Introduction to Graphs

Graphs

Introduction to Graphs

- We begin a major new topic: Graphs.

- Graphs are important discrete structures because they are a flexible mathematical model for many application problems.

Introduction to Graphs

Any time there is

- a set of objects and

- there is some sort of "connection" or "relationship" or "interaction" between pairs of objects,

a graph is a good way to model this.

Introduction to Graphs

Examples:
- computer and communication networks
- transportation networks, e.g., roads
- VLSI, logic circuits
- surface meshes for shape description in computer-aided design and GIS

Introduction to Graphs

- transportation networks, e.g., roads
- VLSI, logic circuits
- surface meshes for shape description in computer-aided desigr and GIS
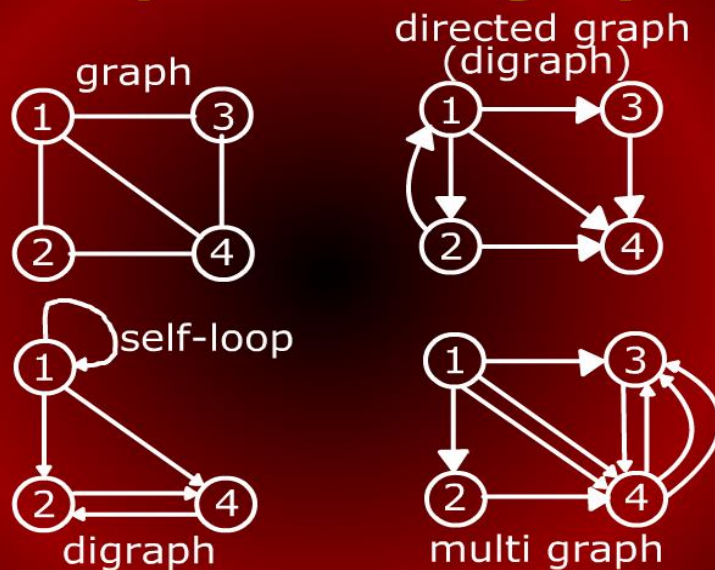- precedence constraints in scheduling systems.

Graphs and Digraphs

A graph G = (V, E) consists of

- a finite set of vertices V (or nodes) and
- E, a binary relation on V called edges.

Graphs and Digraphs

- E is a set of pairs from V.

- If a pair is *ordered*, we have a *directed* graph.

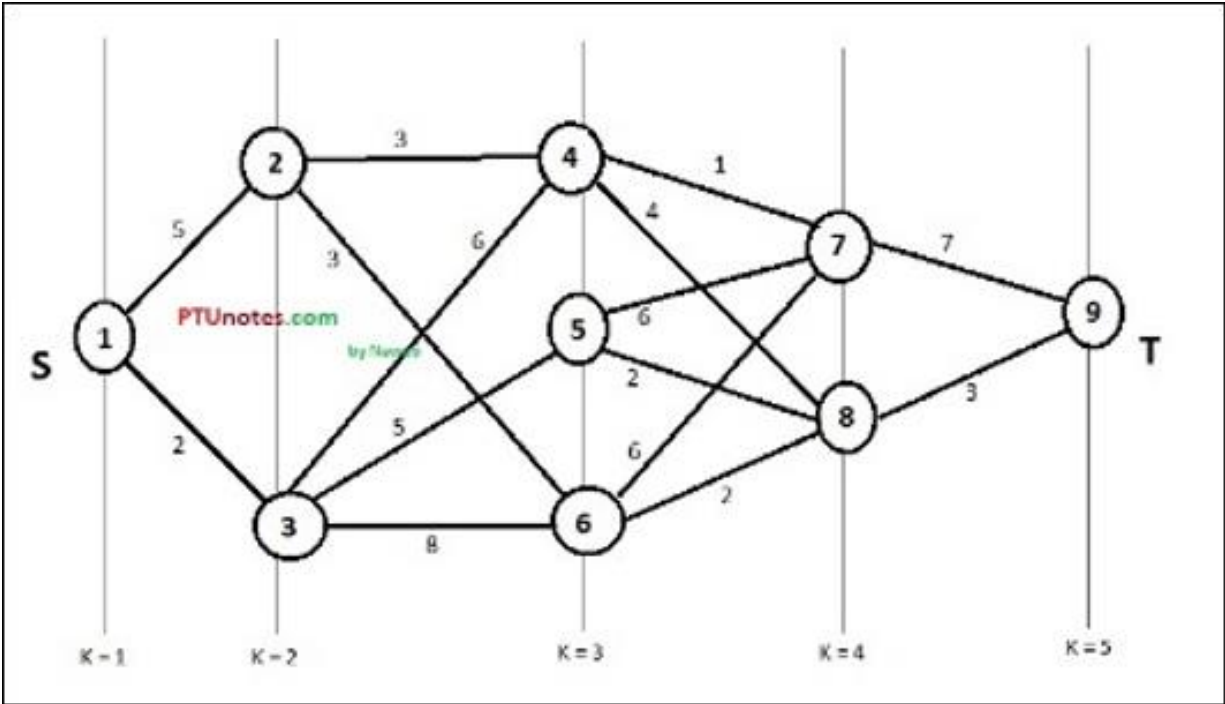- For *unordered* pair, we have an *undirected* graph.

Graphs and Digraphs

Multistage Graph (Shortest Path) $O(|V| + |E|)$

The multistage graph problem is finding the path with minimum cost from source S to sink T.

We have the formula:

$$cost(Stage,Vertex)=\min\{c(Vertex,l)+cost(Stage+1,l)\}$$

According to the formula, we have to calculate the cost (stage, vertex) using the following steps, the small c denote the cost of the edge, and l denote the destination stage of the current vertex.

1. Create a table

Nodes									
Cost									
Destination									

In Figure K = Stage, can be denoted by any alphabet; here I am using K

Where S is the starting point T is the ending point and nodes are Vertex

The multistage graph problem can be solved in two ways using dynamic programming

- Forward approach
- Backward approach

We use the Backward Approach and fill the above table

As we move from backward we start from K=5

- **Stage-5**

$Cost(5,9) = 0$

The cost of vertex 9 is 0 as it is terminal and if you start from one node and end on the same node then its cost would be zero and the destination would be the same as started.

Nodes									9
Cost									0
Destination									9

- **Stage = 4**

In stage 4 we have two nodes so we calculate their costs individually one for vertex 7 and one for vertex 8

$Cost(4,7) = 7$

$Cost(4,8) = 3$

If you move from node 7 to node 9, the cost is written on the transition arrow between two nodes)

Nodes							7	8	9
Cost							7	3	0
Destination							9	9	9

- **Stage = 3**

If you may observe that on stage 3 there are three nodes and each node have more than one transition connecting with node 7 and 8, now remember that we only calculate the cost between stage-3 nodes and stage-4 nodes, the rest of the nodes (7, 8, and 9) we already calculated and filled the above table accordingly, so we don't need to re-calculate them.

