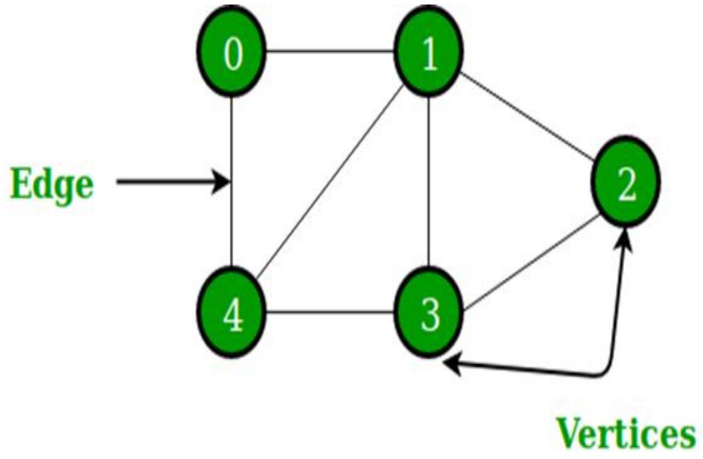
GRAPH ALGORITHMS

Submitted by:
Name: Yuvraj Singh Champawat
Roll : 181210064
Batch: CSE 2 year

Graph Algorithms

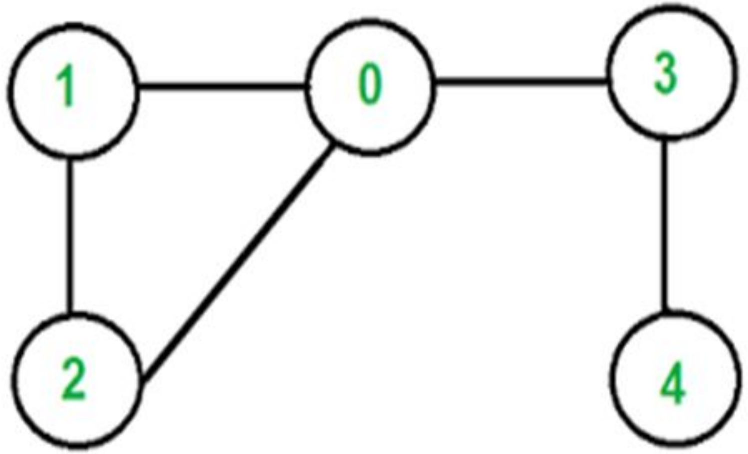
Graph is a non linear data structure that consists of a set of nodes (*vertices*) and a set of edges that relate the nodes to each other.

- The set of edges describes relationships among the vertices .
- A graph *G* is defined as follows: **G = (V, E)**
    - **V(G):** a finite, nonempty set of vertices
    - **E(G):** a set of edges (pairs of vertices)



Example:

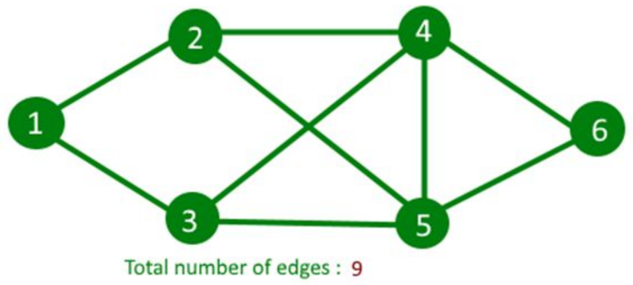
- V = {0, 1, 2, 3, 4}
- E = { (0, 1) , (1, 2) , (0, 3) , (0, 2) , (4, 3) }



Types of Graph:

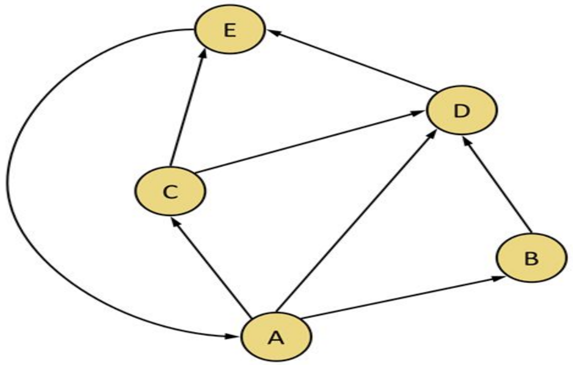
1. Undirected Graphs

An undirected graph is a graph that has no orientation. That is the edge (x, y) is identical to (y, x) i.e. they are not ordered pairs. The maximum number of edges possible in an undirected graphs are n*(n-1)/2 and the minimum number of edges in an undirected graphs are (n-1).



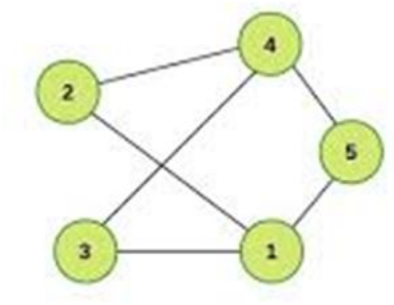
2. Directed Graphs

A graph with orientations. The edge pair (x, y) is not the same as (y, x) i.e. it is an ordered pair. The maximum number of edges in a directed graph is n*(n-1), (twice the number in undirected graphs).

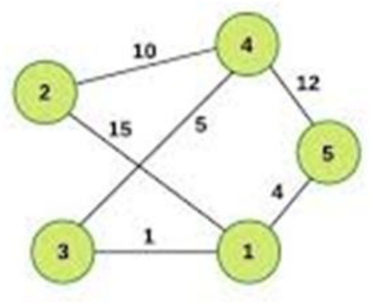


3. <u>Weighted & Unweighted graphs</u>

A weighted graph associates a value (weight) with every edge in the graph. Words, cost or length can also be used instead of weights.And, an unweighted graph does not associate any value (weight) with any edge of the graph.



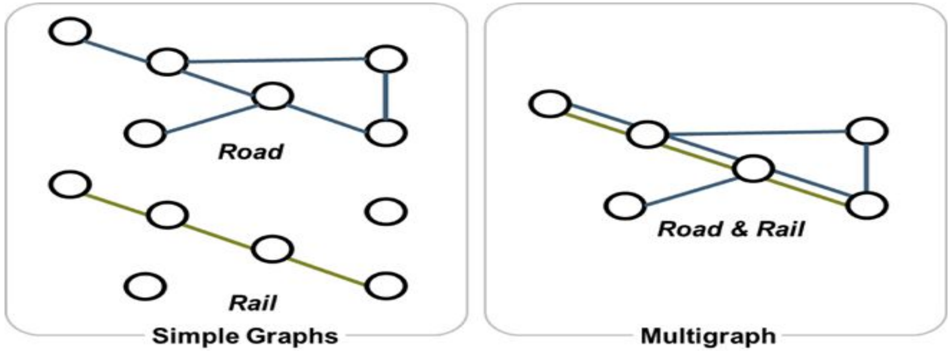
unweighted graph

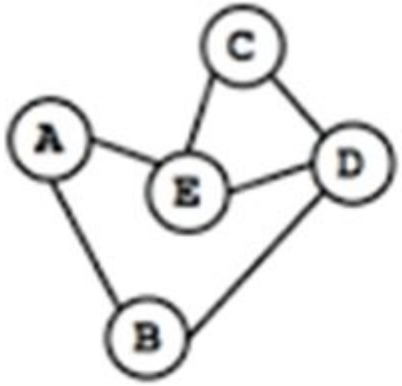


weighted graph

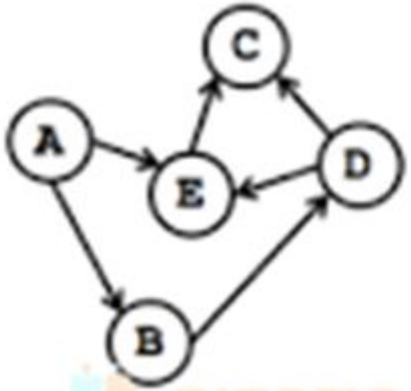
4. Multigraph & Simple Graph

A multigraph is an undirected graph in which multiple edges are allowed (sometimes) loops. Multiple edges refer to two or more edges that connect the same vertices. A simple graph as opposed to a multigraph, is an undirected graph in which both multiple edges and loops are disallowed.

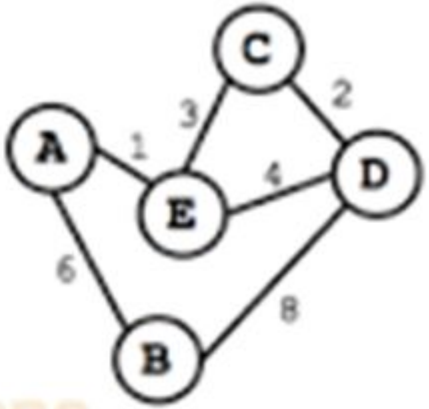




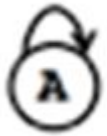
(A) Undirected Graph



(B) Directed Graph



(C) Weighted Graph

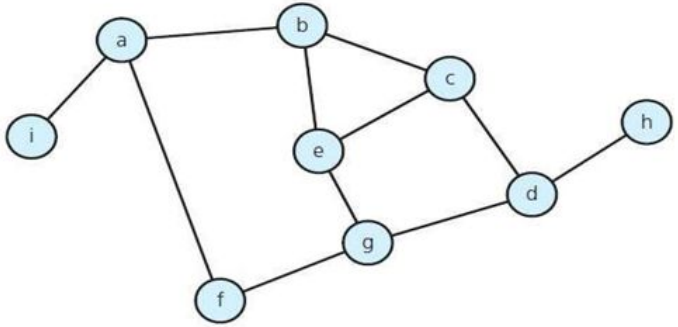


(D) Self-Loop

Number of edges - Undirected Graph

The no. of possible pairs of vertices in an n vertex graph is n*(n-1).

Since edge *(u, v)* is **the same** as edge *(v, u)*, the number of edges in an undirected graph is <= n*(n- 1)/2.

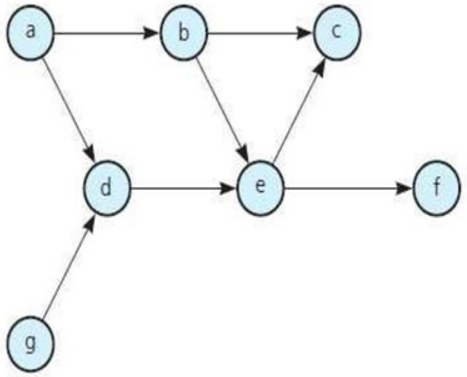


Number of edges - Directed Graph

The no. of possible pairs in an n vertex graph is n*(n-1).

Since edge *(u, v)* is **not the same** as edge *(v, u)*, the number of edges in a directed graph is n*(n-1)

Thus, the number of edges in a directed graph is **<= n*(n-1).**

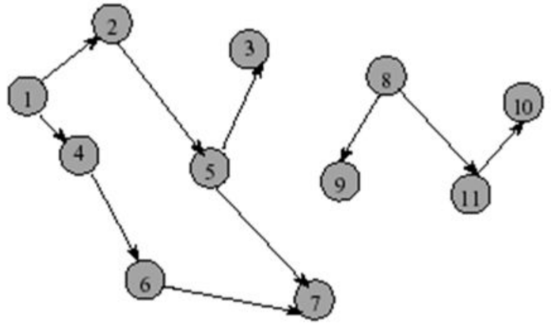


- In-degree of vertex *i* is the number of edges incident to *i* (i.e., the number of incoming edges).

  e.g., in-degree(2) = 1, in-degree(8) = 0

- Out-degree of vertex *i* is the number of edges incident from *i* (i.e., the number of outgoing edges).

  e.g., out-degree(2) = 1, out-degree(8) = 2



Representation of Graph:

Graph is a data structure that consists of following two components:

1. A finite set of vertices also called as nodes.

2. A finite set of ordered pair of the form (u, v) called as edge.

The pair is ordered because (u, v) is not same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

There are two standard ways to represent graphs, namely called as adjacency list and adjacency matrix.

- Adjacency List : The ***adjacency-list representation*** of a graph G = (V, E) consists of an array *Adj* of
- V lists, one of each vertex in V. For each u in V, the adjacency list *Adj[u]* contains all the
- vertices such that there is an edge u in E. That is, *Adj[u]* consists of all the vertices
- adjacent to u in G.
-
- Adjacency Matrix : The ***adjacency-matrix representation*** of a graph G consists of a V×V matrix A_{ij}
- such that :

A_{ij} = 1, if (i, j) ϵ E

0, otherwise

Adjacency Matrix:

- A square grid of boolean (1/0) values.
- If the graph contains N vertices, then the grid contains **N rows and N columns**.
- For two vertices numbered i and j, the element at row i and column j is true(1), if there is an edge from i to j, otherwise false(0).

Adjacency List:

- A graph of n nodes is represented by a one- dimensional array (L) of linked lists, where L[i] is the linked list containing all the nodes adjacent from node i. The nodes in the list L[i] are in no particular order.

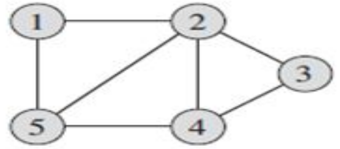
->Adjacency list : for each vertex $v \in V$, store a list of vertices adjacent to *v.*

<u>How much storage is required?</u>

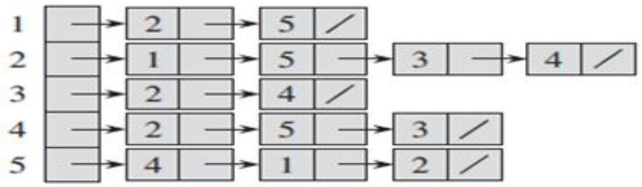
- The *degree* of a vertex v = # incident edges
- (*Directed graphs have in-degree, out-degree*)

  For directed graphs, # of items in adjacency lists is Σ out-degree(v) = |E|

  For undirected graphs, # items in adjacency lists is Σ degree(v) = 2* |E|
- So: Adjacency lists take O(V+E) storage



(a)

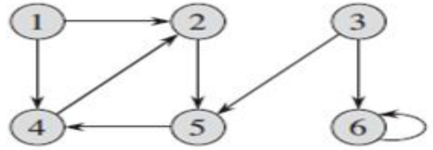


(b)

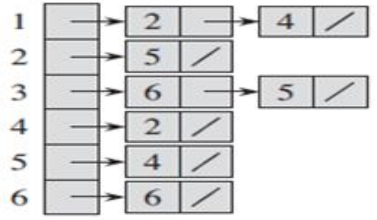
$$\begin{array}{c|ccccc}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 1 & 1 & 0 & 1 & 0
\end{array}$$

(c)

Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G with 5 vertices and 7 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.



(a)



(b)

$$\begin{array}{c|cccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 \\
4 & 0 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}$$

(c)

Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Graph Search Algorithm:

•Given a Graph G = (V, E) directed or undirected. The goal is to methodically explore every vertex and every edge.

There are two well known ***graph search algorithms*** :

1. Breadth First Search
2. Depth First Search

Breadth First Search:

•**Breadth-first search** (**BFS**) is an algorithm for traversing or searching tree or graph data structures. We need to know two important terminology to understand the process of BFS traversal.

Visiting a Vertex : This mean we are currently present on thatvertex in the graph or tree.

Exploration of a Vertex : This refers to visiting all the adjacent vertices of the source vertex or the visiting vertex.

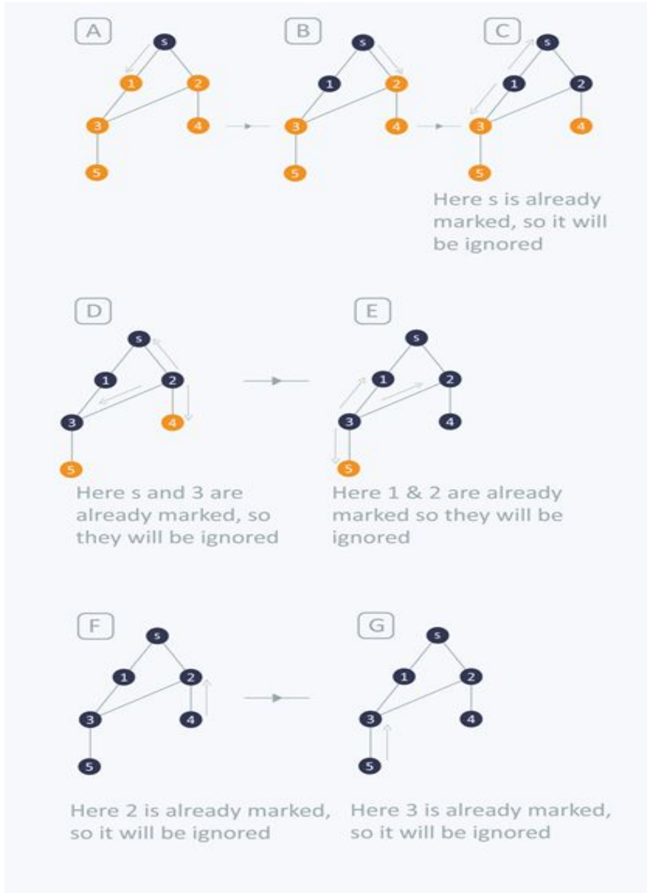
•BREADTH-FIRST TRAVERSAL (BFS) :

->The first step is to pick any vertex, that becomes are start vertex.

->Next step is to explore the current vertex we are on and explore all the adjacent vertices to it.

->We repeat the above two steps until all the vertices of the graph are visited at least once.

->We have explained the above steps using an example. Note that the vertex marked with dark colour are the visited or explored vertices.

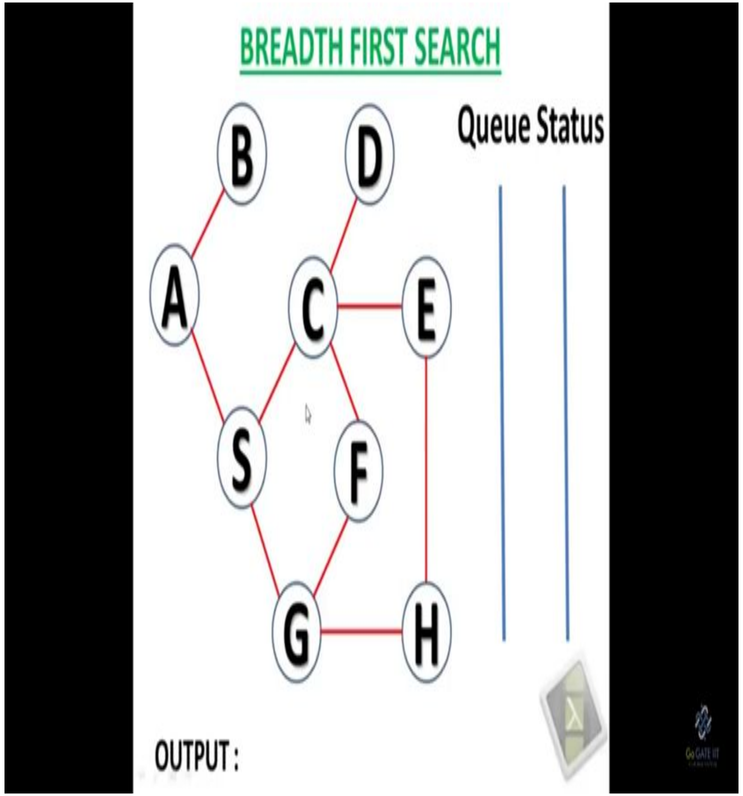

A
B
C
Here s is already marked, so it will be ignored
D
E
Here s and 3 are already marked, so they will be ignored
Here 1 & 2 are already marked so they will be ignored
F
G
Here 2 is already marked, so it will be ignored
Here 3 is already marked, so it will be ignored

•Like in previous example, in order to keep track of our BFS traversal we will use different colour i.e. white, grey and black, to denote vertices with different attributes.

Ø All vertices start out white. ***White represents un-visited or un-explored vertices.***

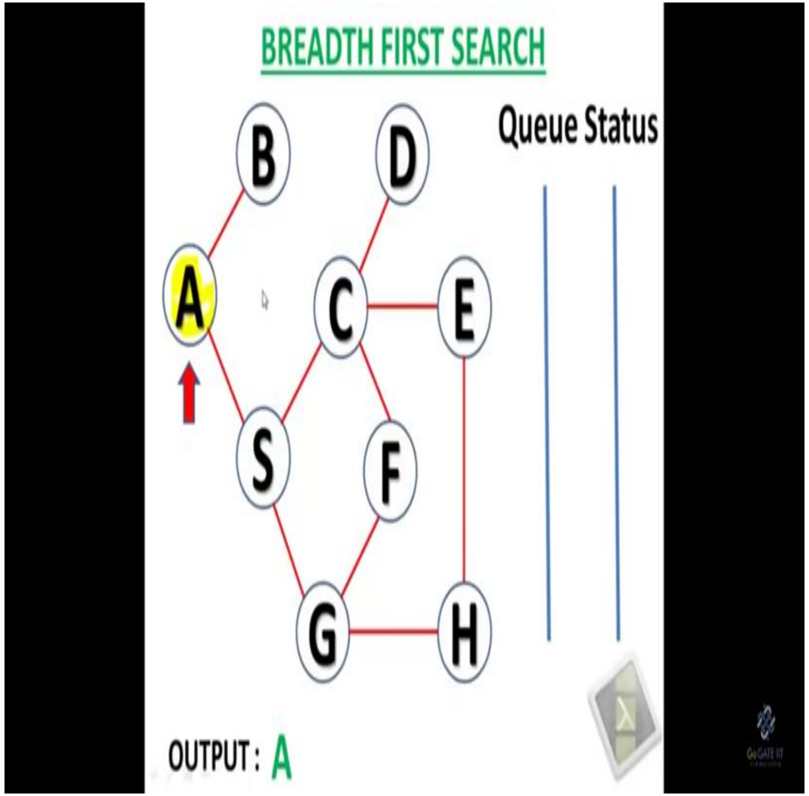
Ø A vertex becomes non-white if it discovered the first time. We associate grey for such vertices. ***Grey represents the visited or first-discovered vertices.***

ØA vertex becomes black if it is explored and we are required to begin the exploration of it's adjacent visited vertices. ***Black represents the vertices that are explored.***


BREADTH FIRST SEARCH

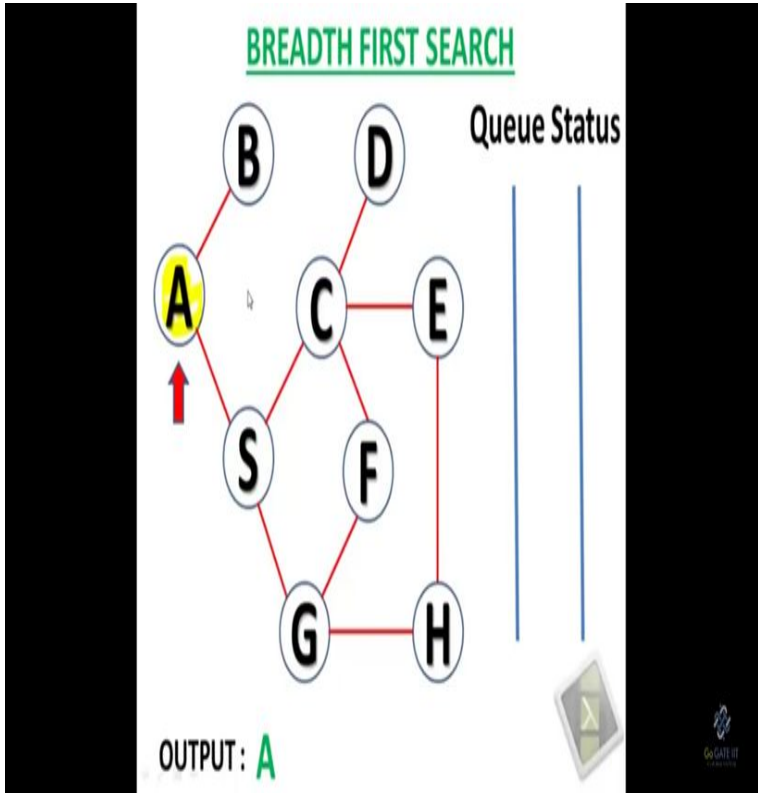
Queue Status

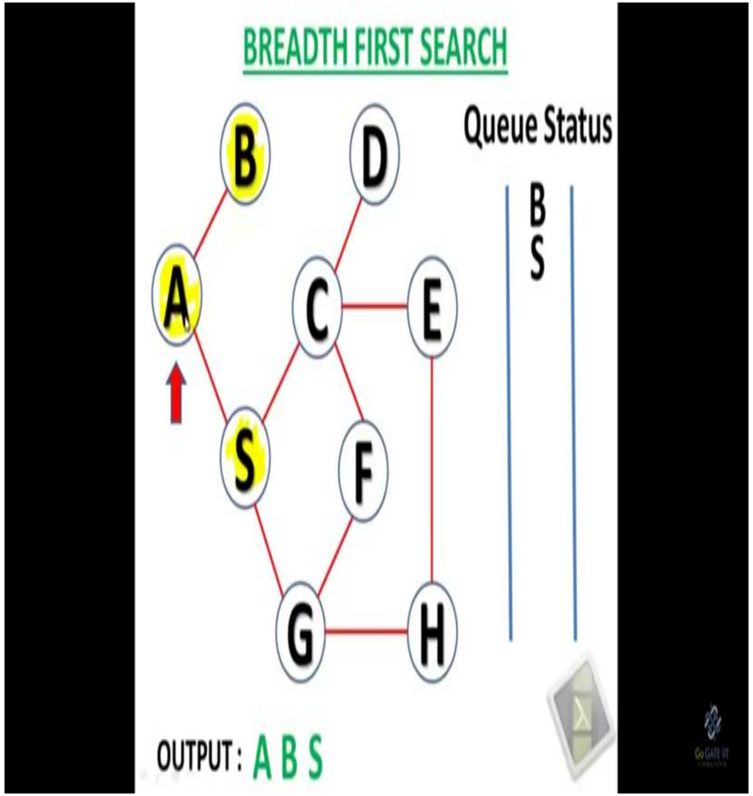
OUTPUT :

