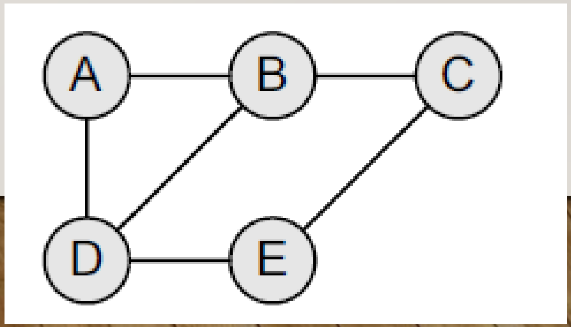
GRAPHS

UNDIRECTED GRAPH

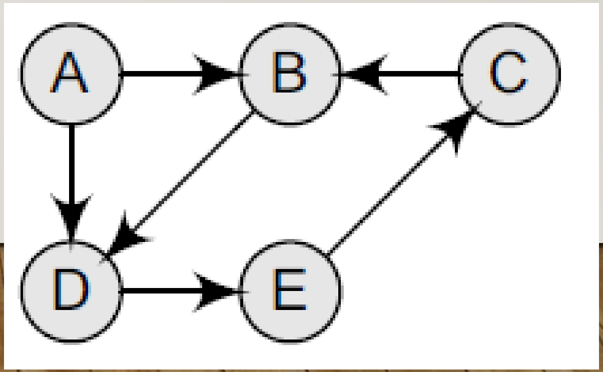
Definition

- A graph `G` is defined as an ordered set `(V, E)`, where `V(G)` represents the set of vertices and `E(G)` represents the edges that connect these vertices.
- Graph `V(G) = {A, B, C, D, E}` and `E(G) = {(A, B), (B, C), (A, D), (B, D), (D, E), (C, E)}`.
- Note that there are five vertices or nodes and six edges in the graph.



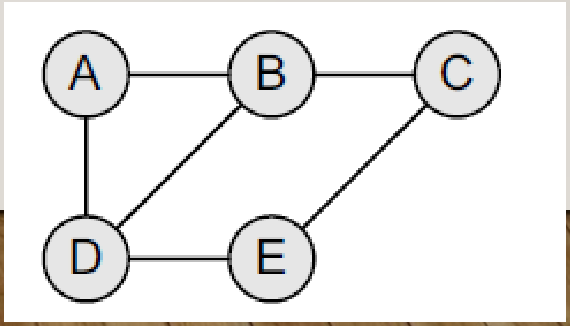
DIRECTED GRAPH

- In an undirected graph, edges do not have any direction associated with them. That is, if an edge is drawn between nodes `A` and `B`, then the nodes can be traversed from `A` to `B` as well as from `B` to `A`.
- In a directed graph, edges form an ordered pair. If there is an edge from `A` to `B`, then there is a path from `A` to `B` but not from `B` to `A`. The edge `(A, B)` is said to initiate from node `A` (also known as initial node) and terminate at node `B` (terminal node).



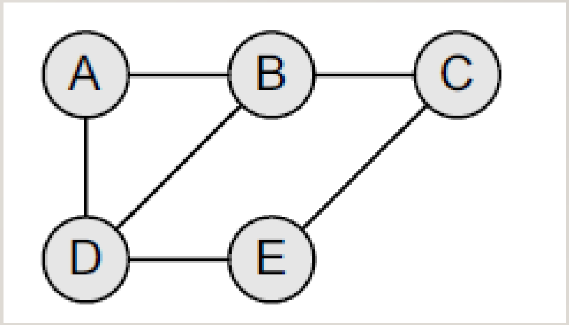
GRAPH TERMINOLOGIES

- ***Adjacent nodes or neighbors*** For every edge, `e = (u, v)` that connects nodes `u` and `v`, the nodes `u` and `v` are the end-points and are said to be the adjacent nodes or neighbours.
- ***Degree of a node*** Degree of a node `u`, `deg(u)`, is the total number of edges containing the node `u`.
- If `deg(u) = 0`, it means that `u` does not belong to any edge and such a node is known as an isolated node.



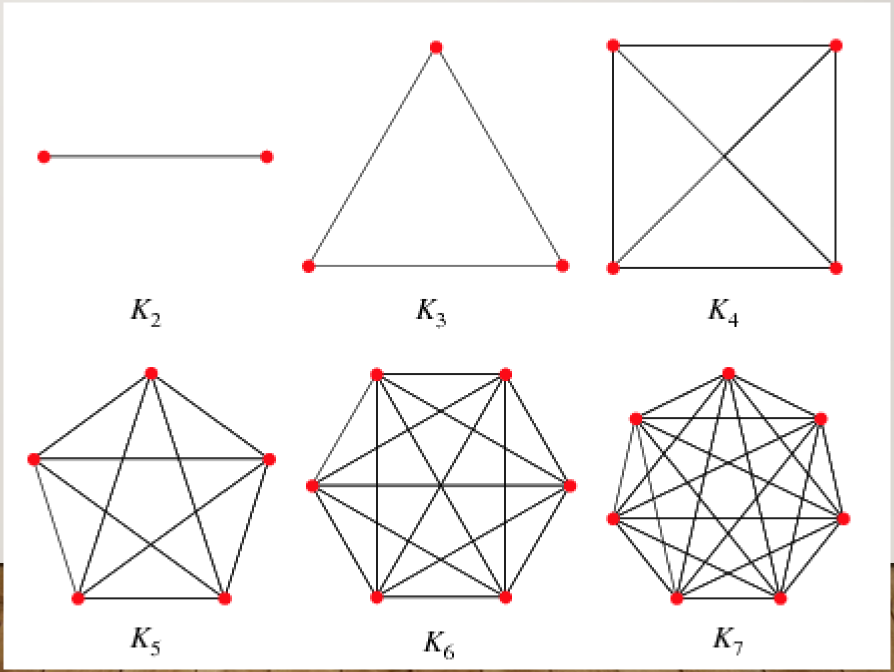
GRAPH TERMINOLOGIES

- ***Path*** A path `P` written as `P = {v0, v1, v2, ..., vn)`, of length `n` from a node `u` to `v` is defined as a sequence of `(n+1)` nodes. Here, `u = v0, v = vn` and `vi-1` is adjacent to `vi` for `i = 1, 2, 3,..., n`.
- ***Cycle*** A path in which the first and the last vertices are same. A *simple cycle* has no repeated edges or vertices (except the first and last vertices).



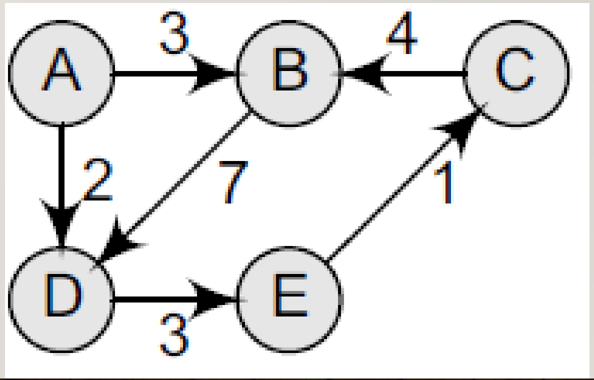
GRAPH TERMINOLOGIES

- ***Complete graph*** A graph `G` is said to be complete if all its nodes are fully connected. That is, there is a path from one node to every other node in the graph. A complete graph has `n(n−1)/2` edges, where `n` is the number of nodes in `G`.



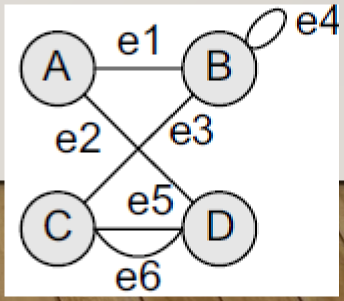
GRAPH TERMINOLOGIES

- ***Labelled graph or weighted graph*** A graph is said to be labelled if every edge in the graph is assigned some data. In a weighted graph, the edges of the graph are assigned some weight or length. The weight of an edge denoted by `w(e)` is a positive value which indicates the cost of traversing the edge. Figure 13.4(c) shows a weighted graph.



GRAPH TERMINOLOGIES

- ***Multiple edges*** Distinct edges which connect the same end-points are called multiple edges. That is, `e = (u, v)` and `e' = (u, v)` are known as multiple edges of `G`.
- ***Loop*** An edge that has identical end-points is called a loop. That is, `e=(u,u)`.
- ***Multi-graph*** A graph with multiple edges and/or loops is called a multi-graph.
- ***Size of a graph*** The size of a graph is the total number of edges in it.



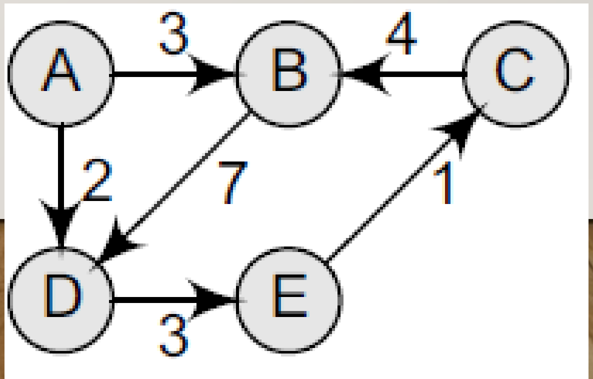
DIRECTED GRAPHS

- A directed graph `G`, also known as a *digraph*, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair `(u, v)` of nodes in `G`. For an edge `(u, v)`,
- The edge begins at `u` and terminates at `v`.
- `u` is known as the **origin** or initial point of `e`. Correspondingly, `v` is known as the **destination** or terminal point of `e`.
- `u` is the predecessor of `v`. Correspondingly, `v` is the successor of `u`.
- Nodes `u` and `v` are adjacent to each other.

DIRECTED GRAPH TERMINOLOGY

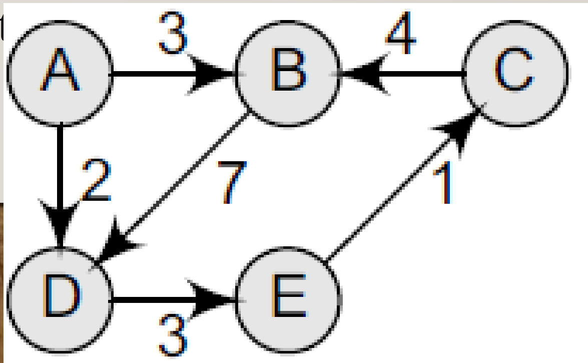
- ***Out-degree of a node*** The out-degree of a node `u`, written as `outdeg(u)`, is the number of edges that originate at `u`.
- ***In-degree of a node*** The in-degree of a node `u`, written as `indeg(u)`, is the number of edges that terminate at `u`.
- ***Degree of a node*** The degree of a node, written as `deg(u),` is equal to the sum of in-degree and out-degree of that node. Therefore,

  `deg(u) = indeg(u) + outdeg(u).`
- ***Isolated vertex*** A vertex with degree zero. Such a vertex is not an end-point of any edge.



DIRECTED GRAPH TERMINOLOGY

- ***Source*** A node u is known as a source if it has a positive out-degree but a zero in-degree.
- ***Sink*** A node u is known as a sink if it has a positive in-degree but a zero out-degree.
- ***Reachability*** A node v is said to be reachable from node u, if and only if there exists a (directed) path from node u to node v.
- ***Strongly connected directed graph*** A digraph is said to be strongly connected if and only if there exists a path between every pair of nodes in G. That is, if there is a path from node u to v, then there must be a pat


$$\begin{tikzcd}
A \arrow[r, "3"] \arrow[d, "2"] & B \arrow[dl, "7"] & C \arrow[l, "4"'] \\
D \arrow[r, "3"'] & E \arrow[ur, "1"'] &
\end{tikzcd}$$

APPLICATIONS

- Graphs are widely used to model any situation where entities or things are related to each other in pairs. For example, the following information can be represented by graphs:
  - *Family trees* in which the member nodes have an edge from parent to each of their children.
  - *Transportation networks* in which nodes are airports, intersections, ports, etc. The edges can be airline flights, one-way roads, shipping routes, etc.