Secondly, if a node has more than one outgoing transition then we calculate their cost individually and pick the minimum cost among them to fill the above table, respectively.

Now apply the formula:

$Cost(3,4) =$	min	$c(4,7) + cost(4,7)$ $1 + 7 = 8$	
		$c(4,8) + cost(4,8)$ $4 + 3 = 7$	✓

$Cost(3,5) =$	min	$c(5,7) + cost(4,7)$ $6 + 7 = 13$	
		$c(5,8) + cost(4,8)$ $2 + 3 = 5$	✓

$Cost(3,6) =$	min	$c(6,7) + cost(4,7)$ $6 + 7 = 13$	
		$c(6,8) + cost(4,8)$ $2 + 3 = 5$	✓

Nodes				4	5	6	7	8	9
Cost				7	5	5	7	3	0
Destination				8	8	8	9	9	9

- **Stage = 2**

$Cost(2,2) =$	min	$c(2,4) + cost(3,4)$ $3 + 7 = 10$	
		$c(2,6) + cost(3,6)$ $3 + 5 = 8$	✓

$Cost(2,3) =$	min	$c(3,4) + cost(3,4)$ $6 + 7 = 13$	

		$c(3,5) + cost(3,5)$ $5 + 5 = 10$	✓
		$c(3,6) + cost(3,6)$ $8 + 5 = 13$	

Nodes		2	3	4	5	6	7	8	9
Cost		8	10	7	5	5	7	3	0
Destination		6	5	8	8	8	9	9	9

- **Stage = 1**

			$c(1,2) + cost(2,2)$ $5 + 8 = 13$	
$Cost(1,1) =$	min		$c(1,3) + cost(2,3)$ $2 + 10 = 12$	✓

Nodes	1	2	3	4	5	6	7	8	9
Cost	12	8	10	7	5	5	7	3	0
Destination	3	6	5	8	8	8	9	9	9

Shortest Path:
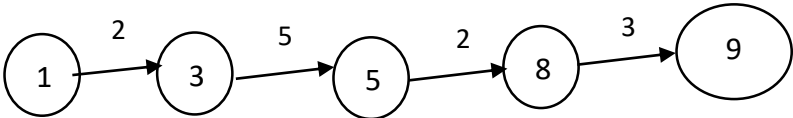
// Graph stored in the form of an adjacency Matrix

e.g.

$nodes \rightarrow$ ↓	1	2	3	4	5	6	7	8	9
1	INF	5	2	INF	INF	INF	INF	INF	INF
2	INF	INF	INF	3	INF	3	INF	INF	INF
3	INF	INF	INF	6	5	8	INF	INF	INF
4	INF	INF	INF	INF	INF	INF	1	4	INF
5	INF	INF	INF	INF	INF	INF	6	2	INF
6	INF	INF	INF	INF	INF	INF	6	2	INF
7	INF	INF	INF	INF	INF	INF	INF	INF	7
8	INF	INF	INF	INF	INF	INF	INF	INF	3
9	INF	INF	INF	INF	INF	INF	INF	INF	INF

```
int[][] graph = new int[][]{
        {INF,  5,   2,   INF, INF, INF, INF, INF, INF},
        {INF,  INF, INF, 3,   INF, 3,   INF, INF, INF},
        {INF,  INF, INF, 6,   5,   8,   INF, INF, INF},
        {INF,  INF, INF, INF, INF, INF, 1,   4,   INF},
        {INF,  INF, INF, INF, INF, INF, 6,   2,   INF},
```

```
{INF,   INF,  INF,  INF,  INF,  INF, 6,    2,    INF},
{INF,   INF,  INF,  INF,  INF,  INF, INF,  INF, 7},
{INF,   INF,  INF,  INF,  INF,  INF, INF,  INF, 3}};
```

1. $N \leftarrow 9$ // number of nodes
2. $INF \leftarrow \infty$

$Shortest\_Distance\ (int\ [\ ][\ ]\ graph)$

1. $int[\ ]\ dist \leftarrow N$ //is going to shortest distance from node 1 to node N-1
2. $dist[N-1] \leftarrow 0;$
3. $for\ i \leftarrow N-2\ downto\ 1$
4. $\quad dist[i] \leftarrow INF;$
5. $\quad\quad for\ j \leftarrow i\ to\ N-1$
6. $\quad\quad\quad if\ graph[i][j] = INF\ then$
7. $\quad\quad\quad\quad continue;$ //immediate jump for next iteration
8. $\quad\quad\quad end\ if$
9. $\quad\quad\quad dist[i] = Math.\min(dist[i], graph[i][j] + dist[j])$ //we apply recursive equation to distance to target through j and compare with minimum distance so far
10. $\quad\quad end\ for$
11. $end\ for$
12. $return\ dist[0];$

Spanning Tree (Greedy Approach)

A connected sub-graph 'S' of Graph G(V,E) is said to be spanning if and if only (iff):

1. 'S' should contain all vertices of 'G'