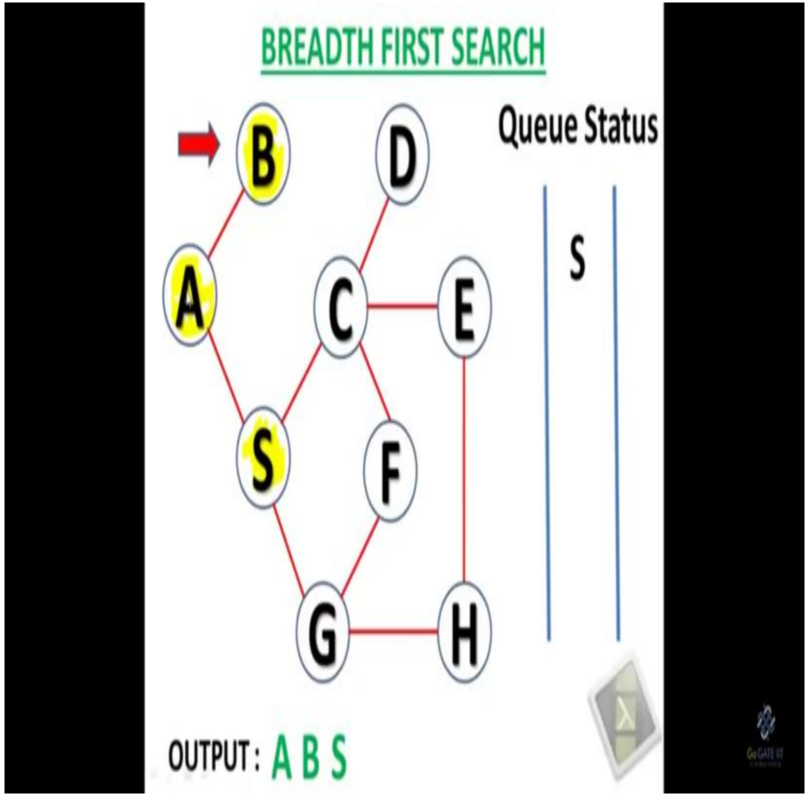

BREADTH FIRST SEARCH

Queue Status

OUTPUT : A


BREADTH FIRST SEARCH
Queue Status
B
D
A
C
E
S
F
G
H
OUTPUT : A

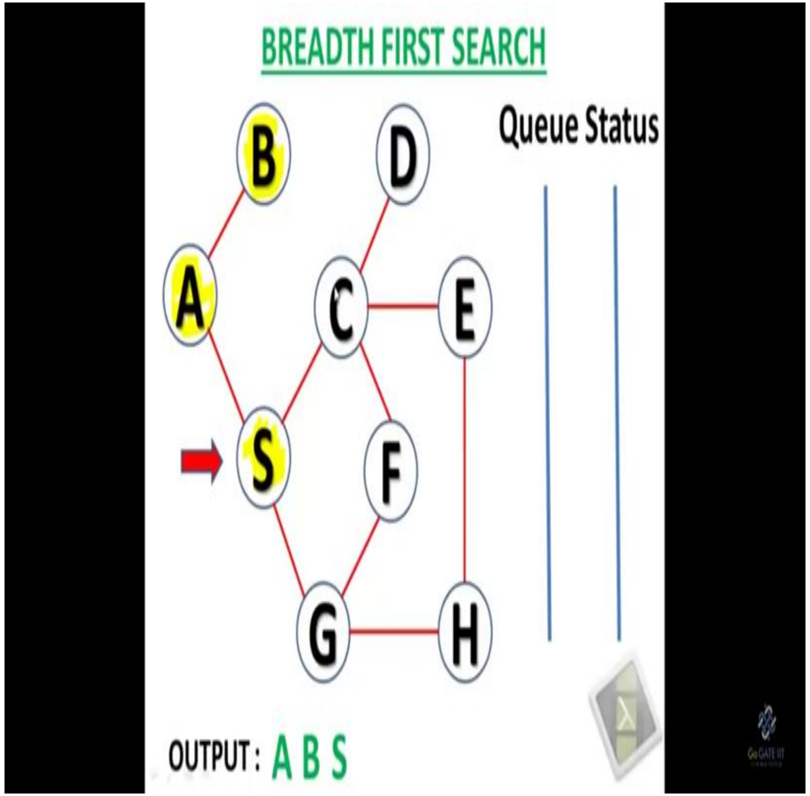

BREADTH FIRST SEARCH
Queue Status
B
S
B
D
A
C
E
S
F
G
H
OUTPUT : A B S


BREADTH FIRST SEARCH

Queue Status

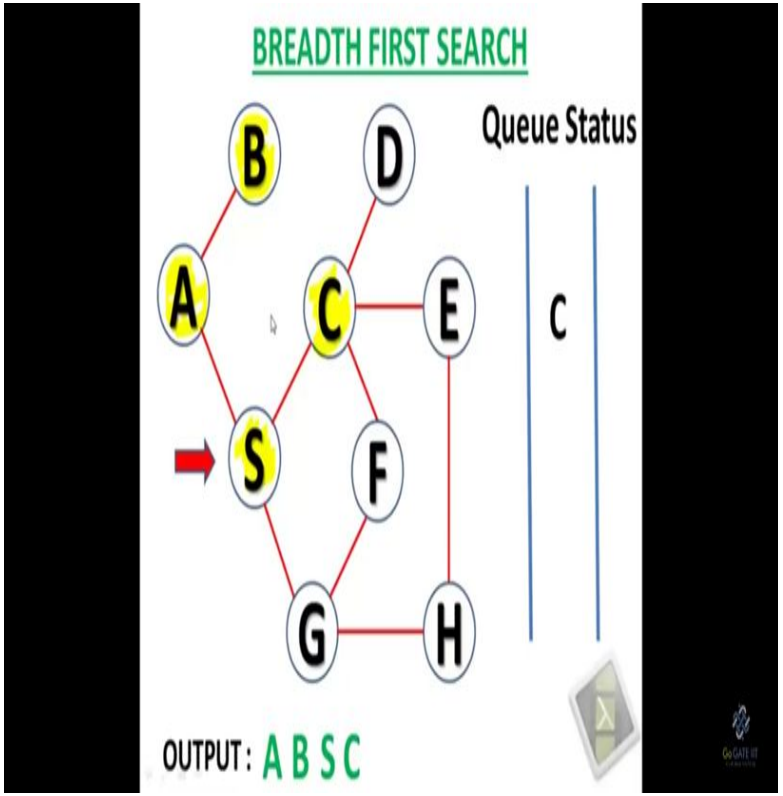
S

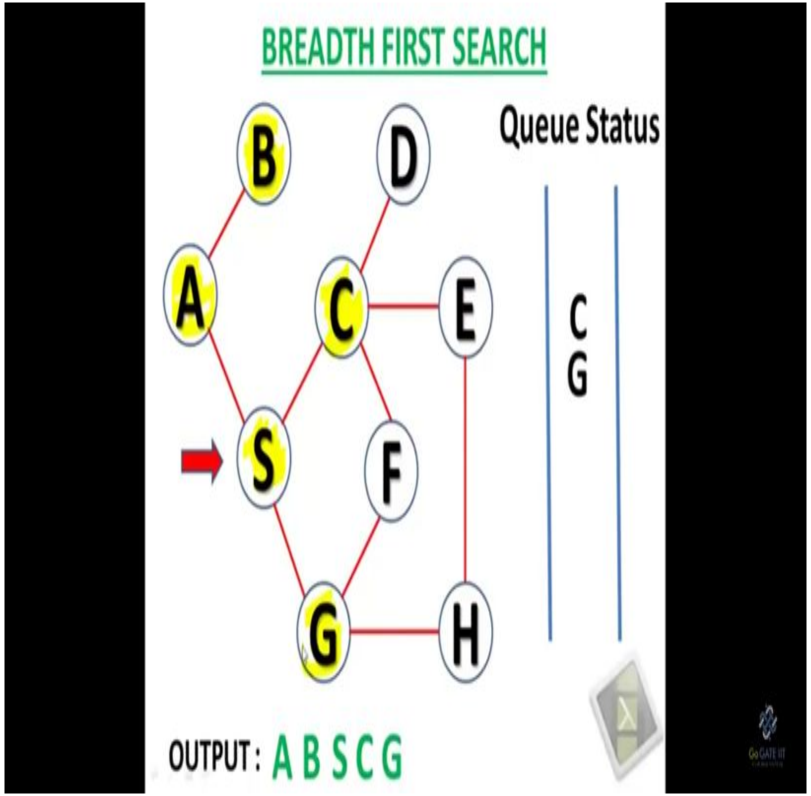
OUTPUT : A B S

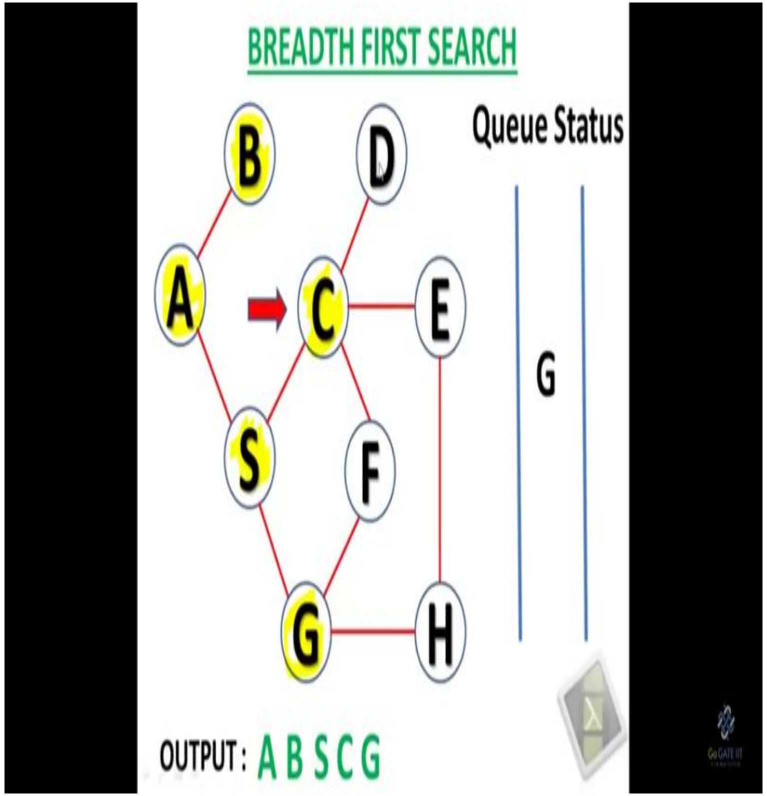

BREADTH FIRST SEARCH

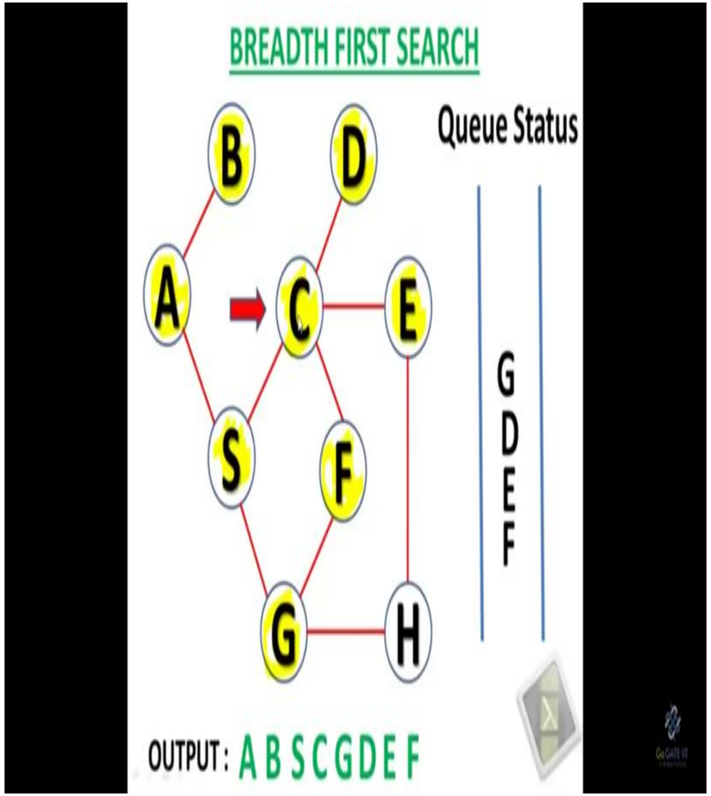
Queue Status

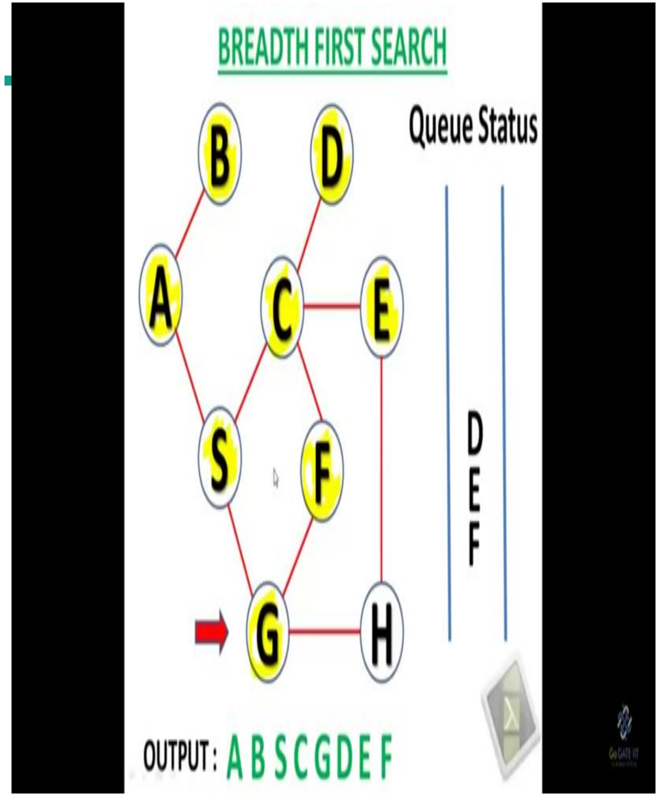
OUTPUT : A B S


BREADTH FIRST SEARCH
Queue Status
C
A
B
S
C
D
E
F
G
H
OUTPUT : A B S C

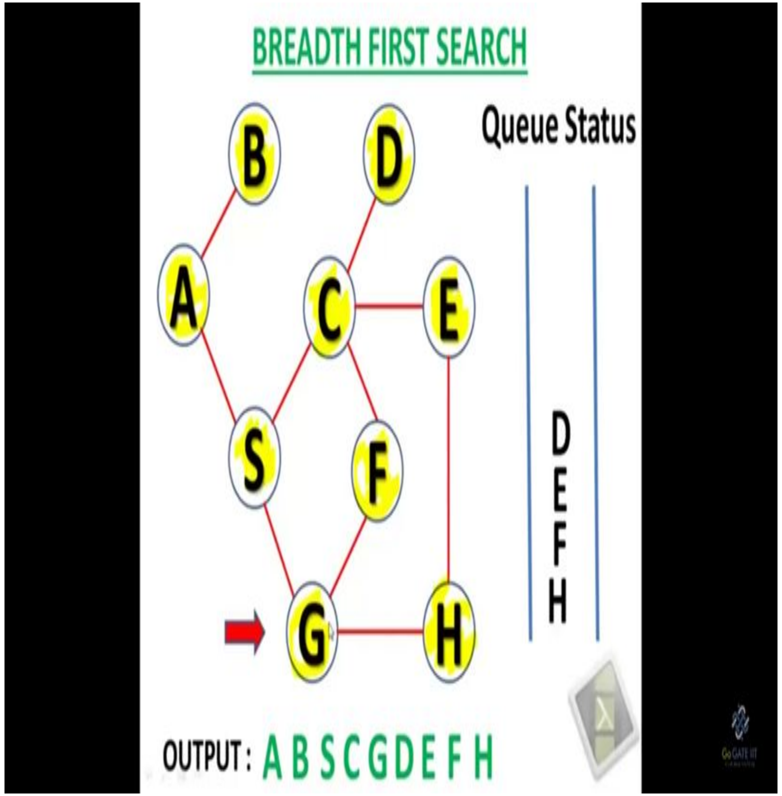

BREADTH FIRST SEARCH
Queue Status
C
G
A
B
S
C
D
E
F
G
H
OUTPUT : A B S C G


BREADTH FIRST SEARCH
Queue Status
G
OUTPUT : A B S C G

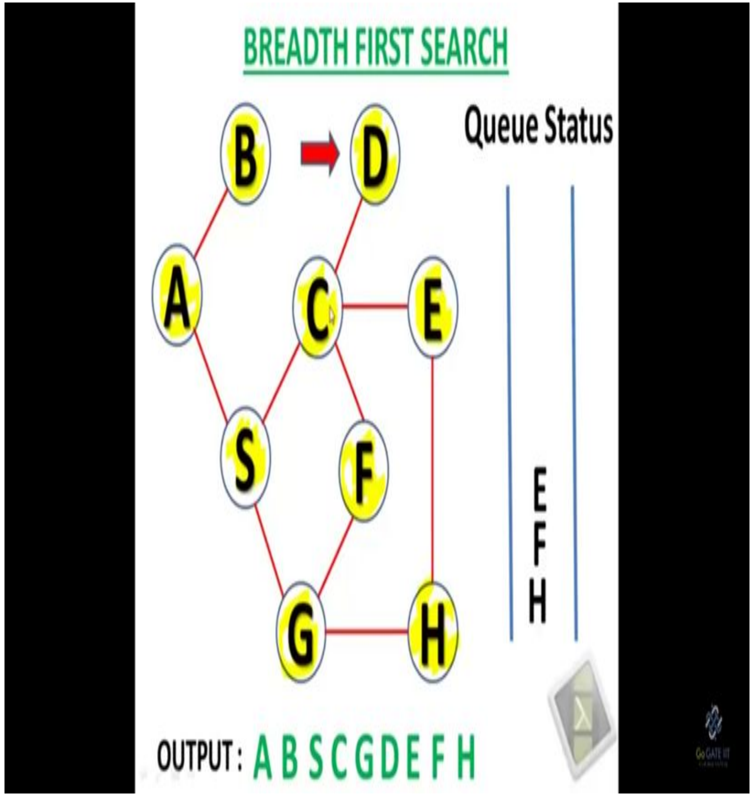

BREADTH FIRST SEARCH
Queue Status
G
D
E
F
OUTPUT : A B S C G D E F



BREADTH FIRST SEARCH
Queue Status
B
D
A
C
E
S
F
G
H
D
E
F
OUTPUT : A B S C G D E F

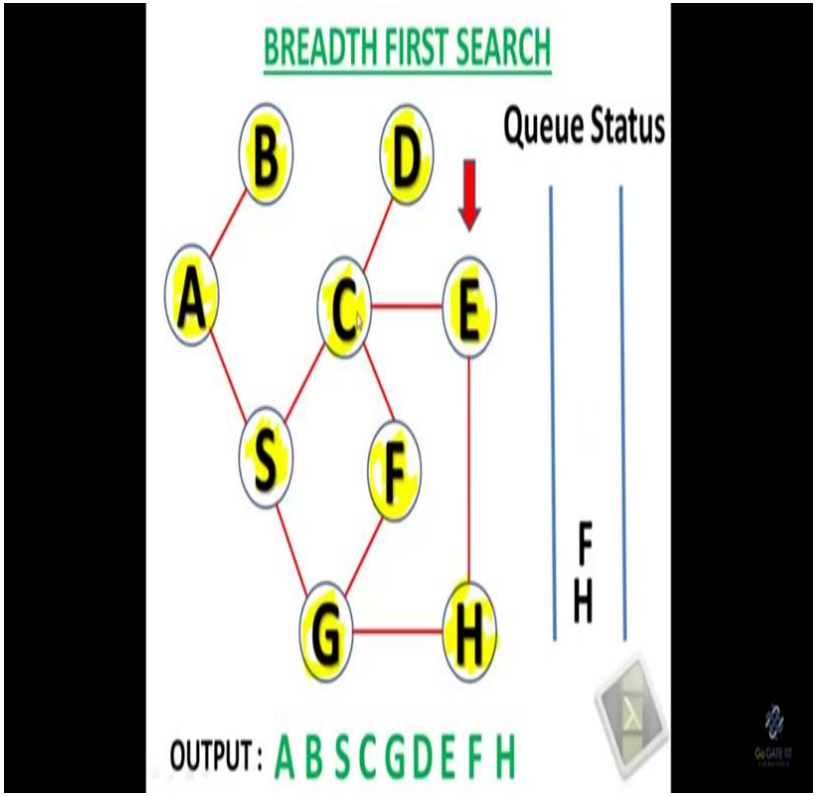


BREADTH FIRST SEARCH
Queue Status
B
D
A
C
E
S
F
G
H
D
E
F
H
OUTPUT : A B S C G D E F H



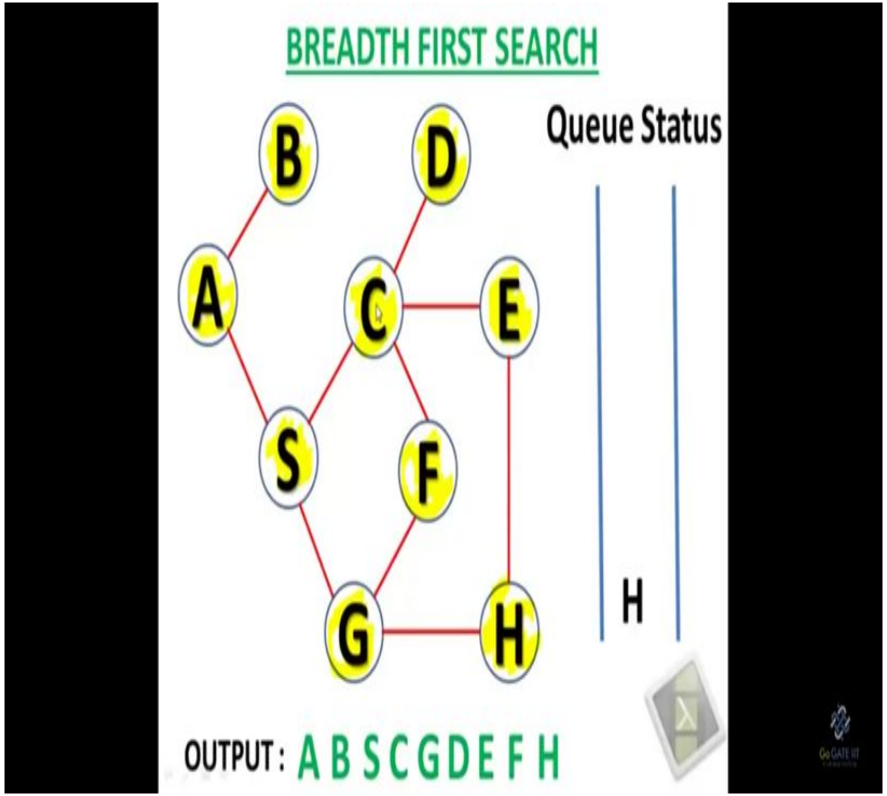
BREADTH FIRST SEARCH

OUTPUT : A B S C G D E F H

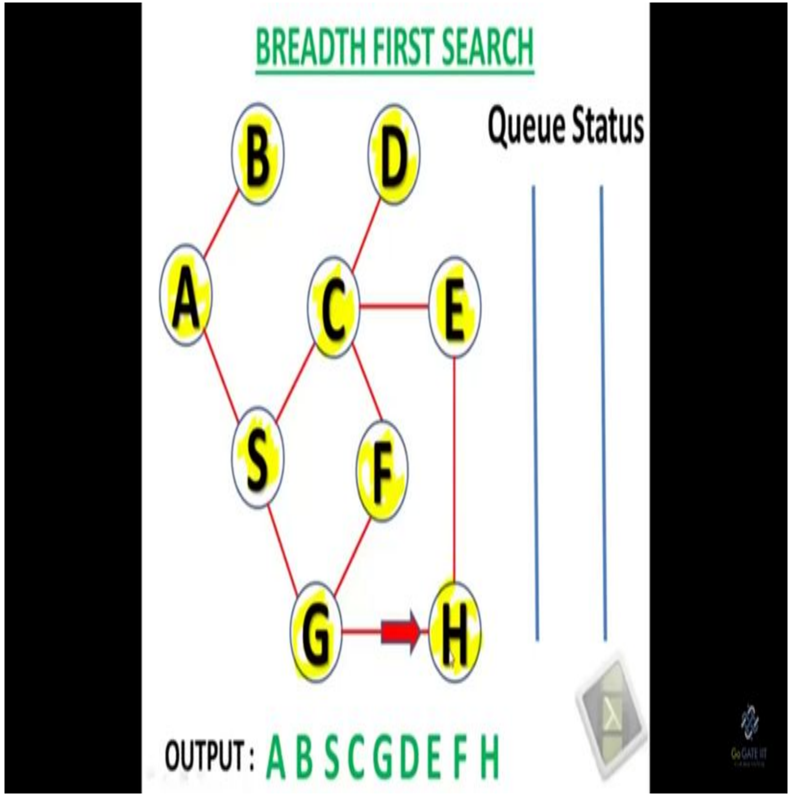


BREADTH FIRST SEARCH

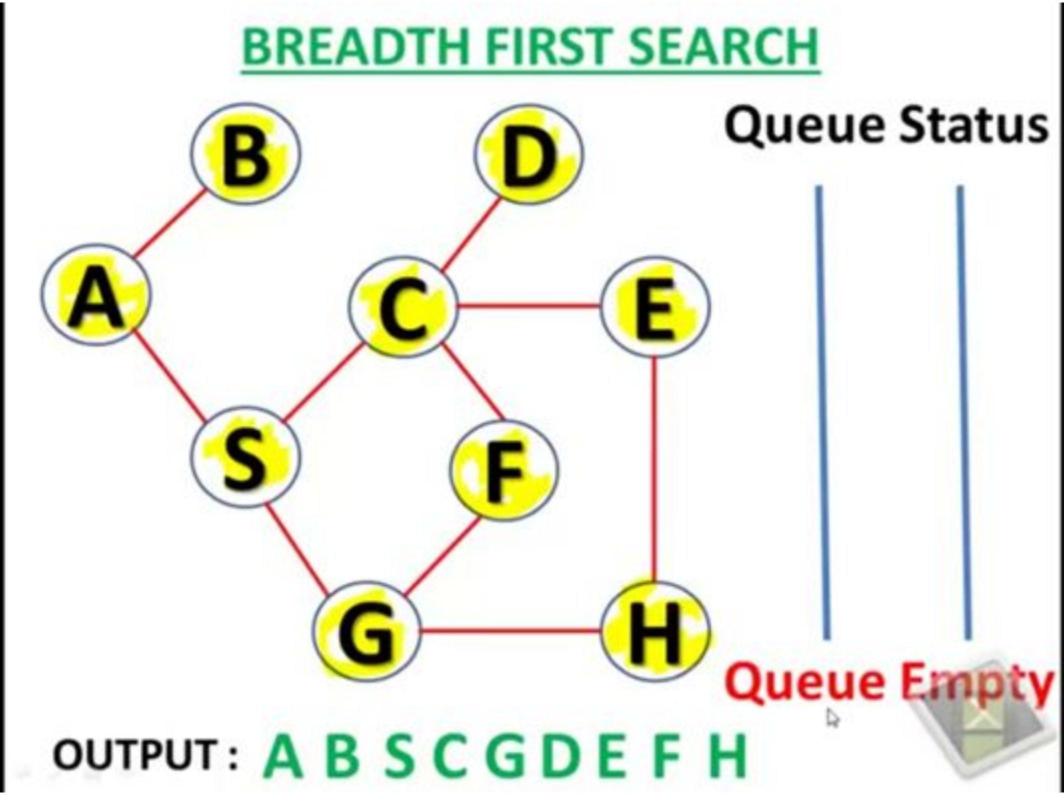
OUTPUT : A B S C G D E F H



OUTPUT : A B S C G D E F H



OUTPUT : A B S C G D E F H



BREADTH FIRST SEARCH

OUTPUT : A B S C G D E F H

Algorithm:

- The breadth-first search procedure assumes that the input graph

  G = (V, E) is represented using adjacency lists.
- We store the colour of each vertex u ϵ V in the attribute ***u.color*** and the predecessor of u in the attribute in ***u. π.***
- If u has no predecessor then ***u.π*** **= NIL.**
- The attribute ***u.d*** holds the distance from the source s to vertex u.
- The algorithm also uses first-in, first-out queue ***Q.***

```
BFS$(G, s)$
 1  for each vertex $u \in G.V - \{s\}$
 2      $u.color$ = WHITE
 3      $u.d = \infty$
 4      $u.\pi$ = NIL
 5  $s.color$ = GRAY
 6  $s.d = 0$
 7  $s.\pi$ = NIL
 8  $Q = \emptyset$
 9  ENQUEUE$(Q, s)$
10  while $Q \neq \emptyset$
11      $u$ = DEQUEUE$(Q)$
12      for each $v \in G.Adj[u]$
13          if $v.color$ == WHITE
14              $v.color$ = GRAY
15              $v.d = u.d + 1$
16              $v.\pi = u$
17              ENQUEUE$(Q, v)$
18      $u.color$ = BLACK
```

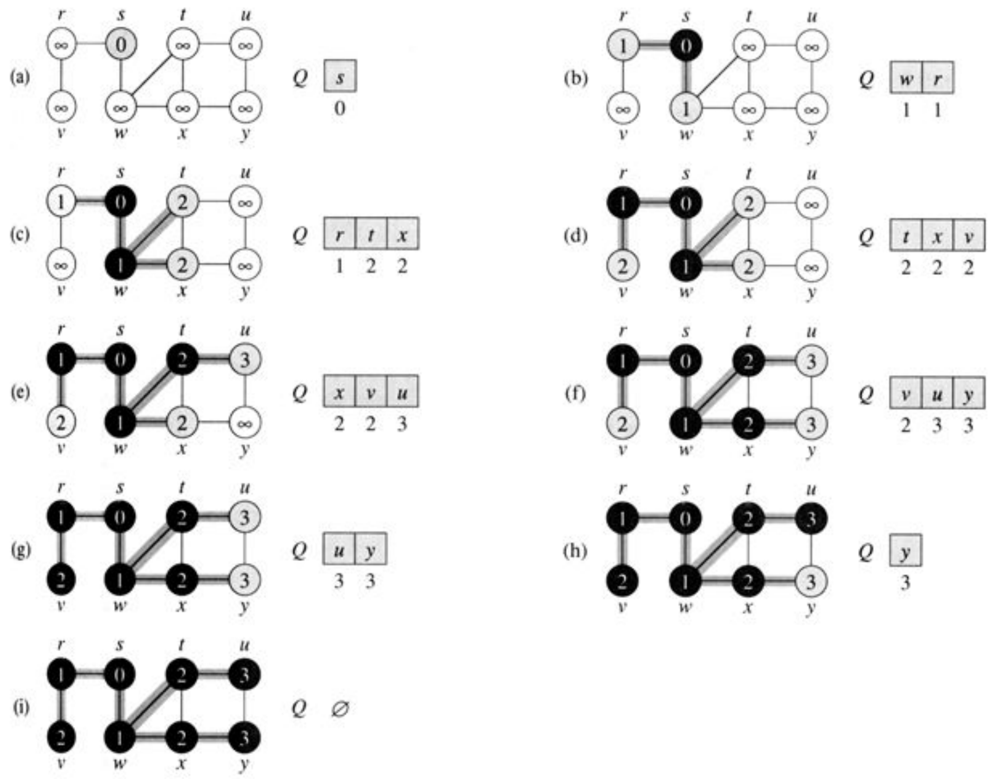


Figure 23.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 9–18. Vertex distances are shown next to vertices in the queue.

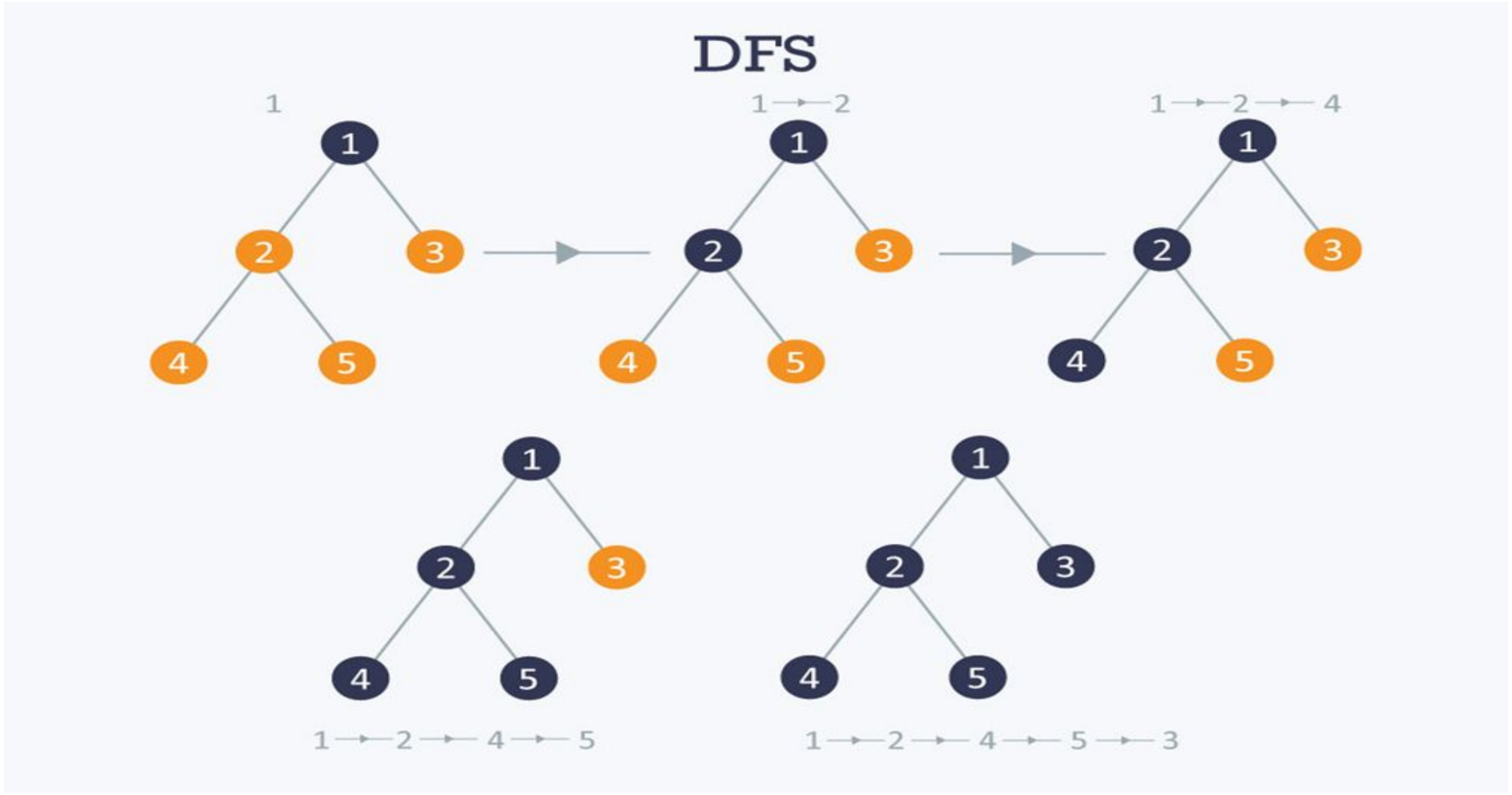
Depth First Search Algorithm:

•The strategy followed by depth-first search is, as its name implies, to search ***deeper*** in the graph whenever possible.

•DEPTH-FIRST SEARCH TRAVERSAL (DFS):

Ø Depth-first search explores edges out of the most recently discovered vertex v that still has un-explored edges leaving it.

Ø Once all of v’s edges have been explored, the search “backtracks” to explore edges leaving the vertex from which v was discovered.



DFS
1
1 → 2
1 → 2 → 4
1 → 2 → 4 → 5
1 → 2 → 4 → 5 → 3
1
2
3
4
5

Algorithm:

•Whenever depth-first search discovers a vertex v during a scan of the adjacency list of an already discovered vertex u, it records this event by setting v's predecessor attribute ***v.π*** to u.

•The ***predecessor subgraph*** of a depth-first search slightly differently from that of a breadth-first search :

$$G_\pi = (V, E_\pi)$$

$$\text{where , } E_\pi = \{(v.\pi, v) : v \in V \text{ and } v.\pi \neq \text{NIL}\}$$

•Depth-first search also uses colours to mark the vertices with different attributes.

Ø Each vertex is initially white. ***White is used to represent the vertex which is un-visited and un-explored.***

Ø When the vertex is discovered and visited it's marked non-white. ***Grey represents the vertex that is discovered or visited.***

Ø When all the adjacent vertices of the current discovered vertices are visited, the colour changes to black. ***Black is used to determine the exploration of vertex is completed or in other words, when the adjacency list of the vertex is completed.***

```
DFS(G)
1  for each vertex $u \in G.V$
2      $u.color$ = WHITE
3      $u.\pi$ = NIL
4  $time = 0$
5  for each vertex $u \in G.V$
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 $v \in G.Adj[u]$            // explore edge $(u, v)$
 5      if $v.color$ == WHITE
 6          $v.\pi = u$
 7          DFS-VISIT$(G, v)$
 8  $u.color$ = BLACK                    // blacken $u$; it is finished
 9  $time = time + 1$
10  $u.f = time$
```

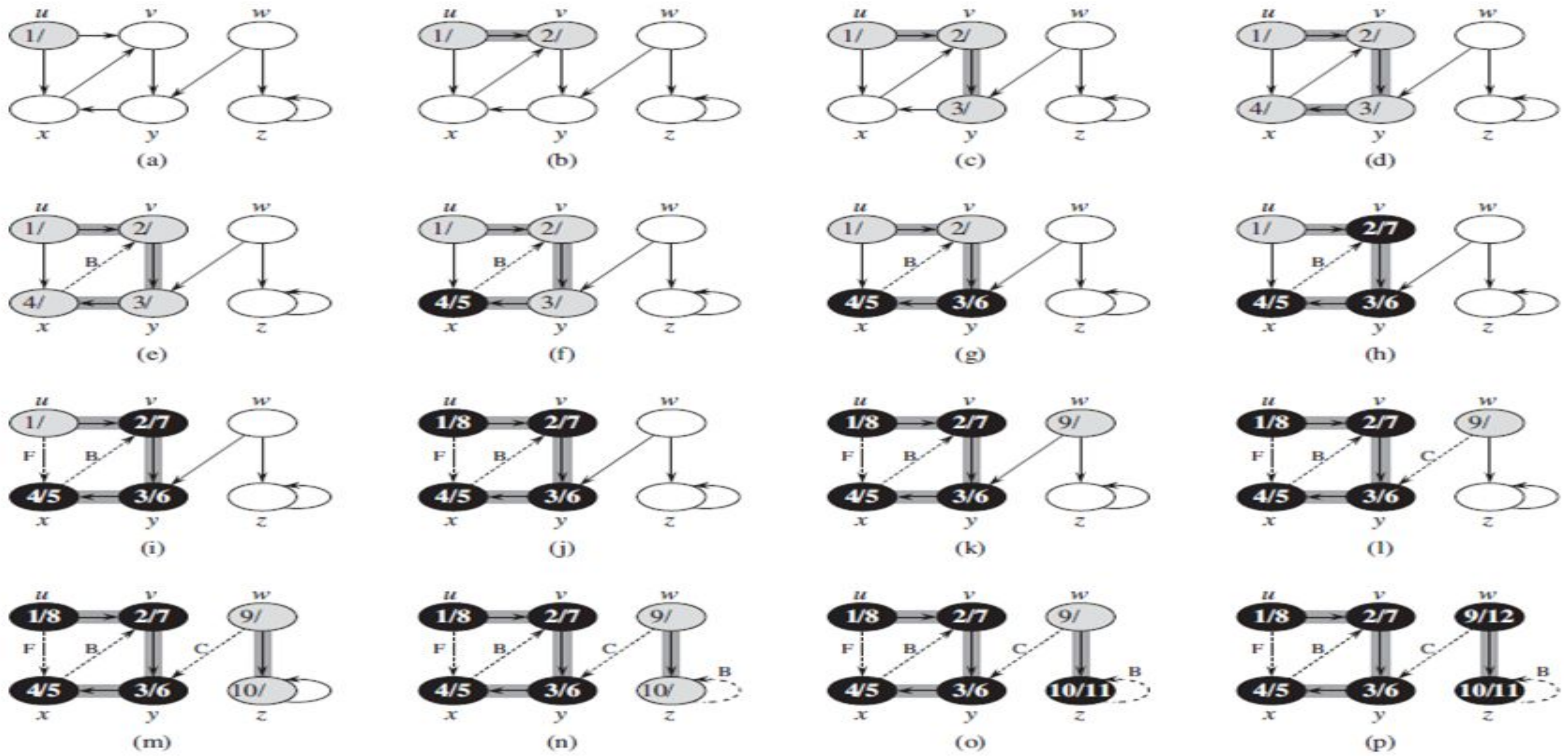
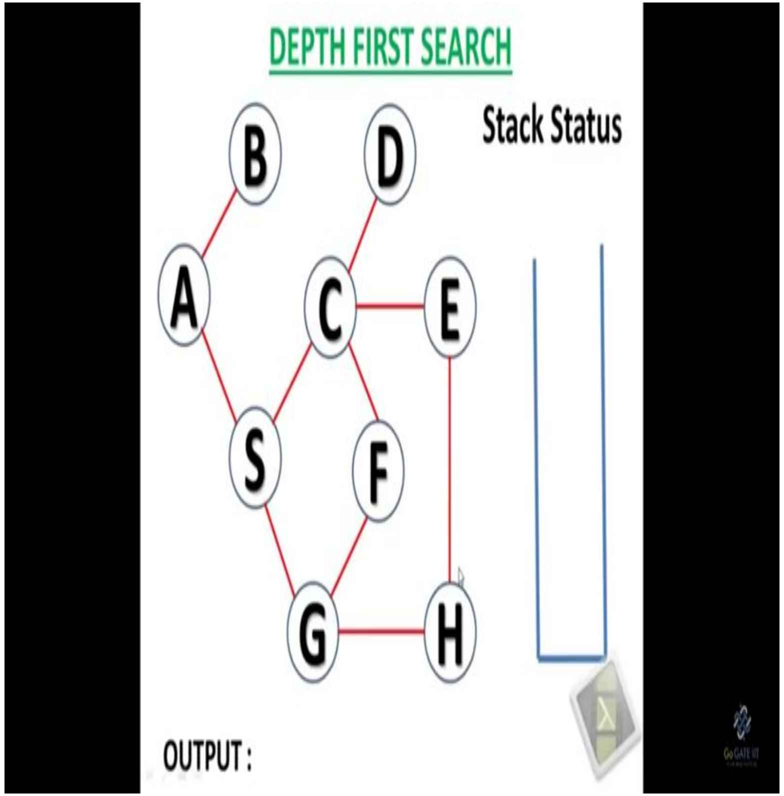
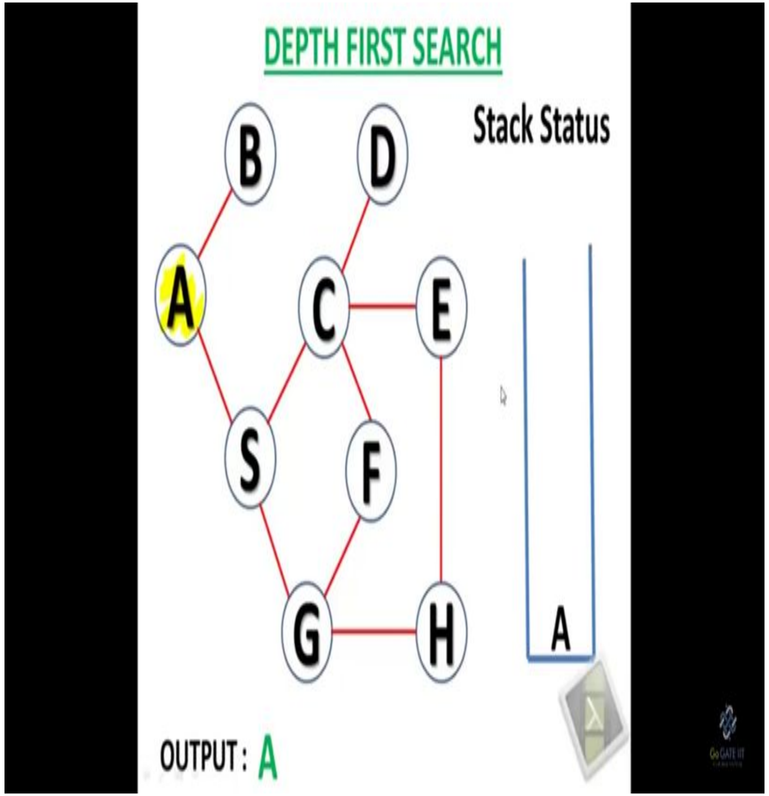
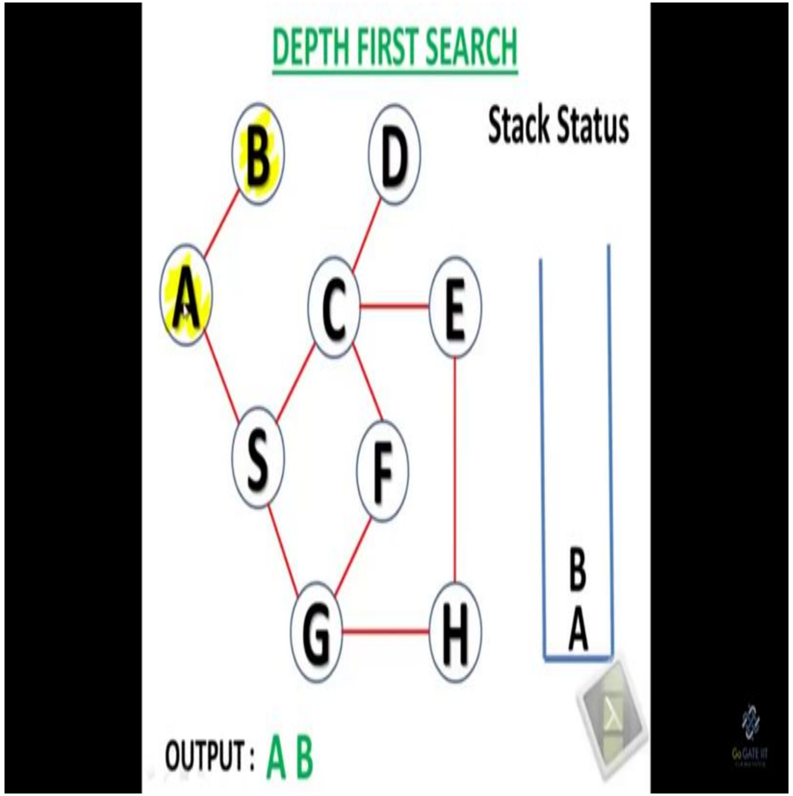


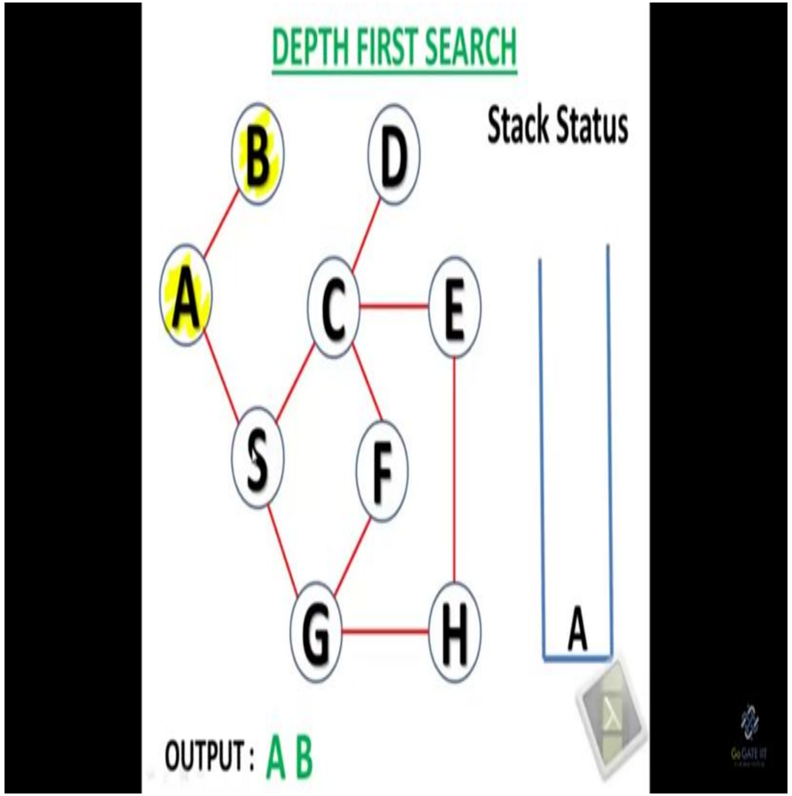
Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Timestamps within vertices indicate discovery time/finishing times.