REPRESENTATION OF GRAPHS

REPRESENTATION OF GRAPHS

Three common ways of storing graphs:

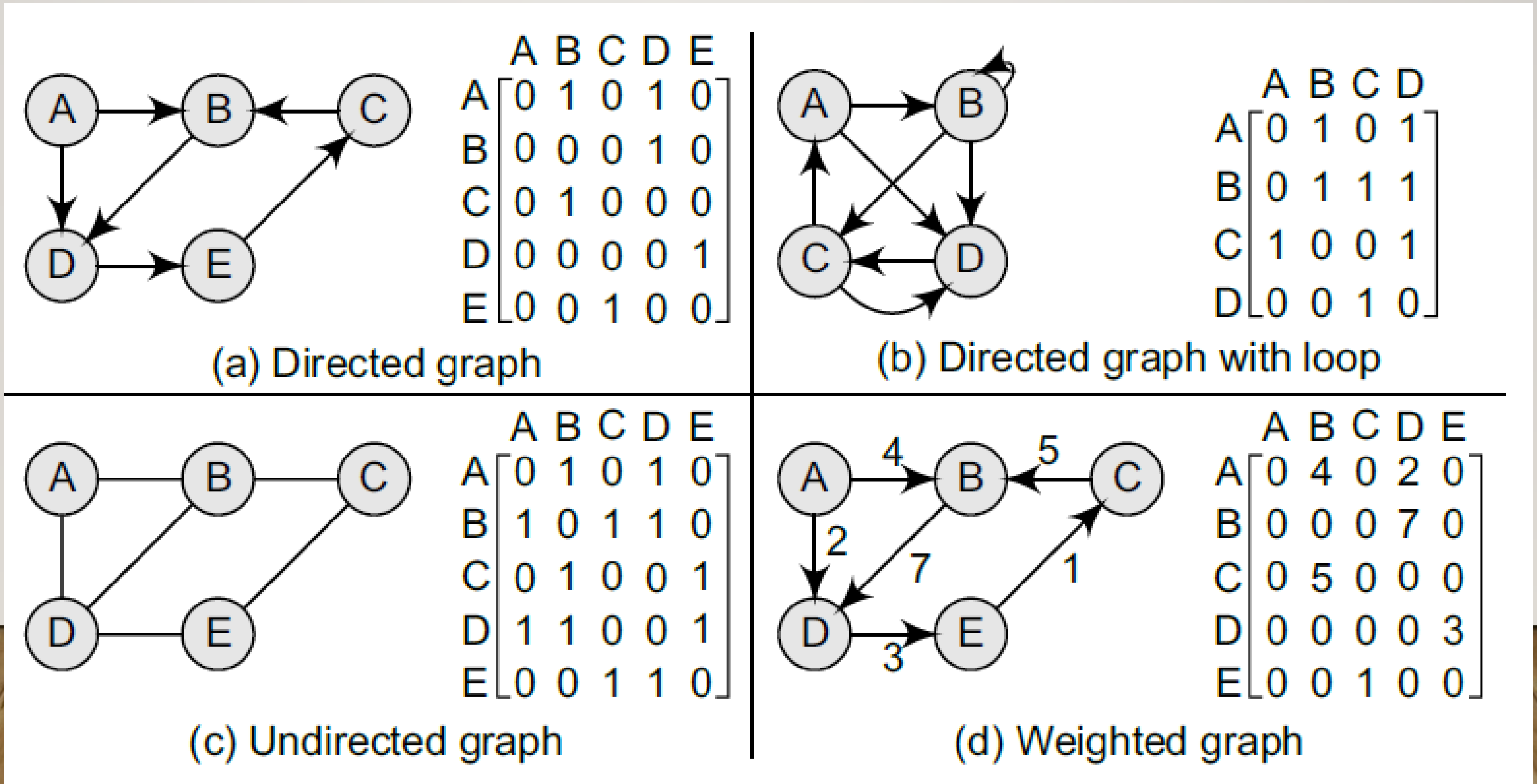
1. *Sequential representation by using an adjacency matrix.*

2. *Linked representation by using an adjacency list that stores the neighbors of a node using a linked list.*

3. *Adjacency multi-list which is an extension of linked representation.*

ADJACENCY MATRIX

In an adjacency matrix, the rows and columns are labelled by graph vertices. An entry `aij` in the adjacency matrix will contain 1, if vertices `vi` and `vj` are adjacent to each other. However, if the nodes are not adjacent, `aij` will be set to zero.



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 1 & 0 \\ B & 0 & 0 & 0 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 0 \\ D & 0 & 0 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 0 & 0 \end{array}$$

(a) Directed graph

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 0 & 1 \\ B & 0 & 1 & 1 & 1 \\ C & 1 & 0 & 0 & 1 \\ D & 0 & 0 & 1 & 0 \end{array}$$

(b) Directed graph with loop

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 1 & 0 \\ B & 1 & 0 & 1 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 1 \\ D & 1 & 1 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 1 & 0 \end{array}$$

(c) Undirected graph

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 4 & 0 & 2 & 0 \\ B & 0 & 0 & 0 & 7 & 0 \\ C & 0 & 5 & 0 & 0 & 0 \\ D & 0 & 0 & 0 & 0 & 3 \\ E & 0 & 0 & 1 & 0 & 0 \end{array}$$

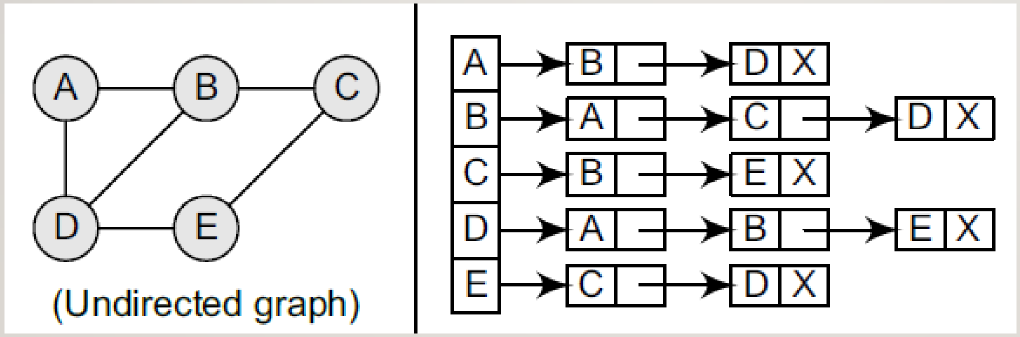
(d) Weighted graph

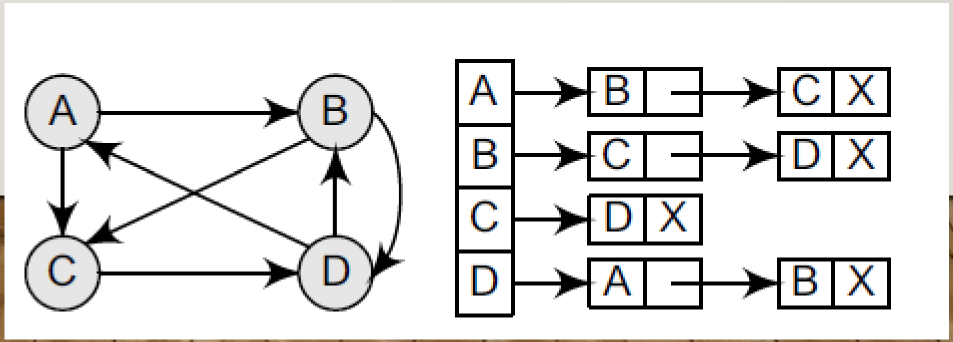
OBSERVATIONS

- The adjacency matrix of an undirected graph is symmetric.
- The memory use of an adjacency matrix is `O(n^2)`, where `n` is the number of nodes in the graph.
- Number of 1s (or non-zero entries) in an adjacency matrix is equal to the number of edges in the graph.
- The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.

ADJACENCY LIST REPRESENTATION

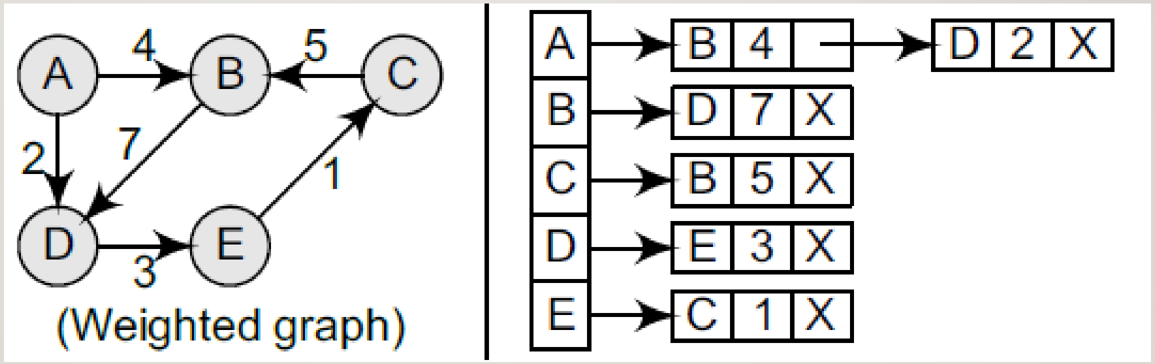
- This structure consists of a list of all nodes in G. Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it.





ADJACENCY LIST REPRESENTATION FOR WEIGHTED GRAPH

- Adjacency lists can also be modified to store weighted graphs



OBSERVATIONS

- The adjacency matrix of an undirected graph is symmetric.
- It is easy to follow and clearly shows the adjacent nodes of a particular node.
- It is often used for storing graphs that have a small-to-moderate number of edges. That is, an adjacency list is preferred for representing sparse graphs in the computer’s memory; otherwise, an adjacency matrix is a good choice.
- Adding new nodes in G is easy and straightforward when G is represented using an adjacency list.
- Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered.

GRAPH TRAVERSAL ALGORITHMS

GRAPH TRAVERSAL ALGORITHMS

- There are two standard methods of graph traversal. These two methods are:

    1. Breadth-first search

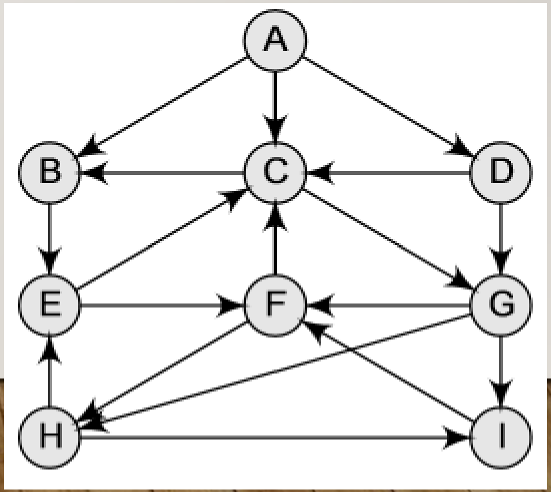
        - uses QUEUE

    2. Depth-first search

        - uses STACK

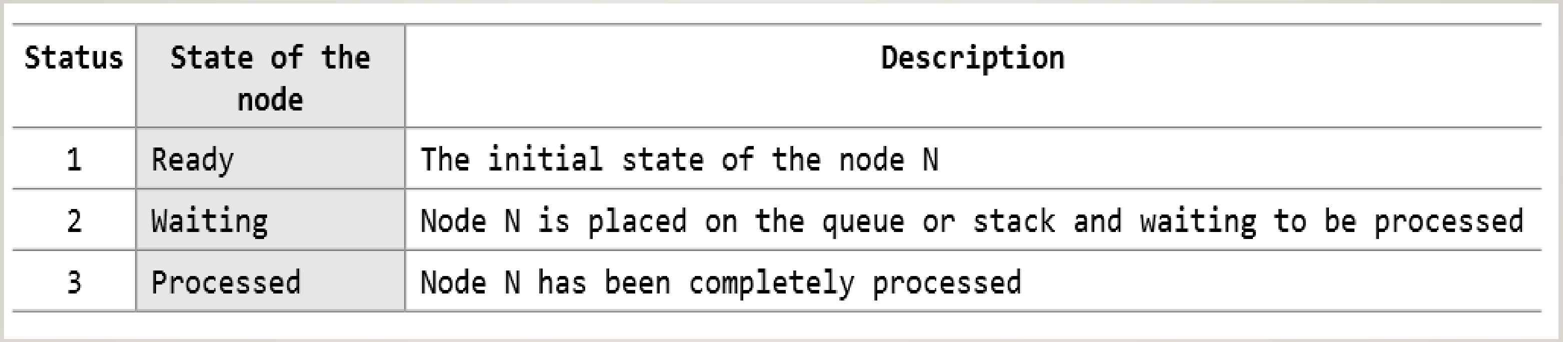
BREADTH-FIRST SEARCH ALGORITHM (BFS)

- Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighboring nodes.
- Then for each of those nearest nodes, the algorithm explores their unexplored neighbor nodes, and so on, until it finds the goal



STATUS OF VERTEX IN ALGORITHMS

- Value of status and its significance



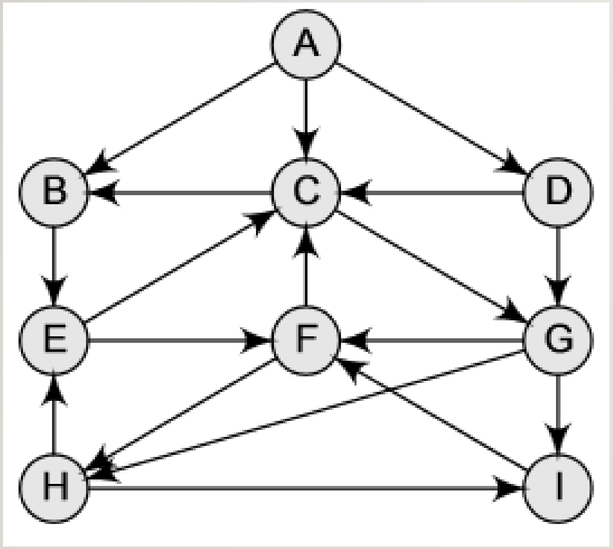
Status	State of the node	Description
1	Ready	The initial state of the node N
2	Waiting	Node N is placed on the queue or stack and waiting to be processed
3	Processed	Node N has been completely processed

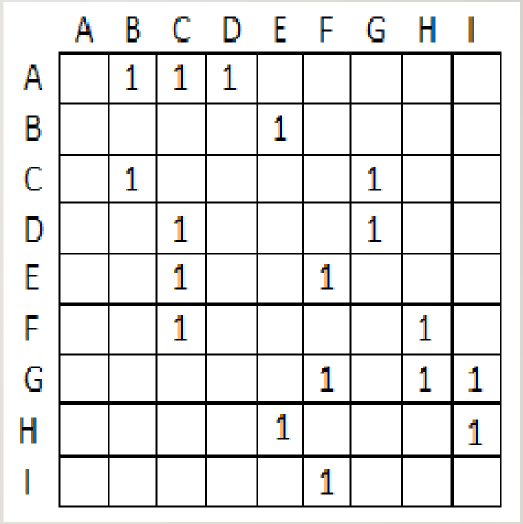
BREADTH-FIRST SEARCH ALGORITHM (BFS)