2. 'S' should contain $(|V| - 1)$ edge's

3. As it is tree so there is no cycle between vertices

For Example: The below Graph 'G'
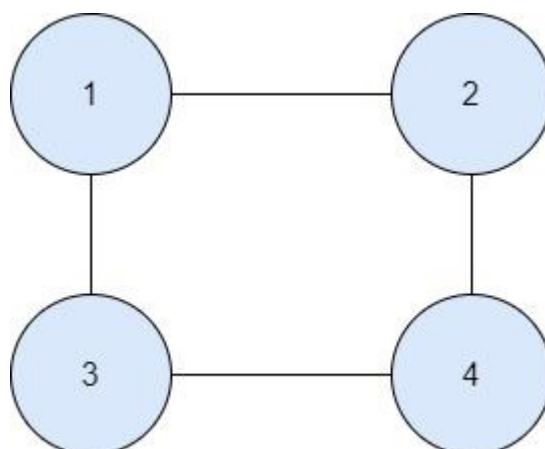
Now create the spanning tree
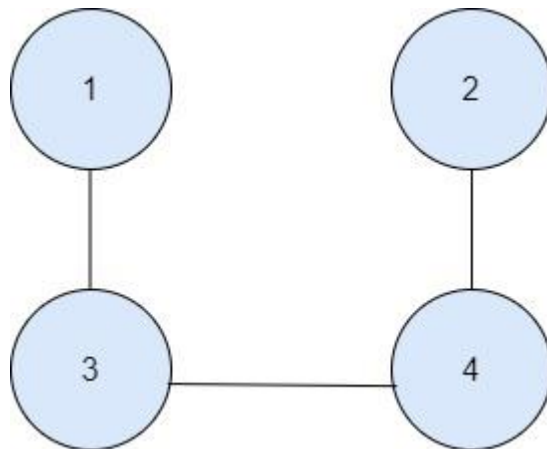
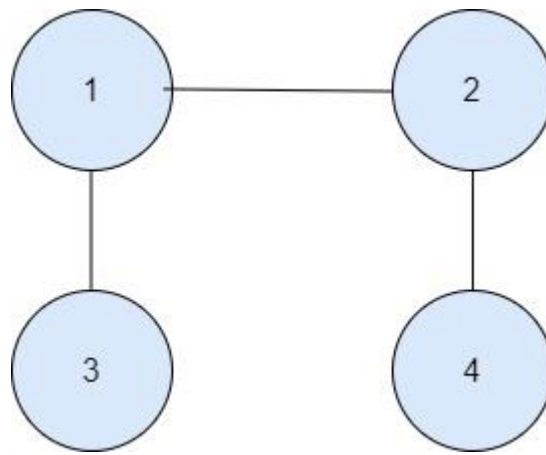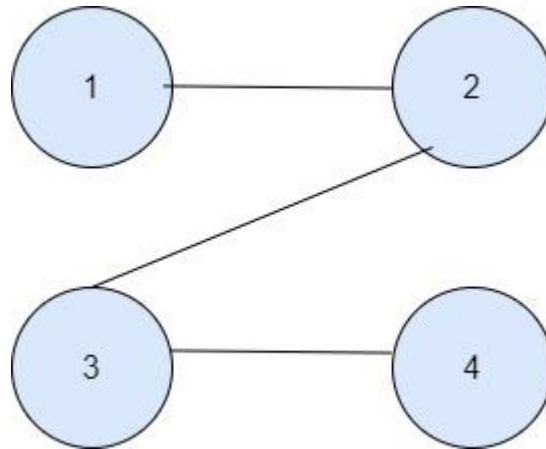1. Should contain all vertex
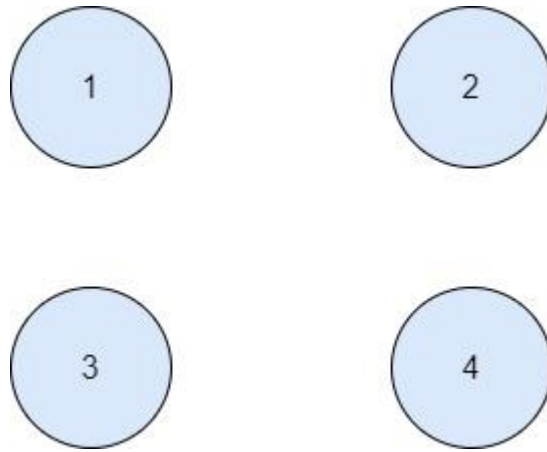
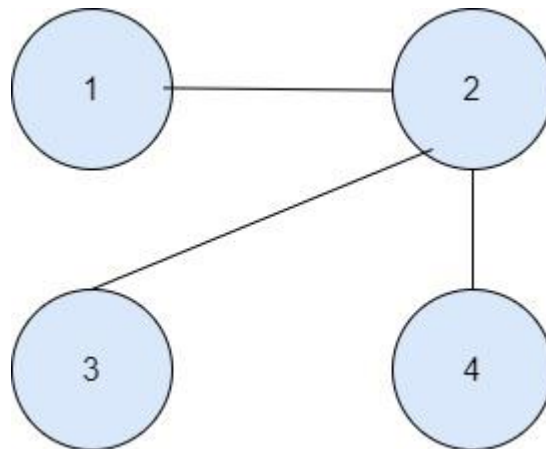2. Should contain edges ($|V| - 1$)
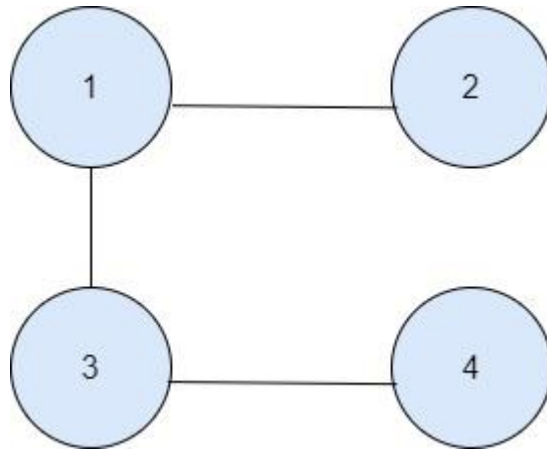
Or
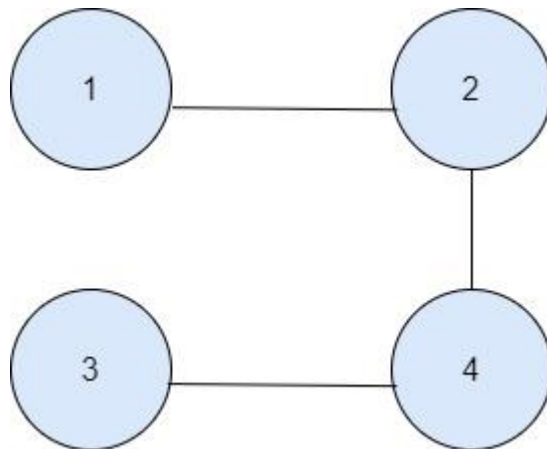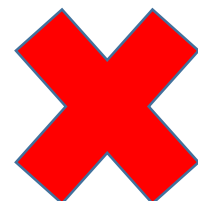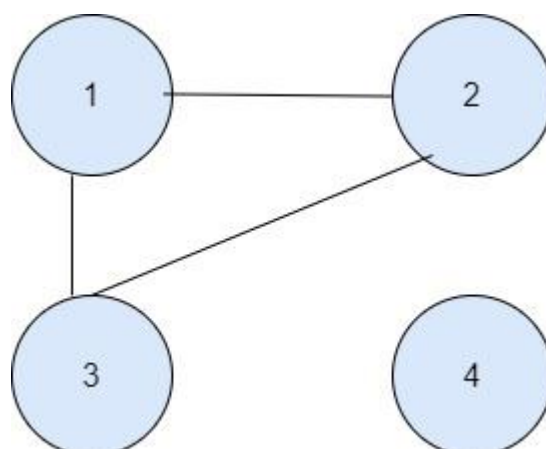

OR

OR

OR

OR

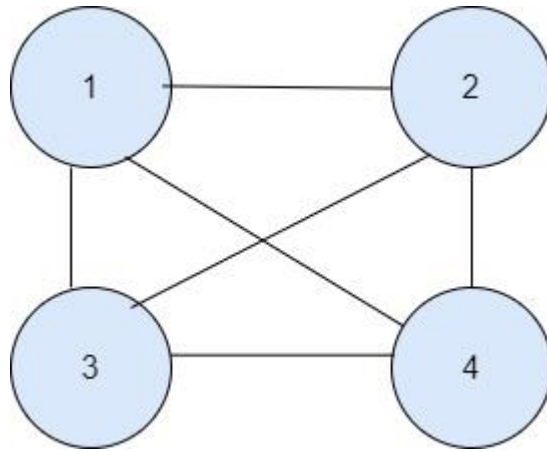Wrong Spanning Tree would be:

1. Containing isolated vertices

2. Containing cycle between vertices

e.g

Possible questions may occur:

1. Find out the number of spanning tree can be possible of the given graph below:

You may apply formula to calculate the possible spanning tree if and only if the given graph is complete graph as given above.

$$n^{n-2}$$

Where n is the number of vertices in the graph.

