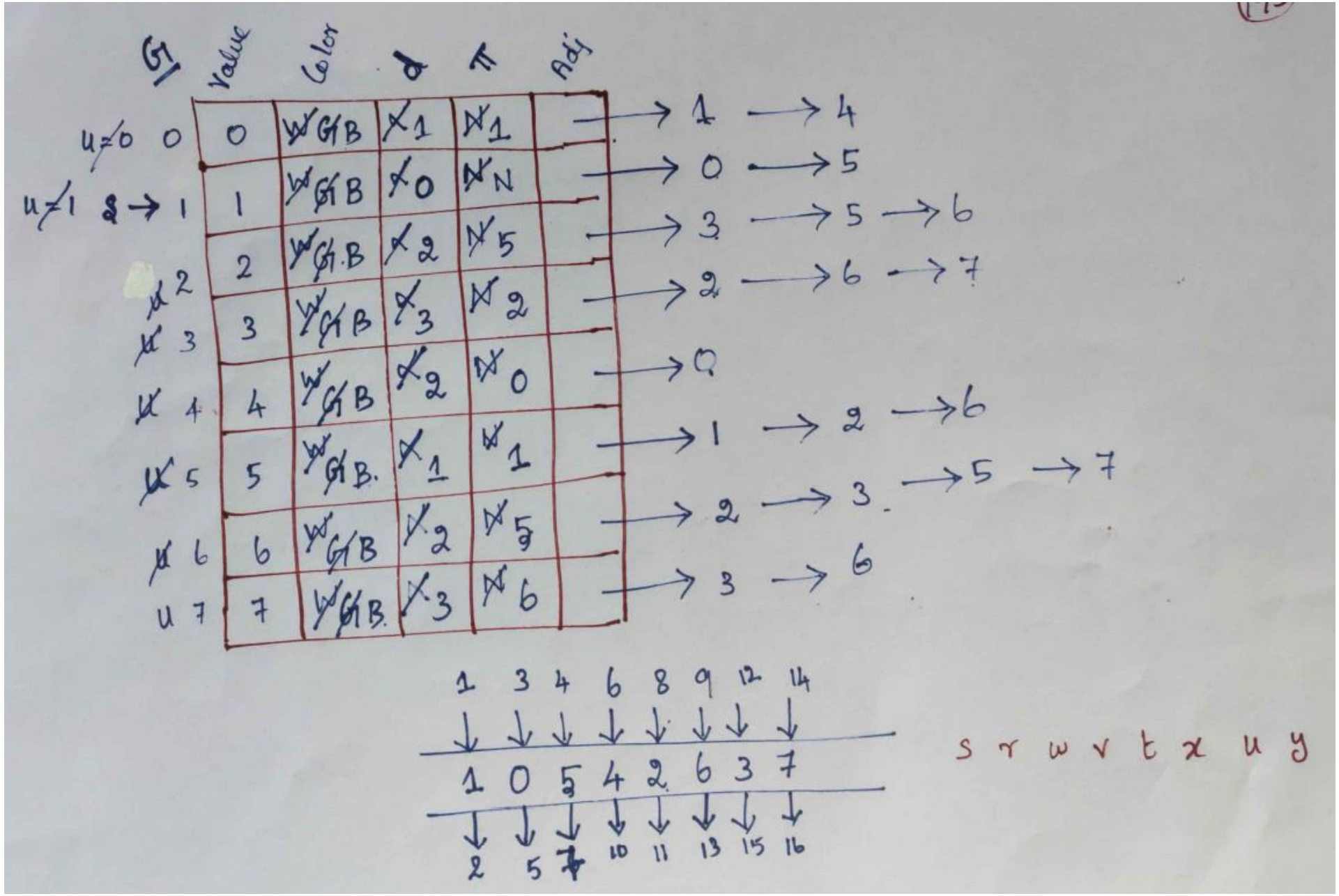
Breadth First Search [BFS] - Algorithm

Each Vertex
- Color
- Distance (d)
- Parent (π)
- Value
- Adjacency List

Color
- White – Unvisited
- Gray – Visited. But Adj. Not Visited
- Black – Visited





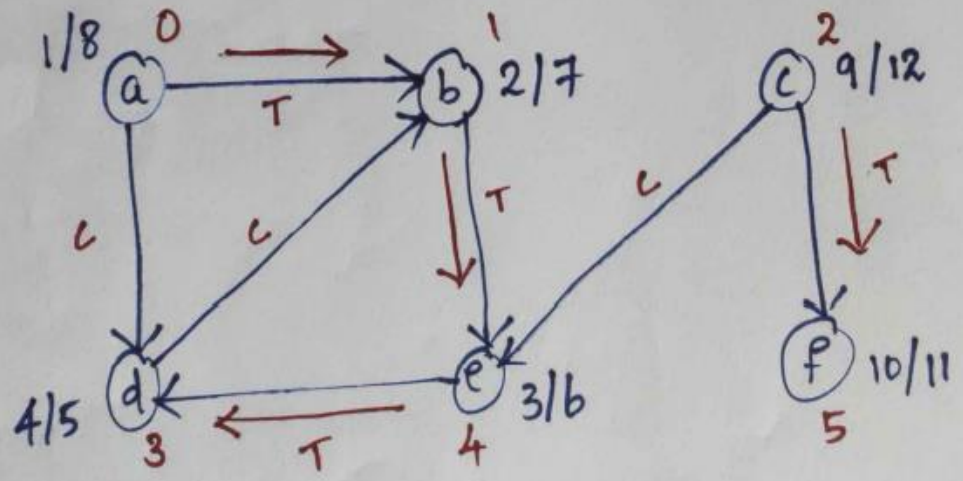

G
Value
Color
d
π
Adj
u≠0
u≠1
s → 1
0 1 2 3 4 5 6 7
1 → 4
0 → 5
3 → 5 → 6
2 → 6 → 7
0
1 → 2 → 6
2 → 3 → 5 → 7
3 → 6
1 3 4 6 8 9 12 14
1 0 5 4 2 6 3 7
2 5 7 10 11 13 15 16
s r w v t x u y

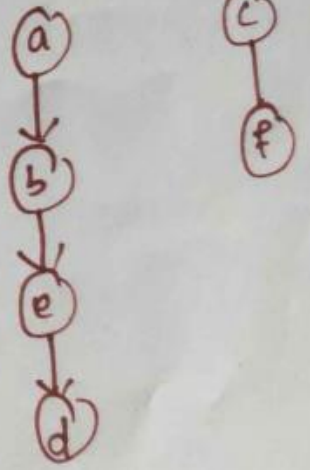
```
Alg BFS(G, s)

Step 1   for each vertex v ∈ G.V
             v.color ← WHITE
             v.d     ← ∞
             v.π     ← NIL
         end for

Step 2   s.color ← GRAY
         s.d     ← 0
         s.π     ← NIL

Step 3   // Let Q be a Queue to store list of Vertices
         Q ← ϕ
         EnQ(Q, s)

Step-4   While Q ≠ ϕ do
             u ← DeQ(Q)
             for each v ∈ G.Adj[u] do
                 if v.color = WHITE then
                     v.color ← GRAY
                     v.d     ← u.d + 1
                     v.π     ← u
                     EnQ(Q, v)
                 end if
             end for
             u.color ← BLACK
         end Loop
     end BFS
```

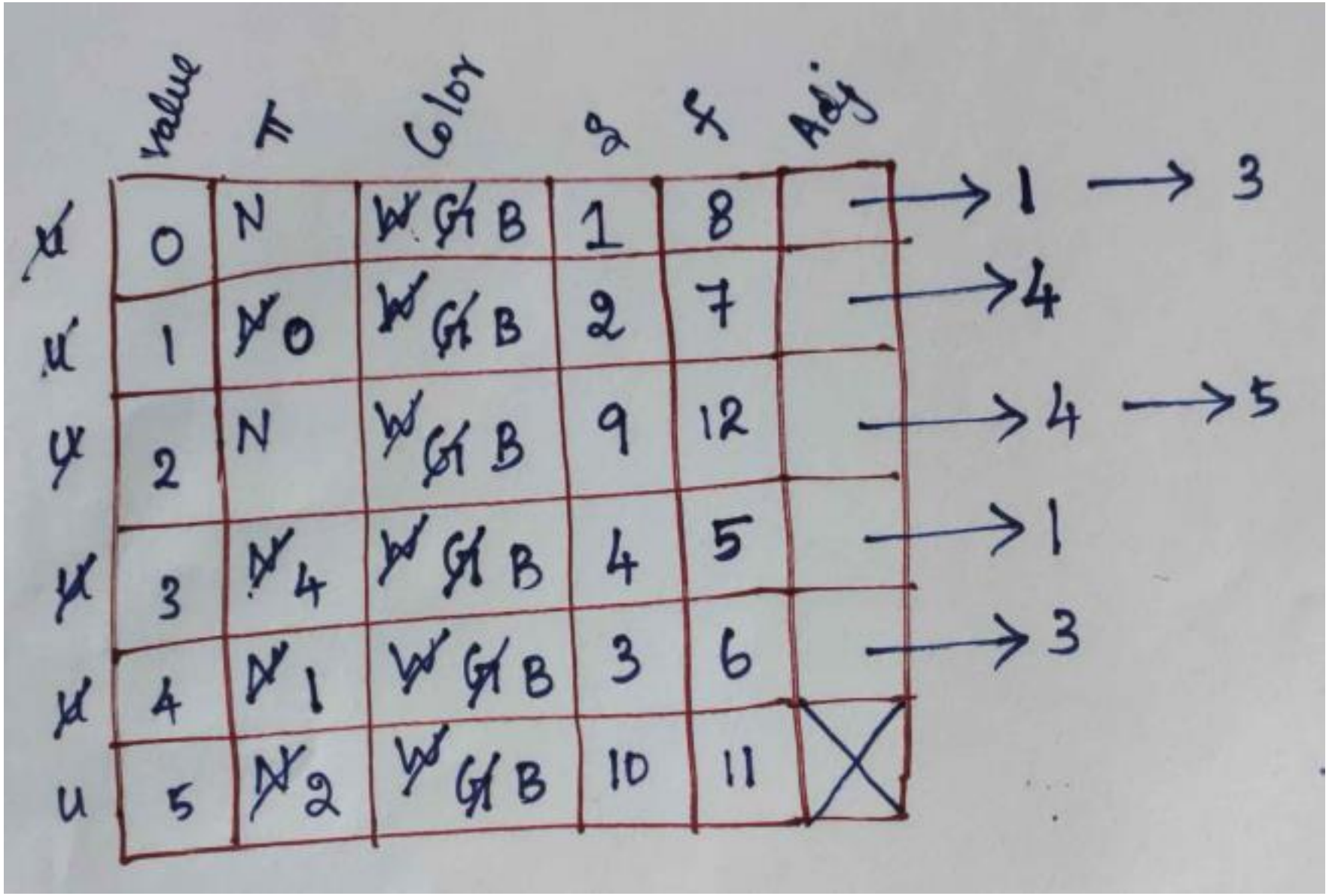
Traversal Algorithms – Depth First Search

Depth First Search: [DFS]