DEPTH FIRST SEARCH
Stack Status
B
D
A
C
E
S
F
G
H
OUTPUT :

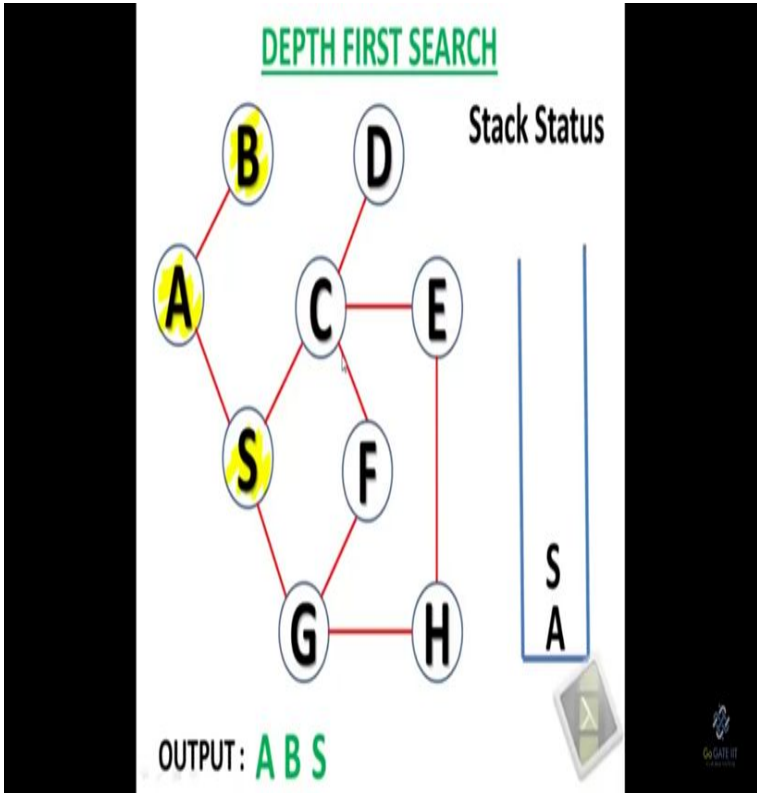

DEPTH FIRST SEARCH
Stack Status
B
D
A
C
E
S
F
G
H
A
OUTPUT : A

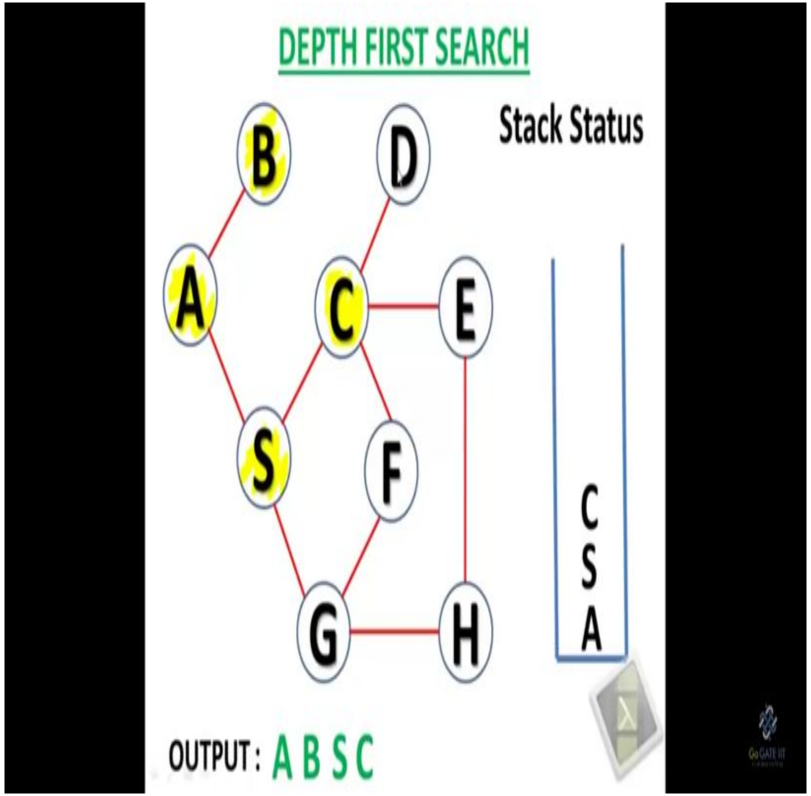

DEPTH FIRST SEARCH

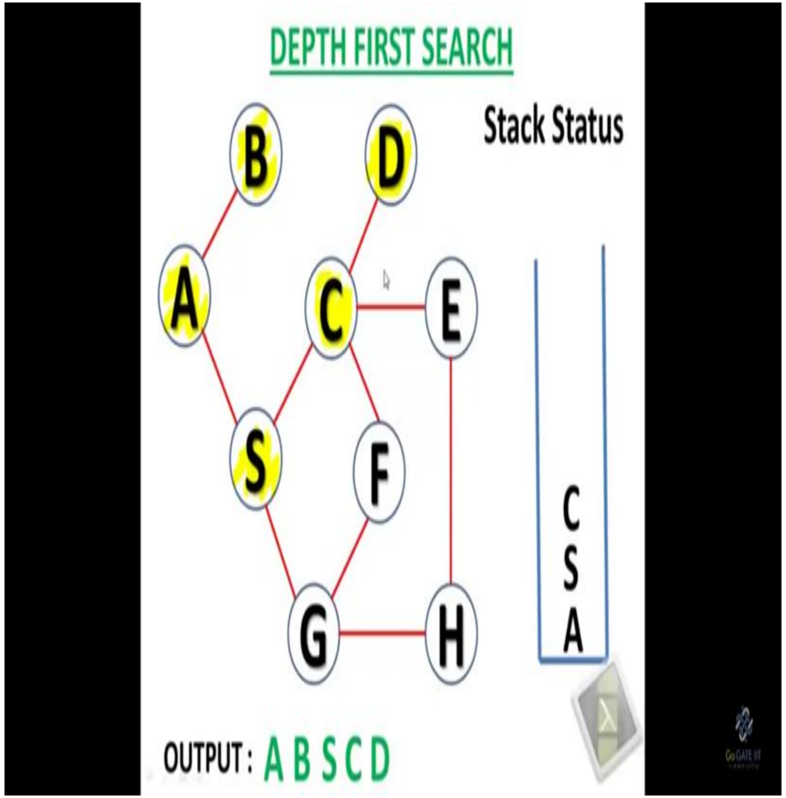
OUTPUT : A B

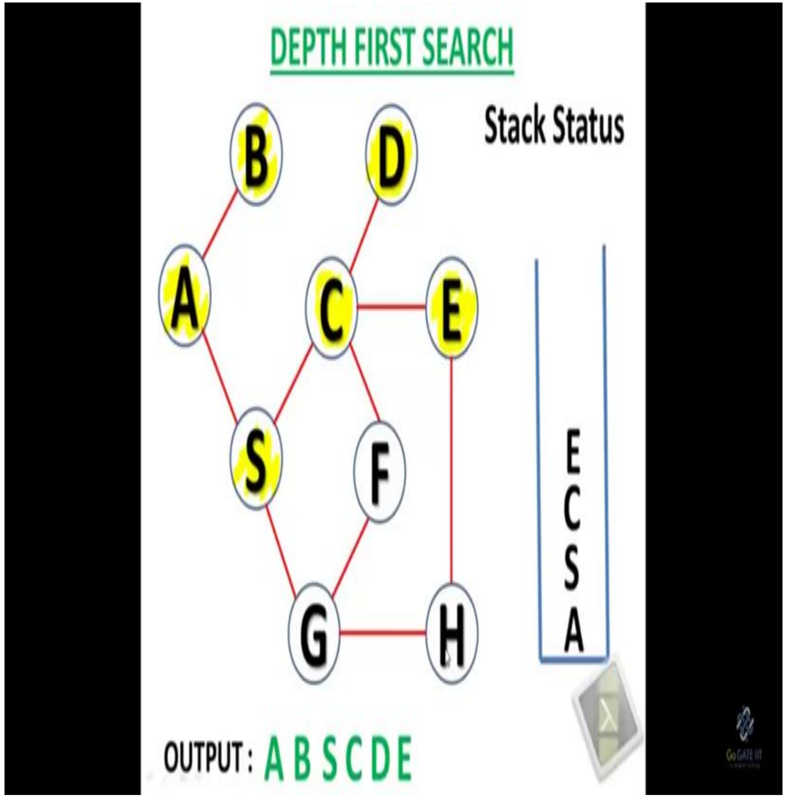

DEPTH FIRST SEARCH

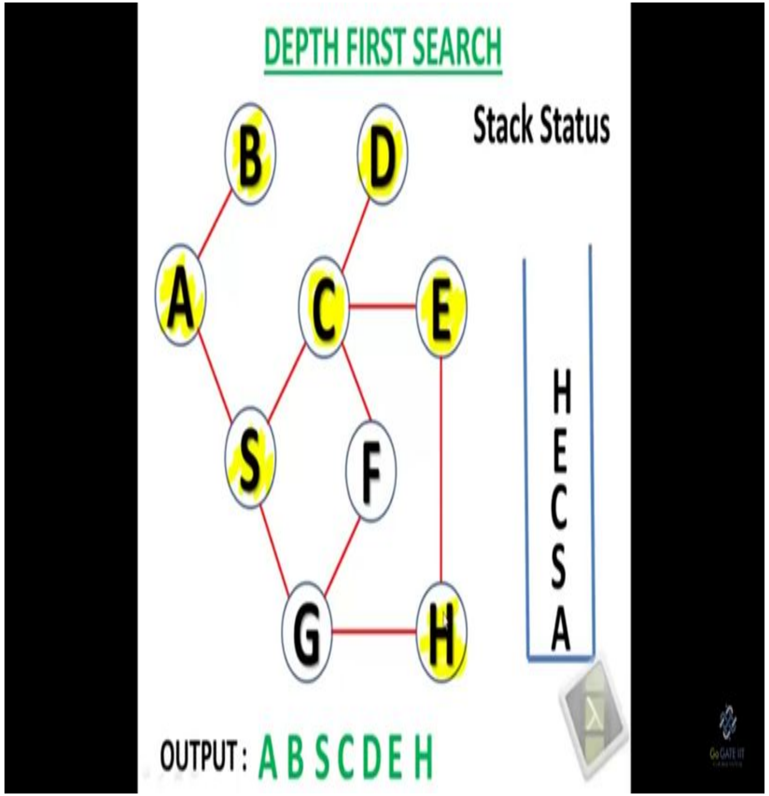
OUTPUT : A B


DEPTH FIRST SEARCH
Stack Status
B
D
A
C
E
S
F
G
H
S
A
OUTPUT : A B S


DEPTH FIRST SEARCH
Stack Status
B
D
A
C
E
S
F
G
H
C
S
A
OUTPUT : A B S C

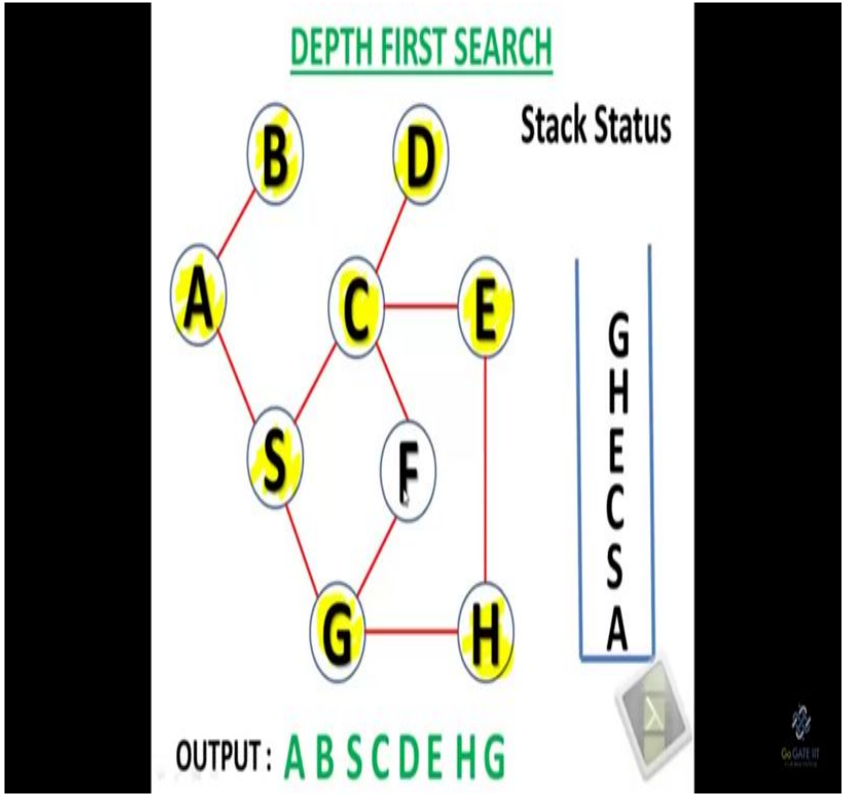

DEPTH FIRST SEARCH
B
D
Stack Status
A
C
E
S
F
G
H
C
S
A
OUTPUT : A B S C D


DEPTH FIRST SEARCH
B
D
Stack Status
A
C
E
S
F
G
H
E
C
S
A
OUTPUT : A B S C D E


DEPTH FIRST SEARCH

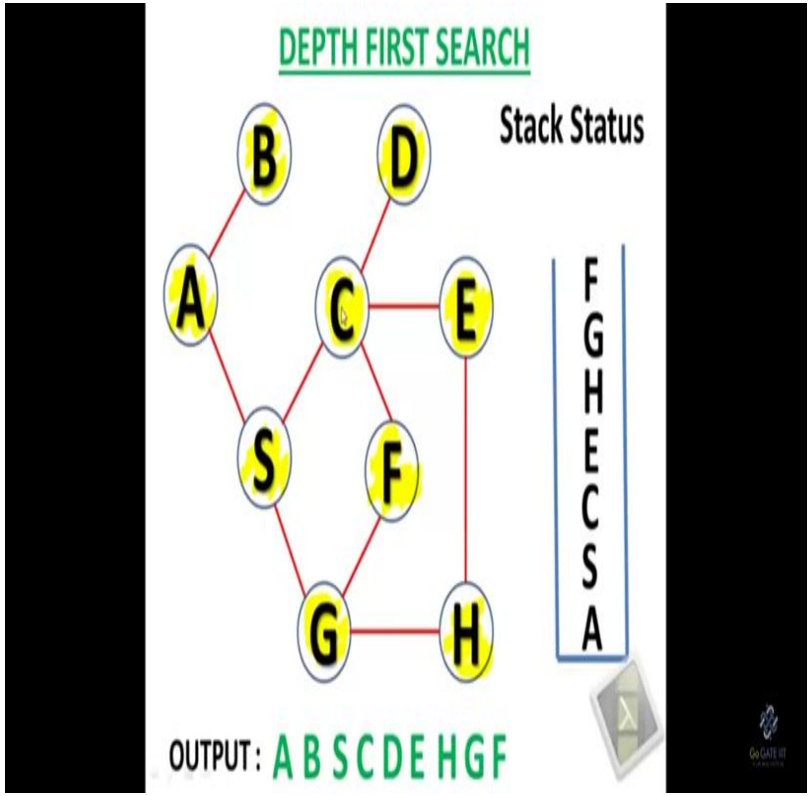
Stack Status

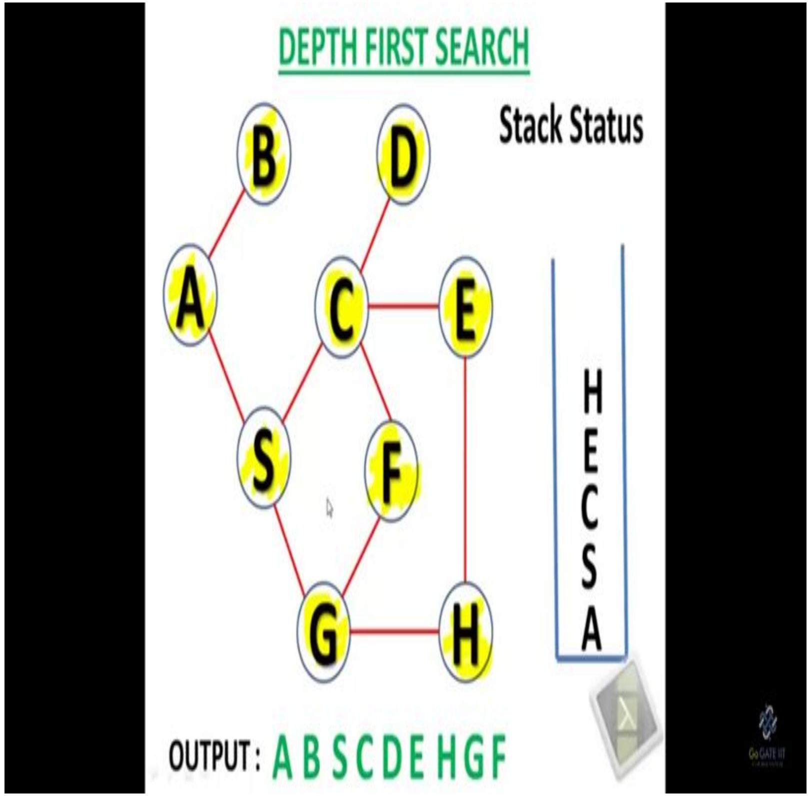
OUTPUT : A B S C D E H

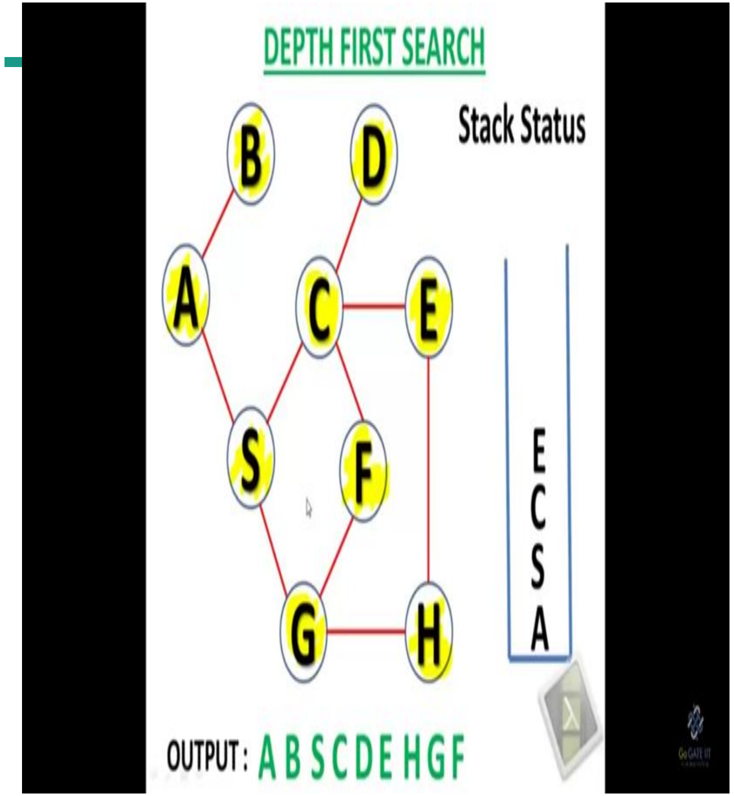

DEPTH FIRST SEARCH

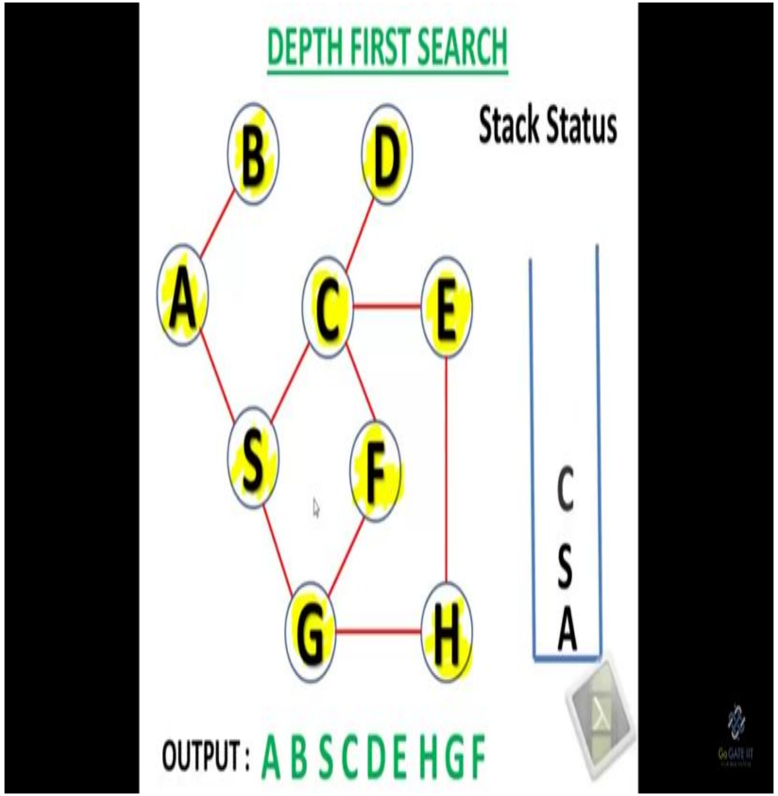
Stack Status

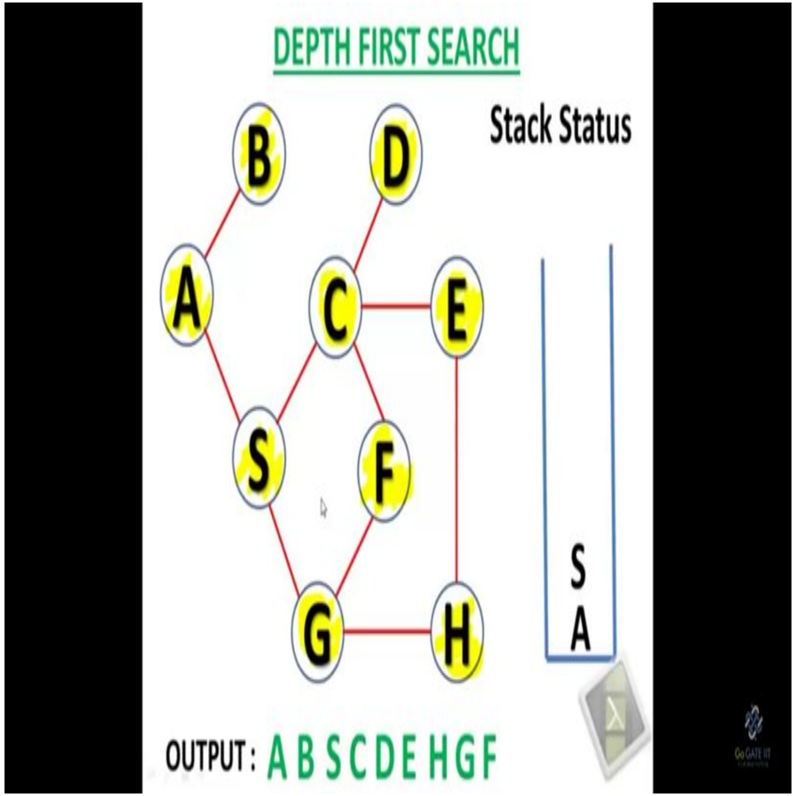
OUTPUT : A B S C D E H G

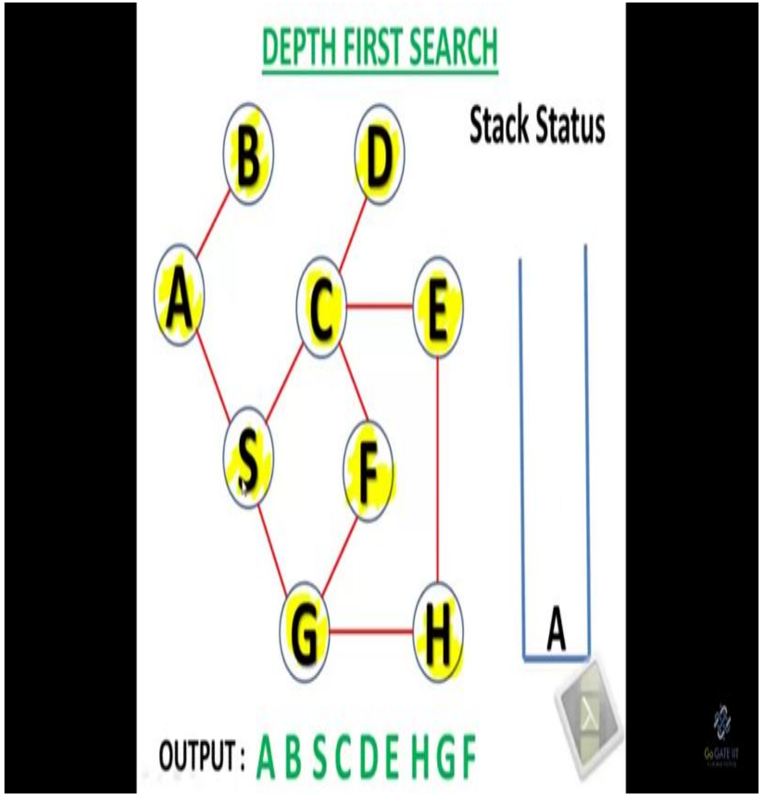

DEPTH FIRST SEARCH
Stack Status
A
B
C
D
E
S
F
G
H
F
G
H
E
C
S
A
OUTPUT : A B S C D E H G F


DEPTH FIRST SEARCH
Stack Status
A
B
C
D
E
S
F
G
H
H
E
C
S
A
OUTPUT : A B S C D E H G F

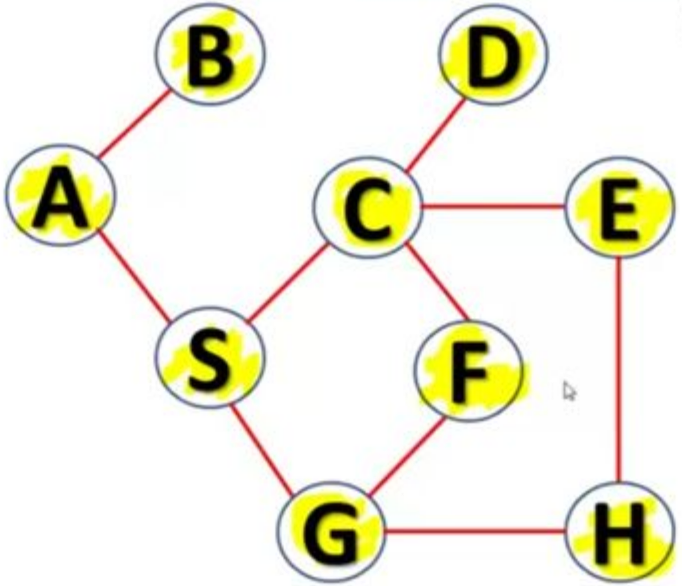

DEPTH FIRST SEARCH
Stack Status
B
D
A
C
E
S
F
G
H
E
C
S
A
OUTPUT : A B S C D E H G F


DEPTH FIRST SEARCH
Stack Status
B
D
A
C
E
S
F
G
H
C
S
A
OUTPUT : A B S C D E H G F


DEPTH FIRST SEARCH
Stack Status
B
D
A
C
E
S
F
G
H
S
A
OUTPUT : A B S C D E H G F


DEPTH FIRST SEARCH
Stack Status
B
D
A
C
E
S
F
G
H
A
OUTPUT : A B S C D E H G F

DEPTH FIRST SEARCH

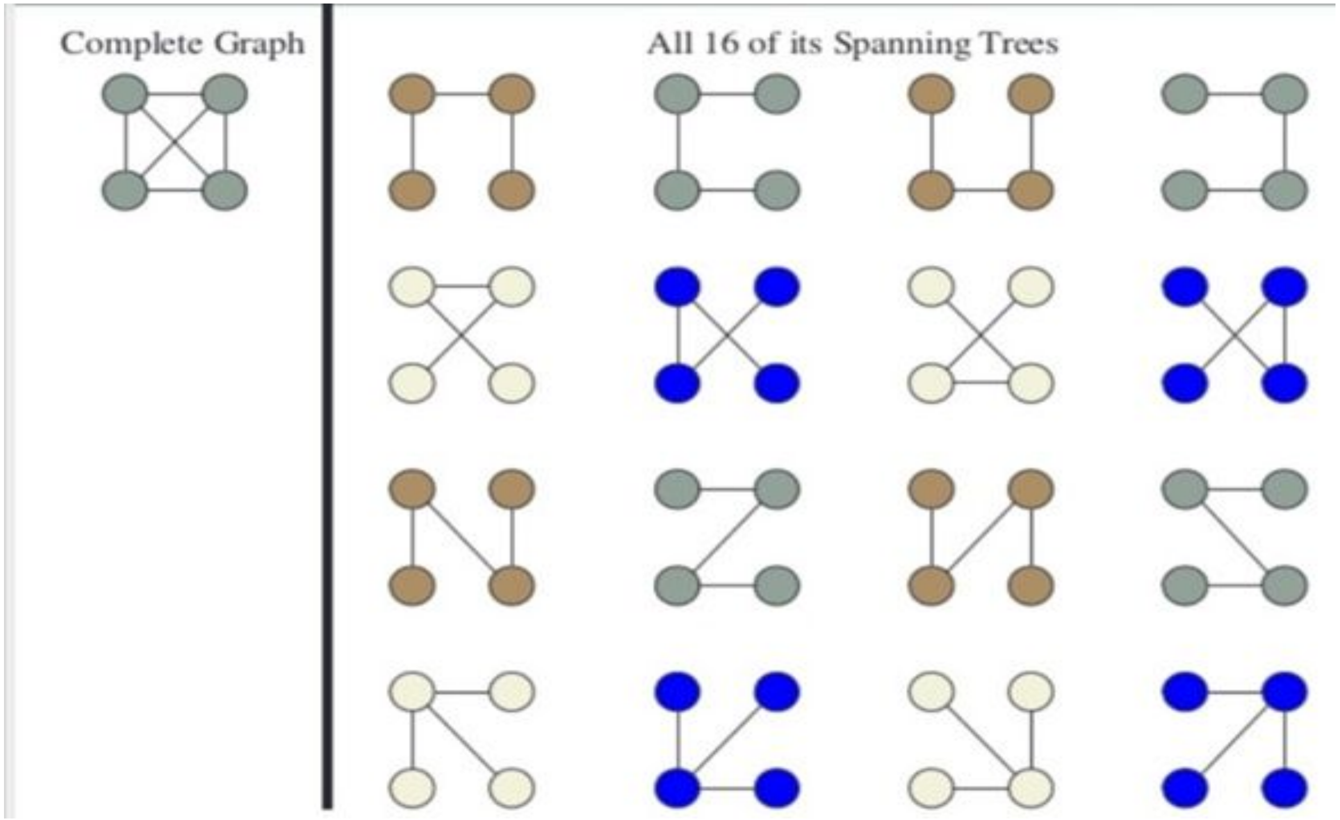


Stack Status

OUTPUT : A B S C D E H G F

Spanning Tree:

- A spanning tree T of an undirected graph G = (V, E) is a sub-graph of G, with the minimum possible number of edges.
- A graph may have several spanning trees.
- A graph that is not connected does not have any spanning tree.