If the given Graph is not a complete graph then we use Kirchhoff's theorem.

2. Find the minimum cost spanning tree

This is to find the shortest path in the spanning tree using Krushal Algorithm and Prim's Algorithm.

Kruskal's Algorithm

- Kruskal's algorithm works by adding edges in increasing order of weight (lightest edge first).
- If the next edge does not induce a cycle among the current set of edges, then it is added to A.
- If it does, we skip it and consider the next in order.

Kruskal's Algorithm

- As the algorithm runs, the edges in A induce a *forest* on the vertices.
- The trees of this forest are eventually merged until a single tree forms containing all vertices.

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

1. Sort all the edges in increasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. The Greedy Choice is to pick the smallest weight edge that does not

cause a cycle in the MST constructed so far. Let us understand it with an example:
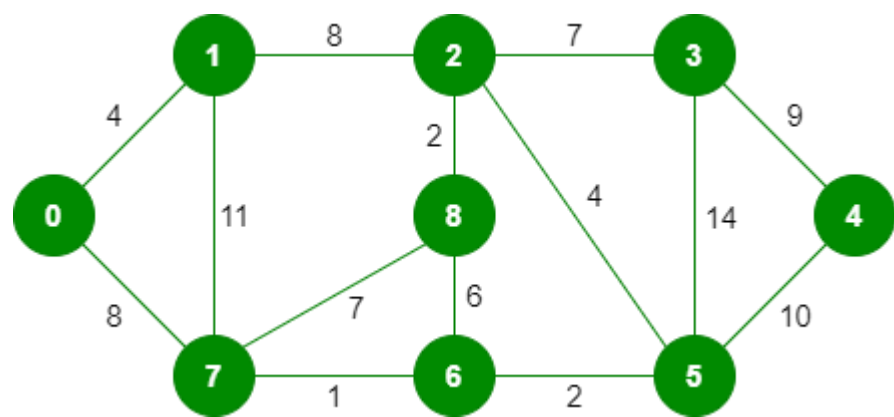
The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

Step-1: sort in increasing order according to weightage/cost:

Weight	Source	Destination	Selected
1	7	6	
2	8	2	
2	6	5	
4	0	1	
4	2	5	
6	8	6	
7	2	3	
7	7	8	
8	0	7	
8	1	2	
9	3	4	
10	5	4	
11	1	7	
14	3	5	

Step-2: Now pick all edges one by one from the sorted list of edges:
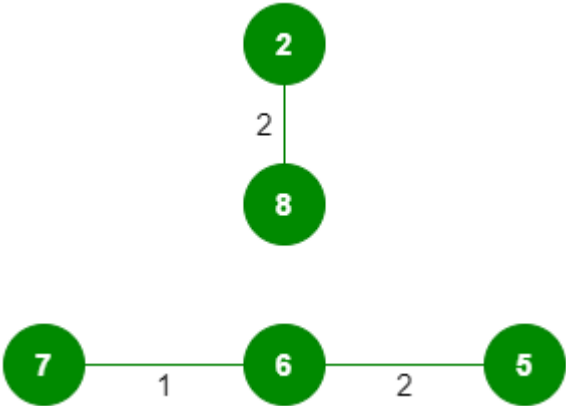
Skip those edges which makes cycle

Weight	Source	Destination	Selected
1	7	6	✓
2	8	2	
2	6	5	
4	0	1	
4	2	5	
6	8	6	
7	2	3	
7	7	8	
8	0	7	
8	1	2	
9	3	4	
10	5	4	
11	1	7	
14	3	5	

Weight	Source	Destination	Selected
1	7	6	✓
2	8	2	✓
2	6	5	
4	0	1	
4	2	5	
6	8	6	
7	2	3	
7	7	8	
8	0	7	
8	1	2	
9	3	4	
10	5	4	
11	1	7	
14	3	5	

Weight	Source	Destination	Selected
1	7	6	✓
2	8	2	✓
2	6	5	✓
4	0	1	
4	2	5	
6	8	6	
7	2	3	
7	7	8	
8	0	7	
8	1	2	
9	3	4	
10	5	4	
11	1	7	
14	3	5	

Weight	Source	Destination	Selected
1	7	6	✓
2	8	2	✓
2	6	5	✓
4	0	1	✓
4	2	5	
6	8	6	
7	2	3	
7	7	8	
8	0	7	
8	1	2	
9	3	4	
10	5	4	
11	1	7	
14	3	5	

Weight	Source	Destination	Selected
1	7	6	✓
2	8	2	✓
2	6	5	✓
4	0	1	✓
4	2	5	✓
6	8	6	
7	2	3	
7	7	8	
8	0	7	
8	1	2	
9	3	4	
10	5	4	
11	1	7	
14	3	5	

Weight	Source	Destination	Selected
1	7	6	✓
2	8	2	✓
2	6	5	✓
4	0	1	✓
4	2	5	✓
6	8	6	✗
7	2	3	✓
7	7	8	
8	0	7	
8	1	2	
9	3	4	
10	5	4	
11	1	7	
14	3	5	

Weight	Source	Destination	Selected
1	7	6	✓
2	8	2	✓
2	6	5	✓
4	0	1	✓
4	2	5	✓
6	8	6	✗
7	2	3	✓
7	7	8	✗
8	0	7	✓
8	1	2	
9	3	4	
10	5	4	
11	1	7	
14	3	5	

Weight	Source	Destination	Selected
1	7	6	✓
2	8	2	✓
2	6	5	✓
4	0	1	✓
4	2	5	✓
6	8	6	✗
7	2	3	✓
7	7	8	✗
8	0	7	✓
8	1	2	✗
9	3	4	✓
10	5	4	
11	1	7	
14	3	5	