[Each Vertex
- s - Starting time
- f - Finishing Time
- Color {W, G, B}
- Parent (π)
- Value
- Adjacency List






value
T
Color
d
f
Adj
0
N
W G B
1
8
1
7
2
N
9
12
3
4
5
4
1
3
6
5
2
10
11
1
3
4
5
4
1
3

```
Alg DFS(G)
    for each u ∈ G.V do
        u.π ← NIL
        u.color ← WHITE
    end for
    time ← 0
    for each u ∈ G.V do
        if u.color = WHITE then
            DFS_VISIT(G, u, time)
        endif
    end for
end DFS
```

```
Alg. DFS_VISIT (G, u, time)

    u.Color ← GRAY
    time ← time + 1
    u.s ← time

        For each v ∈ G.Adj[u] do
            if v.Color = WHITE then
                v.π ← u
                DFS_VISIT (G, v, time)
            end if
        end for

    u.Color ← BLACK
    time ← time + 1
    u.f ← time

end DFS_VISIT
```

Topological Sort of Edges

Topological Sort: - Application of DFS

→ Linear Ordering of vertices in a Directed Acyclic Graph [DAG]

Definition:

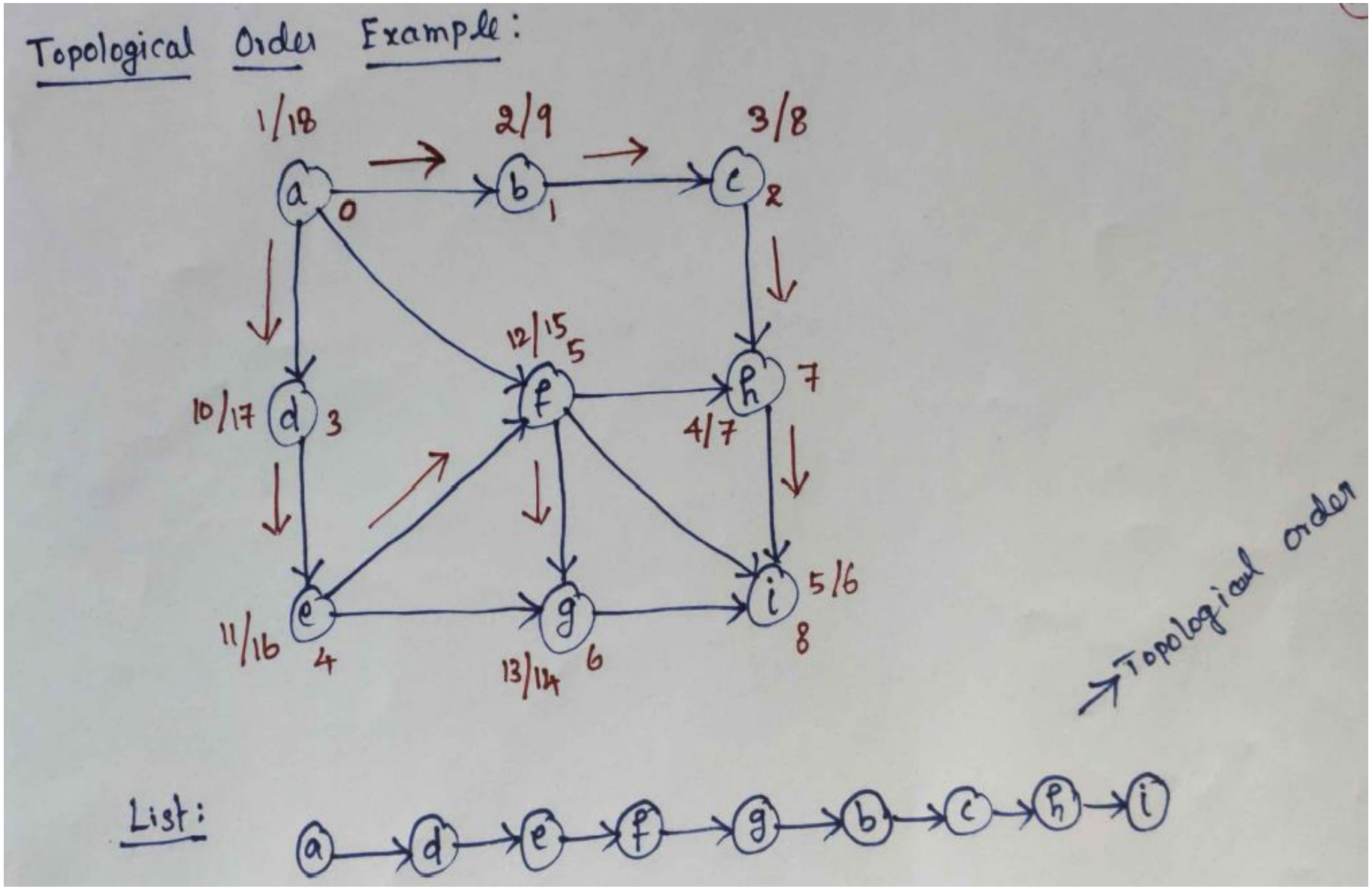
A topological sort of a DAG, $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an Edge (u, v), then the vertex u appears before v in the ordering.

Algorithm:

Alg. Topological Sort (G)

1. Call DFS (G) to compute the finishing time for every vertex v.
2. As each vertex is finished, Insert it onto the Front of the linked list.
3. Return the linked list.

Topological Order Example:



Running Time of Traversal Algorithms:

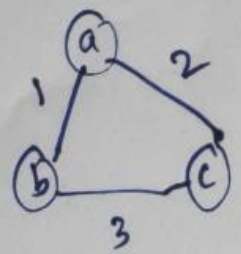
1. BFS - $O(|V| + |E|)$
2. DFS - $O(|V| + |E|)$
3. Topological Sort - $O(|V| + |E|)$

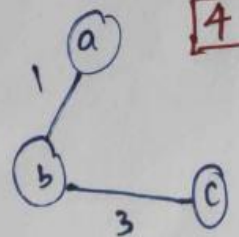
Minimum Spanning Tree – Prim's

Minimum Spanning Tree : (of a Graph)

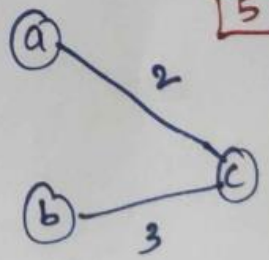
- is a Tree constructed from a weighted-Undirected Graph
- which Connects all the vertices of the Graph without any cycle.
- The resultant spanning Tree's Cost (Sum of Cost of all edges) Should be minimum
- Connects 'n' vertices by using 'n-1' edges.

Example :



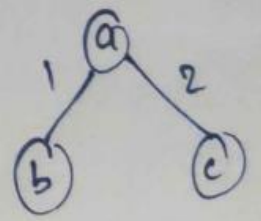


4



5

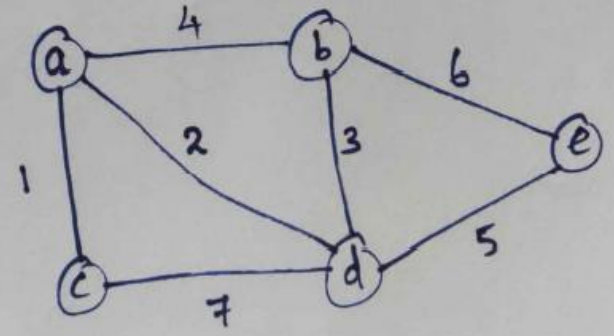
✓ MST



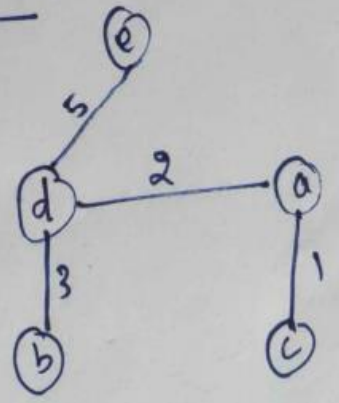
3

Algorithms for MST:

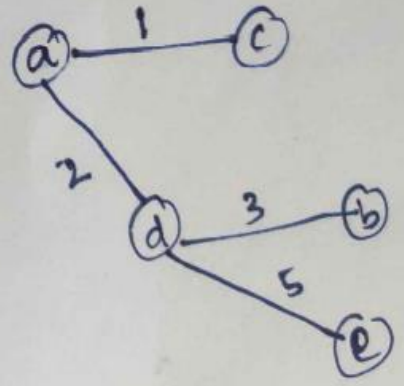
- Prim's
- Kruskal's



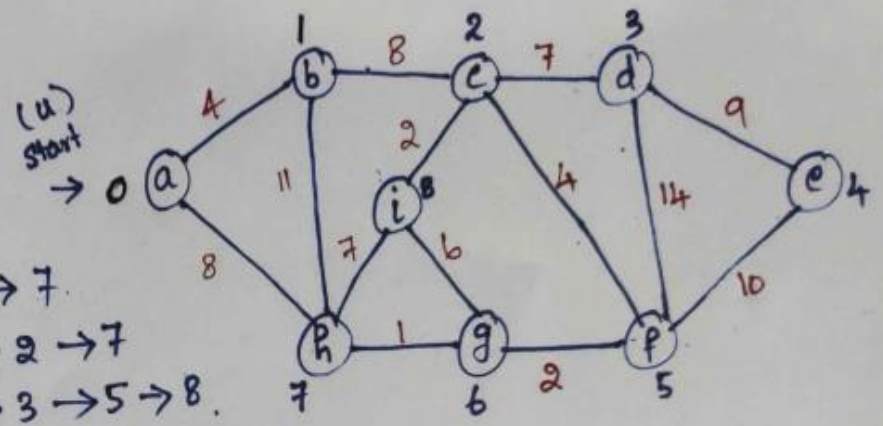
Prim's:



Kruskal's

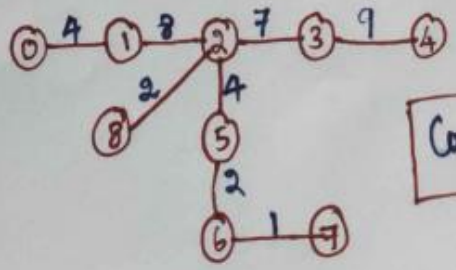


Prim's Algorithm:



Vertex	π	Color	Cost	Adj
0	N	B	0	→ 1 → 7
1	0	B	4	→ 0 → 2 → 7
2	1	B	8	→ 1 → 3 → 5 → 8
3	2	B	7	→ 2 → 4 → 5
4	3	B	9	→ 3 → 5
5	3	B	14	→ 2 → 3 → 4 → 6
6	5	B	2	→ 5 → 7 → 8
7	0	B	8	→ 0 → 1 → 6 → 8
8	2	B	2	→ 2 → 6 → 7

w	0	1	2	3	4	5	6	7	8
0	0	4	0	0	0	0	0	8	0
1	4	0	8	0	0	0	0	11	0
2	0	8	0	7	0	4	0	0	2
3	0	0	7	0	9	14	0	0	0
4	0	0	0	9	0	10	0	0	0
5	0	0	4	14	10	0	2	0	0
6	0	0	0	0	0	2	0	1	8
7	8	11	0	0	0	0	1	0	7
8	0	0	2	0	0	0	8	7	0



Cost = 37

```
Alg Prims (G, w, u)

For each v ∈ G.V do
    v.π ← NIL
    v.Color ← WHITE
    v.Cost ← ∝
end for

u.Cost ← 0

Let a Min-Priority Queue, Q
Q ← G.V

While Q ≠ ϕ do
    s ← Extract_Min(Q)
    For each v ∈ G.Adj[s] do
        if v.Color = WHITE then
            if w(s,v) < v.Cost then
                v.Cost ← w(s,v)
                v.π ← s
            endif
        end if
    end for
    s.Color ← BLACK
end while

end Prims
```