- Step 1: SET STATUS=1(ready state)for each node in G
- Step 2: Enqueue the starting node A and set its STATUS=2(waiting state)
- Step 3: Repeat Steps 4 and 5 until QUEUE is empty
- Step 4: Dequeue a node N. Process it and set its STATUS=3(processed state).
- Step 5: Enqueue all the neighbors of N that are in the ready state(whose STATUS=1) and set their STATUS=2(waiting state)[END OF LOOP]
- Step 6: EXIT

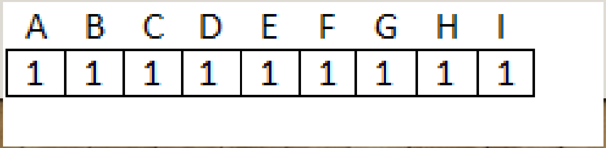
BREADTH-FIRST SEARCH ALGORITHM (BFS)

Initialization

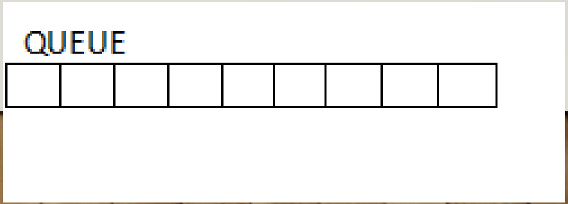




	A	B	C	D	E	F	G	H	I
A		1	1	1					
B					1				
C		1					1		
D			1				1		
E			1			1			
F			1					1	
G						1		1	1
H					1				1
I						1			



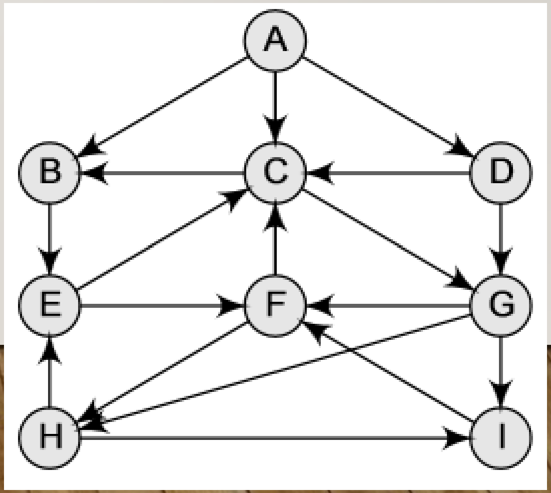
A	B	C	D	E	F	G	H	I
1	1	1	1	1	1	1	1	1



QUEUE

DEPTH-FIRST SEARCH ALGORITHM (DFS)

- The depth-first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered.
- When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored

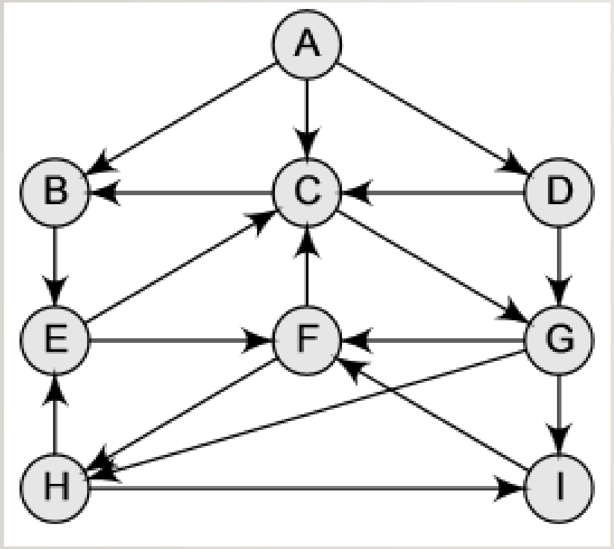


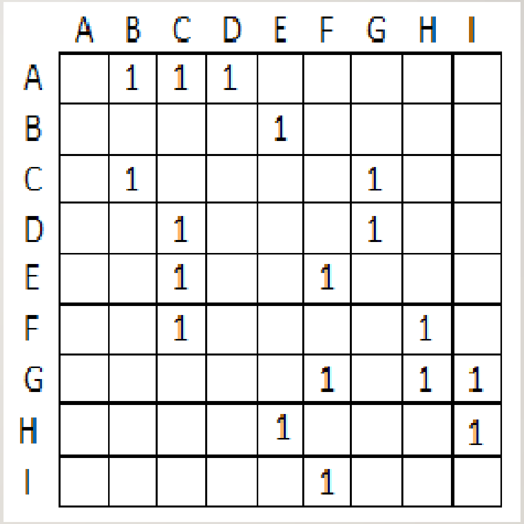
DEPTH-FIRST SEARCH ALGORITHM (DFS)

- Step 1: SET STATUS=1(ready state) for each node in G
- Step2: Push the starting node A on the stack and set its STATUS=2(waiting state)
- Step3: Repeat Steps 4 and 5 until STACK is empty
- Step4: Pop the top node N. Process it and set its STATUS=3(processed state)
- Step5: Push on the stack all the neighbors of N that are in the ready state (whose STATUS=1) and set their STATUS=2(waiting state)[END OF LOOP]Step
- 6: EXIT

DEPTH-FIRST SEARCH ALGORITHM (DFS)

Initialization





	A	B	C	D	E	F	G	H	I
A		1	1	1					
B					1				
C		1					1		
D			1				1		
E			1			1			
F			1					1	
G						1		1	1
H					1				1
I						1			

A	B	C	D	E	F	G	H	I
1	1	1	1	1	1	1	1	1

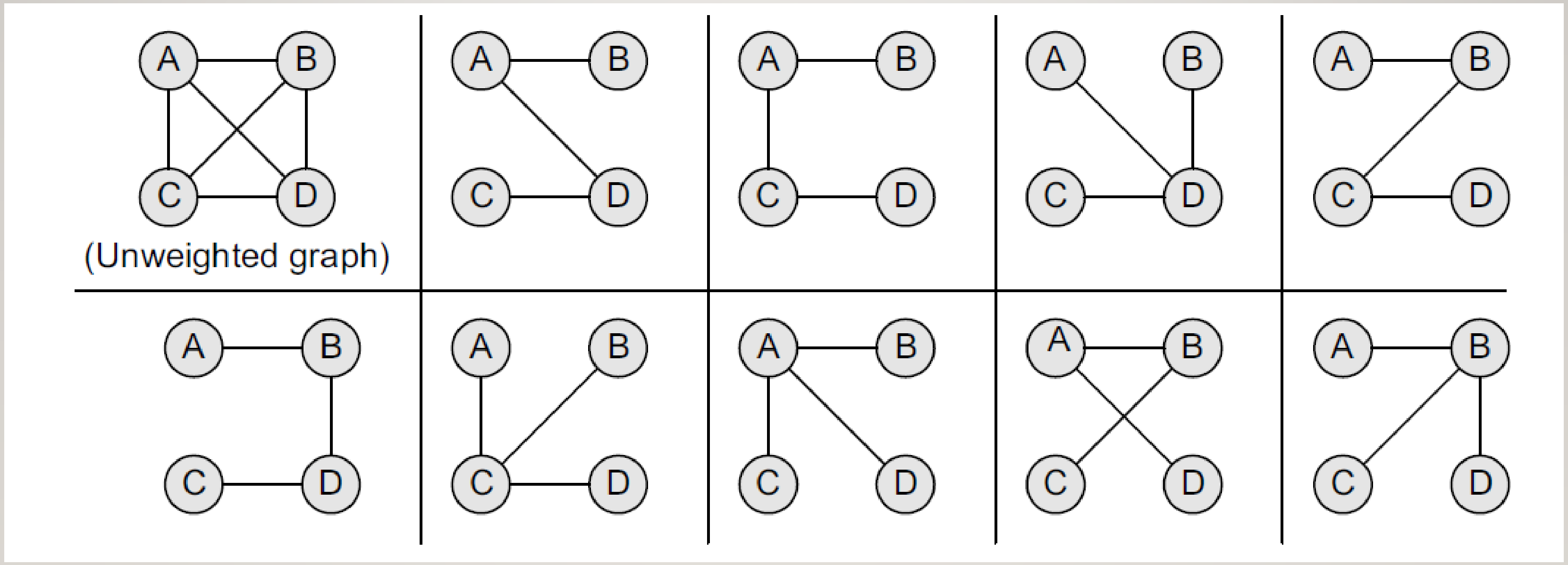
STACK

SHORTEST PATH ALGORITHM

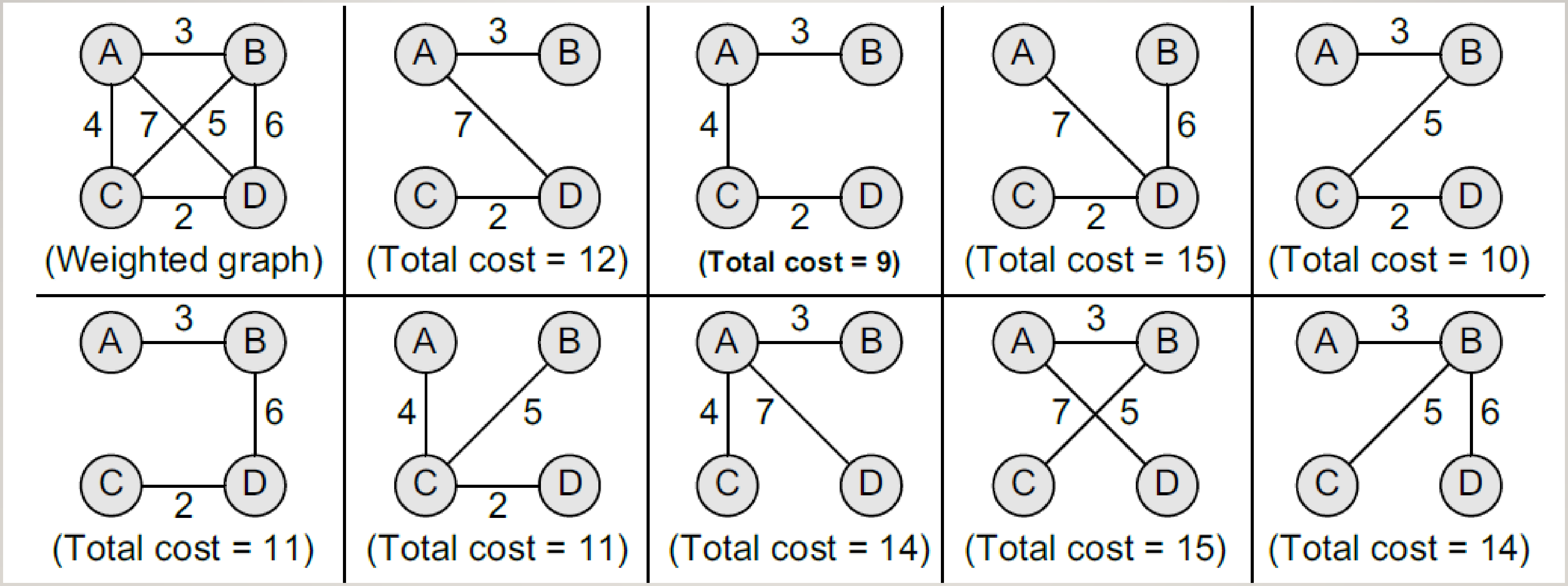
MINIMUM SPANNING TREE

- A spanning tree of a connected, undirected graph G is a sub-graph of G which is a tree that connects all the vertices together
- A graph G can have many different spanning trees
- A *minimum spanning tree* (MST) is defined as a spanning tree with weight less than or equal to the weight of every other spanning tree

MINIMUM SPANNING TREE



MINIMUM SPANNING TREE



PROPERTIES OF MST

- ***Possible multiplicity:*** There can be multiple minimum spanning trees of the same weight.
- ***Uniqueness:*** When each edge in the graph is assigned a different weight, then there will be only one unique minimum spanning tree.
- ***Minimum-cost subgraph:*** If the edges of a graph are assigned *non-negative* weights, then a minimum spanning tree is in fact the minimum-cost subgraph or a tree that connects all vertices.
- ***Cycle property:*** If there exists a cycle C in the graph G that has a weight larger than that of other edges of C, then this edge cannot belong to an MST.
- ***Usefulness:*** Minimum spanning trees can be computed quickly and easily to provide optimal solutions. These trees create a sparse subgraph that reflects a lot about the original graph.
- ***Simplicity*** The minimum spanning tree of a weighted graph is nothing but a spanning tree of the graph which comprises of n–1 edges of minimum total weight.

APPLICATIONS OF MINIMUM SPANNING TREES

1. **MSTs are widely used for designing networks**. For instance, people separated by varying distances wish to be connected together through a telephone network. A minimum spanning tree is used to determine the least costly paths with no cycles in this network, thereby providing a connection that has the minimum cost involved.

2. **MSTs are used to find shortest path**. While the vertices in the graph denote cities, edges represent the routes between places. More the distance between the cities, higher will be the amount charged. Therefore, MSTs are used to optimize airline routes by finding the least costly path with no cycles.

PRIM'S ALGORITHM

- Prim's algorithm is a greedy algorithm that is used to form a minimum spanning tree for a connected weighted undirected graph

PRIMS ALGORITHM

- ***Tree vertices*** Vertices that are a part of the minimum spanning tree T.
- ***Fringe vertices*** Vertices that are currently not a part of T, but are adjacent to some tree vertex.
- ***Unseen vertices*** Vertices that are neither tree vertices nor fringe vertices fall under this category.