Since the number of edges included in the MST equals to (V − 1), so the algorithm stops here.
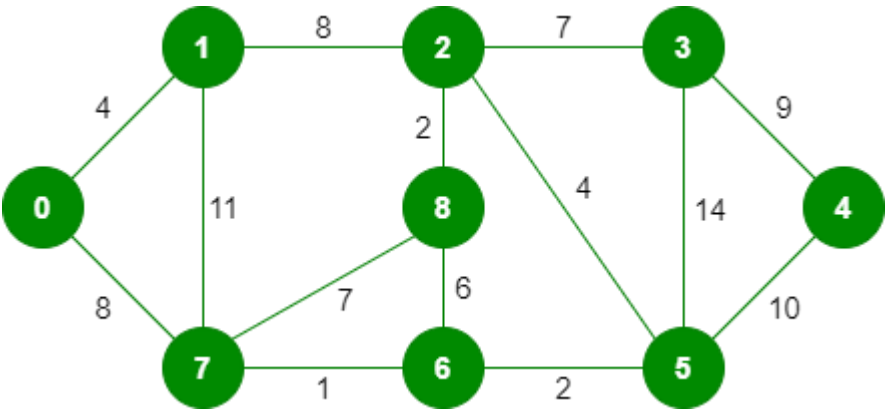
Original Graph
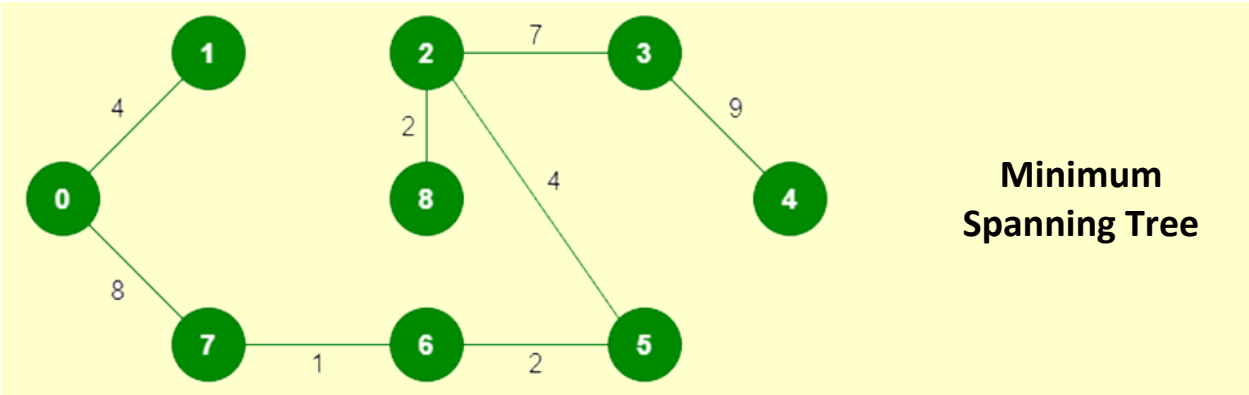
Minimum Spanning Tree

Kruskal's Algorithm

```
KRUSKAL(G = (V, E))
1  A ← {}
2  for ( each u ∈ V)
3  do create_set(u)
4  sort E in increasing order by weight w
5  for ( each (u, v) in sorted edge list)
6  do if (find(u) ≠ find(v))
7        then add (u, v) to A
8              union(u, v)
9  return A
```

Kruskal's Analysis

- Since graph is connected, we may assume that $E \geq V - 1$.

- Sorting edges (4) takes $\Theta(E \log E)$ (4)

- The for loop (5) performs O(E) find and O(V) union operations.

```
KRUSKAL(G = (V, E))
1  A ← {}
2  for ( each u ∈ V)
3  do create_set(u)
4  sort E in increasing
   order by weight w
5  for (each (u, v) in
   sorted edge list)
6  do if (find(u) ≠ find(v))
7     then add (u, v) to A
8           union(u, v)
9  return A
```

Kruskal's Analysis

- Total time for union - find is $O(E\alpha(V))$ where $\alpha(V)$ is the inverse Ackerman function.
- $\alpha(V) < 4$ for V less the number of atoms in the entire universe.
- Thus the time is dominated by sorting.
- Overall time for Kruskal is $\Theta(E \log E) = \Theta(E \log V)$ if the graph is sparse.

Prim's Algorithm
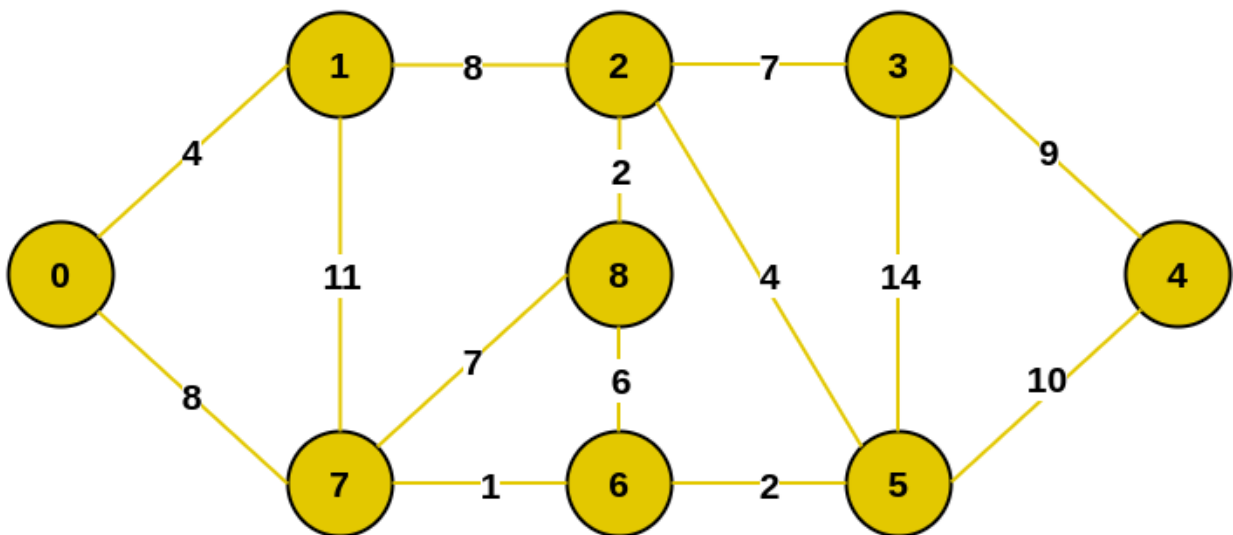
Step1: choose an arbitrary start vertex