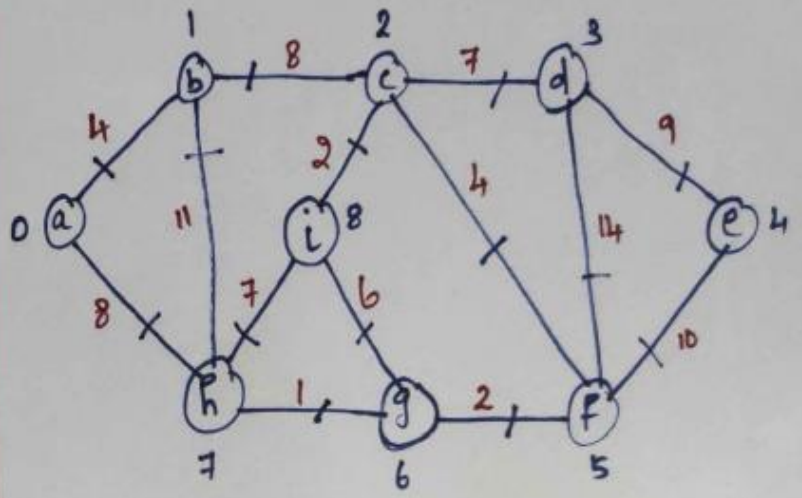
Minimum Spanning Tree – Kruskal's

Kruskal's Algorithm: [for MST]

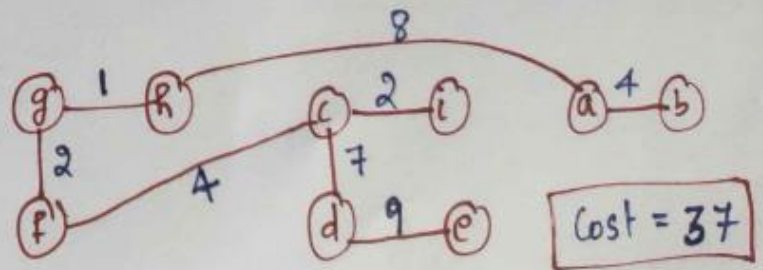
(1,2) → Edge A = { }

Alg. Kruskals (G, ω) → Graph (V, E); ω → weight matrix

```
A ← ϕ
for each vertex v ∈ G.V
    MAKE_SET(v)
end for
Sort the Edges of G [G.E]
    into non-decreasing order
        of its weight
For each edge (u,v) ∈ G.E do
  if FIND-SET(u) ≠ FIND-SET(v) then
        A ← A U {(u,v)}
  end if UNION(u,v)
end for
return A
end Kruskals
```



A = { (g,h), (c,i), (g,f), (a,b), (c,f), (c,d), (a,h), (d,e) }



Cost = 37

Edge	Connected Components	Decision
	{a} {b} {c} {d} {e} {f} {g} ✓ {h} ✓ {i}	
(g, h) - 1	{a} {b} {c} * {d} {e} {f} {g, h}, {i} *	✓
(c, i) - 2	{a} {b} {c, i}, {d}, {e}, {f} * {g, h} *	✓
(g, f) - 2	{a} * {b} * {c, i} {d} {e} {f, g, h}	✓
(a, b) - 4	{a, b} {c, i} * {d} {e} {f, g, h} *	✓
(c, f) - 4	{a, b} {c, i, f, g, h} * * {d} {e}	✓
(g, i) - 6	{a, b} {c, i, f, g, h} * {d} * {e}	✗
(c, d) - 7	{a, b} {c, i, f, g, h, d} * * {e}	✓

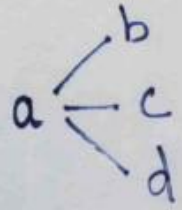
Edge	Connected Components	Decision
(h, i) - 7	{a, b} {c, i, f, h, g, d} {e}	X
(a, h) - 8	{a, b, c, i, f, g, h, d} {e}	✓
(b, c) - 8	{a, b, c, i, f, g, h, d} {e}	X
(d, e) - 9	{a, b, c, i, f, g, h, d, e}	✓
(e, f) - 10	{a, b, c, i, f, g, h, d, e}	X
(b, h) - 11	{a, b, c, i, f, g, h, d, e}	X
(d, f) - 14	{a, b, c, i, f, g, h, d, e}	X

Shortest Path Algorithms – Dijkstra's

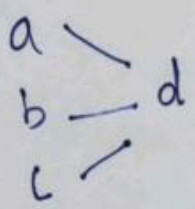
Shortest Path Algorithms:

↳ A path between 2 vertices of a Graph such that the Sum of the weights of its consituent edges is minimized.

1. Single Source



2. Single Destination

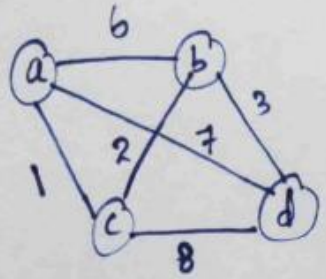


3. Single Pair (a, d)

4. All-Pairs

(a, d)
(a, b)
(a, c)
(a, d)
(b, c)
(b, d)

} For each & every pair

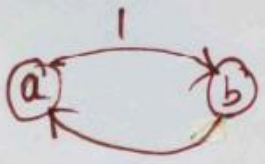


Path b/w a & d:

1 : ad — 7
2 : abd — 9
3 : acd — 9
4 : acbd — 6 ✓

Algorithms:

1. Bellman-Ford → Single-Source
   → Main application is: To verify a Graph containing Negative weight Cycle or Not.

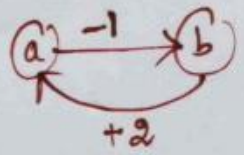


aba → 3

1 – ab ✓
4 – abab
7 – ababab
⋮

2. Dijikstra's → Single-Source
   → Cannot apply, if Gr contains -Ve weight edge

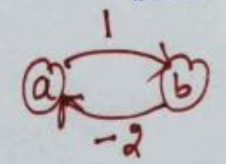
3. Floyd-Warshal
   ↳ All pairs



aba → 1

✓ ab → -1
abab → 0
ababab → 1
⋮ →

aba → -1

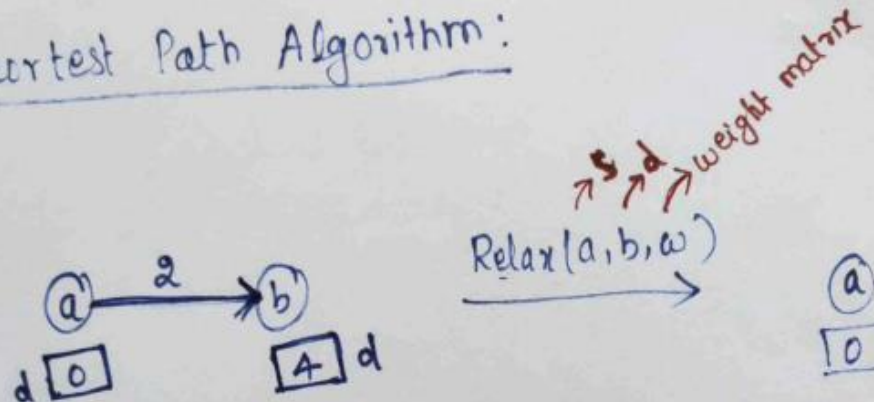


4. Johnson's
   ↳ All Pairs
   ↳ Use Bellman-ford and Dijikstra's

ab → 1
abab → 0
ababab → -1
⋮ →

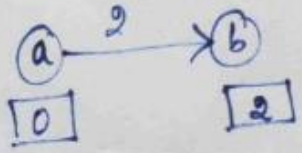
Dijikstra's Shortest Path Algorithm:

Relax:

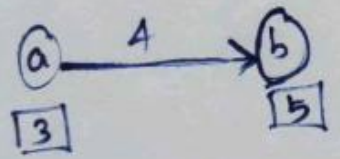


Relax(a, b, w)

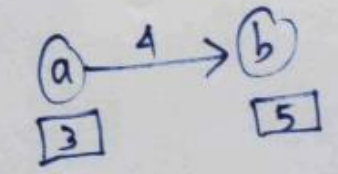
s d weight matrix



$(0+2) < 4 \Rightarrow$ update



Relax(a, b, w)



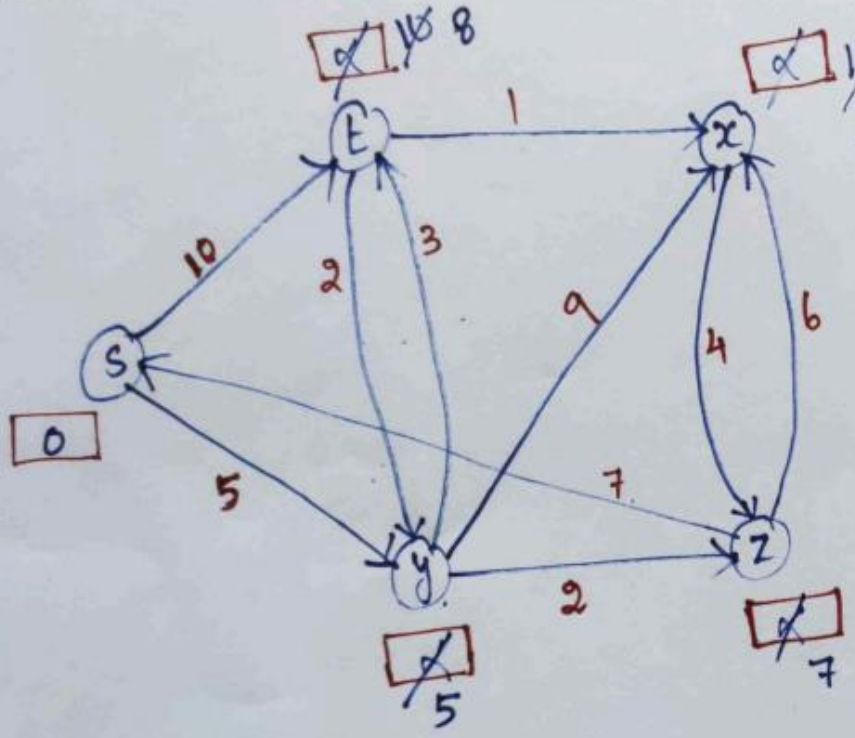
$(3+4) < 5 \Rightarrow \times$, No updation

```
if ((a.d + w(a,b)) < b.d)
{
        b.d ← a.d + w(a,b)
}
```

```
Alg Dijikstra ( G, ω, s )

    for each v ∈ G.V do
        v.π ← NIL
        v.d ← ∝
    endfor
    s.d ← 0
    Let Q be a min Priority Queue
    Q ← G.V
    while Q ≠ ϕ do
        u ← Extract Min ( Q )
        for each vertex v ∈ G.Adj[u] do
            Relax ( u, v, ω )
        end for
    end while
end Alg.
```

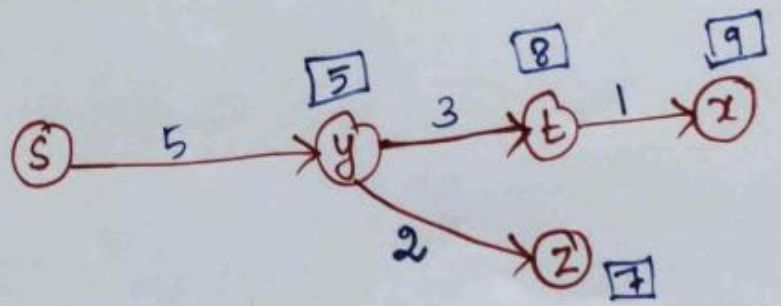
```
Alg Relax ( u, v, ω )

    if (u.d + ω(u,v)) < v.d then
        v.d ← u.d + ω(u,v)
        v.π ← u
    end if
end Relax.
```

Dijikstra's Algorithm:



u	Value d	π	Adj
s	0	N	t → y
t	∞ 8	s y	x → y
x	∞ 14 13 9	y t z	→ z
y	5	s	x → t → z
z	7	y	x → s

Running Time : $\Theta(|E| \cdot \log |V|)$



Shortest Path Algorithms – Bellman-Ford

Bellman-Ford Algorithm

- ✓ Used to find shortest path from the given starting vertex to all other vertices in a given graph.
- ✓ Also, used to check whether a graph contains Negative Weight Cycle (NWC) or not.
- ✓ So, for BF algorithm, negative edges are allowed, where as the Dijkstra's algorithm won't work, if graph contains negative weight cycle.
- ✓ BF algorithm report "False", if graph contains any NWC.
- ✓ BF algorithm report "True" and calculate the shortest distance from the starting vertex to all other vertices, only if the graph has no NWC.