PRIM'S ALGORITHM

Step 1: Select a starting vertex

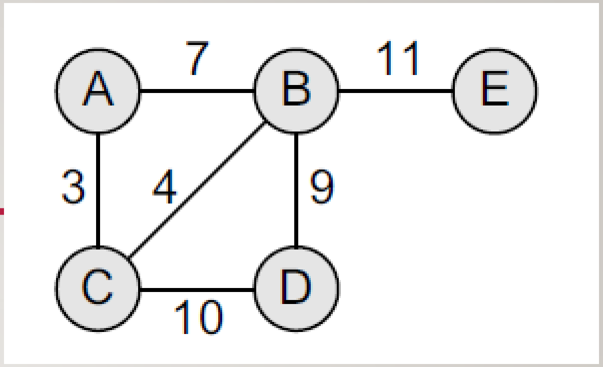
Step 2: Repeat Steps 3 and 4 until there are fringe vertices

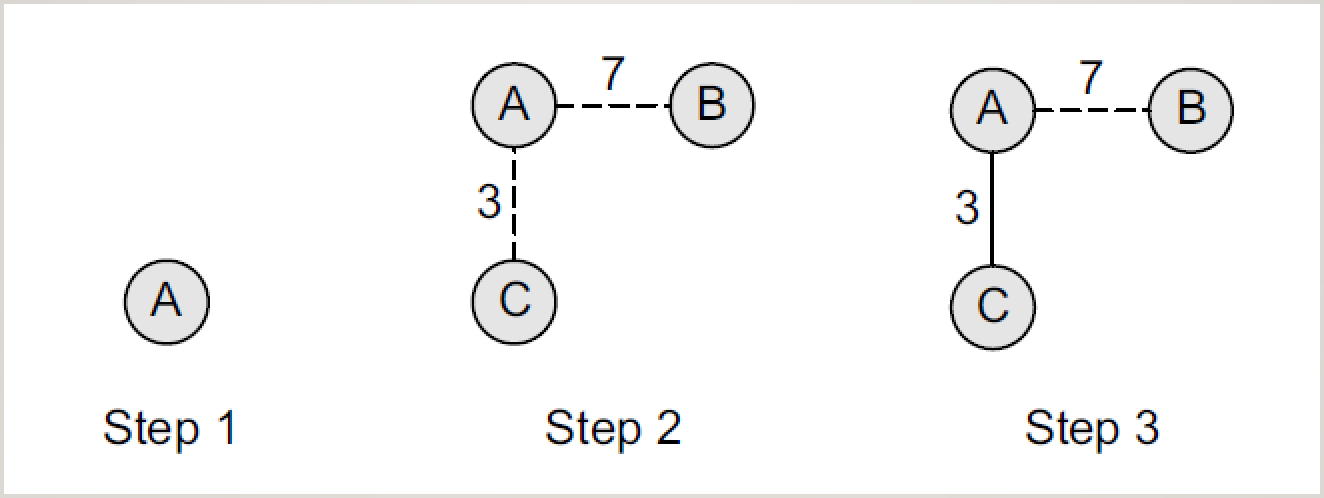
Step 3: Select an edge e connecting the tree vertex and fringe vertex that has minimum weight

Step 4: Add the selected edge and the vertex to the minimum spanning tree T

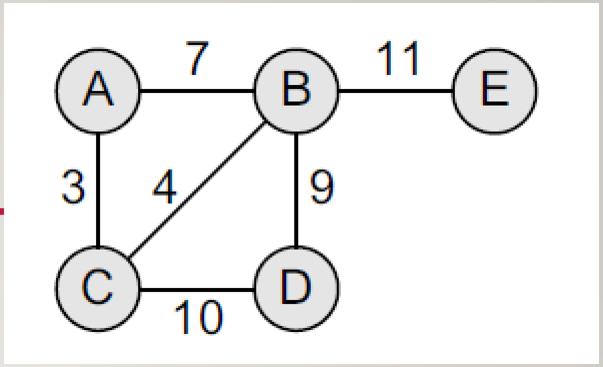
Step 5: EXIT

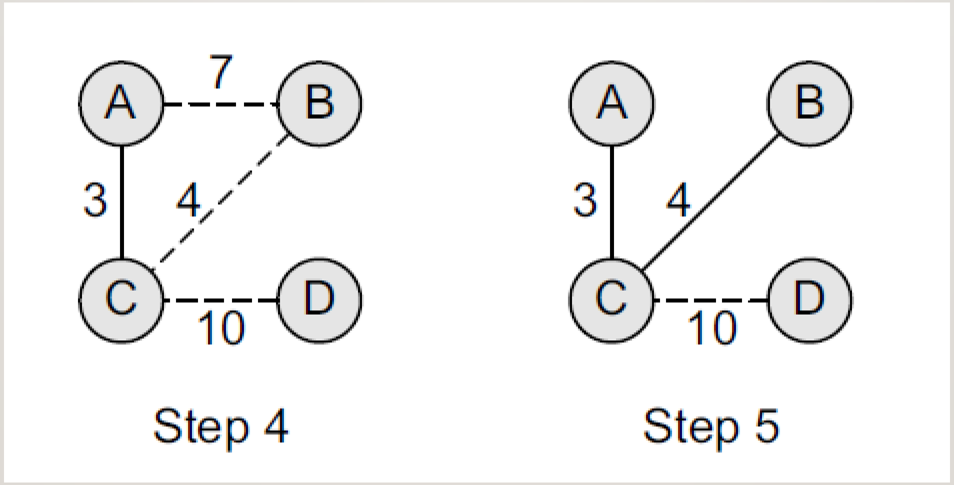
PRIM'S ALGORITHM



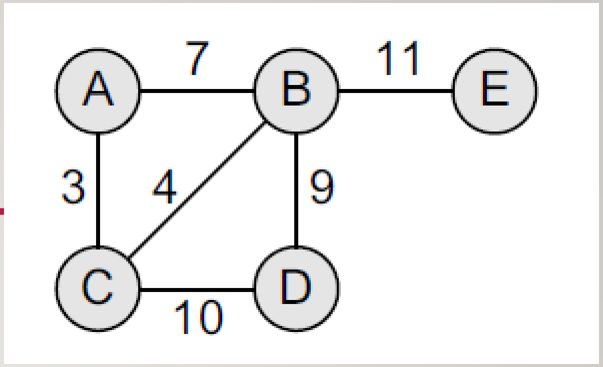


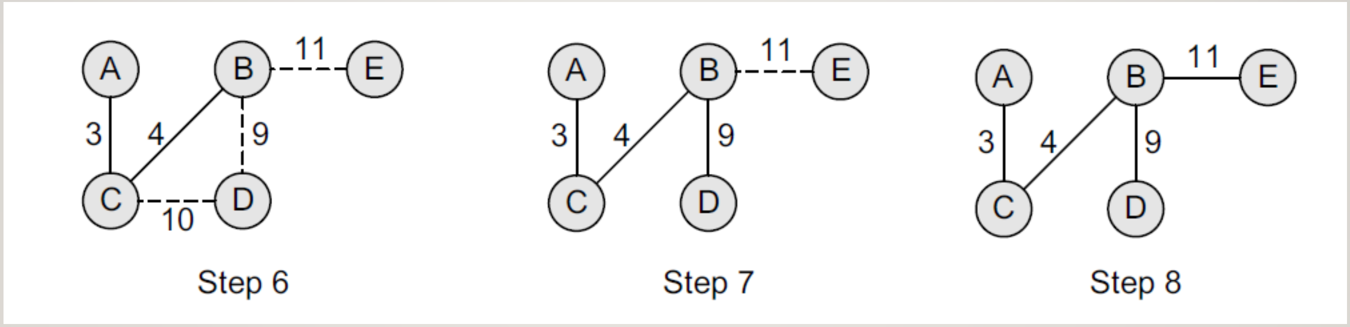
PRIM'S ALGORITHM





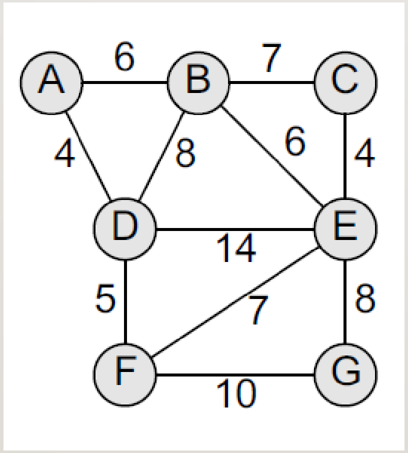
PRIM'S ALGORITHM

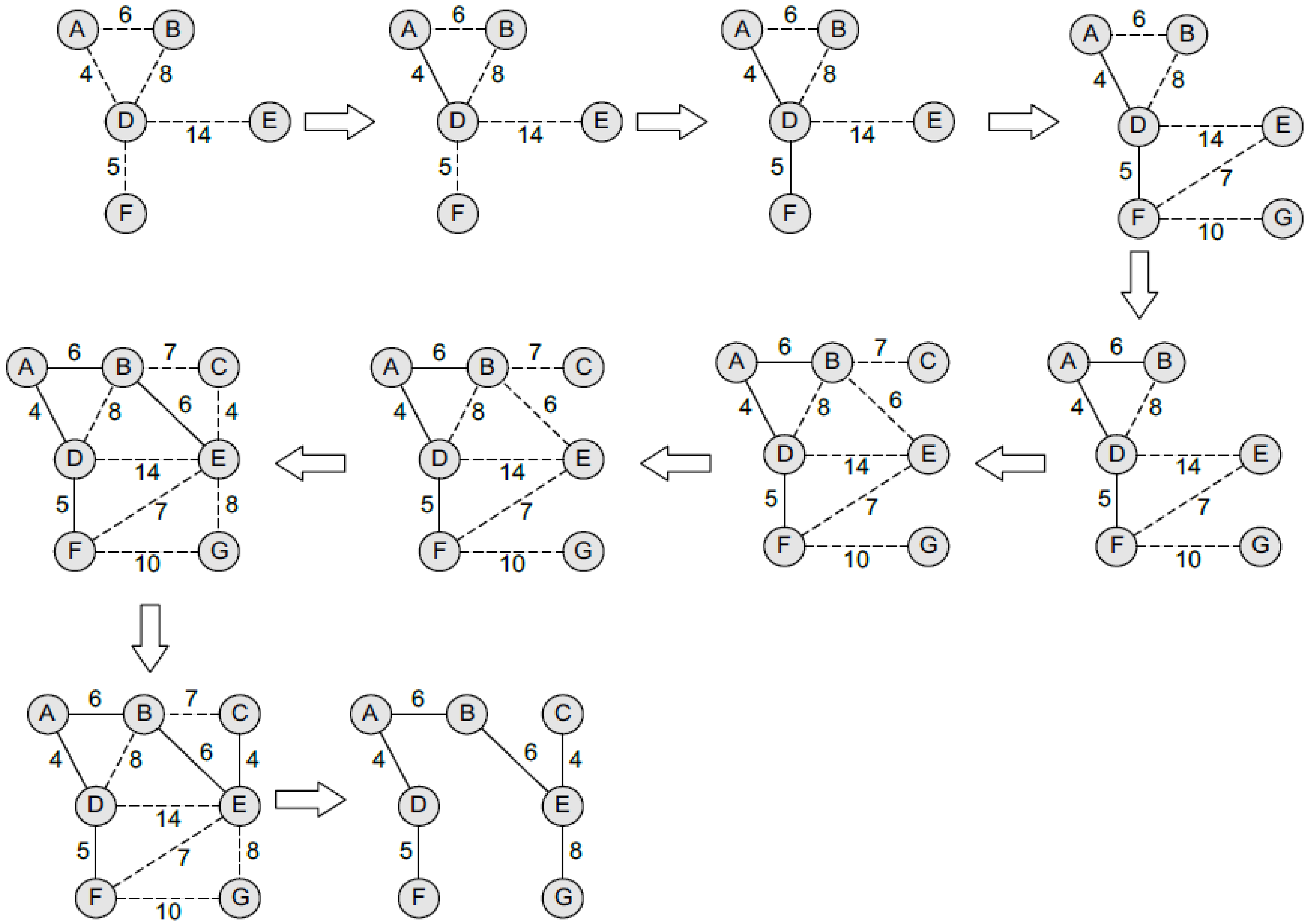




DO IT YOURSELF

- Construct a minimum spanning tree of the graph given in Figure Start the Prim's algorithm from vertex D