Here we have selected vertex 0 as the starting vertex
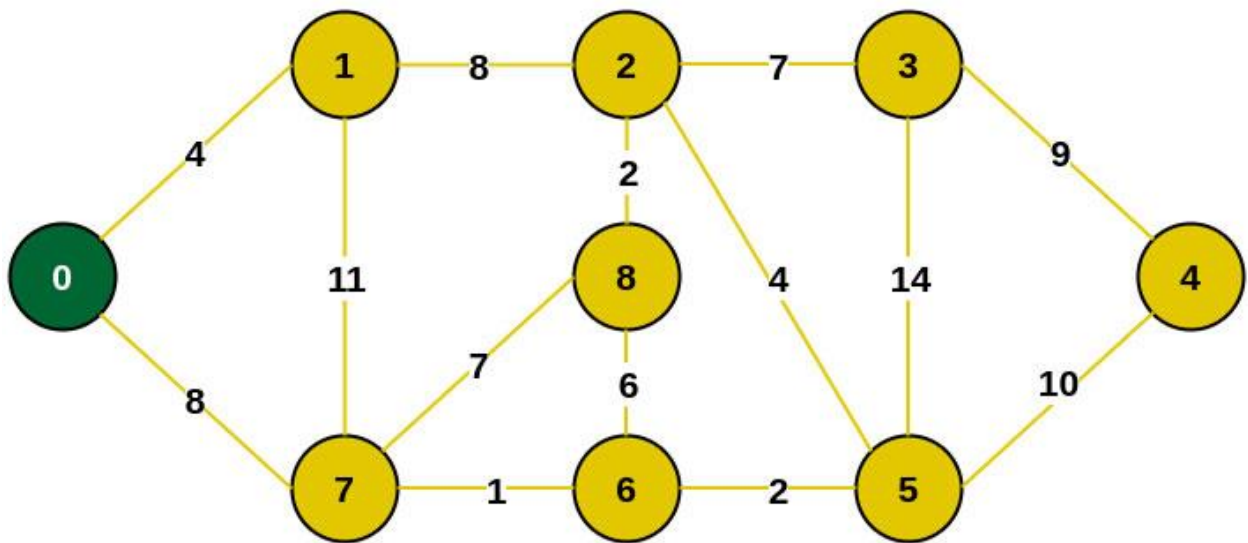
Step2: keep including connected edges having minimum weightage/cost and also keep the track of the not selected/picke edge