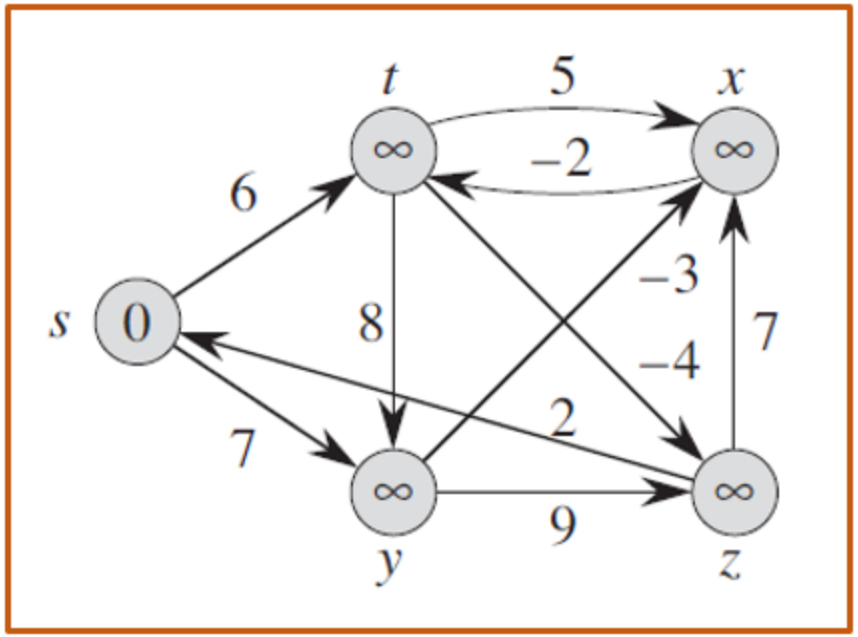
Bellman-Ford Algorithm

```
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
```

```
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
```

BF algorithm – Example – Step-by-Step

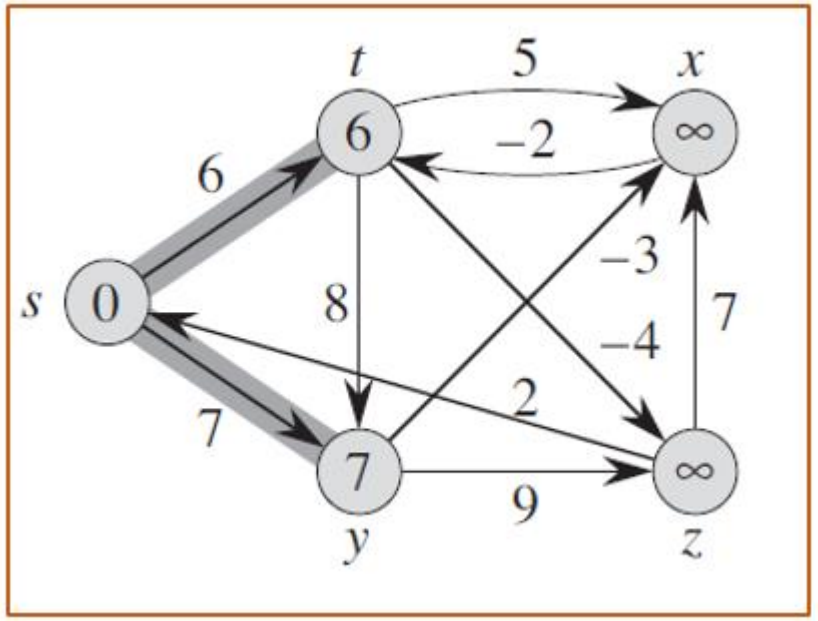


Value	Parent	Distance
s	Nil	0
t	Nil	∞
x	Nil	∞
y	Nil	∞
z	Nil	∞

- ✓ Totally, 'n-1' Iterations. i.e., All the edges needs to be relaxed for n-1 times in a particular order
- ✓ In this, example, n=5, So, Need to relax all the edges for 4 times.
- ✓ Assume the order of edges as:

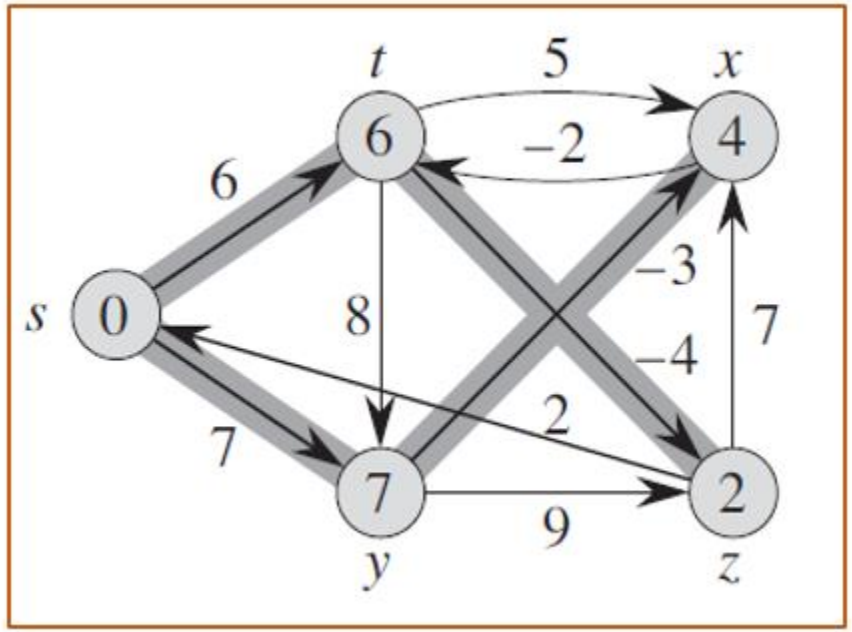
$$(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$$

Relaxing Edges					
S.No	Edges	i=1	i=2	i=3	i=4
1	(t,x)	N			
2	(t,y)	N			
3	(t,z)	N			
4	(x,t)	N			
5	(y,x)	N			
6	(y,z)	N			
7	(z,x)	N			
8	(z,s)	N			
9	(s,t)	Y			
10	(s,y)	Y			



Value	Parent	Distance
s	Nil	0
t	s	6
x	Nil	∞
y	s	7
z	Nil	∞

Relaxing Edges					
S.No	Edges	i=1	i=2	i=3	i=4
1	(t,x)	N	Y		
2	(t,y)	N	N		
3	(t,z)	N	Y		
4	(x,t)	N	N		
5	(y,x)	N	Y		
6	(y,z)	N	N		
7	(z,x)	N	N		
8	(z,s)	N	N		
9	(s,t)	Y	N		
10	(s,y)	Y	N		



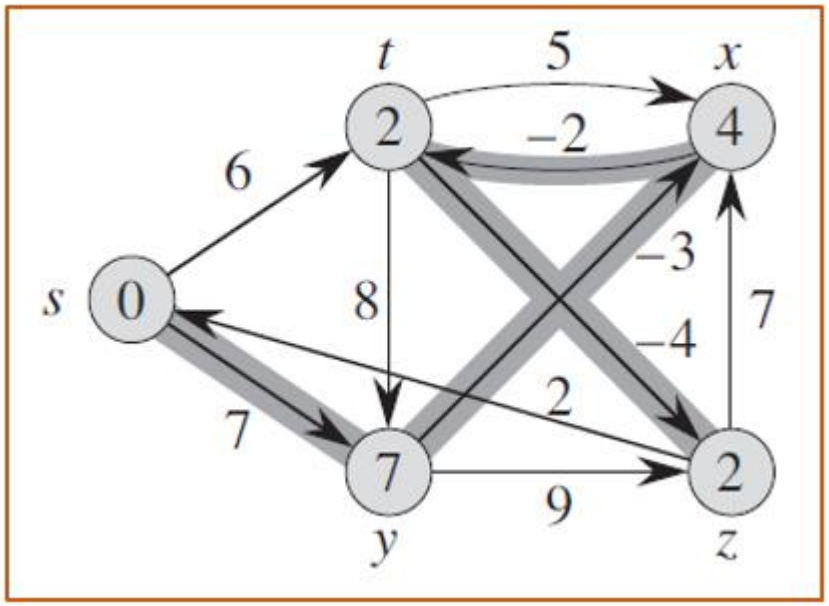
After (t,x)

Value	Parent	Distance
s	Nil	0
t	s	6
x	t	11
y	s	7
z	Nil	∞

After (t,z) & (y,x)

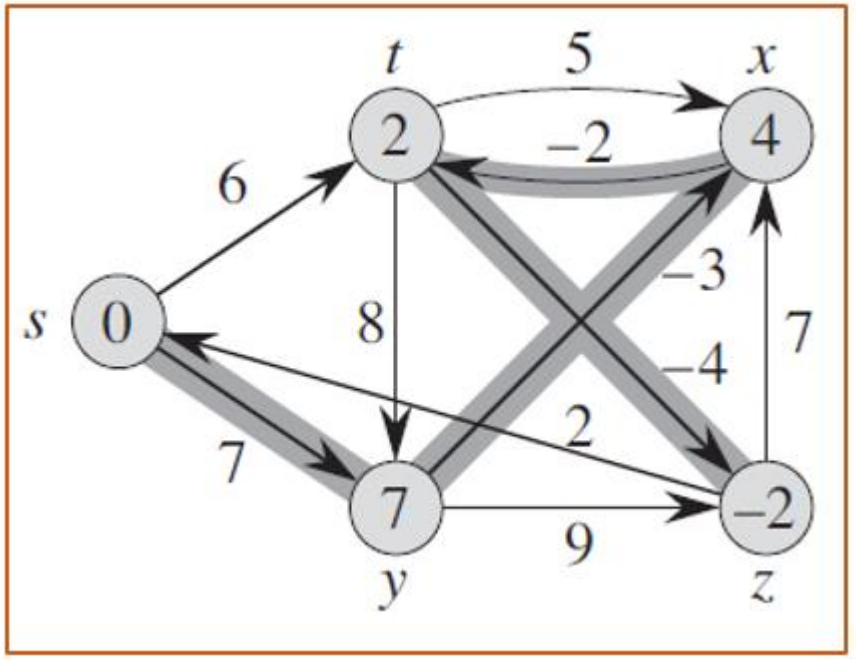
Value	Parent	Distance
s	Nil	0
t	s	6
x	y	4
y	s	7
z	t	2