Complete Graph
All 16 of its Spanning Trees

Minimum Spanning Tree:

ØA minimum spanning tree is a subset of the edges of a connected weighted undirected graph with the minimum possible total edge weights (without any cycle).

ØIt is a spanning tree whose sum of weights is as small as possible. A graph may have several minimum spanning trees

A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

->There are mainly two methods used to find minimum spanning out of a connected weighted graph :-

Ø **Prim's algorithm**

Ø **Kruskal's algorithm**

- ***How many such spanning trees can be formed ?***

Suppose |V| = 6 and |E| = |V|-1 = 5, where V, E ϵ vertices and edges respectively. Thus the possible combinations are :

$$\mathbf{N = {}^{|V|}C_{|E|} = {}^{6}C_{5}}$$

$$\mathbf{N = 6}$$

NOTE : In case the total edges (in the graph) forms (m) cycles.

Then the possible combinations will be :

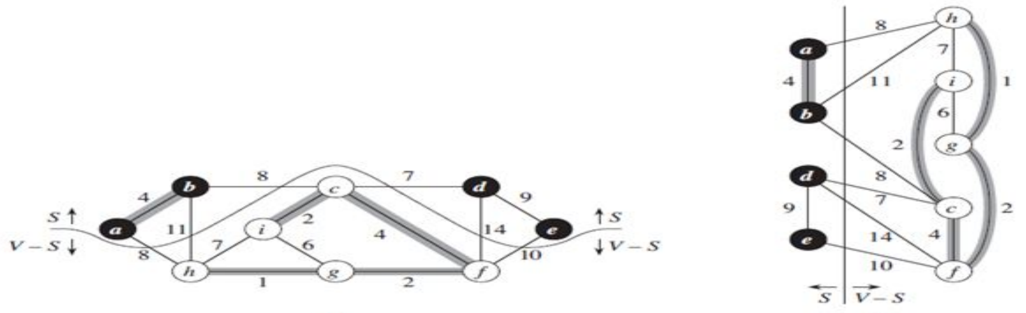
$$\mathbf{N = {}^{|V|}C_{|E|} - m}$$

Prim’s Algorithm:

- The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included.
- At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges.
- After picking the edge, it moves the other endpoint of the edge to the set containing MST.

- A group of edges that connects two set of vertices in a graph is called cut in graph theory.
- *At every step of Prim’s algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the vertices).*
- *Pick the minimum weight edge from the cut and include this vertex to MST Set.*

(The set that contains already included vertices).



(a) (b)

Algorithm:

–In order to implement Prim’s algorithm efficiently, we need a fast way to select a new edge to add to the tree formed by the edges in A.

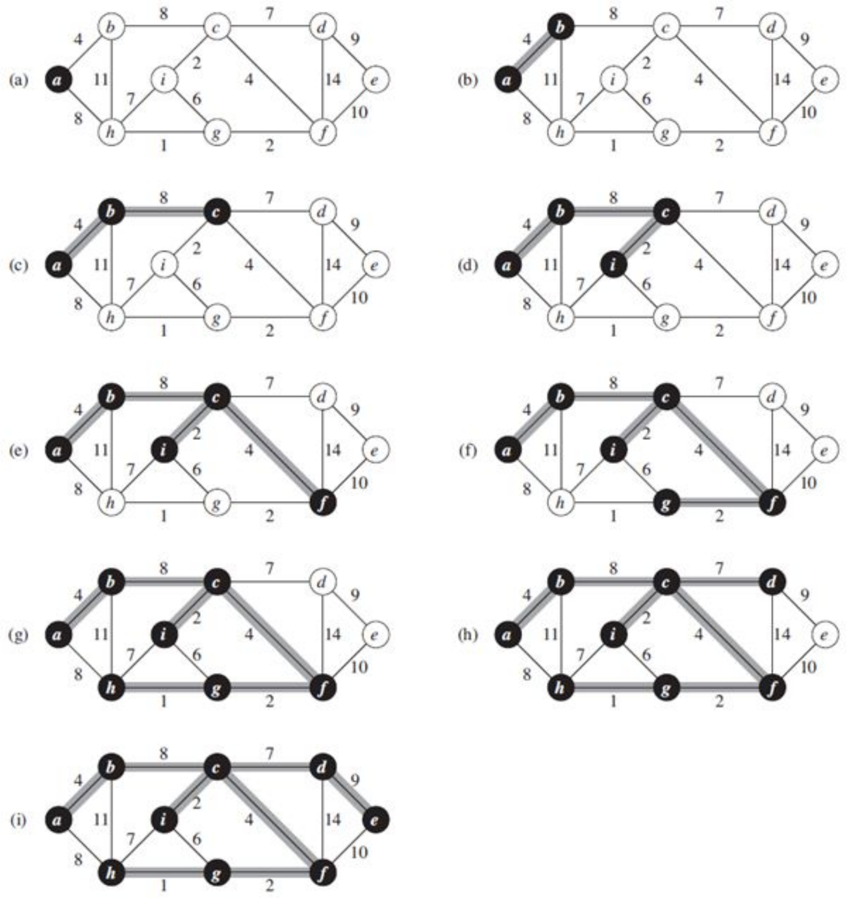
–The connected graph G and the root r of the minimum spanning tree to be grown are inputs to the algorithm.

–During execution of the algorithm, all vertices that are *not* in the tree reside in a min-priority queue Q based on a *key* attribute.

–For each vertex v, the attribute v.*key* is the minimum weight of any edge connecting v to a vertex in the tree; by convention v.key = ∞ if there is no such edge.

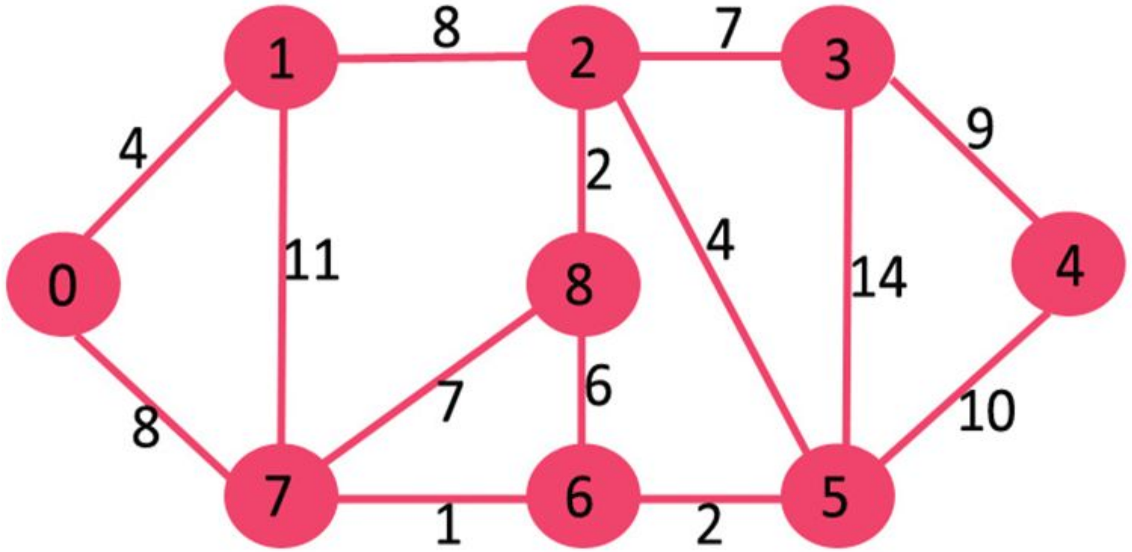
–The attribute ***v.π*** names the parent of in the tree.

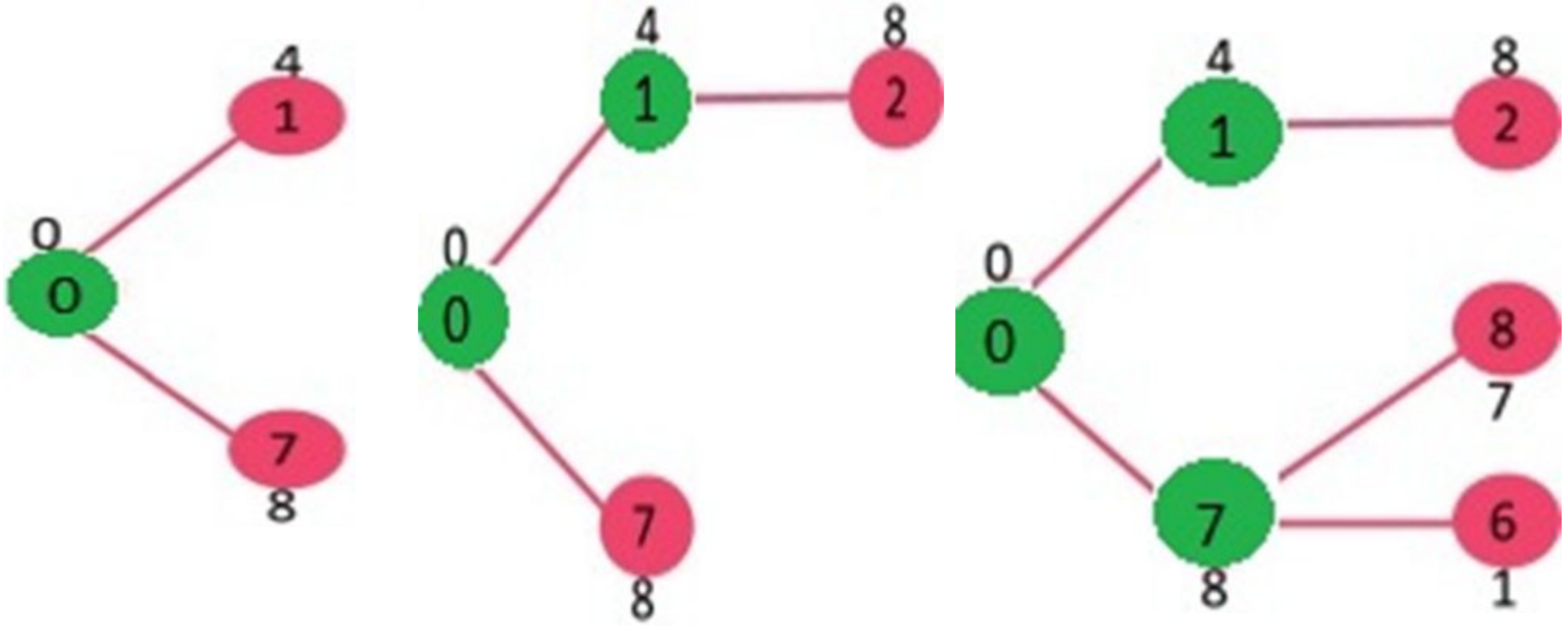
```
MST-PRIM$(G, w, r)$
1  for each $u \in G.V$
2      $u.key = \infty$
3      $u.\pi = \text{NIL}$
4  $r.key = 0$
5  $Q = G.V$
6  while $Q \neq \emptyset$
7      $u = \text{EXTRACT-MIN}(Q)$
8      for each $v \in G.Adj[u]$
9          if $v \in Q$ and $w(u, v) < v.key$
10             $v.\pi = u$
11             $v.key = w(u, v)$
```

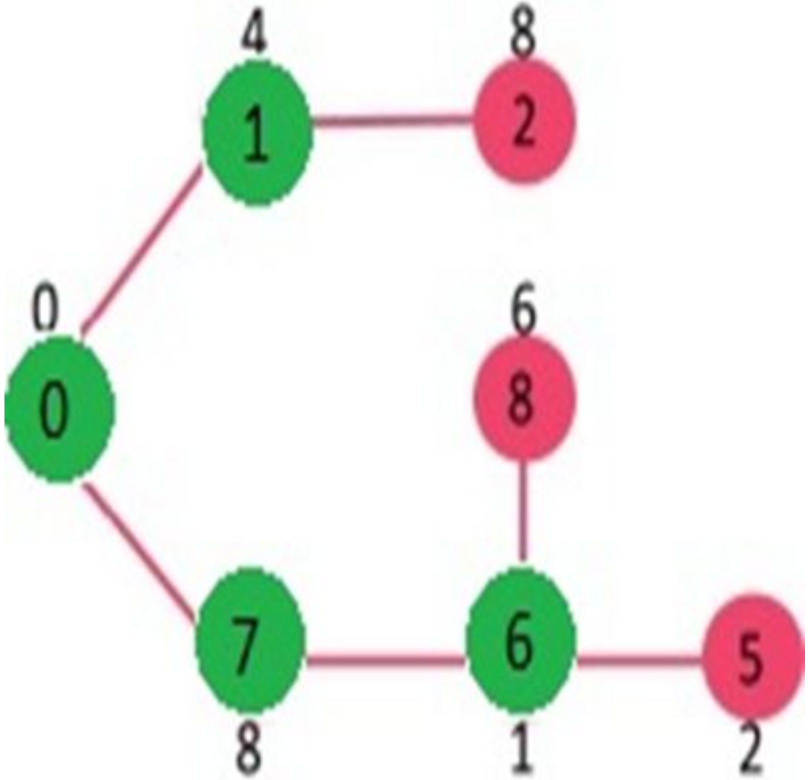


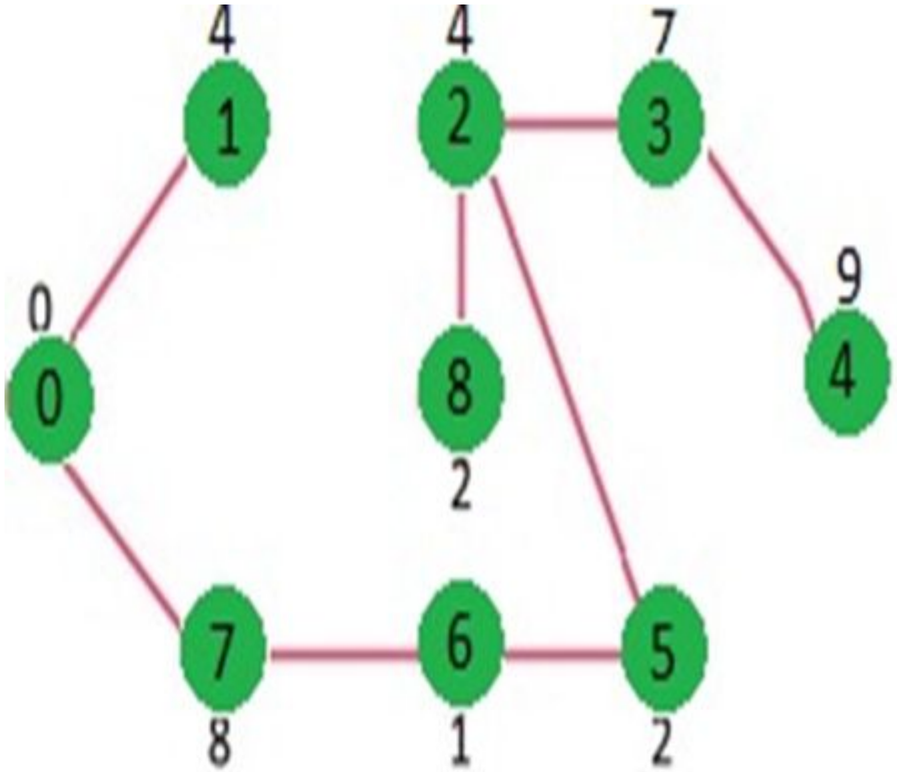
Example:

Find the MST for the following graph using prims algorithm -




0
4
8
0
1
7
0
4
8
0
8
1
2
7
0
4
8
0
8
7
1
1
2
7
8
6


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


4
4
7
1
2
3
0
0
9
8
4
2
7
6
5
8
1
2

Kruskal’s Algorithm:

•**Kruskal's algorithm** is a minimum-spanning-tree **algorithm** which finds an edge of the least possible weight that connects any two trees in the forest. It is a greedy **algorithm** in graph theory as it finds a minimum spanning tree for a connected weighted graph adding increasing cost arcs at each step.

•Below are the steps for finding MST using Kruskal’s algorithm :

Ø *Sort all the edges in non-decreasing order of their weight.*

Ø*Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.*
Repeat step#2 until there are (V-1) edges in the spanning tree

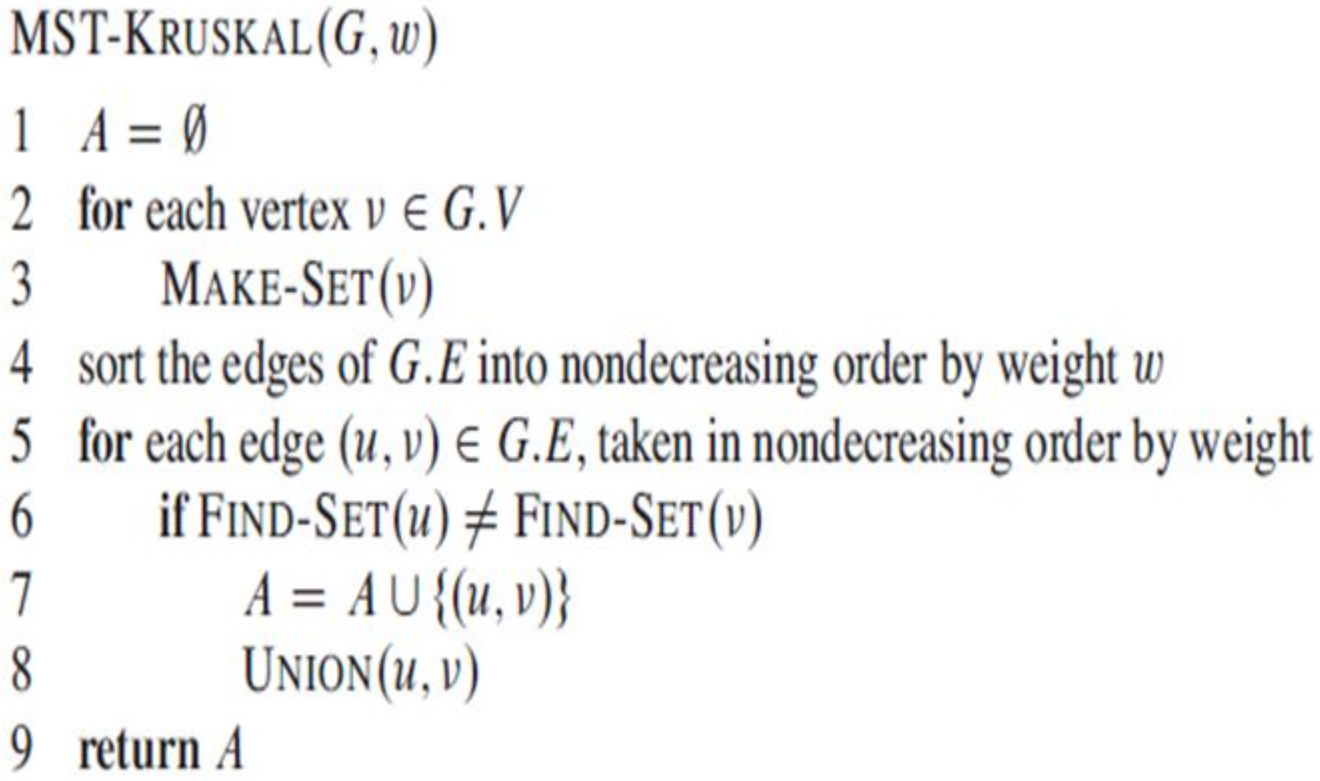
Algorithm:

•It uses a disjoint-set data structure to maintain several disjoint sets of elements.

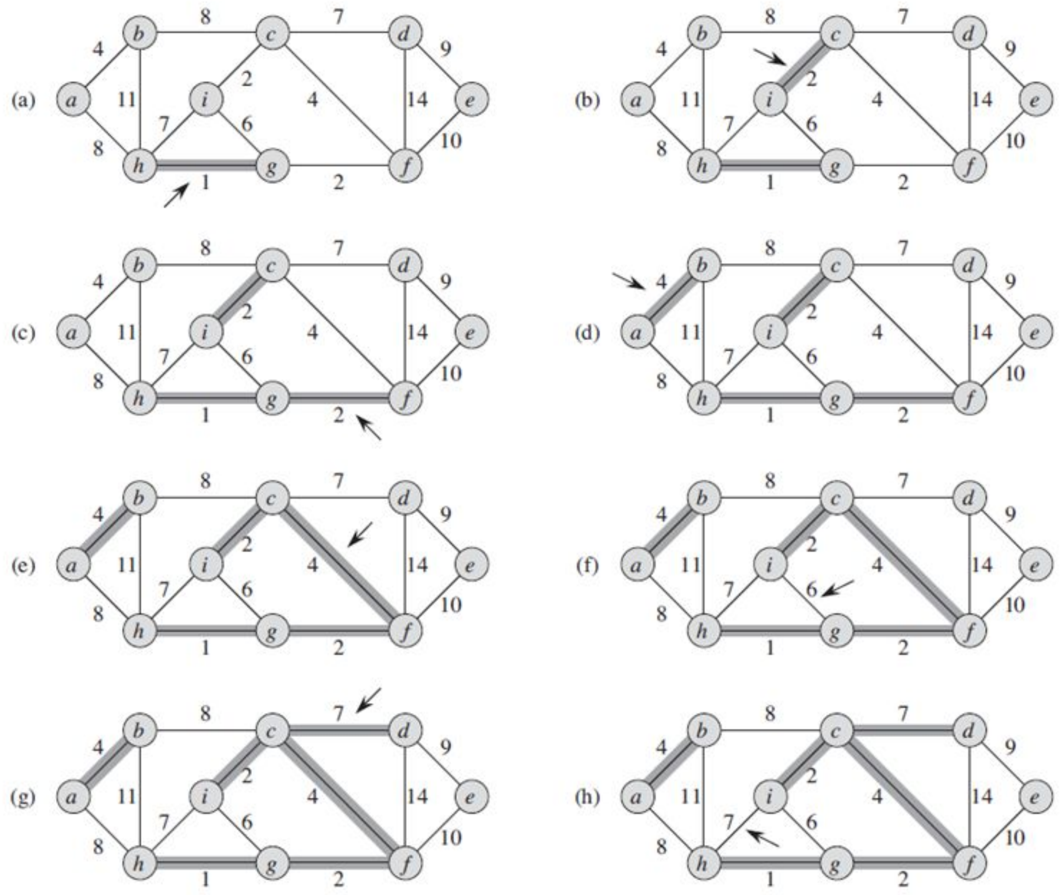
•The operation FIND-SET(u) returns a representative element from the set that contains u.

•We can determine whether two vertices u and v belong to the same tree by testing whether FIND-SET(u) equals FIND-SET(v).

•To combine trees, Kruskal's algorithm calls the UNION procedure

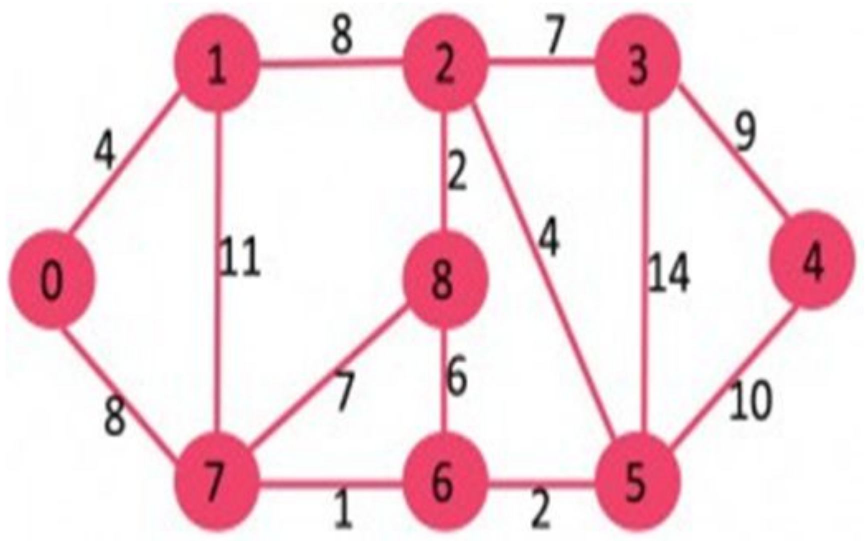


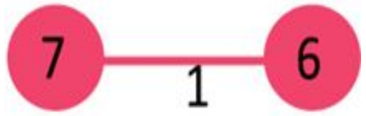
```
MST-KRUSKAL($G, w$)
1  $A = \emptyset$
2  for each vertex $v \in G.V$
3      MAKE-SET($v$)
4  sort the edges of $G.E$ into nondecreasing order by weight $w$
5  for each edge $(u, v) \in G.E$, taken in nondecreasing order by weight
6      if FIND-SET($u$) $\neq$ FIND-SET($v$)
7          $A = A \cup \{(u, v)\}$
8          UNION($u, v$)
9  return $A$
```

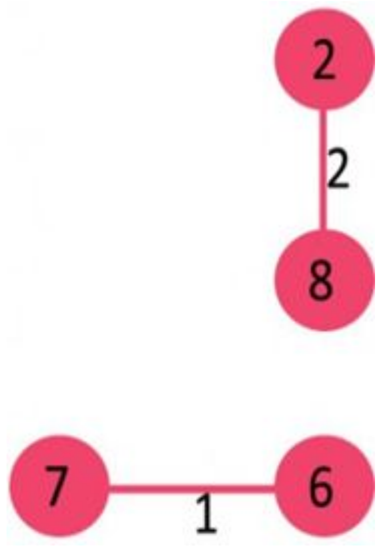

(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
10
8
1
2

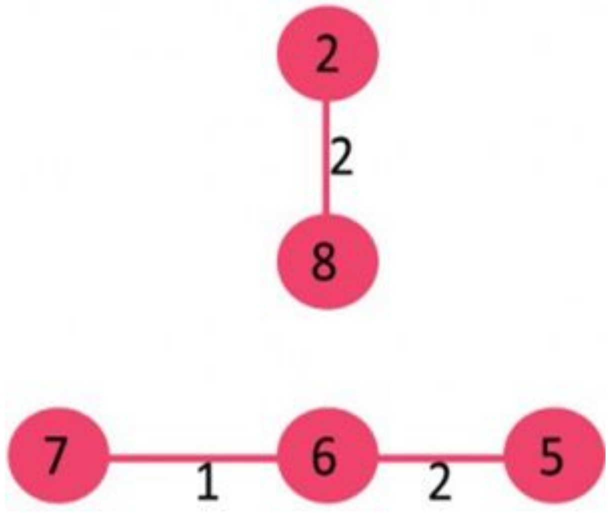
Example:

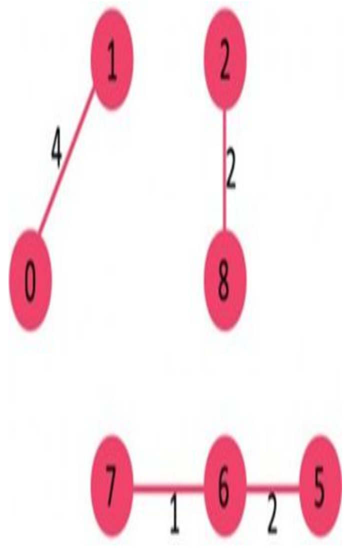
->Find the MST for the following graph using kruskal’s algorithm.