As vertex 0 have two connected vertexes 1 and 7 with the weightage of 4 and 8 respectively, now choose the minimum weightage edge i.e the vertex has edge weightage 4
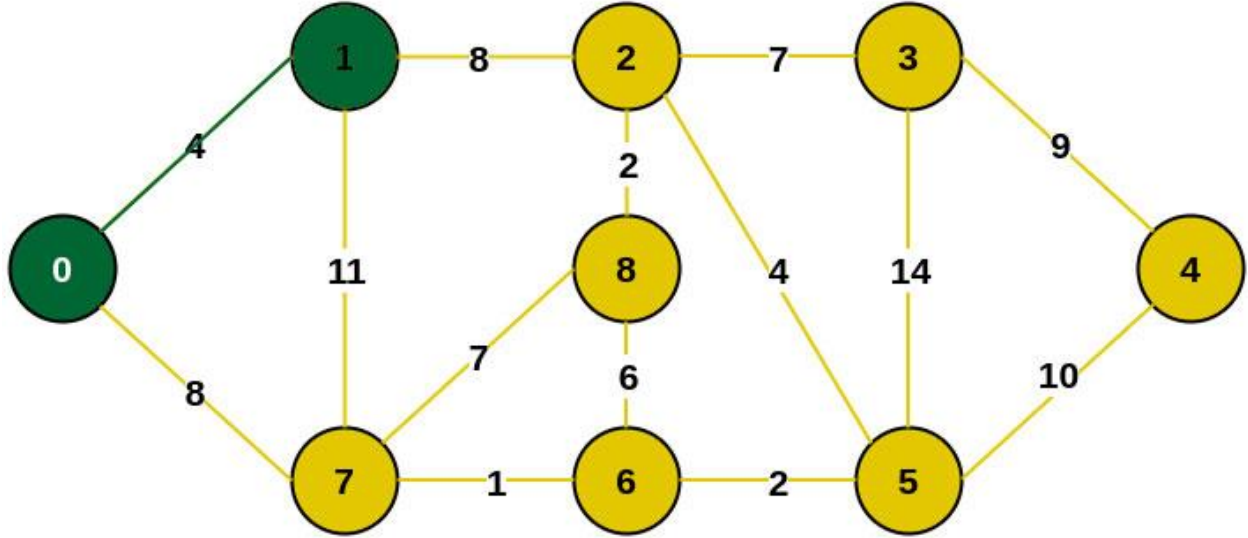
Now MST has another vertex that is 1, check the vertices connected to the 1 and also check the previous unpicked edge weightage also, and pick the minimum weightage edge
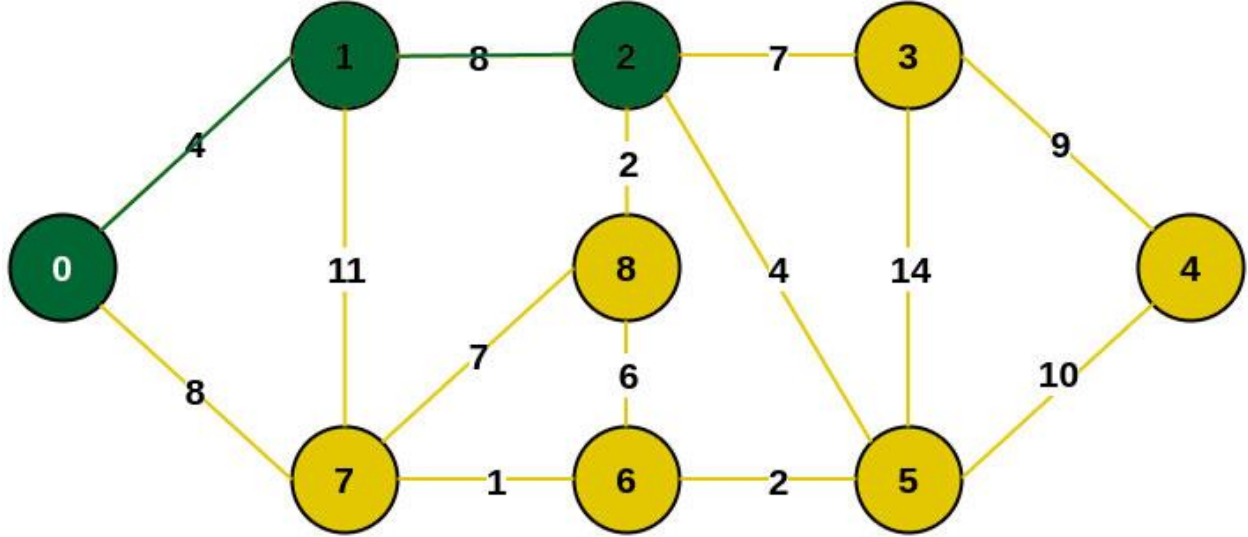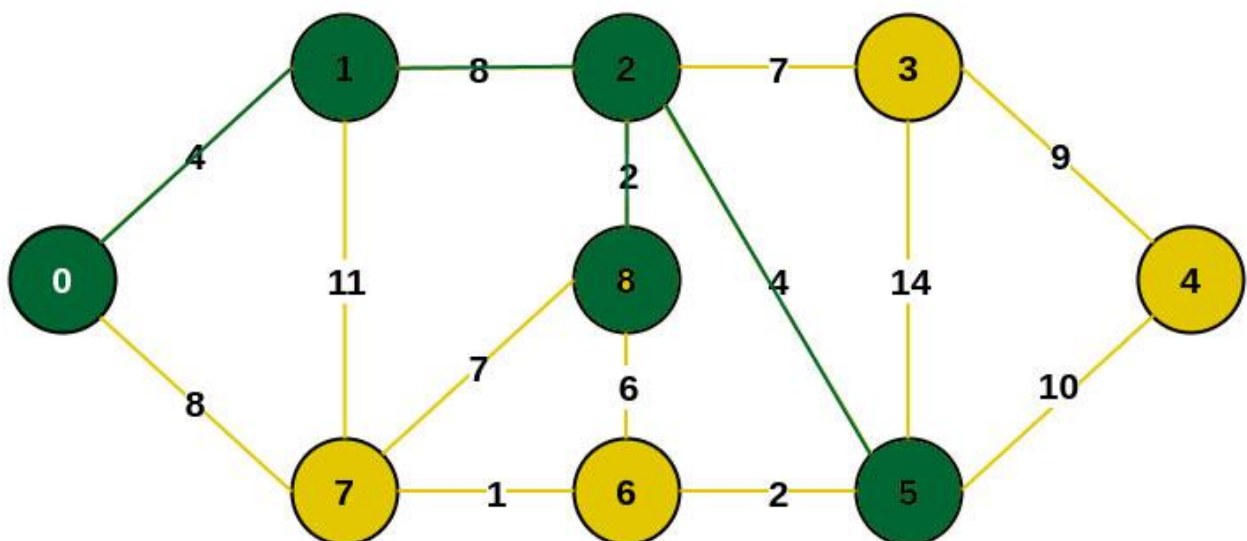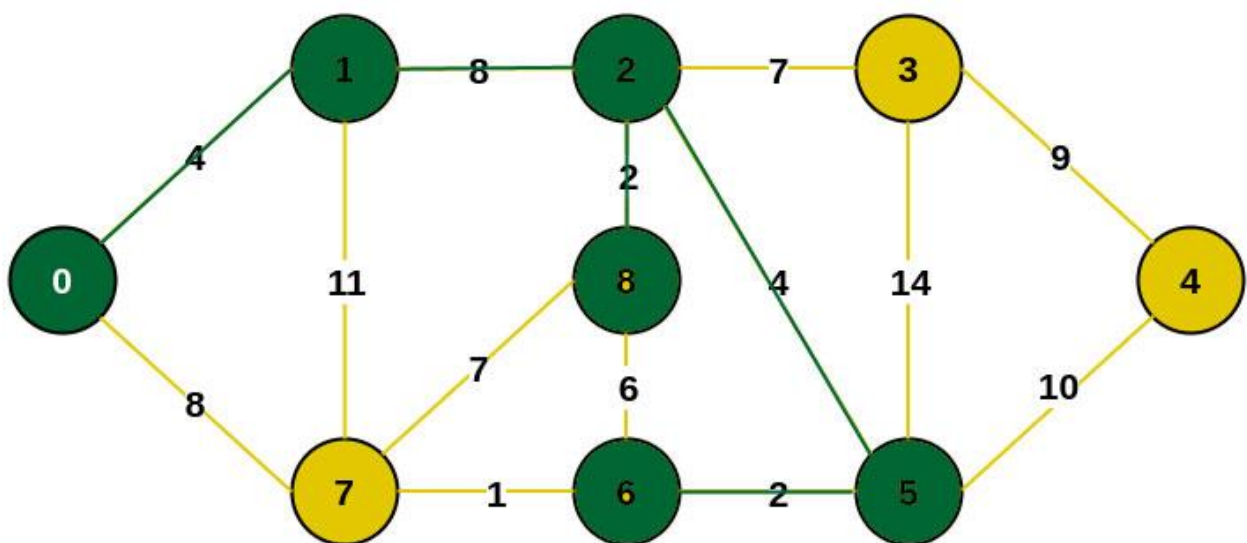
Now MST has another vertex that is 2, check the vertices connected to the 2, and pick the minimum weightage edge
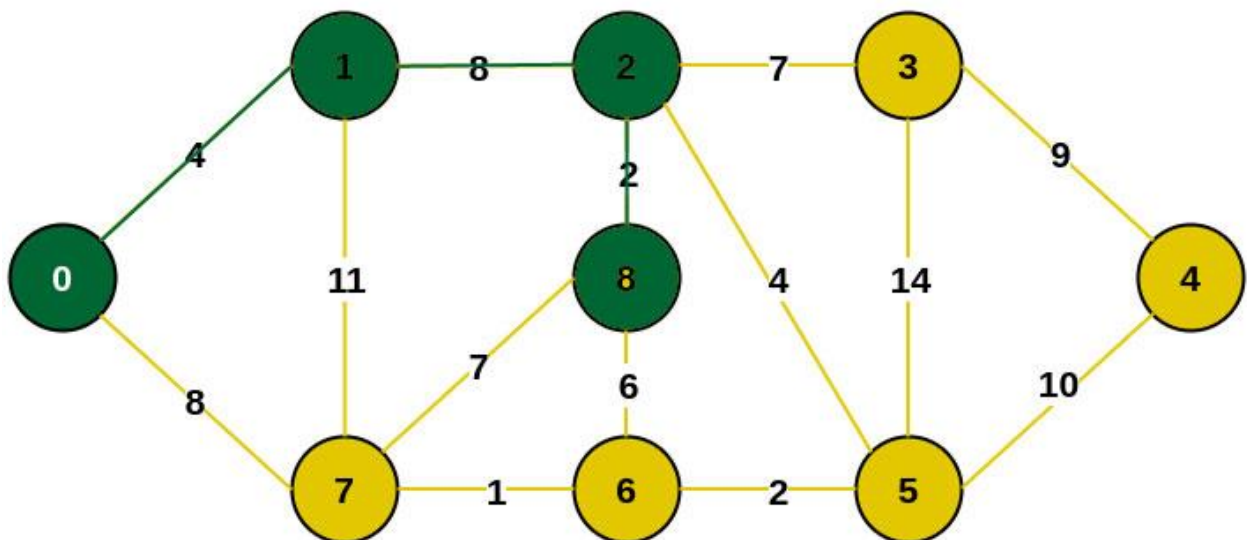
Now MST has another vertex that is 8, check the vertices connected to the 8 and check the previous unpicked edges, and pick the minimum weightage edge
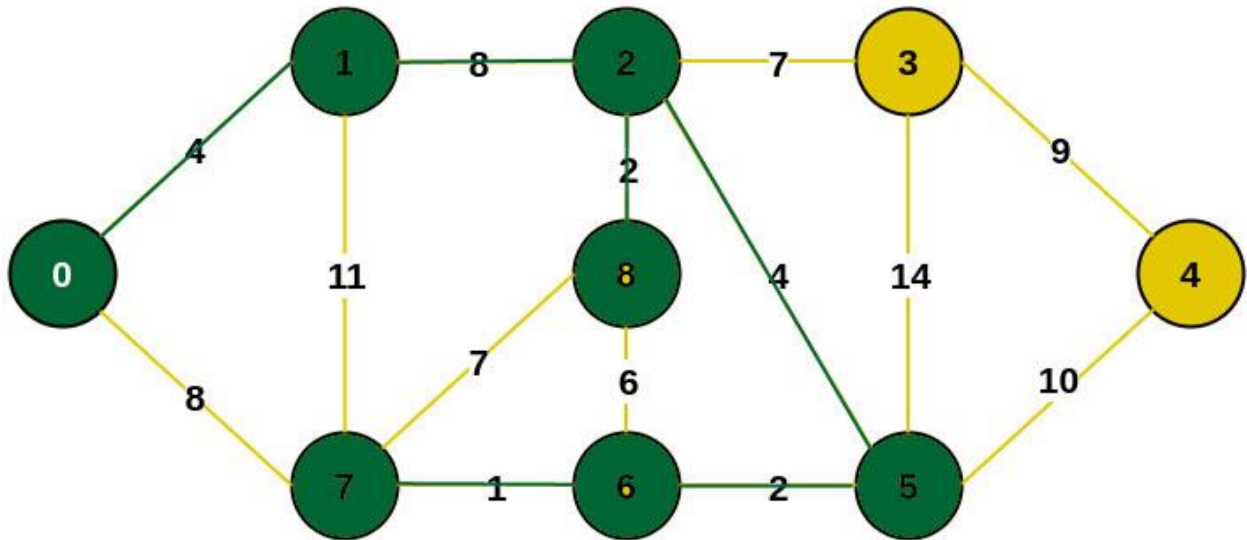