Relaxing Edges					
S.No	Edges	i=1	i=2	i=3	i=4
1	(t,x)	N	Y	N	
2	(t,y)	N	N	N	
3	(t,z)	N	Y	N	
4	(x,t)	N	N	Y	
5	(y,x)	N	Y	N	
6	(y,z)	N	N	N	
7	(z,x)	N	N	N	
8	(z,s)	N	N	N	
9	(s,t)	Y	N	N	
10	(s,y)	Y	N	N	



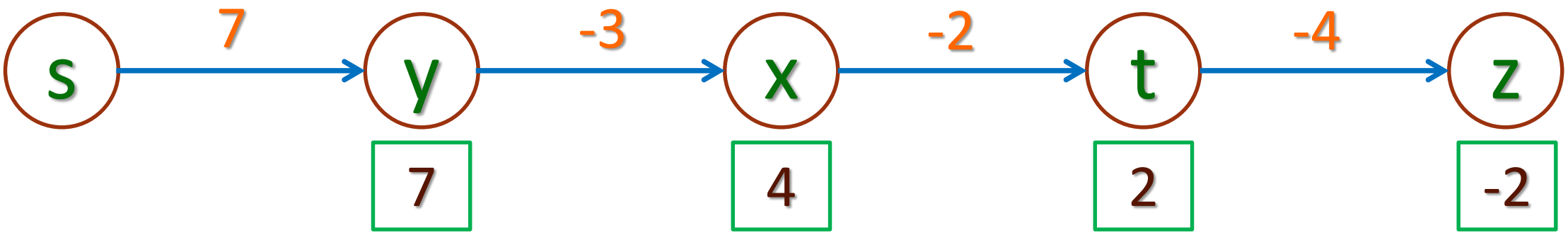
Value	Parent	Distance
s	Nil	0
t	x	2
x	y	4
y	s	7
z	t	2

Relaxing Edges					
S.No	Edges	i=1	i=2	i=3	i=4
1	(t,x)	N	Y	N	N
2	(t,y)	N	N	N	N
3	(t,z)	N	Y	N	Y
4	(x,t)	N	N	Y	N
5	(y,x)	N	Y	N	N
6	(y,z)	N	N	N	N
7	(z,x)	N	N	N	N
8	(z,s)	N	N	N	N
9	(s,t)	Y	N	N	N
10	(s,y)	Y	N	N	N



Value	Parent	Distance
s	Nil	0
t	x	2
x	y	4
y	s	7
z	t	-2

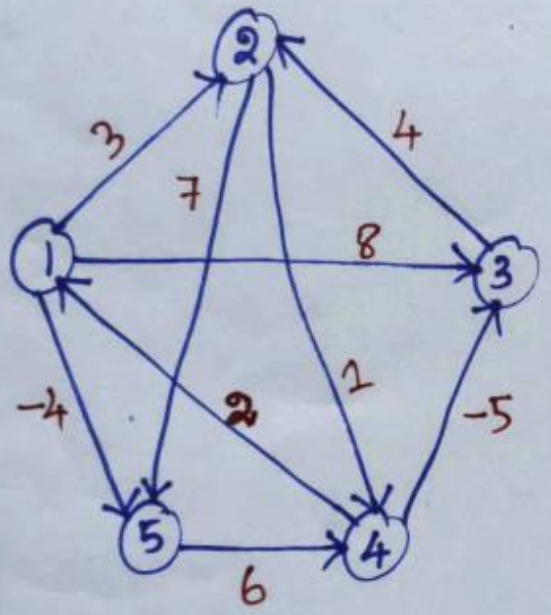
- ✓ At the end of n-1 iterations, need to check all the edges are RELAXED or NOT.
- ✓ In this example, after 4th iteration, all edges are relaxed.
- ✓ So the algorithm will return TRUE with this final Table
- ✓ The following is the shortest Path from s to all others and distances.



Shortest Path Algorithms – Floyd-Warshall

All Pair Shortest Path: Floyd - Warshall Algorithm

(Dynamic Programming Approach - Example)



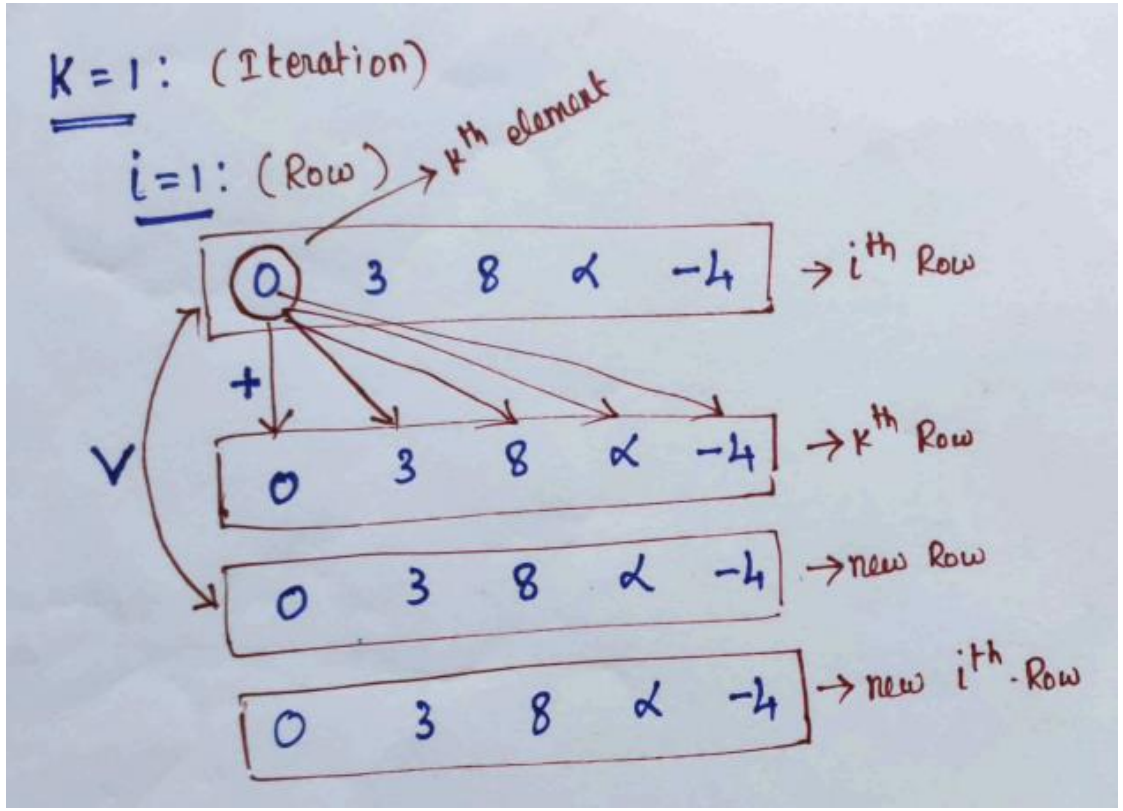
$D^{(0)}$ =

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

* k^{th} Column; * k^{th} Row

$\pi^{(0)}$ =

	1	2	3	4	5
1	N	1	1	N	1
2	N	N	N	2	2
3	N	3	N	N	N
4	4	N	4	N	N
5	N	N	N	5	N



new Row ith Row

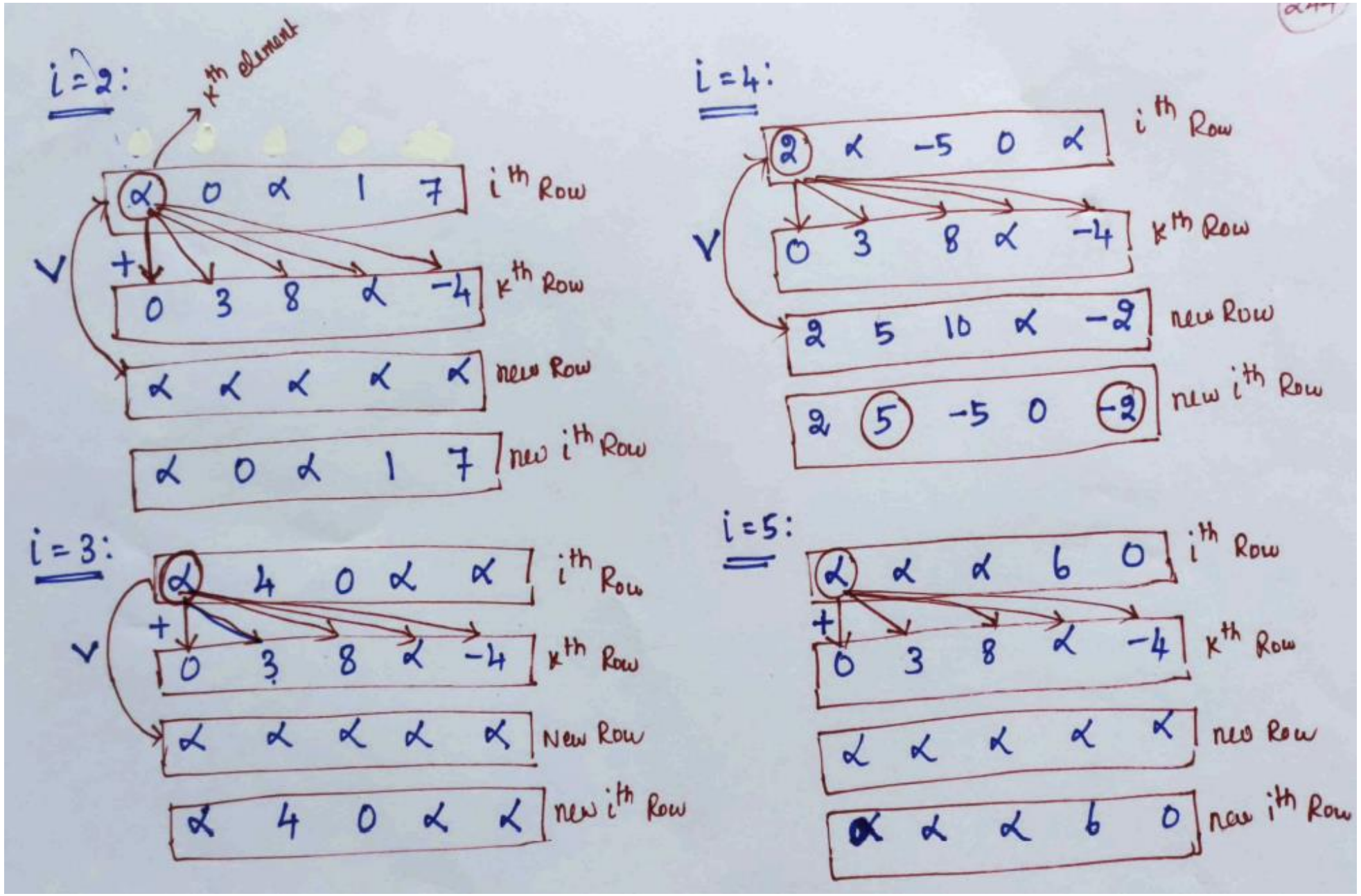
if $d_{ik} + d_{kj} < d_{ij}$

$d_{ij} = d_{ik} + d_{kj}$

else $\pi_{ij} = \pi_{kj}$

$d_{ij} = d_{ij}$

$\pi_{ij} = d_{ij}$


i=2:
kth element
α 0 α 1 7
ith Row
+
0 3 8 α -4
kth Row
α α α α α
new Row
α 0 α 1 7
new ith Row
i=3:
α 4 0 α α
ith Row
+
0 3 8 α -4
kth Row
α α α α α
New Row
α 4 0 α α
new ith Row
i=4:
2 α -5 0 α
ith Row
0 3 8 α -4
kth Row
2 5 10 α -2
new Row
2 5 -5 0 -2
new ith Row
i=5:
α α α 6 0
ith Row
+
0 3 8 α -4
kth Row
α α α α α
new Row
α α α 6 0
new ith Row