PRIMS PROGRAM

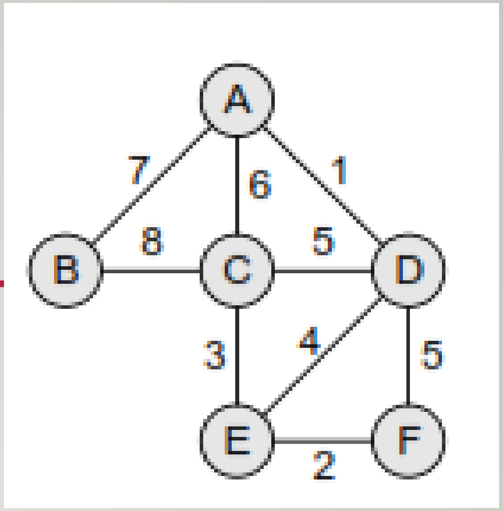
```
void main()
{
     int adjacency[5][5];
     int u,v,n=5,i,j,ne=1;
     int visited[10]={0},min=0,mincost=0;
     printf("\n Enter the number of nodes:");
     scanf("%d",&n);
     printf("\n Enter the adjacency matrix:\n");
     for(i=0;i<n;i++)
         for(j=0;j<n;j++)
         {
                 scanf("%d",& adjacency[i][j]);
                 if(adjacency[i][j]==0)
                     adjacency[i][j]=999;
         }
    visited[0]=1;//visiting the first node
while(ne<n){
     for(i=0,min=999;i<n;i++){
         for(j=0;j<n;j++){
                 if(adjacency[i][j]<min)
                     if(visited[i]!=0){
                           min=adjacency[i][j];
                           u=i;
                           v=j;}}}
       if(visited[u]==0 || visited[v]==0){
             printf("\n Edge %d:(%d %d) cost:%d",ne++,u,v,min);
             mincost+=min;
             visited[v]=1;}
              adjacency[u][v]=adjacency[v][u]=999;
           }
       printf("\n Minimum cost=%d",mincost);
                 }
```

KRUSKAL’S ALGORITHM

- Kruskal’s algorithm is used to find the minimum spanning tree for a connected weighted graph.
- The algorithm aims to find a subset of the edges that forms a tree that includes every vertex. The total weight of all the edges in the tree is minimized. However, if the graph is not connected, then it finds a *minimum spanning forest*. Note that a forest is a collection of trees. Similarly, a minimum spanning forest is a collection of minimum spanning trees.
- Kruskal’s algorithm is an example of a greedy algorithm, as it makes the locally optimal choice at each stage with the hope of finding the global optimum.

```
Step 1: Create a forest in such a way that each graph is a separate tree.

Step 2: Create a priority queue Q that contains all the edges of the graph.

Step 3: Repeat Steps 4 and 5 while Q is NOT EMPTY

Step 4: Remove an edge from Q

Step 5: IF the edge obtained in Step 4 connects two different trees, then Add it to the forest (for combining two trees into one tree).

ELSE Discard the edge

Step 6: END
```

KRUSKAL'S ALGORITHM



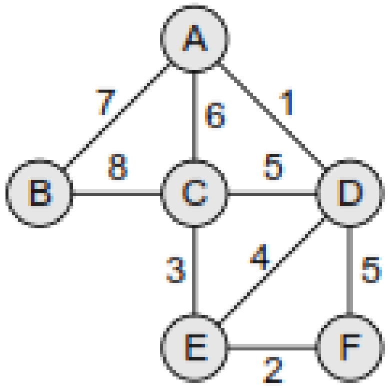
Apply Kruskal's algorithm on the graph given

Initially, we have F = {{A}, {B}, {C}, {D}, {E}, {F}}

MST = {}

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

Step 1: Remove the edge `(A, D)` from `Q` and make the following changes:

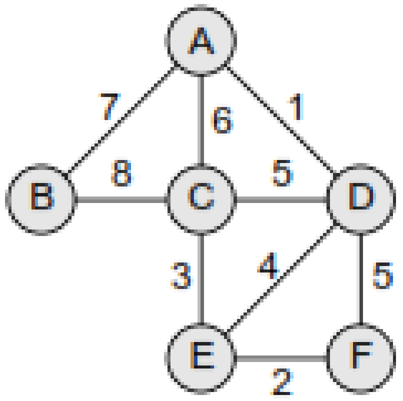


F = {{A, D}, {B}, {C}, {E}, {F}}

MST = {A, D}

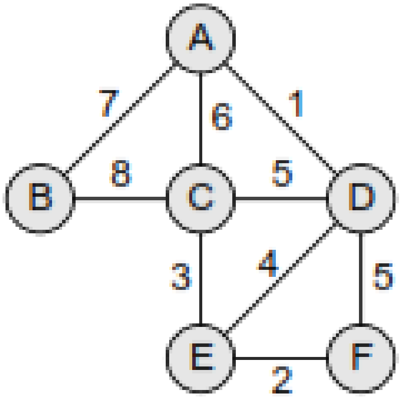
Q = {(E, F), (C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

Step 2: Remove the edge (E, F) from Q and make the following changes:



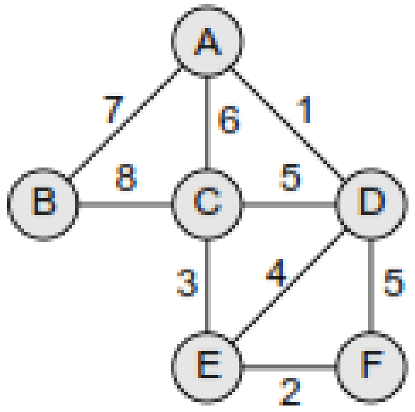
F = {{A, D}, {B}, {C}, {E, F}}
MST = {(A, D), (E, F)}
Q = {(C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

Step 3: Remove the edge `(C, E)` from `Q` and make the following changes:

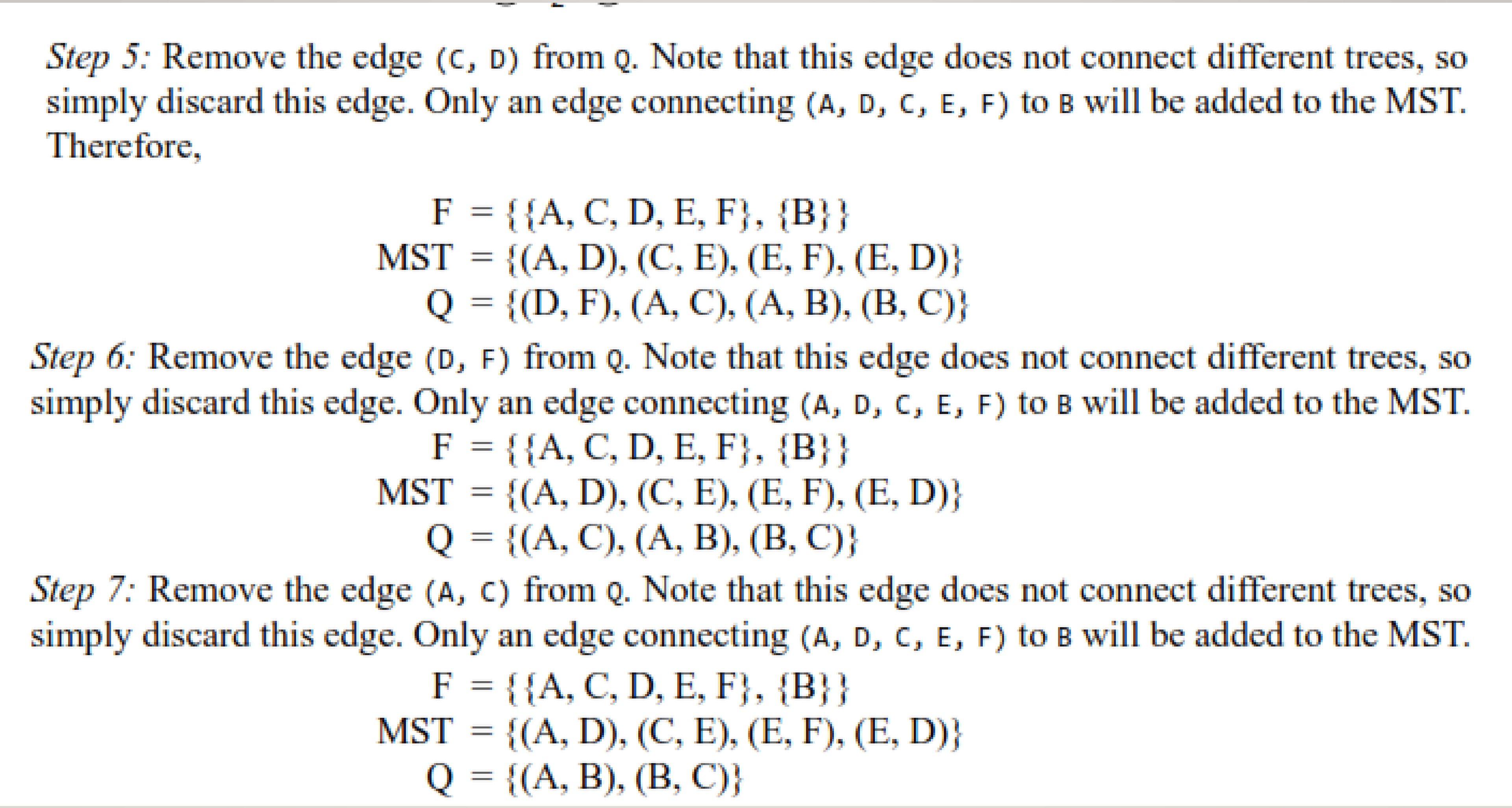


F = {{A, D}, {B}, {C, E, F}}
MST = {(A, D), (C, E), (E, F)}
Q = {(E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

Step 4: Remove the edge `(E, D)` from `Q` and make the following changes:



F = {{A, C, D, E, F}, {B}}
MST = {(A, D), (C, E), (E, F), (E, D)}
Q = {(C, D), (D, F), (A, C), (A, B), (B, C)}



Step 5: Remove the edge (C, D) from Q. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting (A, D, C, E, F) to B will be added to the MST. Therefore,

$$
\begin{aligned}
F &= \{\{A, C, D, E, F\}, \{B\}\} \\
MST &= \{(A, D), (C, E), (E, F), (E, D)\} \\
Q &= \{(D, F), (A, C), (A, B), (B, C)\}
\end{aligned}
$$

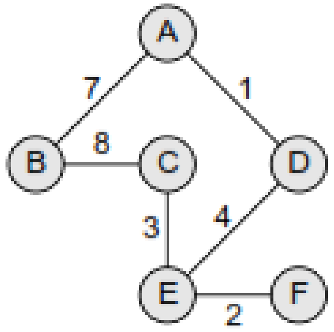
Step 6: Remove the edge (D, F) from Q. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting (A, D, C, E, F) to B will be added to the MST.

$$
\begin{aligned}
F &= \{\{A, C, D, E, F\}, \{B\}\} \\
MST &= \{(A, D), (C, E), (E, F), (E, D)\} \\
Q &= \{(A, C), (A, B), (B, C)\}
\end{aligned}
$$

Step 7: Remove the edge (A, C) from Q. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting (A, D, C, E, F) to B will be added to the MST.

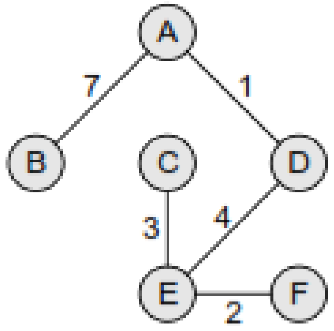
$$
\begin{aligned}
F &= \{\{A, C, D, E, F\}, \{B\}\} \\
MST &= \{(A, D), (C, E), (E, F), (E, D)\} \\
Q &= \{(A, B), (B, C)\}
\end{aligned}
$$

Step 8: Remove the edge `(A, B)` from `Q` and make the following changes:



F = {A, B, C, D, E, F}
MST = {(A, D), (C, E), (E, F), (E, D), (A, B)}
Q = {(B, C)}

Step 9: The algorithm continues until `Q` is empty. Since the entire forest has become one tree, all the remaining edges will simply be discarded. The resultant MS can be given as shown below.



F = {A, B, C, D, E, F}
MST = {(A, D), (C, E), (E, F), (E, D), (A, B)}
Q = {}

DIJKSTRA'S ALGORITHM

- Dijkstra's algorithm, given by a Dutch scientist Edsger Dijkstra in 1959, is used to find the shortest path tree.
- This algorithm is widely used in network routing protocols

DIJKSTRA'S ALGORITHM

1. Select the source node also called the initial node

2. Define an empty set N that will be used to hold nodes to which a shortest path has been found.

3. Label the initial node with 0 , and insert it into N.

4. Repeat Steps 5 to 7 until the destination node is in N or there are no more labelled nodes in N.

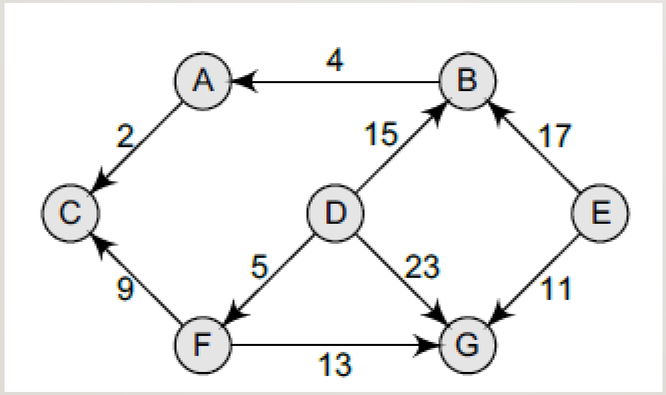
5. Consider each node that is not in N and is connected by an edge from the newly inserted node.

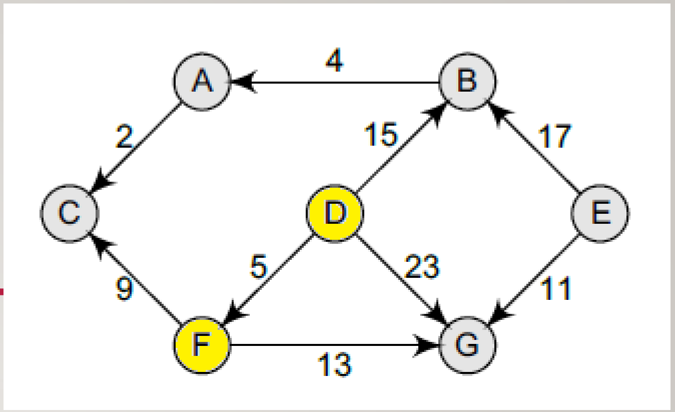
6. (a) If the node that is not in N has no label then SET the label of the node = the label of the newly inserted node + the length of the edge.