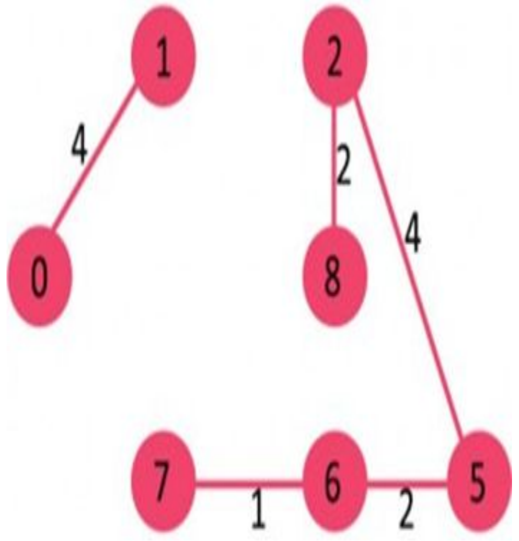


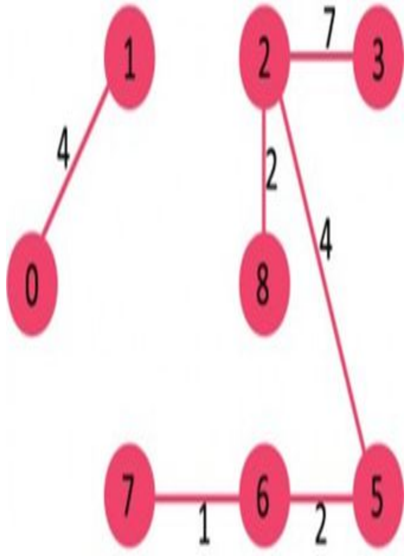

7
1
6

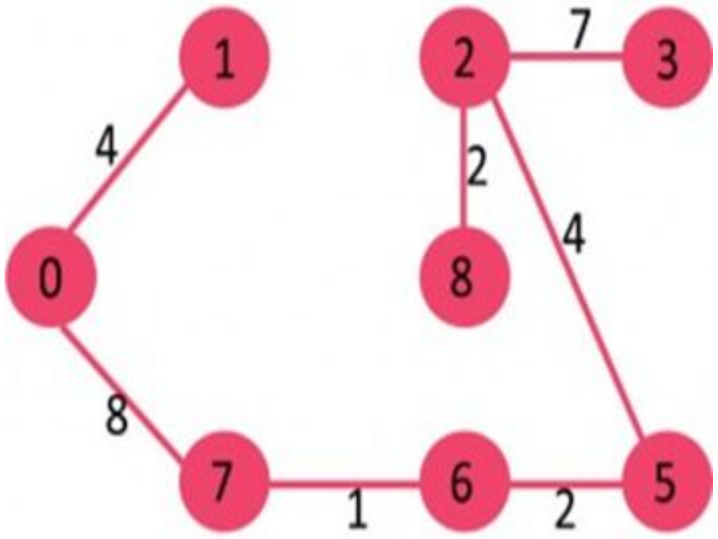

2
2
8
7
1
6


2
2
8
7
1
6
2
5

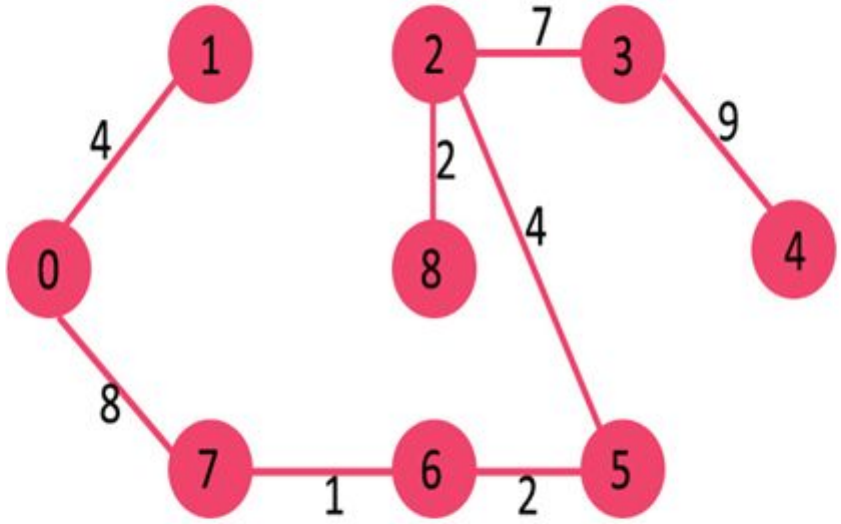

1
2
4
2
0
8
7
6
5
1
2


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


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



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



0
1
2
3
4
8
7
6
5
4
8
1
2
7
2
4
9

Single Source Shortest Path:

•In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.

•In a ***shortest-paths problem***, we are given a weighted, directed graph G(V, E), with weight function w : E à R, mapping edges to real-valued weights.

The ***weight*** w(p) of path p = $\{v_0, v_1, \ldots, v_k\}$ is the sum of the weights of its constituent edges:

$$w(p) = \sum^{k}_{i=1} w(v_{i-1,} v_i)$$

We define the shortest path weight from *u* to *v,*

= min{w(p) : u ~> v}, if there is path from u to v

∞ , otherwise

Single Source Problems:

•The algorithm for the single-source problem can solve many other problems, including the following variants :

Ø***Single Destination Shortest Path Problem :*** Find a shortest path from the given destination vertex t to the vertex v. By reversing the direction of each edge in the graph we can reduce each problem to single-source problem.

Ø ***Single Pair Shortest Path Problem :*** Find a shortest path from u to v for given vertices u and v. If we solve the single-source problem with source vertex u, we solve this problem also.

Ø ***All Pair Shortest Path Problem :*** Find a shortest path from u to v for every pair of vertices of u and v. Although we can solve this problem by running a single-source algorithm once from each vertex, we usually can solve it faster.

Dijkstra’s Algorithm:

•**Dijkstra's algorithm** (or **Dijkstra's** Shortest Path First **algorithm**, SPF **algorithm**) is an **algorithm** for finding the shortest paths between nodes in a graph, which may represent, for example, road networks.

•We generate a *SPT (shortest path tree)* with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree.

•At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

Time Complexity for Dijkstra’s Algorithm is given by

$$O(|V|^2)$$

Using min heap time complexity can be reduced to

$$O(|V|^* \log(|V|)$$

Algorithm:

•Dijkstra’s algorithm solves the single-source shortest-paths problem on a weighted, directed graph G(V, E) for the case in which all edge weights are nonnegative.

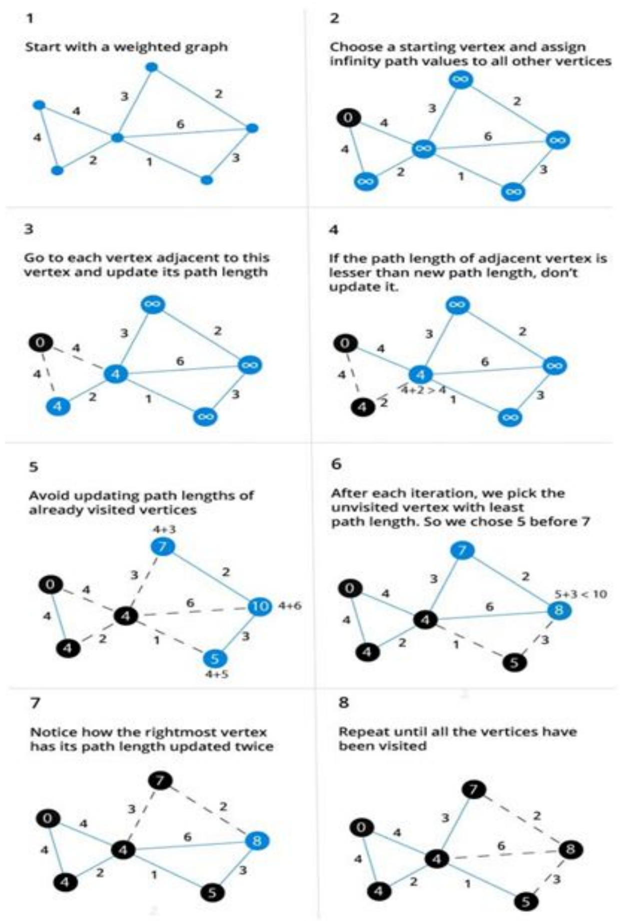
•Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Ø Dijkstra’s Algorithm maintains a set S of vertices whose final shortest path weights from the source s has already been determined.

Ø The algorithm repeatedly selects the vertex u ϵ V-S with minimum shortest path estimate, adds u to S, and relaxes all edges leaving u.