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

$$\pi^{(1)} = \begin{bmatrix} N & 1 & 1 & N & 1 \\ N & N & N & 2 & 2 \\ N & 3 & N & N & N \\ 4 & \textcircled{1} & 4 & N & \textcircled{1} \\ N & N & N & 5 & N \end{bmatrix}$$

K=2:

$$D^{(2)} = \begin{bmatrix} 0 & 3 & 8 & \textcircled{4} & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \textcircled{5} & \textcircled{11} \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}^{*}$$

$$\pi^{(2)} = \begin{bmatrix} N & 1 & 1 & \textcircled{2} & 1 \\ N & N & N & 2 & 2 \\ N & 3 & N & \textcircled{2} & \textcircled{2} \\ 4 & 1 & 4 & N & 1 \\ N & N & N & 5 & N \end{bmatrix}$$

K = 3:

$$D^{(3)} = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} *$$

$$\Pi^{(3)} = \begin{bmatrix} N & 1 & 1 & 2 & 1 \\ N & N & N & 2 & 2 \\ N & 3 & N & 2 & 2 \\ 4 & 3 & 4 & N & 1 \\ N & N & N & 5 & N \end{bmatrix}$$

K = 4:

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

$$\Pi^{(4)} = \begin{bmatrix} N & 1 & 4 & 2 & 1 \\ 4 & N & 4 & 2 & 1 \\ 4 & 3 & N & 2 & 1 \\ 4 & 3 & 4 & N & 1 \\ 4 & 3 & 4 & 5 & N \end{bmatrix}$$

K=5:

$D^{(5)}$ =

	1	2	3	4	5
1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

$\Pi^{(5)}$ =

	1	2	3	4	5
1	N	3	4	5	1
2	4	N	4	2	1
3	4	3	N	2	1
4	4	3	4	N	1
5	4	3	4	5	N

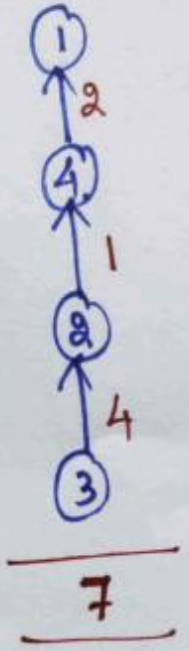
Shortest Path (3, 1):

(3, 1)

(3, 4)

(3, 2)

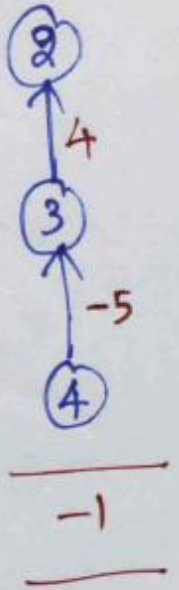
(3, 3)



(4, 2):

(4, 3)

(4, 4)



Alg FloydWarshall ($w[1..n, 1..n]$)

Let $D^{(0)}$ & $\pi^{(0)}$ be the matrices

$D^{(0)} \leftarrow w$

for $i \leftarrow 1$ to n do

for $j \leftarrow 1$ to n do

if $D_{ij}^{(0)} = 0$ OR $D_{ij}^{(0)} = \infty$ then

$\pi_{ij}^{(0)} \leftarrow N$

else $\pi_{ij}^{(0)} \leftarrow i$

end if

end for

end for

```
for k ← 1 to n do
    Let D(k) and π(k) be the matrices
    for i ← 1 to n do
        for j = 1 to n do
            if D(k-1)ik + D(k-1)kj < D(k-1)ij then
                D(k)ij ← D(k-1)ik + D(k-1)kj
                π(k)ij ← π(k-1)kj
            else
                D(k)ij ← D(k-1)ij
                π(k)ij ← π(k-1)ij
            end if
        end for
    end for
end for
```

$O(|V|^3)$

```
return D(n) & π(n)
end FloydWorshall
```

Ford-Fulkerson – Flow Network Problem

Maximum Flow Problem

- ✓ Graph problem.
- ✓ Graph is mapped with Flow Network.
- ✓ Example: Water Flow Network
- ✓ The vertices represent the tanks
- ✓ The edges represent the pipes connected between the tanks
- ✓ Edge cost maps to the capacity of the pipe. How much water it can flow?
- ✓ Problem Objective: Need to find how much stuff can be pushed from the source to the sink.
- ✓ Known as maximum flow problem.
- ✓ Algorithm: Ford-Fulkerson Algorithm
- ✓ Conditions:
  1. Flow<=Capacity
  2. Incoming == Outgoing
- ✓ Some improvements done by Edmonds-Karp. So sometimes called as Edmond-Karp algorithm.

Maximum Flow Problem

Residual Capacity:

- ✓ It is the capacity of the edge after subtracting the flow from the maximum capacity

Residual Graph:

- ✓ A graph with the same vertices and same edges, but we use the residual capacities as capacities.

Augmenting Path:

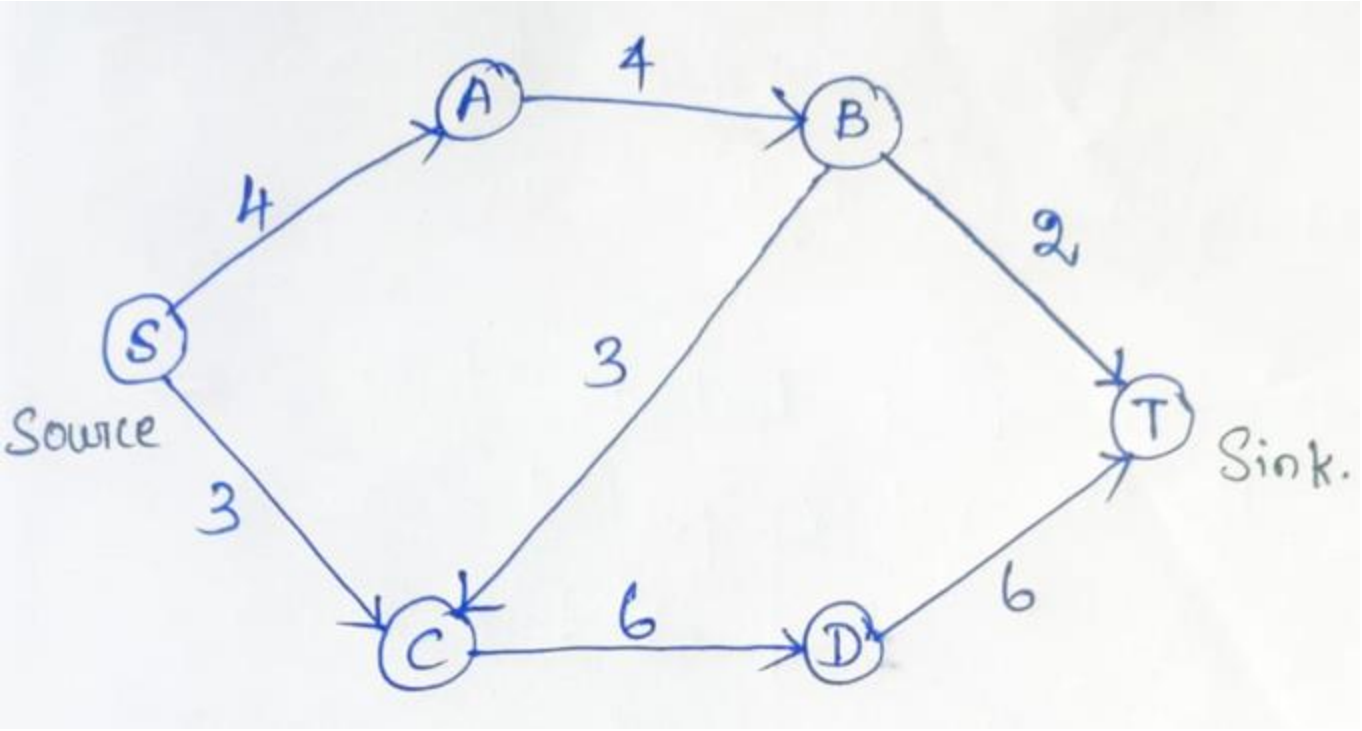
- ✓ An augmenting path is simple path in the residual graph., i.e., along the edges whose residual capacity is positive

Algorithm

```
max_flow = 0

For each edge (u,v) in G:
    flow(u,v) = w(u,v)
End For

While there is a PATH p from s→t in RG:
    min_cap = minimum residual capacity in path p
    max_flow = max_flow + min_cap
    For each edge (u,v) in p:
        flow(u,v) = flow(u,v) – min_cap
        flow(v,u) = flow(v,u) + min_cap
    End For
End While

Return max_flow
```

Note: Use BFS to find the best PATH in each iteration

Maximum Flow Problem



Source – S
Sink – T

Maximum Flow Problem

max_Flow = 0

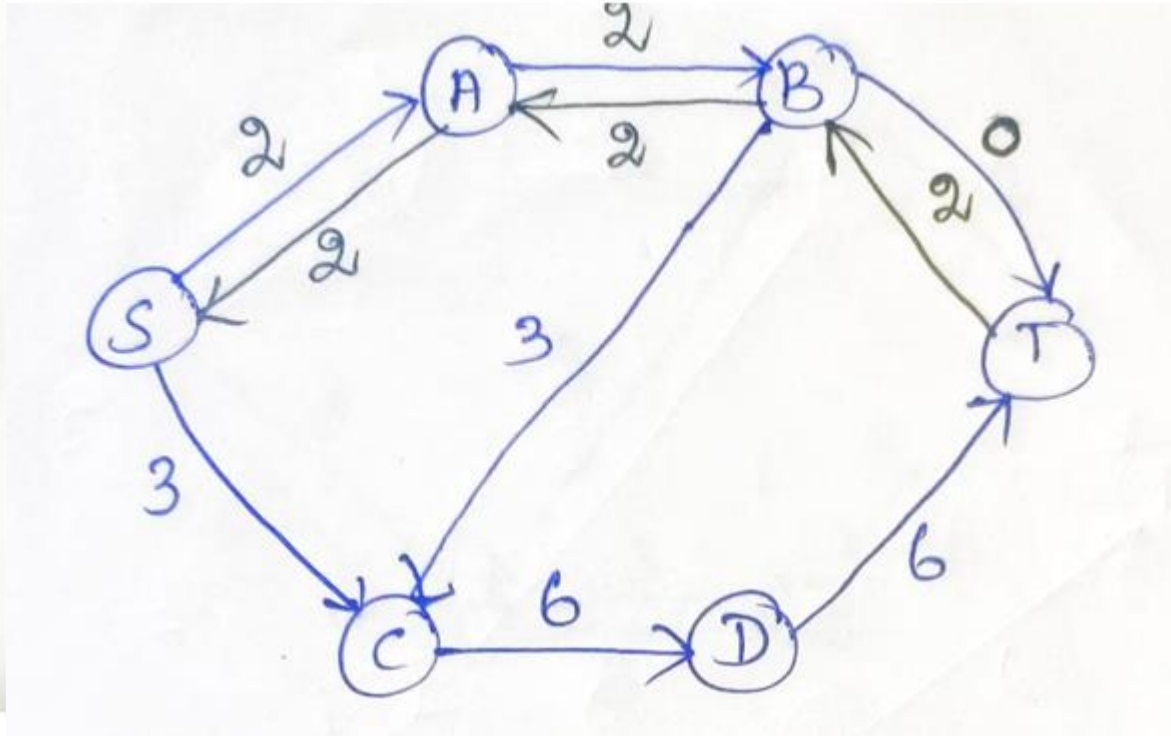
Select a Path from S to T:

PATH = S A B T

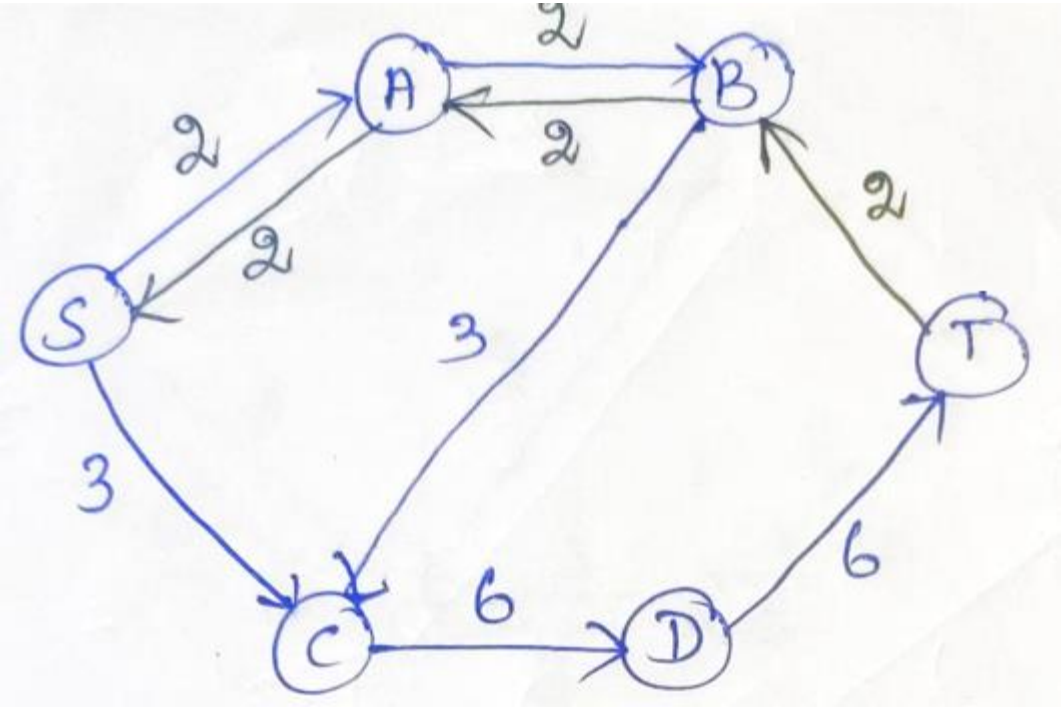
min_cap = min { c(S,A), c(A,B), c(B,T) }

= min { 4, 4, 2 }

min_cap = 2 ⇒ Update Path



Maximum Flow Problem



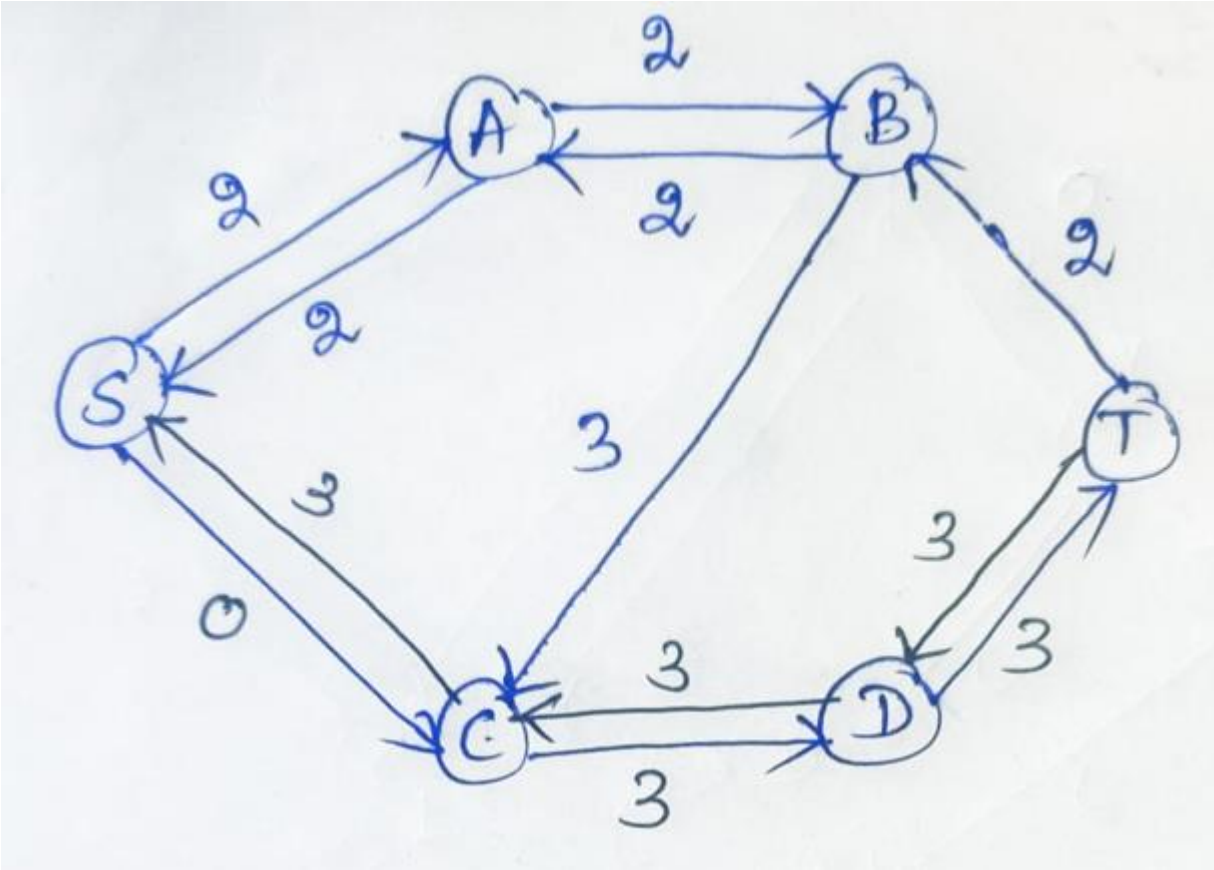
Select a Path from S to T:

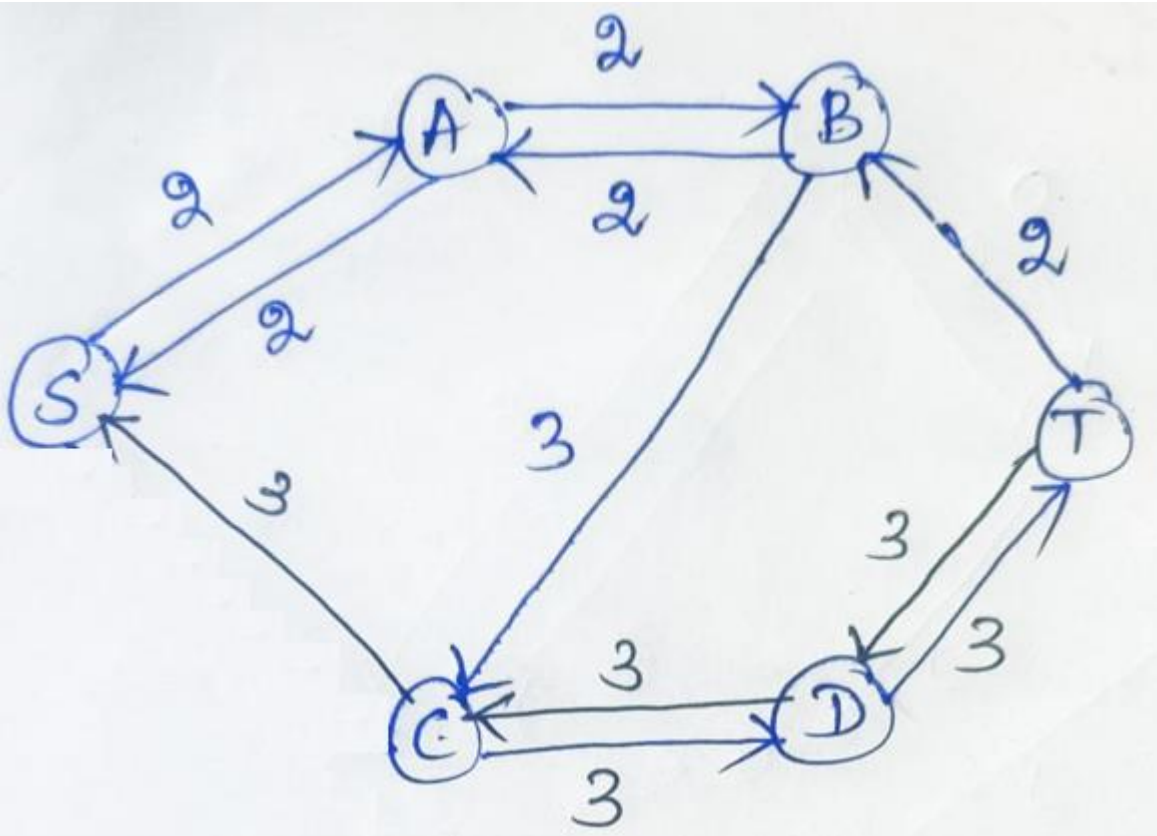
PATH : S C D T

min_cap = min{3, 6, 6}

min_cap = 3 ⇒ update Path

Maximum Flow Problem



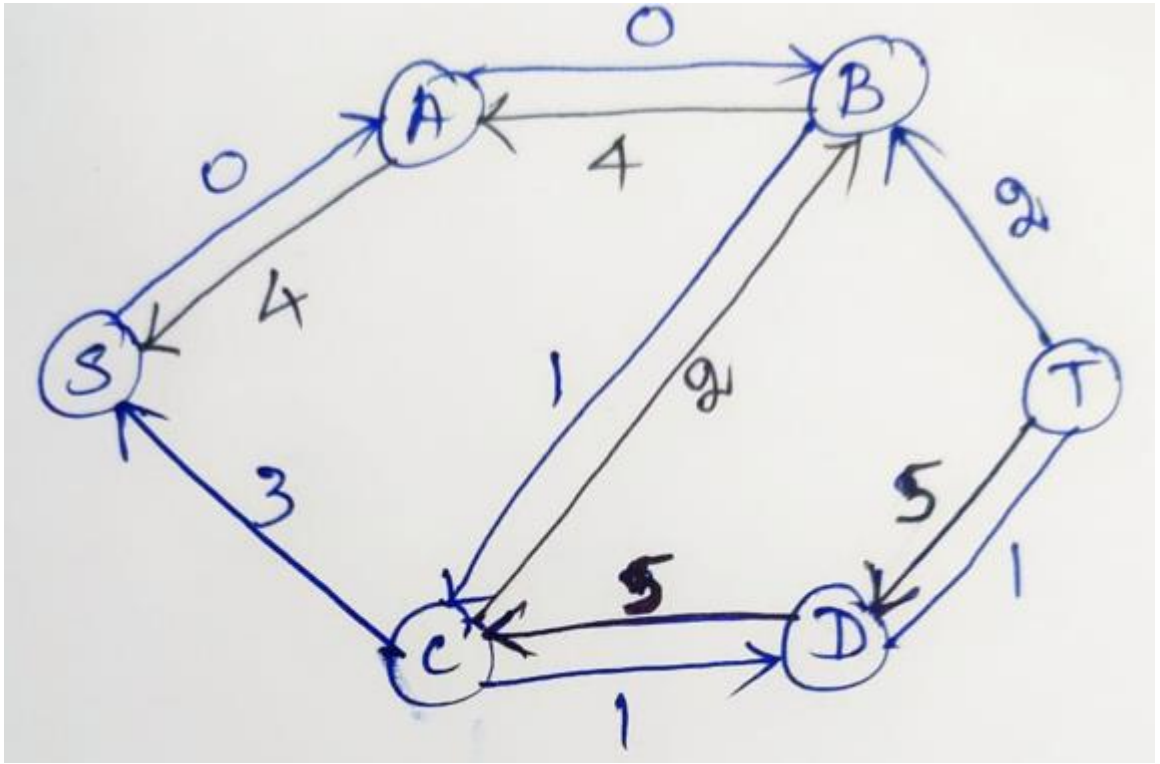


Maximum Flow Problem

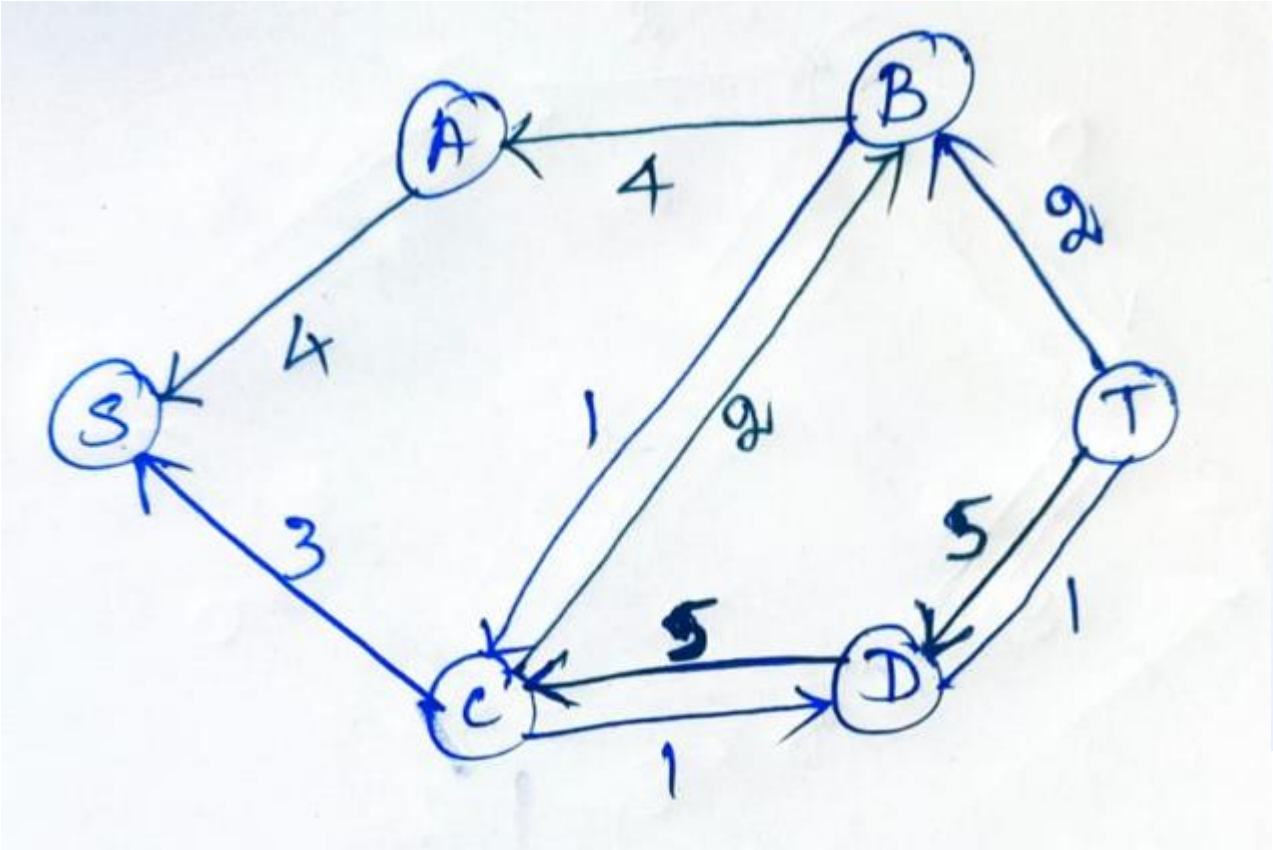
PATH : S A B C D T

min_cap = min{2, 2, 3, 3, 3}

min_cap = 2



Maximum Flow Problem



No More Paths Exists From S to T

Sum of all min-caps is Max-Flow.

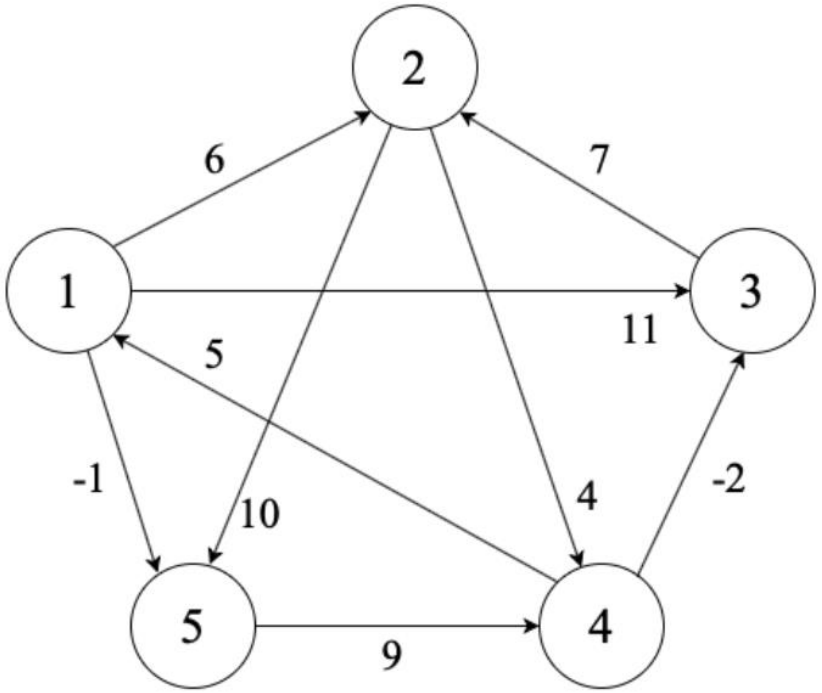
Max-Flow = 2 + 3 + 2

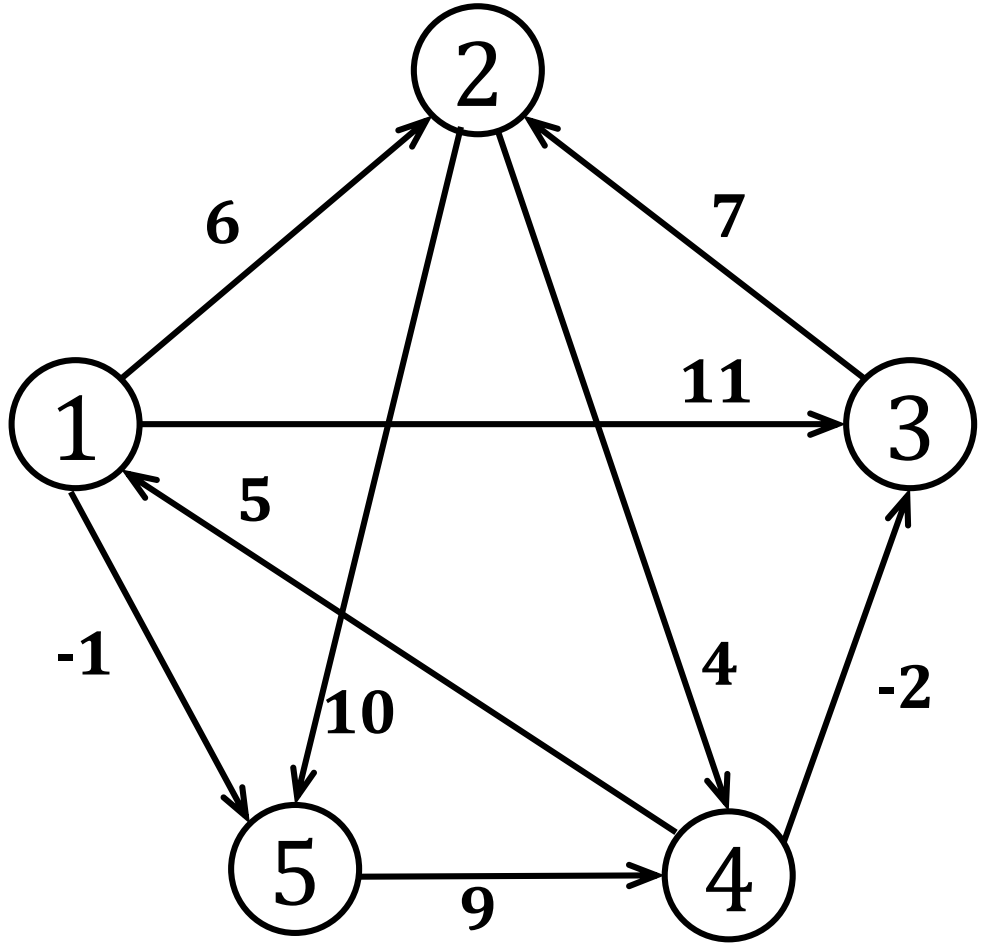
Max Flow = 8

Summary

- ✓ Breadth First Search (BFS) – Greedy Approach – O(V+E)
- ✓ Depth First Search (DFS) – Greedy Approach – O(V+E)
- ✓ Topological Sort - O(V+E)
- ✓ Prim's Algorithm – Greedy Approach – O((V+E)log V)
- ✓ Kruskal's Algorithm – Greedy Approach – O(E log V)
- ✓ Bellman-Ford Algorithm – Dynamic Programming – O(VE)
- ✓ Dijkstra's Algorithm – Greedy Approach - O((E + V) log V)
- ✓ Floyd-Wharshall Algorithm – Dynamic Programming – O(V3)
- ✓ Ford-Fulkerson Algorithm – Greedy Approach - O(max_flow * E)

Problem statement - Algorithm – Example (step-by-step tracing) – Complexity – Design strategy


2
6
7
1
3
11
5
-1
10
4
-2
5
9
4


2
6
7
1
3
11
5
-1
10
4
-2
5
9
4