(b) Else if the node that is not in N was already labelled, then SET its new label = minimum (label of newly inserted vertex + length of edge, old label)

7. Pick a node not in N that has the smallest label assigned to it and add it to N.

- Consider the graph G given in figure below, D is the initial node, execute the Dijkstra's algorithm on it.





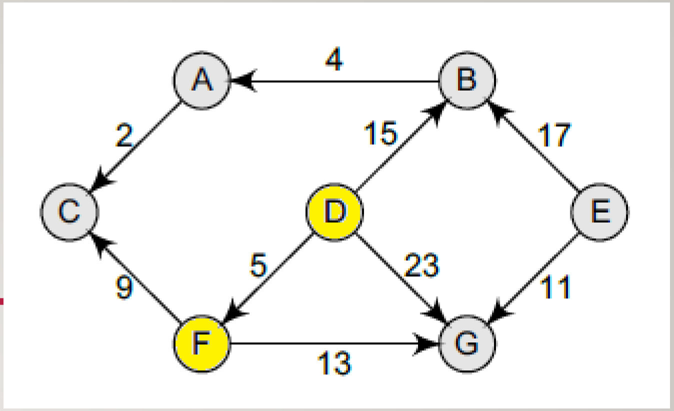
- *Step 1:* Set the label of D = 0 and N = {D}.

Vertex	A	B	C	D	E	F	G
Weight				0			
Previous Vertex				-			

- *Step 2:* Label of D = 0, B = 15, G = 23, and F = 5.

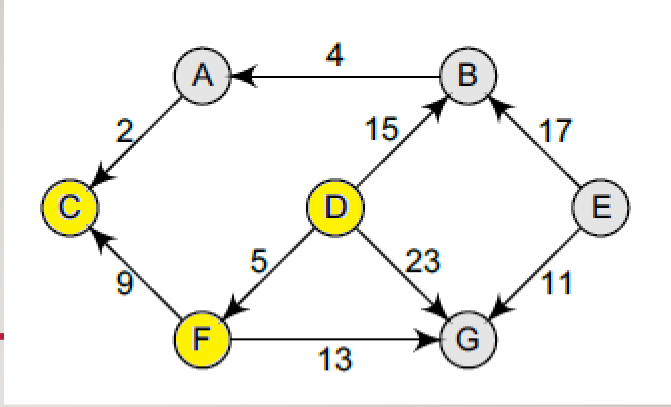
Vertex	A	B	C	D	E	F	G
Weight		15		0		5	23
Previous Vertex		**D**		**-**		**D**	**D**

- Therefore, N = {D, F}.



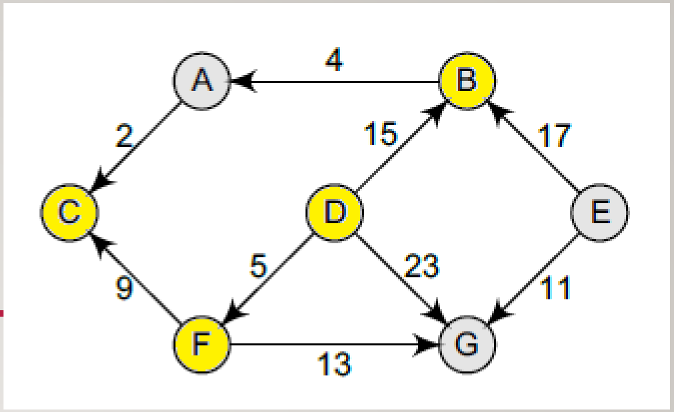
- Label of D = 0, B = 15, G has been re-labelled 18 because minimum (5 + 13, 23) = 18, C has been re-labelled 14 (5 + 9). Therefore, N = {D, F, C}.

Vertex	A	B	C	D	E	F	G
Weight		15	14	0		5	18
Previous Vertex		**D**	**F**	**-**		**D**	**F**



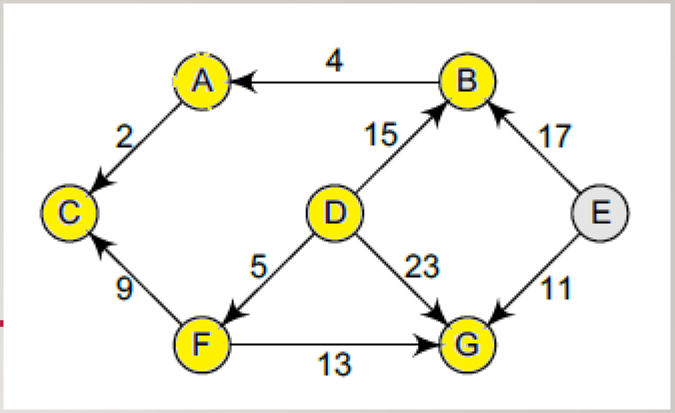
- *Step 4:* Label of D = 0, B = 15, G = 18. Therefore, N = {D, F, C, B}.

Vertex	A	B	C	D	E	F	G
Weight		15	14	0		5	18
Previous Vertex		**D**	**F**	**-**		**D**	**F**



- *Step 5:* Label of D = 0, B = 15, G = 18 and A = 19 (15 + 4). Therefore, N = {D, F, C, B, G}.

Vertex	A	B	C	D	E	F	G
Weight	19	15	14	0		5	18
Previous Vertex	**B**	**D**	**F**	**-**		**D**	**F**



- *Step 6:* Label of D = 0 and A = 19. Therefore, N = {D, F, C, B, G, A}

A	B	C	D	E	F	G
19	15	14	0		5	18
B	**D**	**F**	**-**		**D**	**F**

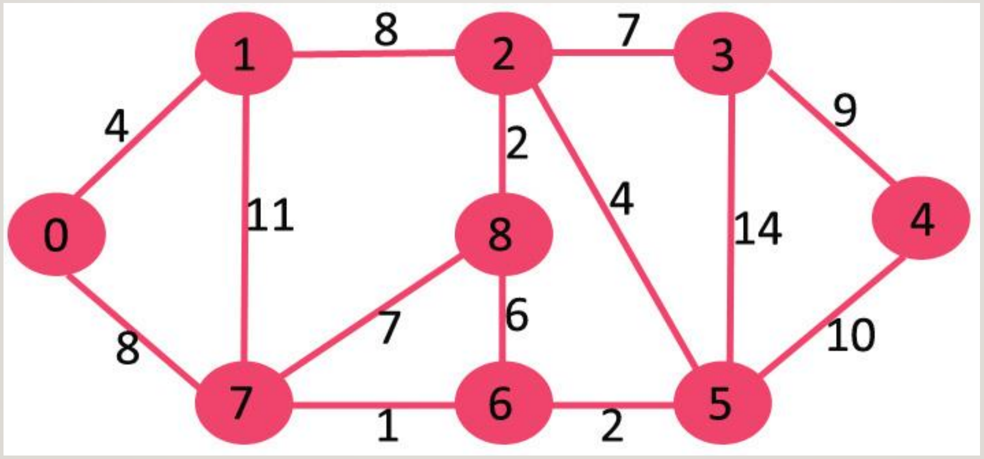
- Note that we have no labels for node E; this means that E is not reachable from D. Only the nodes that are in N are reachable from B.

OBSERVATIONS

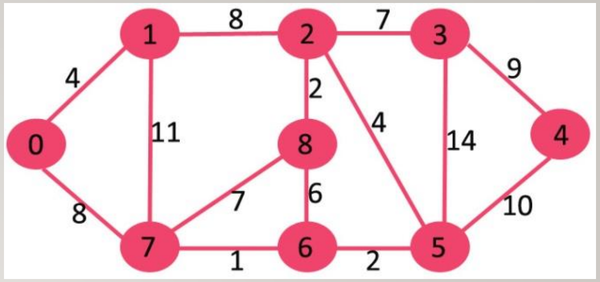
Prim's	Dijkstra's
Minimum spanning tree algorithm is used to traverse a graph in the most efficient manner	calculates the distance from a given vertex to every other vertex in the graph
Prim's algorithm stores a minimum cost edge	Dijkstra's algorithm stores the total cost from a source node to the current node
Prim's algorithm stores at most one minimum cost edge	Dijkstra's algorithm is used to store the summation of minimum cost edges
Both the algorithms begin at a specific node and extend outward within the graph, until all other nodes in the graph have been reached	

DO IT YOURSELF

- Find the Shortest path to all vertices starting with Vertex 0 using Dijkstra's Algorithm

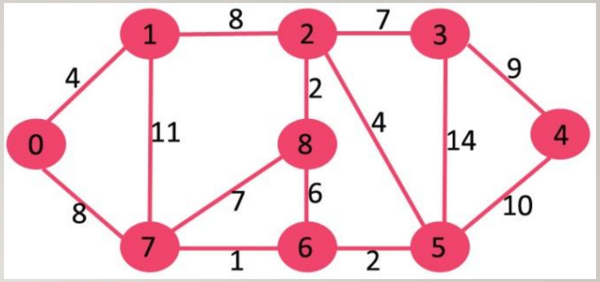


SOLUTION



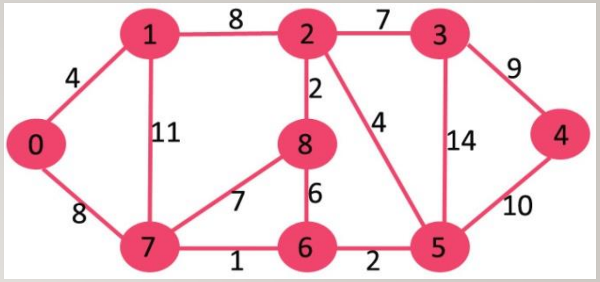
Vertex	0	1	2	3	4	5	6	7	8
Weight	0	**4**	∞	∞	∞	∞	∞	**8**	∞
Previous Vertex	-	0						0	

SOLUTION



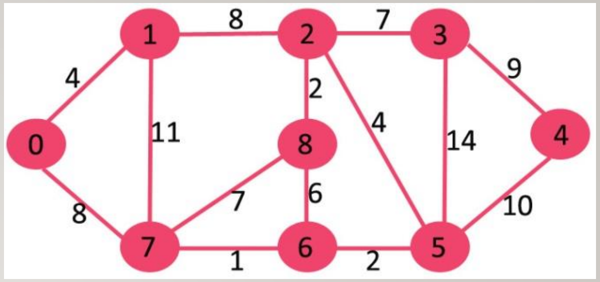
Vertex	0	1	2	3	4	5	6	7	8
Weight	0	**4**	**12**	∞	∞	∞	∞	**8**	∞
Previous Vertex	-	0	1					0	

SOLUTION



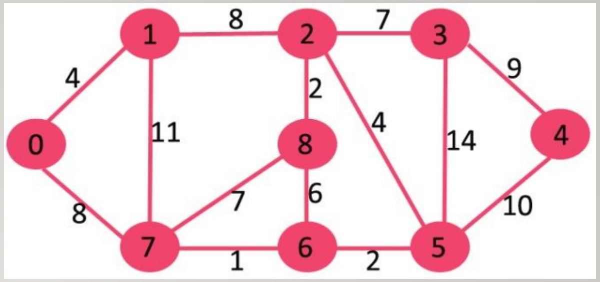
Vertex	0	1	2	3	4	5	6	7	8
Weight	0	**4**	**12**	∞	∞	∞	**9**	**8**	**15**
Previous Vertex	-	0	1				7	0	7

SOLUTION



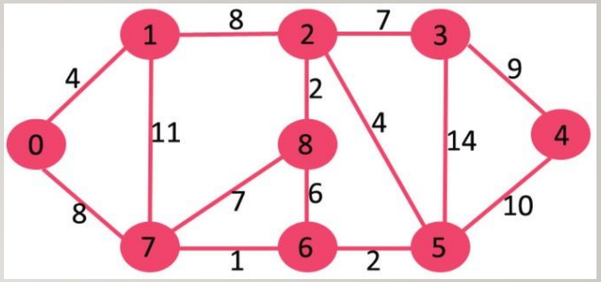
Vertex	0	1	2	3	4	5	6	7	8
Weight	0	**4**	**12**	**∞**	**∞**	**11**	**9**	**8**	**15**
Previous Vertex	-	0	1			6	7	0	7

SOLUTION



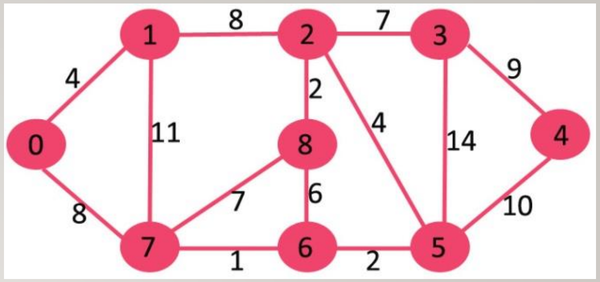
Vertex	0	1	2	3	4	5	6	7	8
Weight	0	**4**	**12**	**25**	**21**	**11**	**9**	**8**	**15**
Previous Vertex	-	0	1	5	5	6	7	0	7