ØIn the following implementation, we use a min-priority queue Q of vertices, keyed by their d value.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  $S = \emptyset$
3  $Q = G.V$
4  while $Q \neq \emptyset$
5      $u$ = EXTRACT-MIN($Q$)
6      $S = S \cup \{u\}$
7      for each vertex $v \in G.Adj[u]$
8          RELAX($u, v, w$)
```


1

Start with a weighted graph

2

Choose a starting vertex and assign infinity path values to all other vertices

3

Go to each vertex adjacent to this vertex and update its path length

4

If the path length of adjacent vertex is lesser than new path length, don't update it.

5

Avoid updating path lengths of already visited vertices

6

After each iteration, we pick the unvisited vertex with least path length. So we chose 5 before 7

7

Notice how the rightmost vertex has its path length updated twice

8

Repeat until all the vertices have been visited

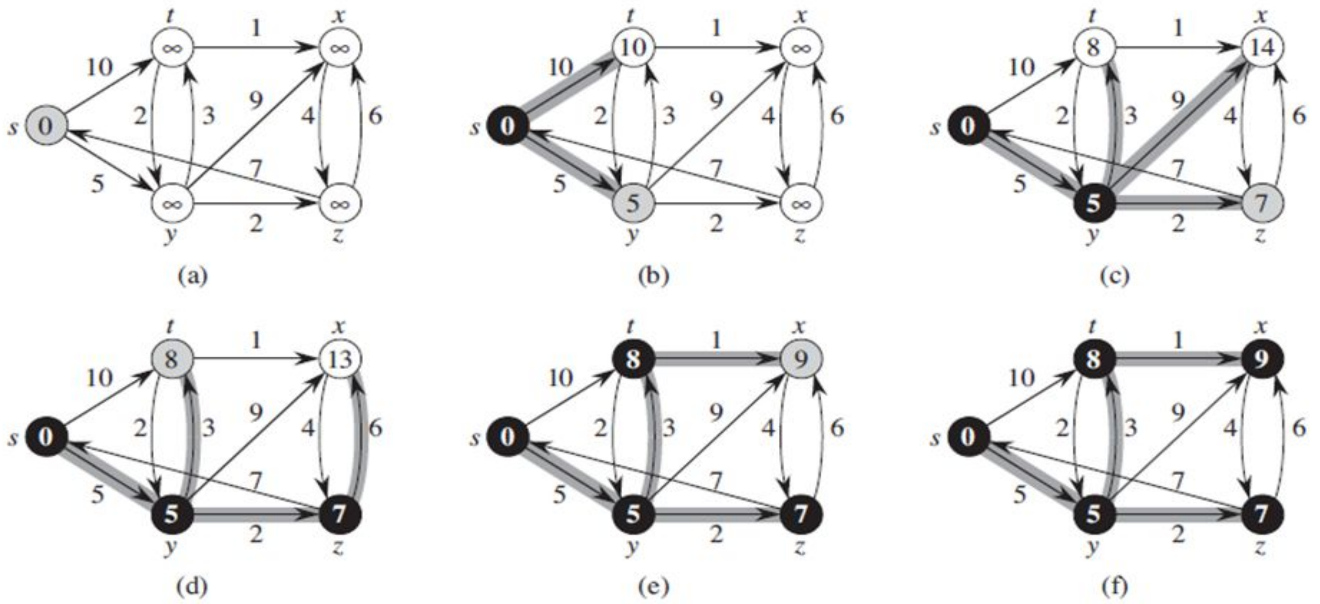
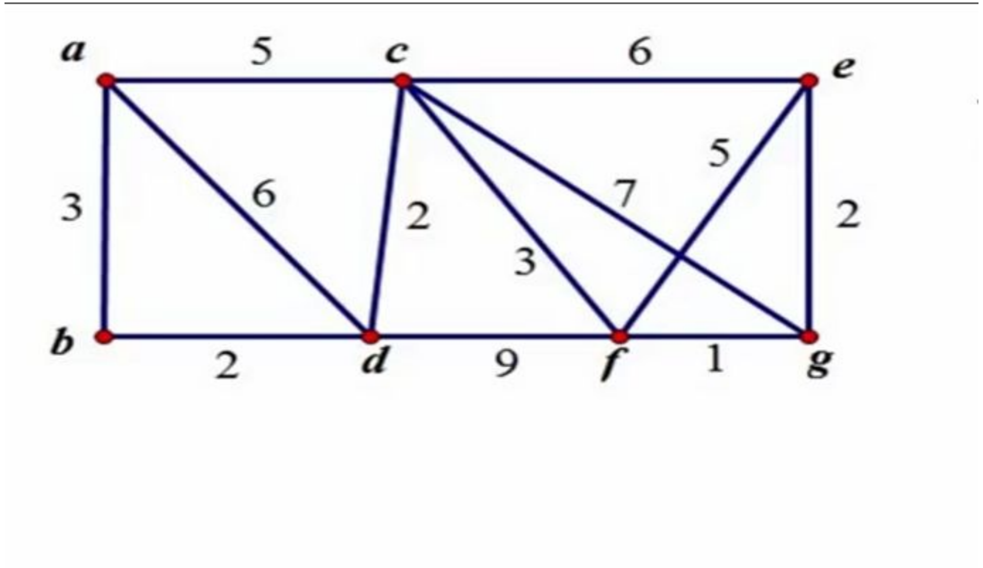
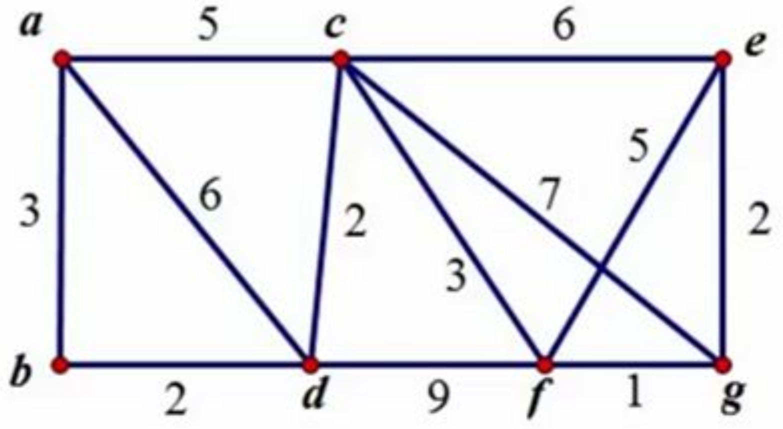


Figure 24.6 The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop of lines 4–8. The shaded vertex has the minimum d value and is chosen as vertex u in line 5. **(b)–(f)** The situation after each successive iteration of the **while** loop. The shaded vertex in each part is chosen as vertex u in line 5 of the next iteration. The d values and predecessors shown in part (f) are the final values.

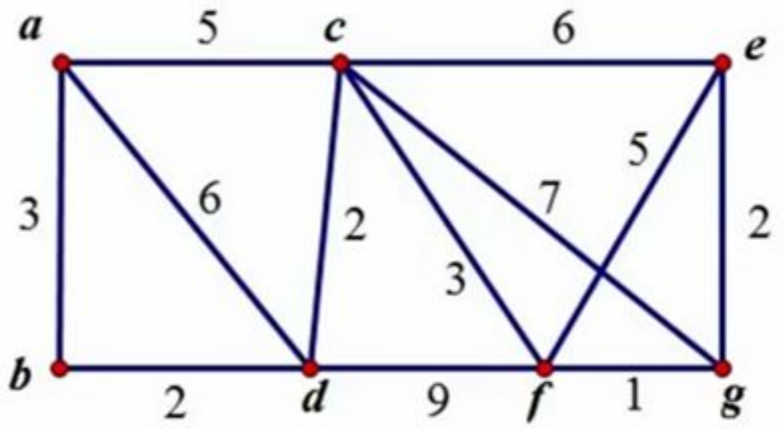
Example:

Find the shortest path from vertex ‘a’ to every other vertex for this graph

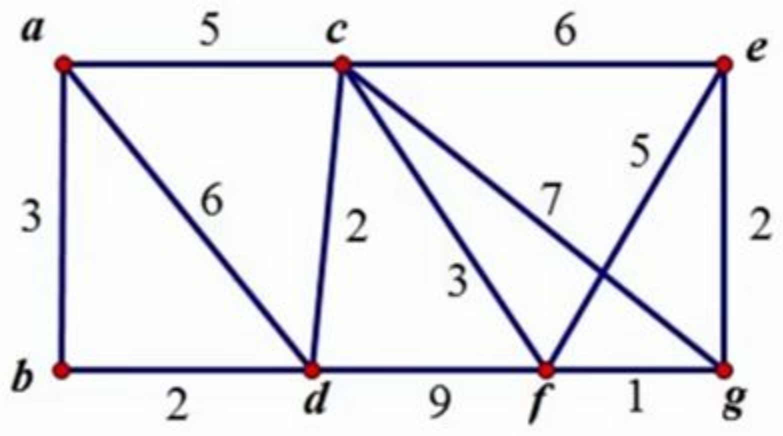




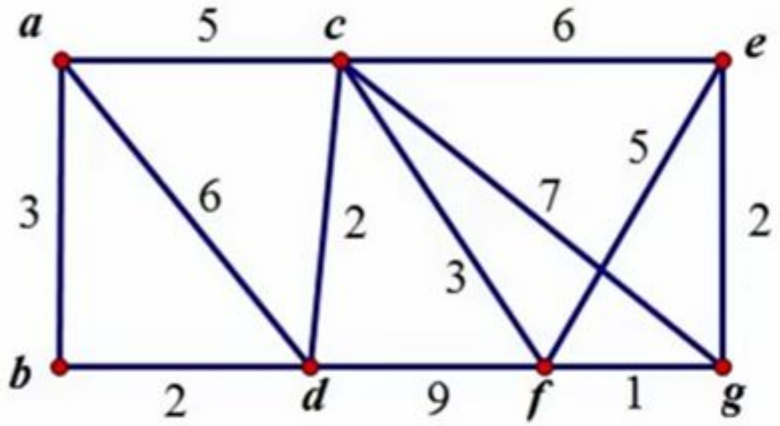
v	a	b	c	d	e	f	g
a							



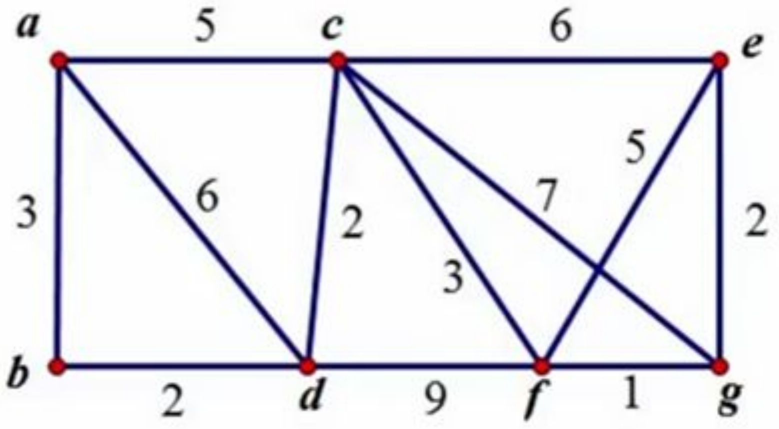
v	a	b	c	d	e	f	g
a	0_a	3_a	5_a	6_a	∞_a	∞_a	∞_a



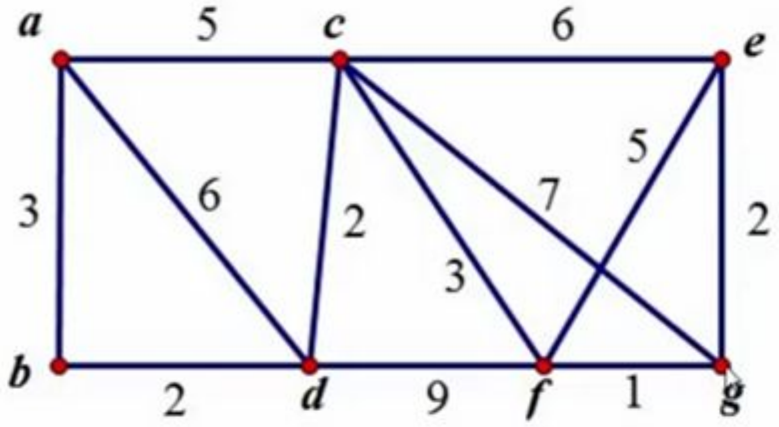
v	a	b	c	d	e	f	g
a	0_a	3_a	5_a	6_a	∞_a	∞_a	∞_a
b	0_a	3_a	5_a	5_b	∞_a	∞_a	∞_a



v	a	b	c	d	e	f	g
a	0_a	3_a	5_a	6_a	∞_a	∞_a	∞_a
b	0_a	3_a	5_a	5_b	∞_a	∞_a	∞_a
c	0_a	3_a	5_a	5_b	11_c	8_c	12_c



v	a	b	c	d	e	f	g
a	0_a	3_a	5_a	6_a	∞_a	∞_a	∞_a
b	0_a	3_a	5_a	5_b	∞_a	∞_a	∞_a
c	0_a	3_a	5_a	5_b	11_c	8_c	12_c
d	0_a	3_a	5_a	5_b	11_c	8_c	12_c



v	a	b	c	d	e	f	g
a	0_a	3_a	5_a	6_a	∞_a	∞_a	∞_a
b	0_a	3_a	5_a	5_b	∞_a	∞_a	∞_a
c	0_a	3_a	5_a	5_b	11_c	8_c	12_c
d	0_a	3_a	5_a	5_b	11_c	8_c	12_c
f	0_a	3_a	5_a	5_b	11_c	8_c	9_f

Bellman Ford Algorithm:

- The **Bellman–Ford algorithm** is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph.
- Negative edge weights are found in various applications of graphs, hence the usefulness of this algorithm.
- If a graph contains a "negative cycle" (i.e. a cycle whose edges sum to a negative value) that is reachable from the source, then there is no *cheapest* path: any path that has a point on the negative cycle can be made cheaper by one more walk around the negative cycle. In such a case, the Bellman–Ford algorithm can detect and report the negative cycle.

Why negative edges:

Negative weight edges might seem useless at first but they can explain a lot of phenomena like : -

ØCash flow

ØHeat released

ØHeat absorbed

For instance, if there are different ways to reach from one chemical A to another chemical B, each method will have sub-reactions involving both heat dissipation and absorption.

ØIf we want to find the set of reactions where minimum energy is required, then we will need to be able to factor in the heat absorption as negative weights and heat dissipation as positive weights.

Algorithm:

•Like Dijkstra's algorithm, Bellman–Ford proceeds by relaxation, in which approximations to the correct distance are replaced by better ones until they eventually reach the solution.

•In both algorithms, the approximate distance to each vertex is always an overestimate of the true distance, and is replaced by the minimum of its old value and the length of a newly found path.

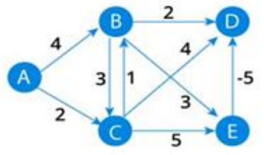
•Dijkstra's algorithm uses a priority queue to greedily select the closest vertex that has not yet been processed, and performs this relaxation process on all of its outgoing edges; by contrast, the Bellman–Ford algorithm simply relaxes *all* the edges, and does this |V|-1 times, where |V| is the number of vertices in the graph.

•Bellman Ford runs in O(|V|*|E|) time, where |V| and |E| are the total number of vertices and edges respectively.

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

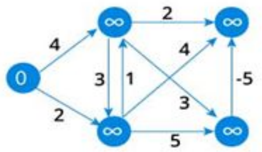
1

Start with a weighted graph



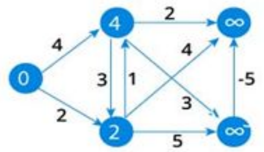
2

Choose a starting vertex and assign infinity path values to all other vertices



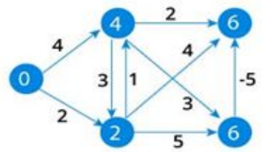
3

Visit each edge and relax the path distances if they are inaccurate



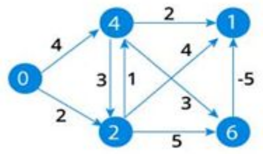
4

We need to do this V times because in the worst case, a vertex's path length might need to be readjusted V times



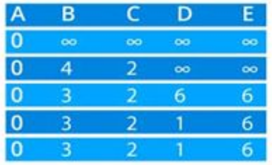
5

Notice how the vertex at the top right corner had its path length adjusted



6

After all the vertices have their path lengths, we check if a negative cycle is present.



A	B	C	D	E
0	∞	∞	∞	∞
0	4	2	∞	∞
0	3	2	6	6
0	3	2	1	6
0	3	2	1	6

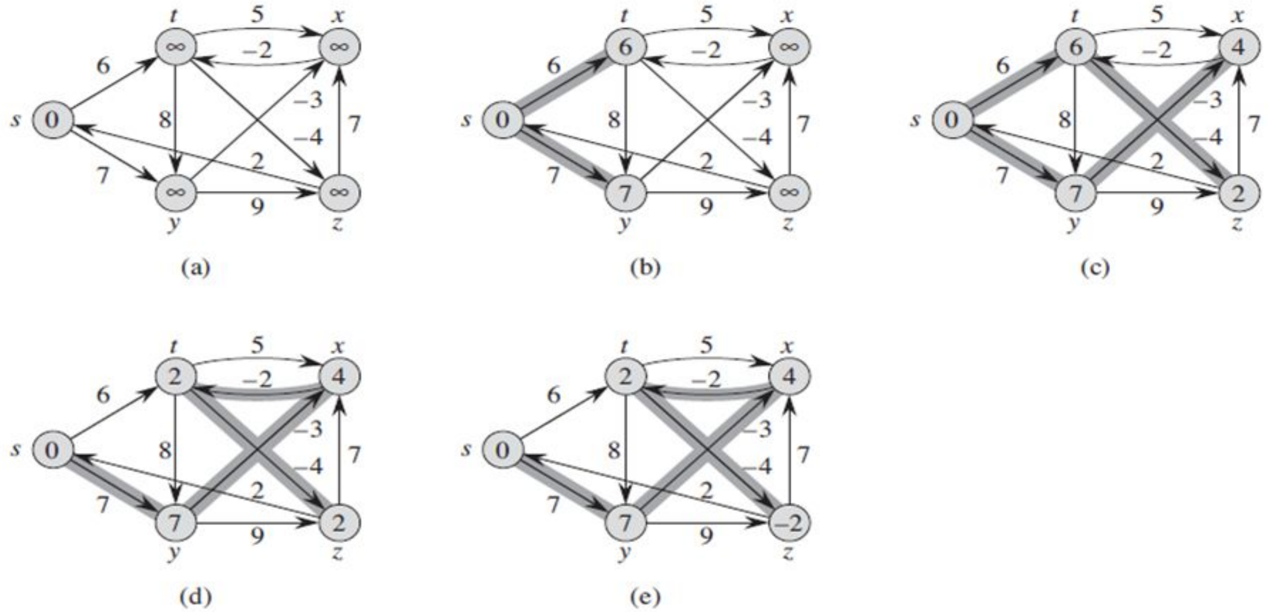
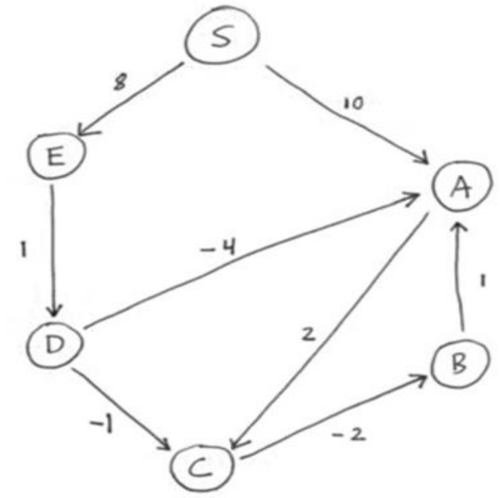


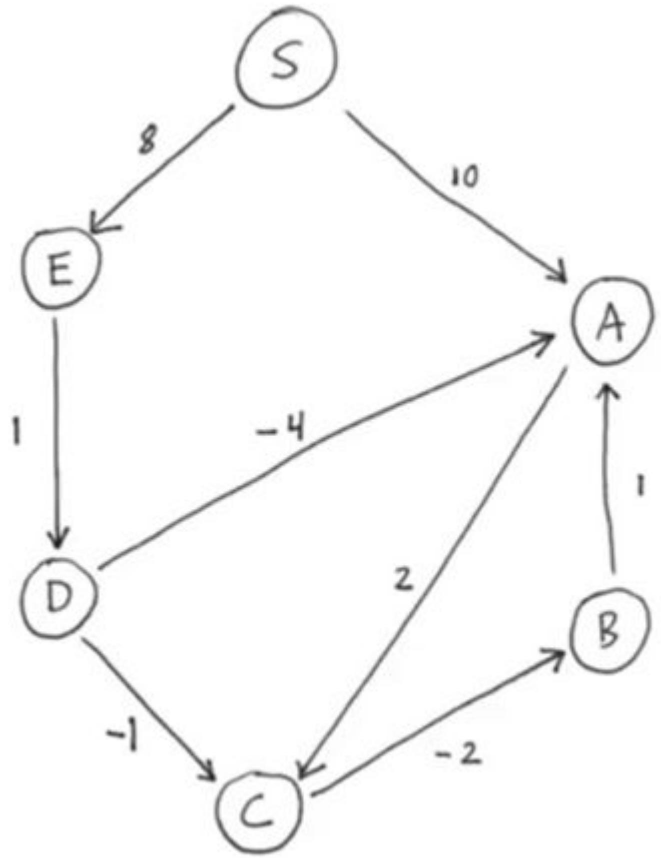
Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values appear within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $v.\pi = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

Example:

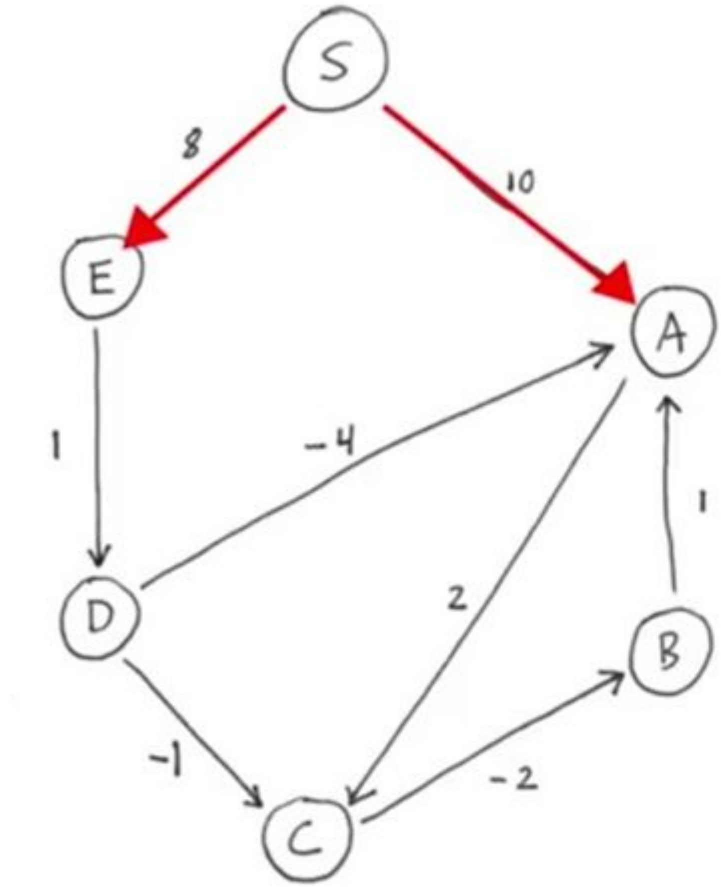
Find the shortest path from vertex **S** all other vertices



6 vertices = 5 iterations

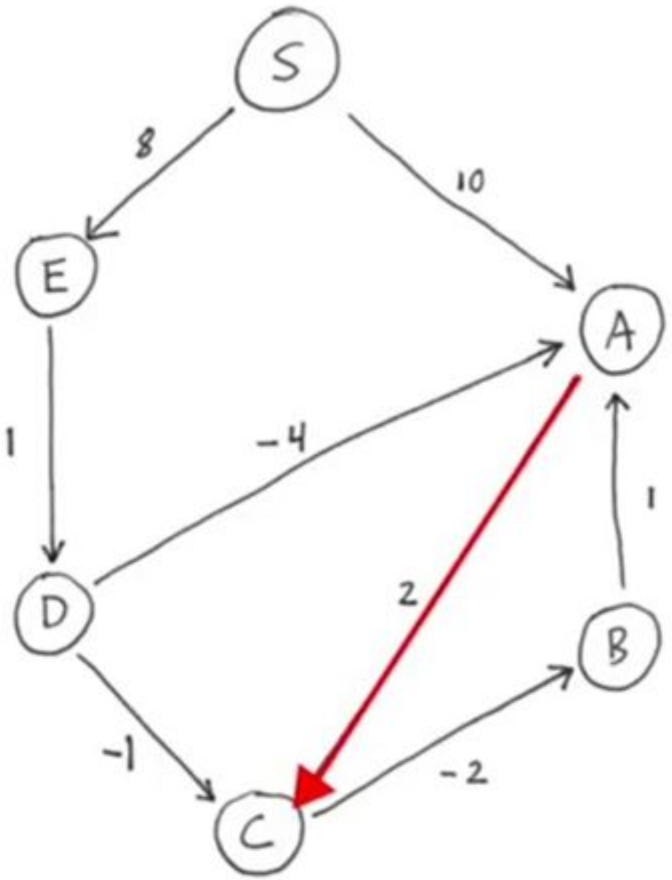


0	∞	∞	∞	∞	∞
S	**A**	**B**	**C**	**D**	**E**



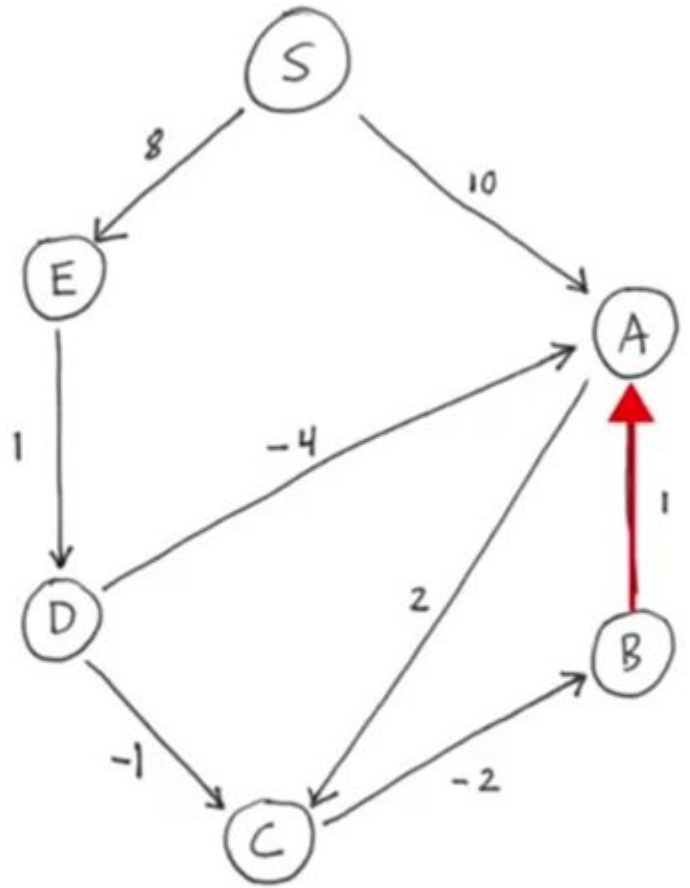
	10				8
0	~~∞~~	∞	∞	∞	~~∞~~
S	**A**	**B**	**C**	**D**	**E**

1st Iteration



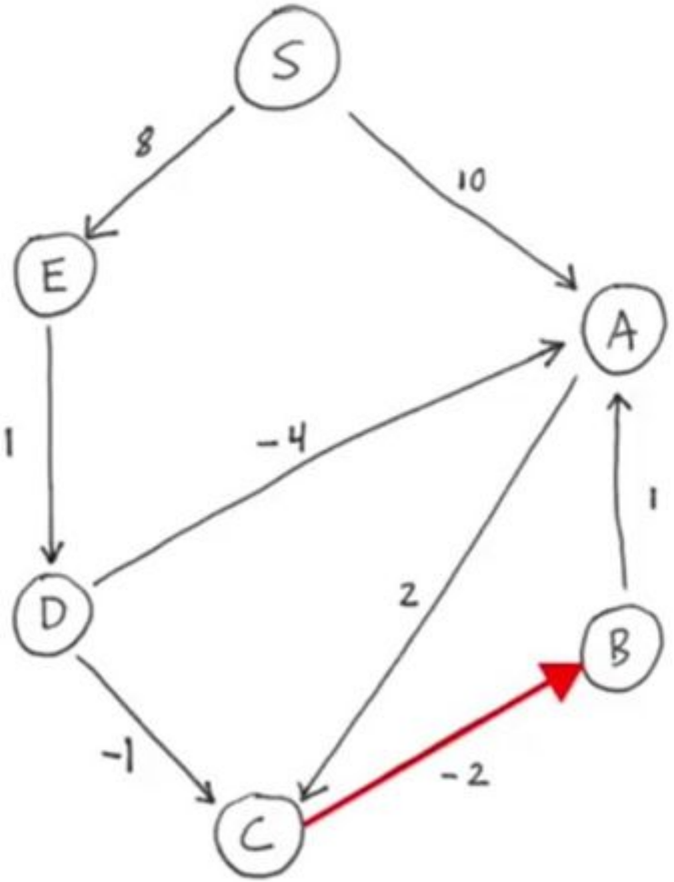
			12		
0	10	∞	~~∞~~	∞	8
S	A	B	C	D	E

1st Iteration



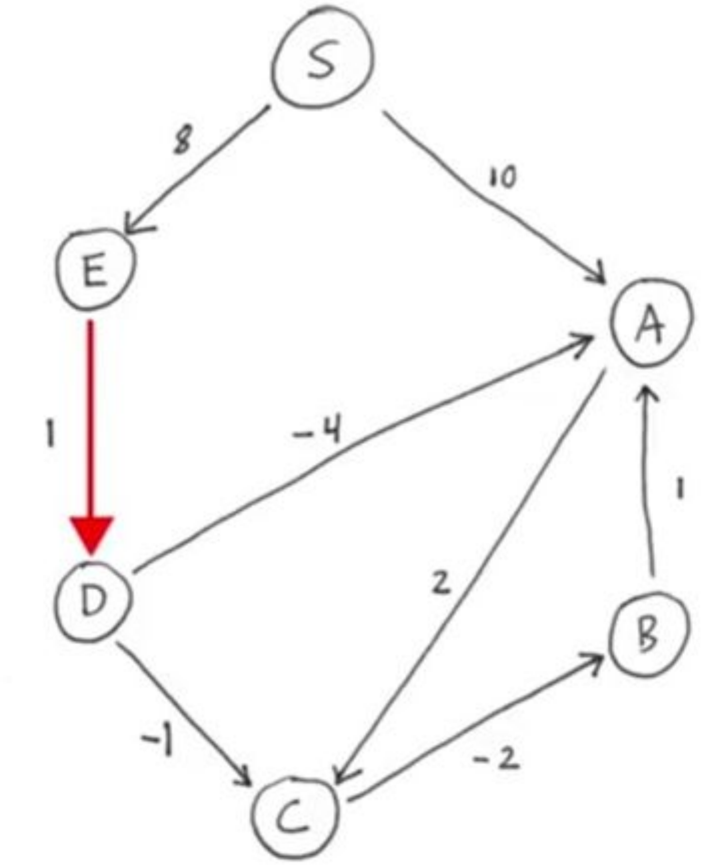
0	10	∞	12	∞	8
S	A	B	C	D	E

1st Iteration



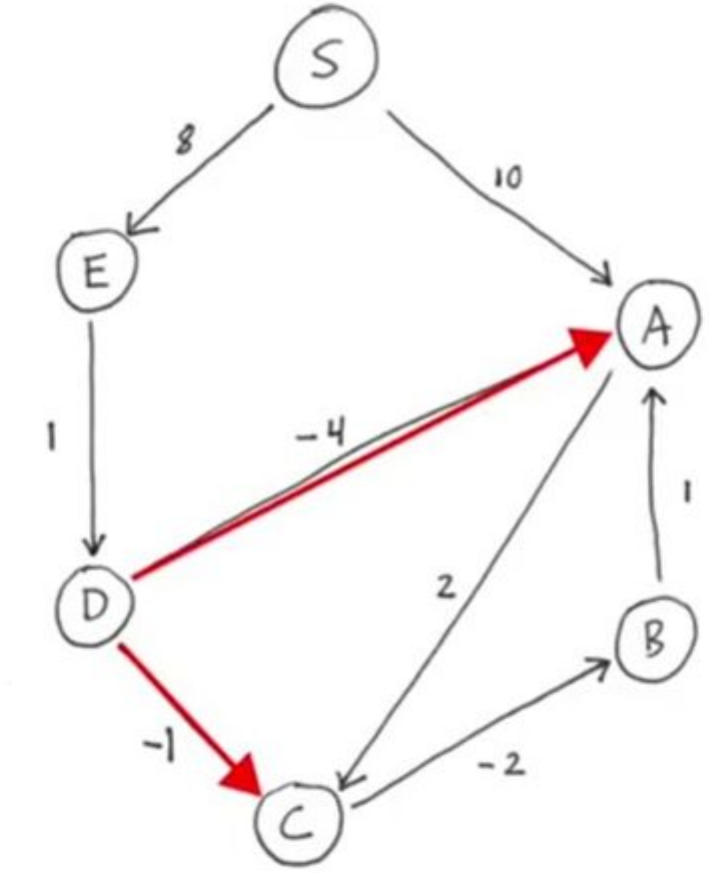
		10			
0	**10**	~~∞~~	**12**	∞	**8**
S	**A**	**B**	**C**	**D**	**E**

1st Iteration



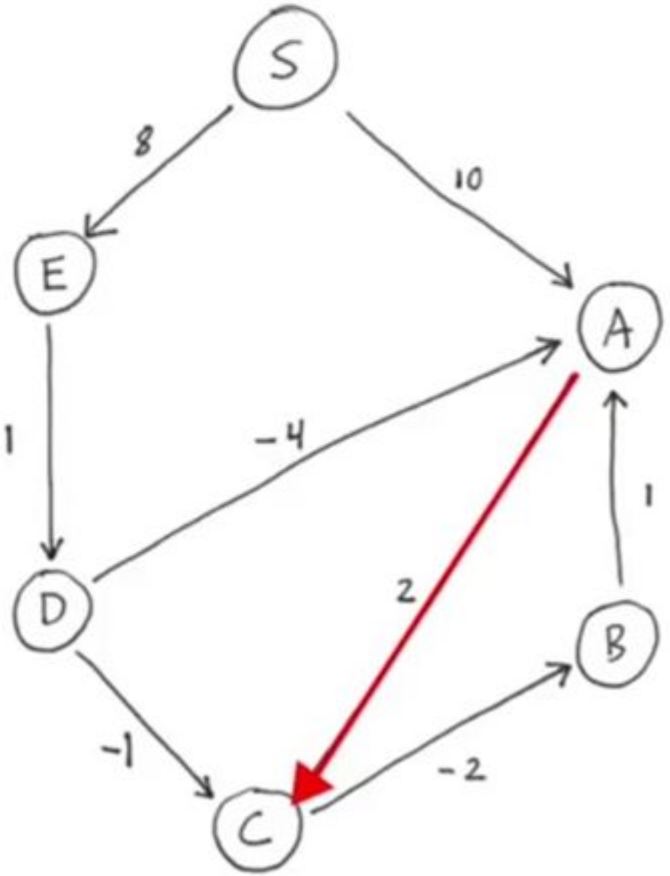
0	**10**	**10**	**12**	~~∞~~ 9	**8**
S	**A**	**B**	**C**	**D**	**E**

1st Iteration



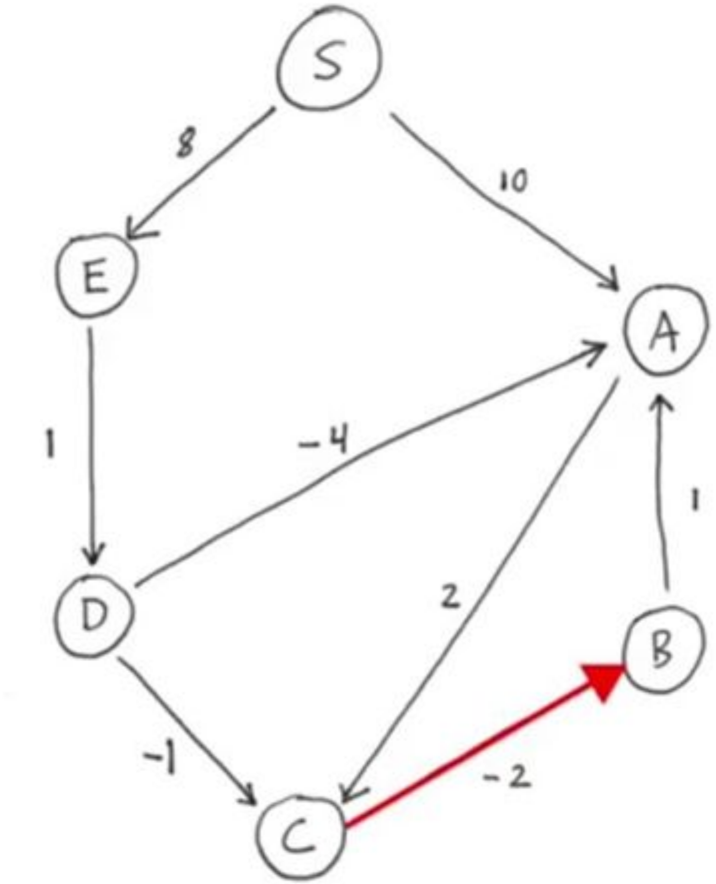
	5		8		
0	~~1~~0	10	~~1~~2	9	8
S	A	B	C	D	E

2nd Iteration



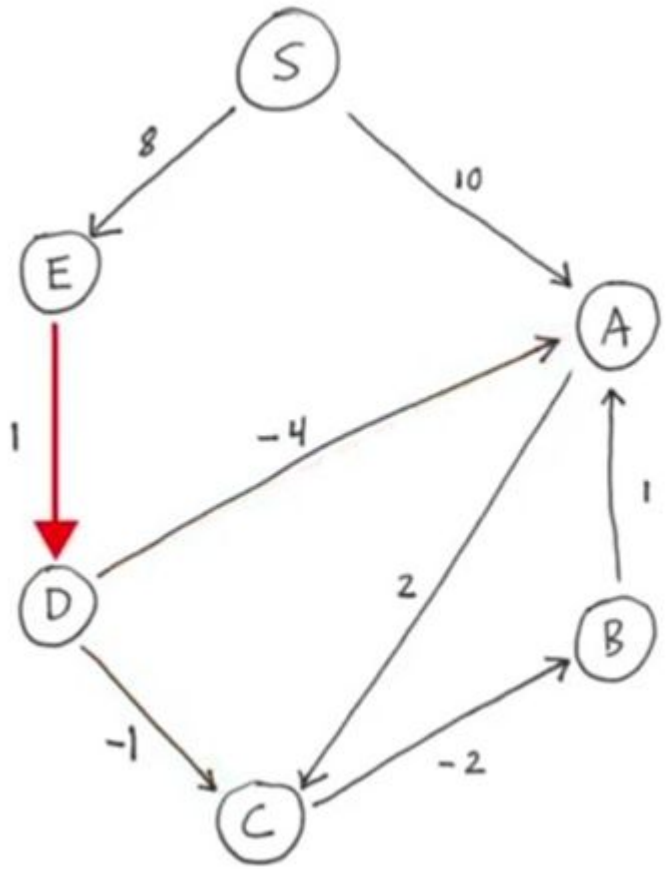
0	5	10	~~8~~ 7	9	8
S	A	B	C	D	E

3rd Iteration



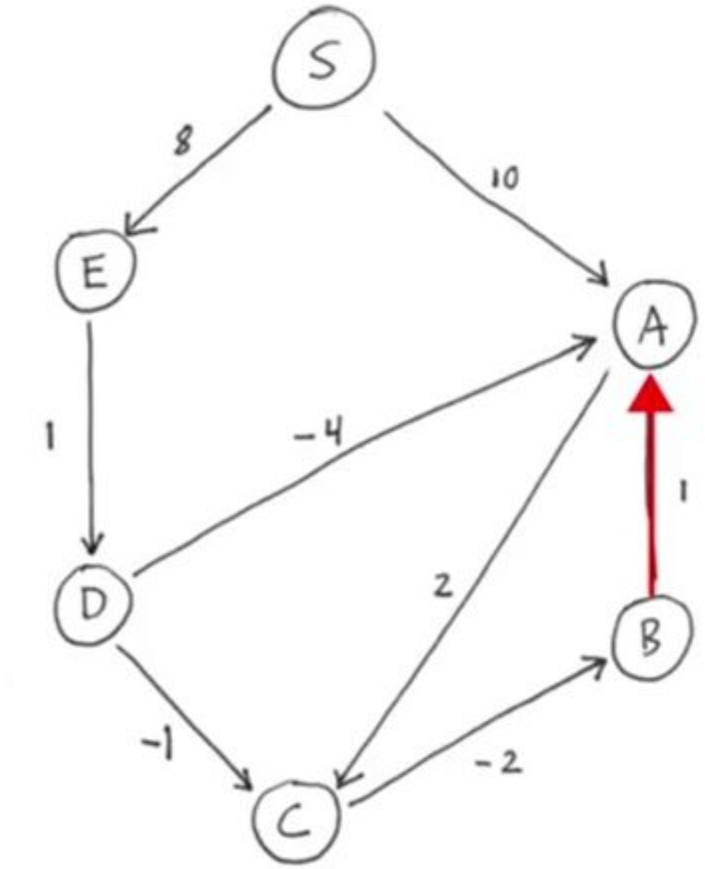
		5			
0	5	~~10~~	7	9	8
S	A	B	C	D	E

3rd Iteration



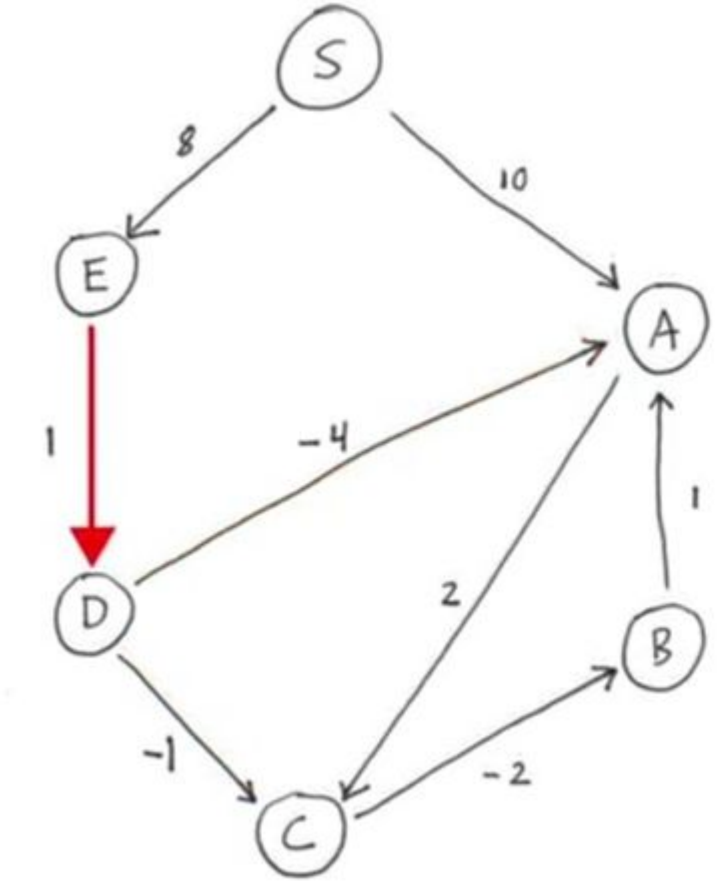
0	5	5	7	9	8
S	A	B	C	D	E

3rd Iteration



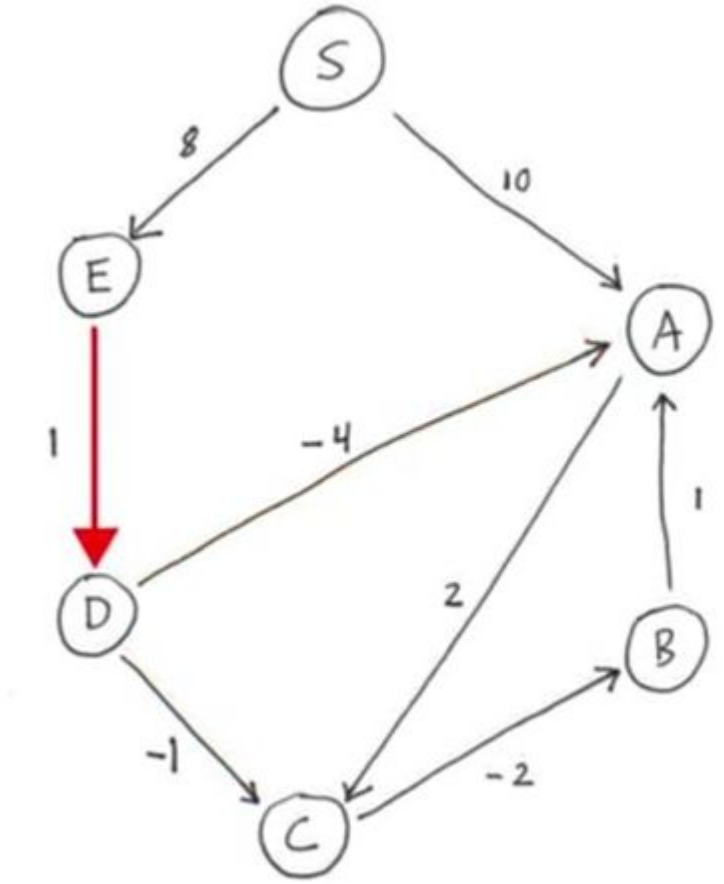
0	5	5	7	9	8
S	A	B	C	D	E

4th Iteration



0	5	5	7	9	8
S	A	B	C	D	E

4th Iteration



0	5	5	7	9	8
S	A	B	C	D	E

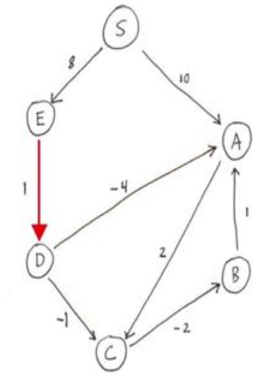
4th Iteration

□As in the 4th iteration no change has taken place. So, it is the final answer.
At most |V|-1 iterations are required for this algorithm to find the shortest to all other vertices from the given vertex

If Bellman ford algorithm is used in graph

G = (V , E) to find the shortest distance from a vertex to all other vertex, then

Time complexity = **O(|V|*|E|)**



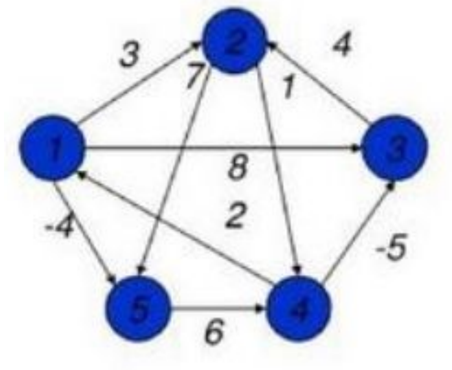
0	5	5	7	9	8
S	A	B	C	D	E

4th Iteration

All Pair Shortest Path Algorithm:

•The all pair shortest path algorithm is also known as Floyd - Warshall algorithm is used to find all pair shortest path problem from a given weighted graph.

•As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.



$$\begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

Floyd Warshall Algorithm:

•We can solve an all-pairs shortest-paths problem by running a single-source shortest-paths algorithm |V| times, once for each vertex as the source.

•Unlike the single-source algorithms, which assume an adjacency-list representation of the graph, Floyd – Warshall algorithm uses an adjacency matrix representation.

•We assume that the vertices are numbered 1, 2, 3, …, |V| so that the input is an n × n matrix W representing the edge weights of an n-vertex directed graph G(V, E).

•The tabular output of the all-pairs shortest-paths algorithms is an n × n matrix D = (d_{ij}) where entry d_{ij} contains the weight of a shortest path from vertex i to vertex j .

Time Complexity for this algorithm

$$O(|n|^3)$$

where n is the number of vertices in graph

Algorithm:

•To solve the all-pairs shortest-paths problem on an input adjacency matrix, we need to compute not only the shortest-path weights but also a ***predecessor matrix*** $\Pi = (\pi_{ij})$ where π_{ij} = NIL, if i = j or there's no path from i to j, otherwise π_{ij} is the predecessor of j on some shortest path from i.

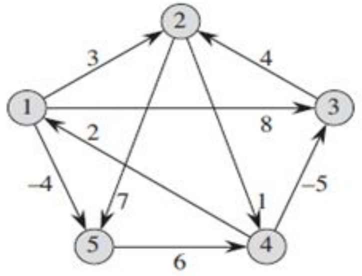
•For each vertex, $i \in V$, we define the predecessor subgraph of G for i as $G_{\pi, i} = (V_{\pi, i}, E_{\pi, i})$, where

$$\mathbf{V_{\pi, i} = \{j \in V : \pi_{ij} \neq NIL\} \cup \{i\}}$$

And,

$$\mathbf{E_{\pi, I} = \{(\pi_{ij}, j) : j \in V_{\pi, i} - \{i\}\}}$$

```
PRINT-ALL-PAIRS-SHORTEST-PATH($\Pi, i, j$)
if $i == j$
    print $i$
elseif $\pi_{ij} ==$ NIL
    print "no path from" $i$ "to" $j$ "exists"
else PRINT-ALL-PAIRS-SHORTEST-PATH($\Pi, i, \pi_{ij}$)
    print $j$
```



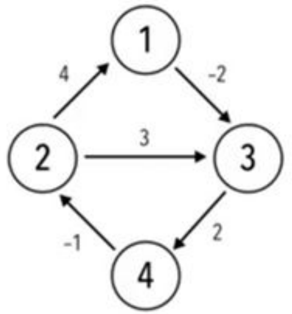
$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Example:

k = 1 2 3 4
i = 1 2 3 4
j = 1 2 3 4

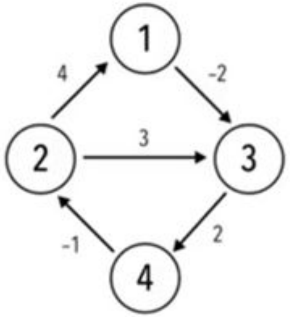
dist [i][j] > dist [i][k] + dist [k][j]
dist [1][1] > dist [1][1] + dist [1][1]
0 > 0 + 0
X 0 > 0



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

k = 1 2 3 4
i = 1 2 3 4
j = 1 2 3 4

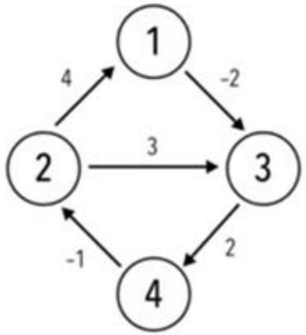
dist [i][j] > dist [i][k] + dist [k][j]
dist [1][2] > dist [1][1] + dist [1][2]
$\infty > 0 + \infty$
X $\infty > \infty$



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

k = 1 2 3 4
i = 1 2 3 4
j = 1 2 3 4

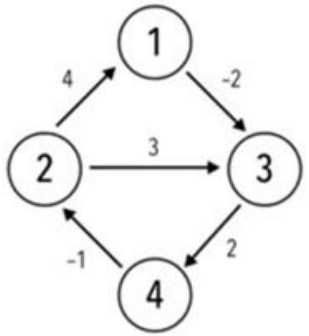
dist [i][j] > dist [i][k] + dist [k][j]
dist [1][3] > dist [1][1] + dist [1][3]
−2 > 0 + −2
X −2 > −2



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

k = 1 2 3 4
i = 1 2 3 4
j = 1 2 3 4

dist [i][j] > dist [i][k] + dist [k][j]
dist [1][4] > dist [1][1] + dist [1][4]
∞ > 0 + ∞
X ∞ > ∞

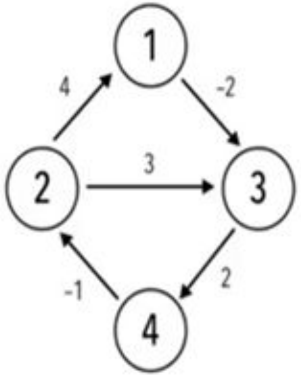


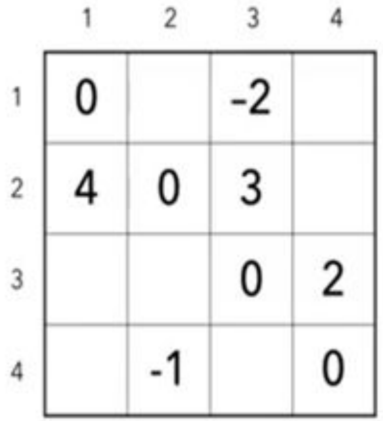
	1	2	3	4
1	0		-2	
2	4	0	3	
3			0	2
4		-1		0

When condition becomes true we would overwrite the position in matrix.

k = 1 2 3 4
i = 1 2 3 4
j = 1 2 3 4

dist [i][j] > dist [i][k] + dist [k][j]
dist [2][3] > dist [2][1] + dist [1][3]
3 > 4 + -2
3 > 2





$$\begin{array}{c|cccc}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & & -2 & \\
2 & 4 & 0 & 3 & \\
3 & & & 0 & 2 \\
4 & & -1 & & 0
\end{array}$$

k = 1 2 3 4

i = 1 2 3 4

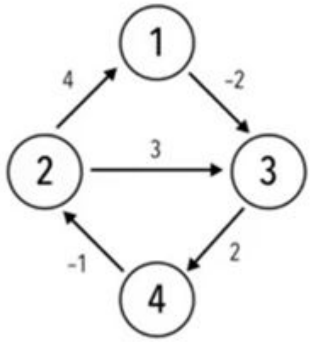
j = 1 2 3 4

dist [i][j] > dist [i][k] + dist [k][j]

dist [2][3] > dist [2][1] + dist [1][3]

3 > 4 + -2

3 > 2



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

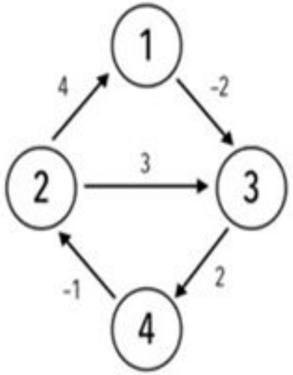
When condition becomes true

we would overwrite the position

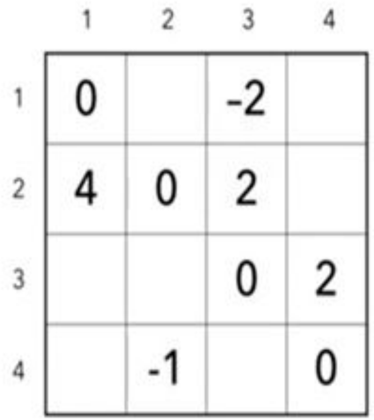
in matrix

k = 1 2 3 4
i = 1 2 3 4
j = 1 2 3 4

dist [i][j] > dist [i][k] + dist [k][j]
dist [4][1] > dist [4][2] + dist [2][1]
$\infty > -1 + 4$
$\infty > 3$



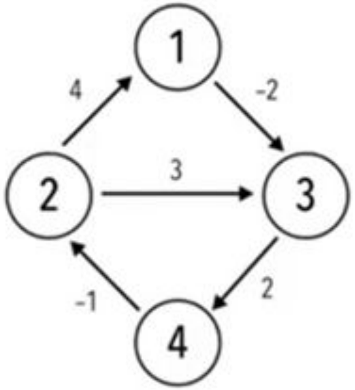
$$\begin{tikzcd}
 & 1 \arrow[dr, "-2"] & \\
2 \arrow[ur, "4"] \arrow[rr, "3"] & & 3 \arrow[dl, "2"] \\
 & 4 \arrow[ul, "-1"] &
\end{tikzcd}$$



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

k = 1 2 3 4
i = 1 2 3 4
j = 1 2 3 4

dist [i][j] > dist [i][k] + dist [k][j]
dist [4][1] > dist [4][2] + dist [2][1]
$\infty > -1 + 4$
$\infty > 3$



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

When condition becomes true

we would overwrite the position

in matrix

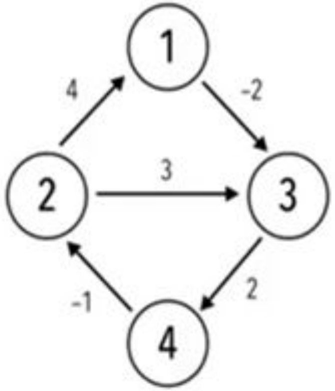
```
k = 1 2 3 4
i = 1 2 3 4
j = 1 2 3 4
```

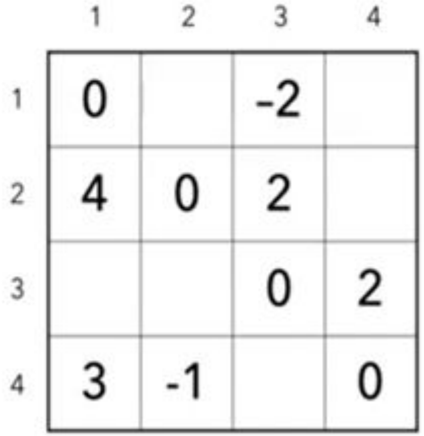
dist [i][j] > dist [i][k] + dist [k][j]

dist [4][3] > dist [4][2] + dist [2][3]

$\infty > -1 + 2$

$\infty > 1$





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

$k = 1\ 2\ 3\ 4$

$i = 1\ 2\ 3\ 4$

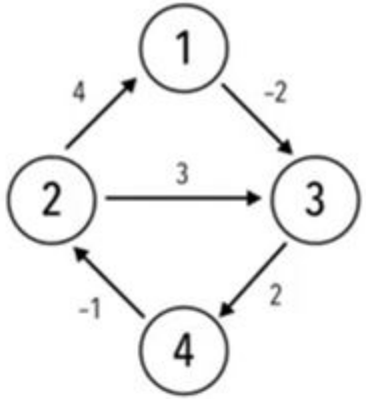
$j = 1\ 2\ 3\ 4$

dist [i][j] > dist [i][k] + dist [k][j]

dist [4][3] > dist [4][2] + dist [2][3]

$\infty > -1 + 2$

$\infty > 1$



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

When condition becomes true

we would overwrite the position

in matrix

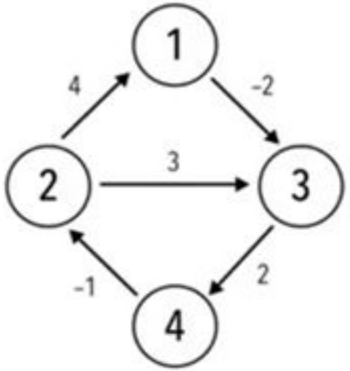
$$\begin{array}{lllll} k = & 1 & 2 & 3 & 4 \\ i = & 1 & 2 & 3 & 4 \\ j = & 1 & 2 & 3 & 4 \end{array}$$

dist [i][j] > dist [i][k] + dist [k][j]

dist [1][4] > dist [1][3] + dist [3][4]

$\infty > -2 + 2$

$\infty > 0$



$$\begin{array}{c|c|c|c|c|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & & -2 & \\ \hline 2 & 4 & 0 & 2 & \\ \hline 3 & & & 0 & 2 \\ \hline 4 & 3 & -1 & 1 & 0 \\ \hline \end{array}$$

When condition becomes true

we would overwrite the position

in matrix

k = 1 2 3 4

i = 1 2 3 4

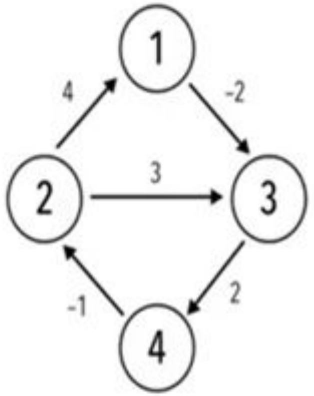
j = 1 2 3 4

dist [i][j] > dist [i][k] + dist [k][j]

dist [1][4] > dist [1][3] + dist [3][4]

$\infty > -2 + 2$

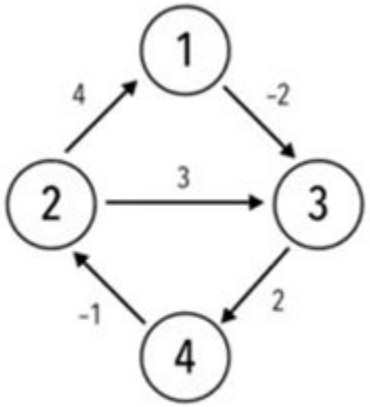
$\infty > 0$



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

k = 1 2 3 4
i = 1 2 3 4
j = 1 2 3 4

dist [i][j] > dist [i][k] + dist [k][j]
dist [2][4] > dist [2][3] + dist [3][4]
∞ > 2 + 2
∞ > 4



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

When condition becomes true

we would overwrite the position

in matrix

k = 1 2 3 4

i = 1 2 3 4

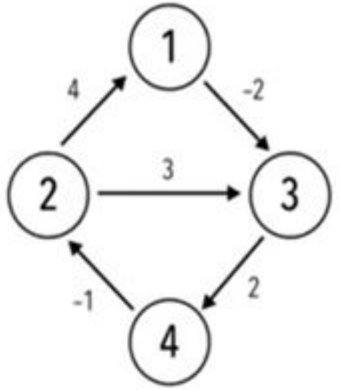
j = 1 2 3 4

dist [i][j] > dist [i][k] + dist [k][j]

dist [2][4] > dist [2][3] + dist [3][4]

$\infty > 2 + 2$

$\infty > 4$



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

k = 1 2 3 4

i = 1 2 3 4

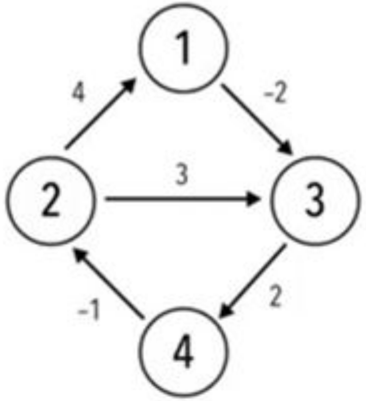
j = 1 2 3 4

dist [i][j] > dist [i][k] + dist [k][j]

dist [1][2] > dist [1][4] + dist [4][2]

∞ > 0 + –1

∞ > –1



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

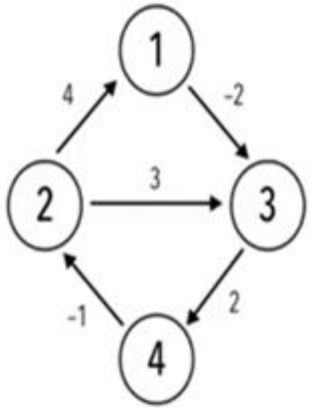
When condition becomes true

we would overwrite the position

in matrix

k = 1 2 3 4
i = 1 2 3 4
j = 1 2 3 4

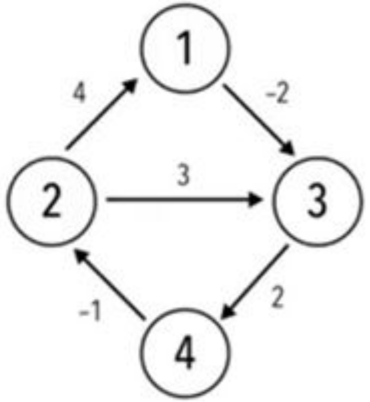
dist [i][j] > dist [i][k] + dist [k][j]
dist [1][2] > dist [1][4] + dist [4][2]
$\infty > 0 + -1$
$\infty > -1$



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

k = 1 2 3 4
i = 1 2 3 4
j = 1 2 3 4

dist [i][j] > dist [i][k] + dist [k][j]
dist [3][1] > dist [3][4] + dist [4][1]
$\infty > 2 + 3$
$\infty > 5$

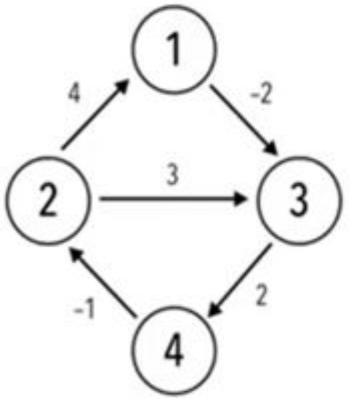


	1	2	3	4
1	0	-1	-2	0
2	4	0	2	4
3	5		0	2
4	3	-1	1	0

When condition becomes true we would overwrite the position in matrix

k = 1 2 3 4
i = 1 2 3 4
j = 1 2 3 4

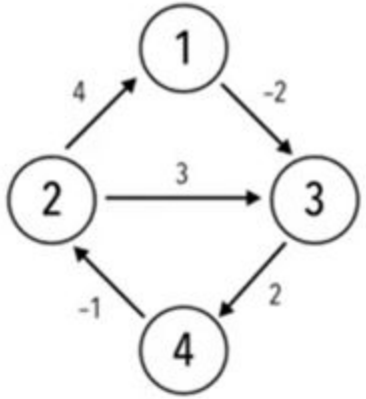
dist [i][j] > dist [i][k] + dist [k][j]
dist [3][2] > dist [3][4] + dist [4][2]
$\infty > 2 + -1$
$\infty > 1$



	1	2	3	4
1	0	-1	-2	0
2	4	0	2	4
3	5		0	2
4	3	-1	1	0

k = 1 2 3 4
i = 1 2 3 4
j = 1 2 3 4

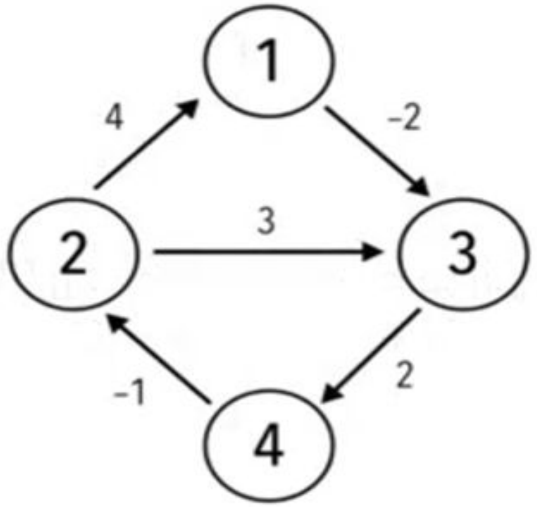
dist [i][j] > dist [i][k] + dist [k][j]
dist [3][2] > dist [3][4] + dist [4][2]
$\infty > 2 + -1$
$\infty > 1$

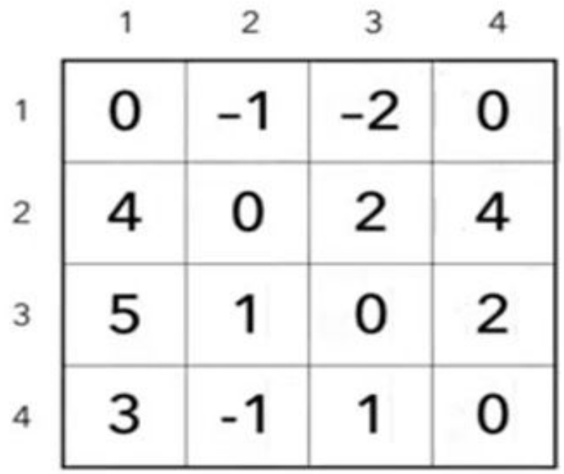


	1	2	3	4
1	0	-1	-2	0
2	4	0	2	4
3	5	1	0	2
4	3	-1	1	0

FINAL RESULTANT

MATRIX





	1	2	3	4
1	0	–1	–2	0
2	4	0	2	4
3	5	1	0	2
4	3	-1	1	0