SOLUTION



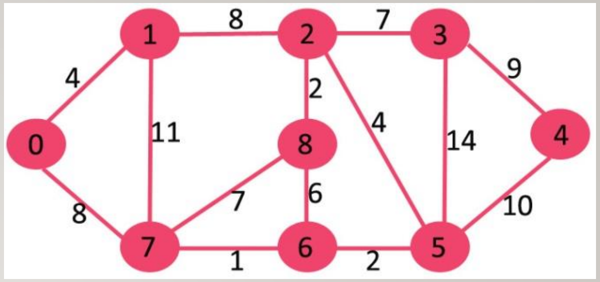
Vertex	0	1	2	3	4	5	6	7	8
Weight	0	**4**	**12**	**19**	**21**	**11**	**9**	**8**	**14**
Previous Vertex	-	0	1	2	5	6	7	0	2

SOLUTION



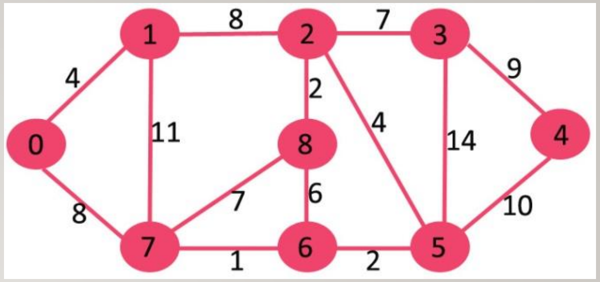
Vertex	0	1	2	3	4	5	6	7	8
Weight	0	**4**	**12**	**19**	**21**	**11**	**9**	**8**	**14**
Previous Vertex	-	0	1	2	5	6	7	0	2

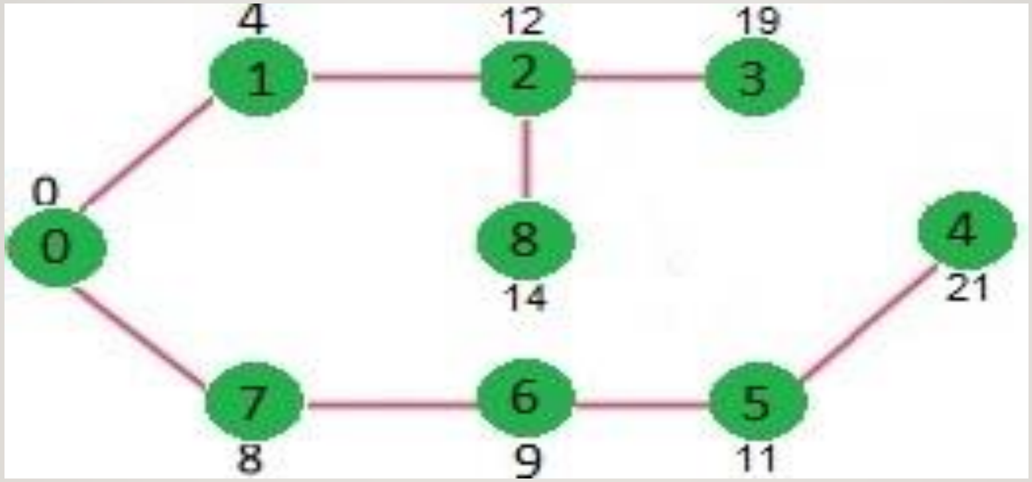
SOLUTION



Vertex	0	1	2	3	4	5	6	7	8
Weight	0	**4**	**12**	**19**	**21**	**11**	**9**	**8**	**14**
Previous Vertex	-	0	1	2	5	6	7	0	2

SOLUTION





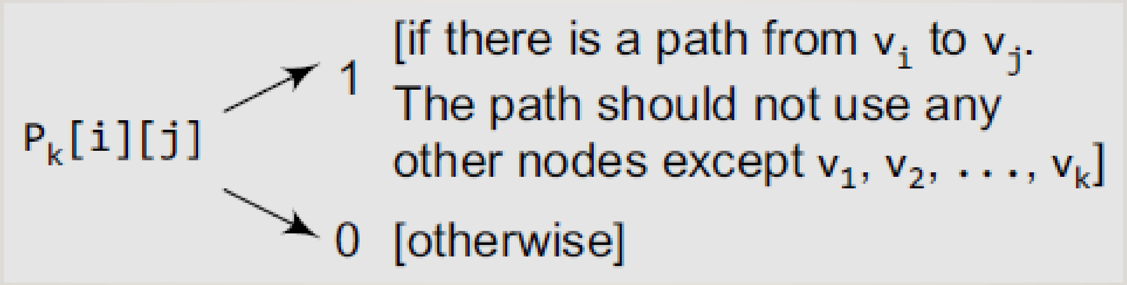
Vertex	0	1	2	3	4	5	6	7	8
Weight	0	4	12	19	21	11	9	8	14
Previous Vertex	-	0	1	2	5	6	7	0	2

FLOYD - WARSHALL'S ALGORITHM

- Used to find Transitive Closure of Matrix
- Finds the length of shortest path between all pair of vertices

FLOYD - WARSHALL’S ALGORITHM

- Warshall’s algorithm defines matrices P0, P1, P2, ... , Pn

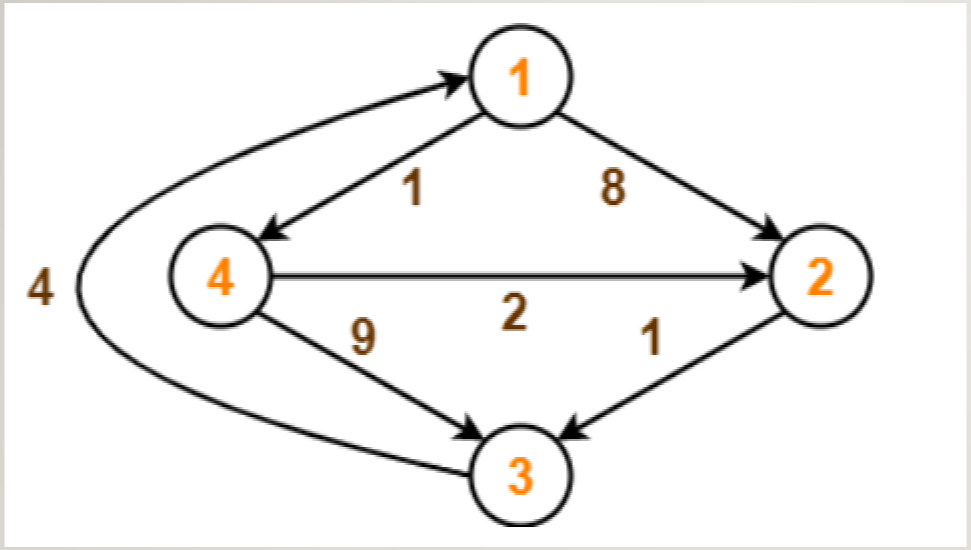


$P_k[i][j]$ → 1 [if there is a path from v_i to v_j. The path should not use any other nodes except $v_1, v_2, \ldots, v_k$]

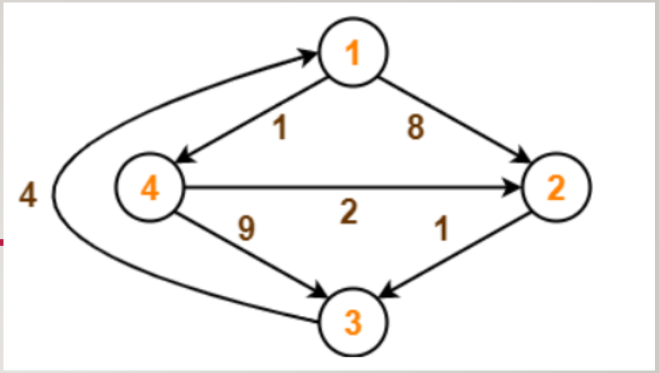
$P_k[i][j]$ → 0 [otherwise]

- if P0[i][j] = 1, then there exists an edge from node vi to vj.
- If P1[i][j] = 1, then there exists an edge from vi to vj that does not use any other vertex except v1.
- If P2[i][j] = 1, then there exists an edge from vi to vj that does not use any other vertex except v1 and v2.

FLOYD - WARSHALL'S ALGORITHM

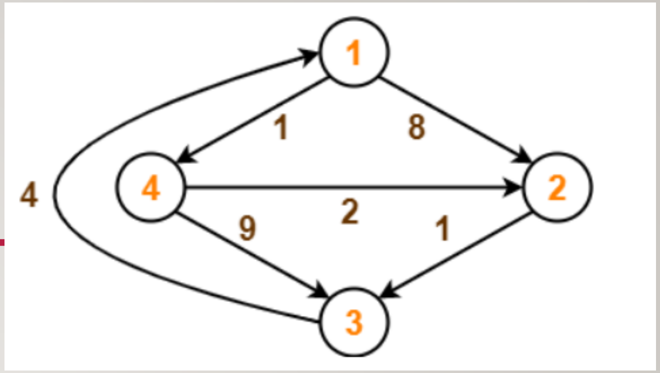


FLOYD - WARSHALL'S ALGORITHM

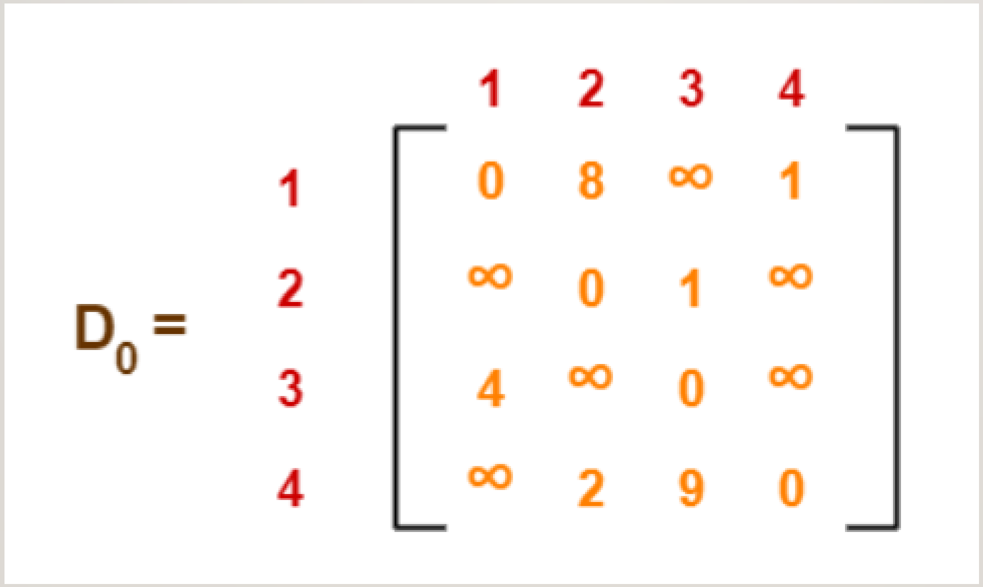


- Write the initial distance matrix.
- It represents the distance between every pair of vertices in the form of given weights.
- For diagonal elements (representing self-loops), distance value = 0.
- For vertices having a direct edge between them, distance value = weight of that edge.
- For vertices having no direct edge between them, distance value = ∞.

FLOYD - WARSHALL'S ALGORITHM

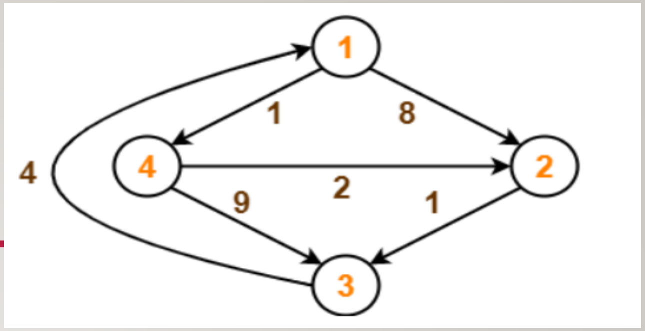


- Initial distance matrix for the given graph is-

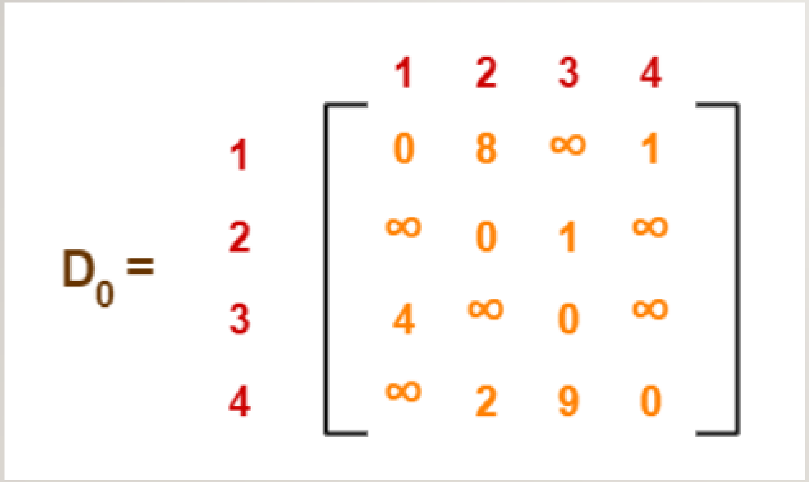


$$D_0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & \infty & 0 & \infty \\ 4 & \infty & 2 & 9 & 0 \end{array}$$

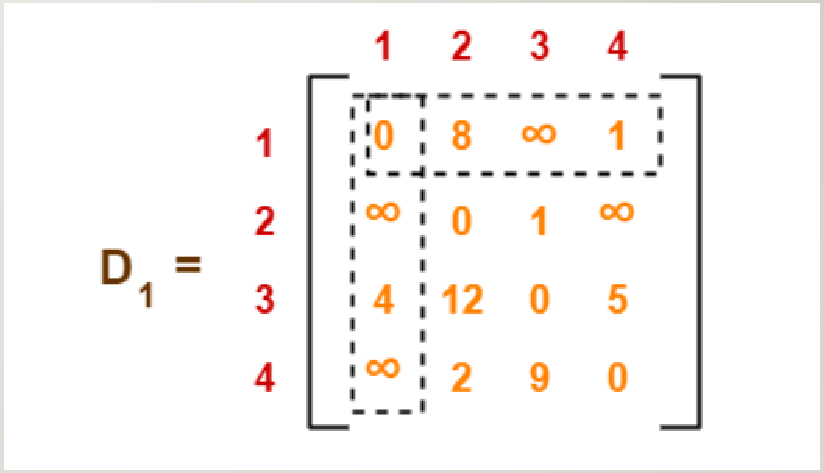
FLOYD - WARSHALL'S ALGORITHM



```
D[i,j] = min(D[i,j],D[i,k]+D[k,j])
```

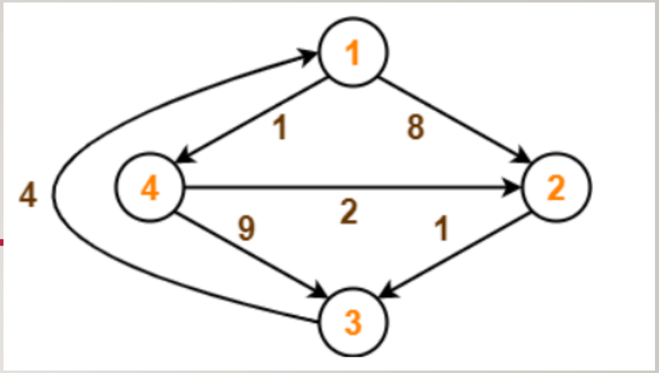


$$D_0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & \infty & 0 & \infty \\ 4 & \infty & 2 & 9 & 0 \end{array}$$

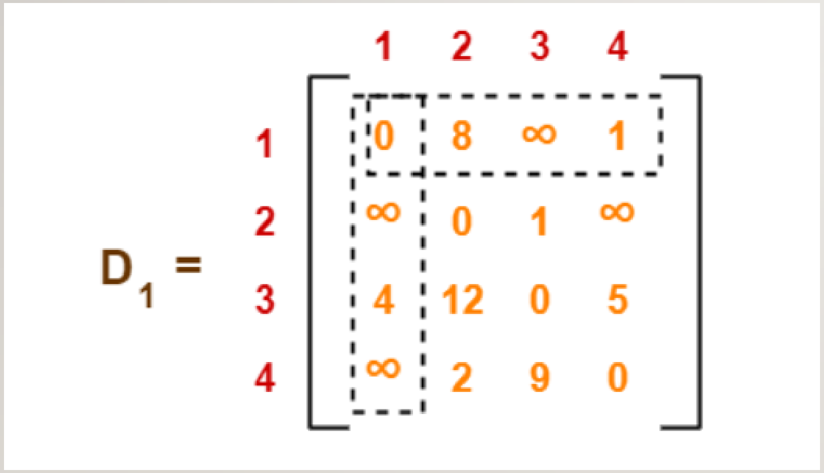


$$D_1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 9 & 0 \end{array}$$

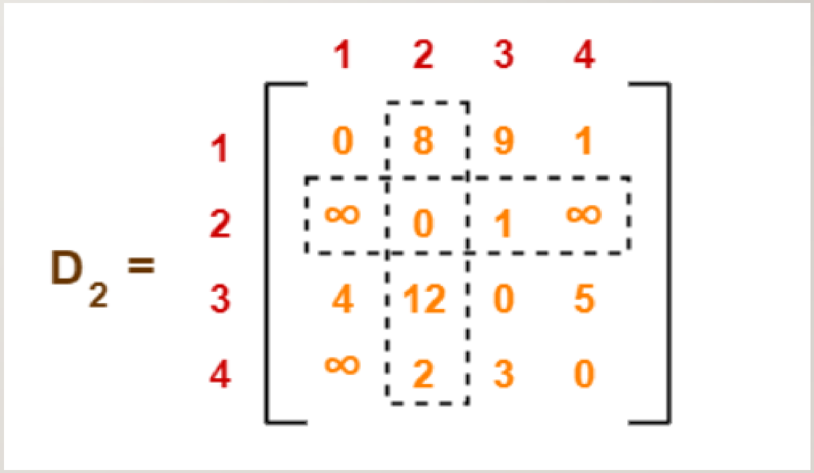
FLOYD - WARSHALL'S ALGORITHM



```
D[i,j] = min(D[i,j],D[i,k]+D[k,j])
```

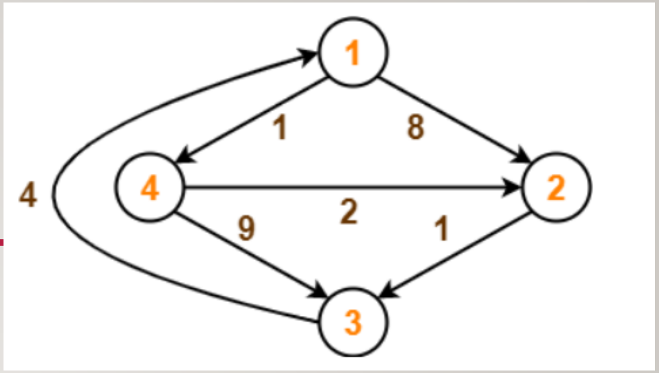


$$D_1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 9 & 0 \end{array}$$

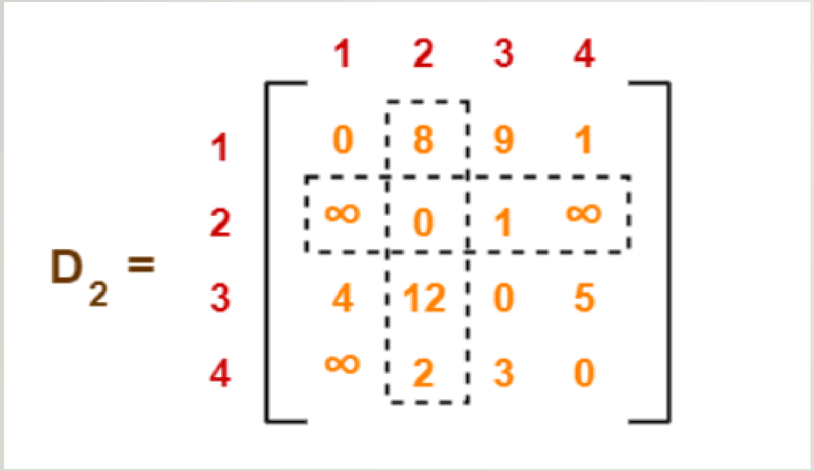


$$D_2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & 9 & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 3 & 0 \end{array}$$

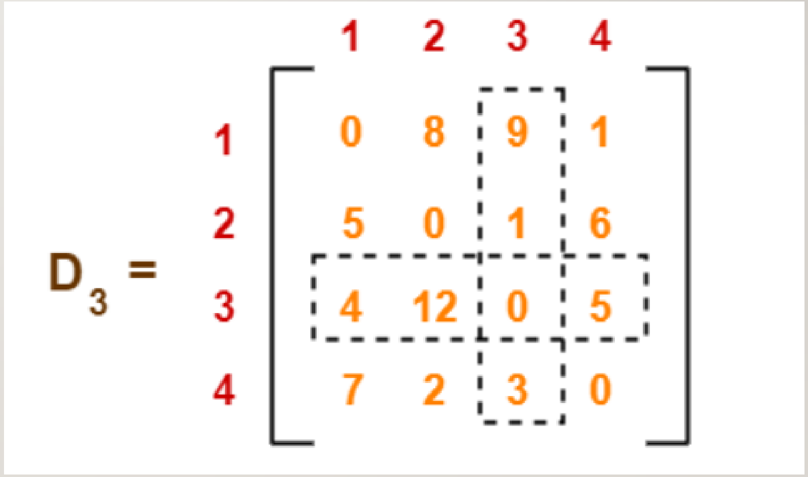
FLOYD - WARSHALL'S ALGORITHM



```
D[i,j] = min(D[i,j],D[i,k]+D[k,j])
```

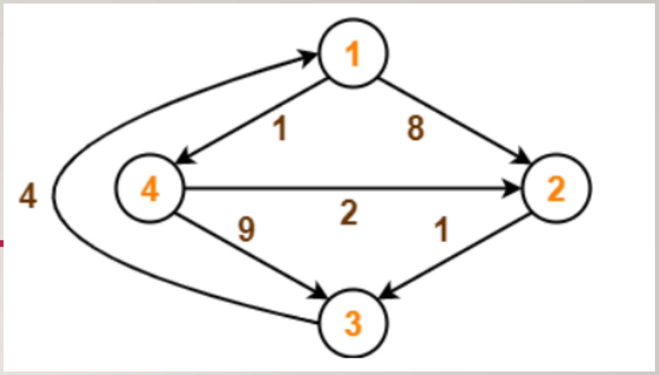


$$D_2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & 9 & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 3 & 0 \end{array}$$



$$D_3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & 9 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 12 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{array}$$

FLOYD - WARSHALL'S ALGORITHM



```
D[i,j] = min(D[i,j],D[i,k]+D[k,j])
```

$$D_3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & 9 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 12 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{array}$$

$$D_4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 4 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 7 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{array}$$

ALGORITHM

```
Create a |V| x |V| matrix // It represents the distance between every pair of vertices as given
For each cell (i,j) in M do-
    if i = = j
        M[ i ][ j ] = 0 // For all diagonal elements, value = 0
    if (i , j) is an edge in E
        M[ i ][ j ] = weight(i,j) // If there exists a direct edge between the vertices, value = weight of edge
    else
        M[ i ][ j ] = infinity // If there is no direct edge between the vertices, value = ∞
for k from 1 to |V|
    for i from 1 to |V|
        for j from 1 to |V|
            if M[ i ][ j ] > M[ i ][ k ] + M[ k ][ j ]
                M[ i ][ j ] = M[ i ][ k ] + M[ k ][ j ]
```

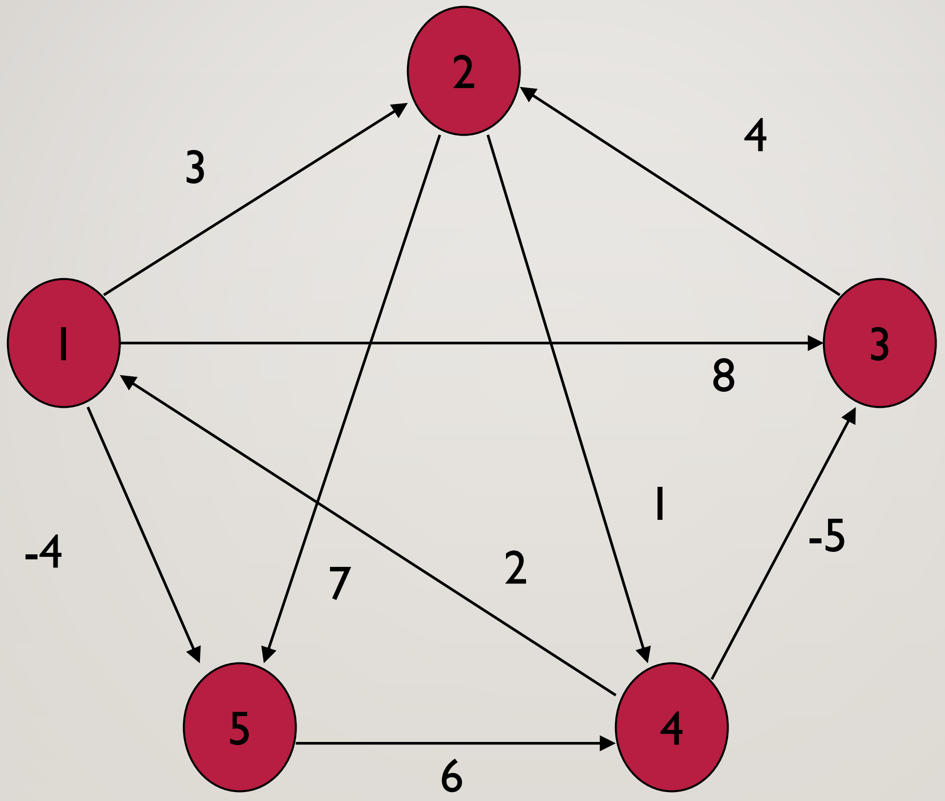
FLOYD - WARSHALL'S ALGORITHM

- **Time Complexity-**

Floyd Warshall Algorithm consists of three loops over all the nodes.The inner most loop consists of only constant complexity operations.Hence, the asymptotic complexity of Floyd Warshall algorithm is $O(n^3)$. Here, n is the number of nodes in the given graph.

- Floyd Warshall Algorithm is best suited for dense graphs.This is because its complexity depends only on the number of vertices in the given graph.

EXAMPLE:



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

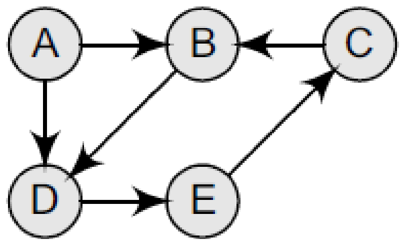
$$D(1)= \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

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

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

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

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



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 1 & 0 \\ B & 0 & 0 & 0 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 0 \\ D & 0 & 0 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 0 & 0 \end{array}$$

(a) Directed graph