Now MST has another vertex that is 5, check the vertices connected to the 5 and check the previous unpicked edges, and pick the minimum weightage edge

Now MST has another vertex that is 6, check the vertices connected to the 6 and check the previous unpicked edges, and pick the minimum weightage edge


Now MST has another vertex that is 7, check the vertices connected to the 7 and check the previous unpicked edges, and pick the minimum weightage edge: we can see that from vertex 7 all vertices creates cycle so we not pick that edges. Now from the unpicked edges we have vertex edge between 2 and 3 with the cost of 7 which is not creating cycle so we choose that edge, we get:
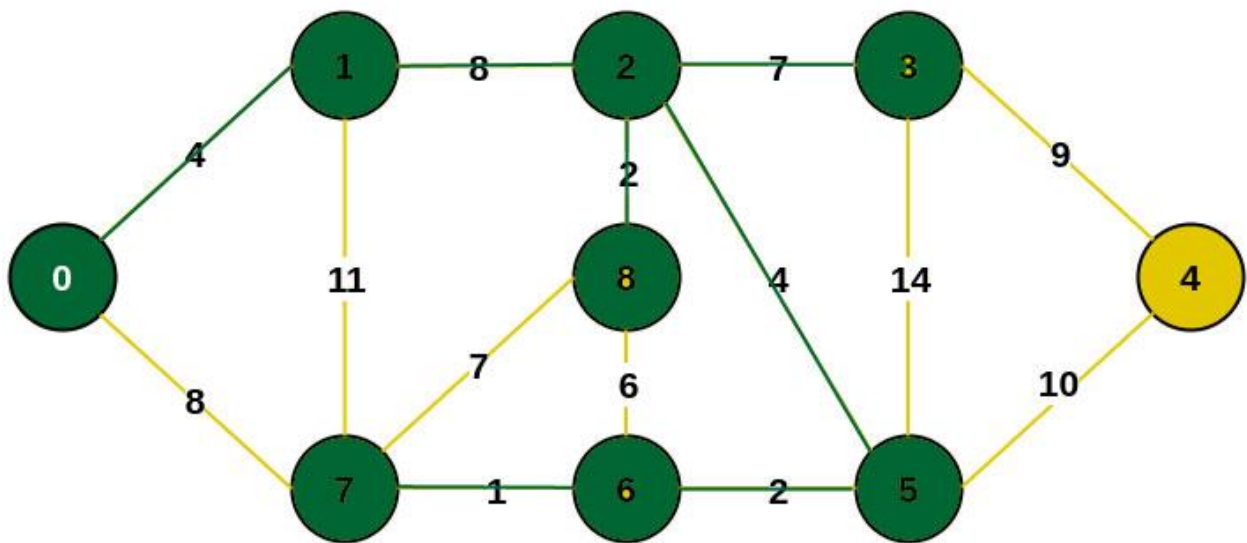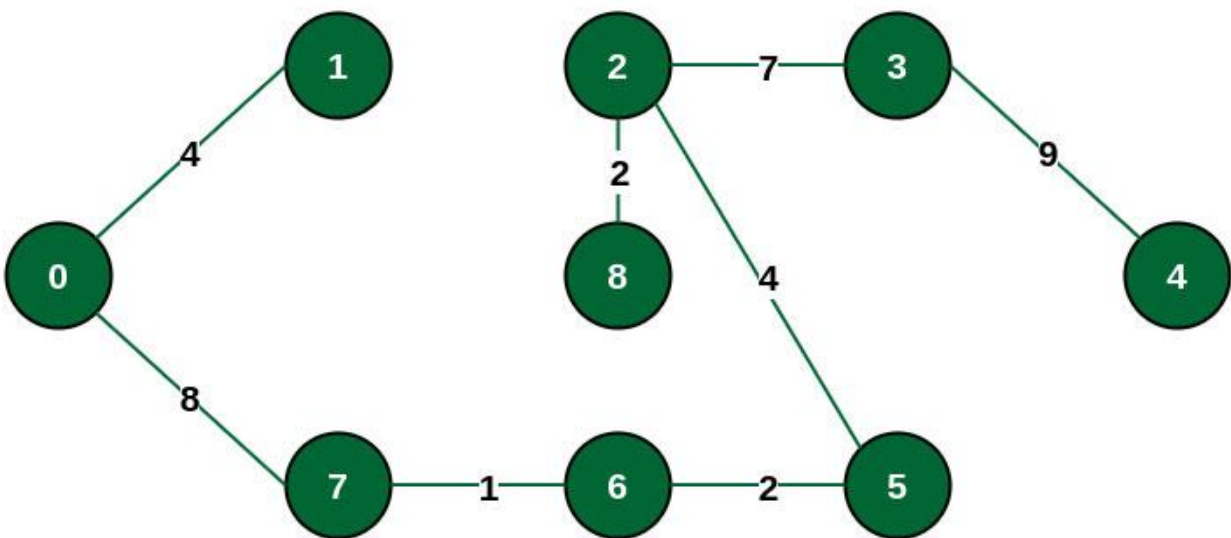

Now, unpicked edges left are {9,10,14} between vertices {5,3,4}, picked that edge which not creating cycle and has minimum weightage/cost, and fulfilling the required of $|V| - 1$ edges.
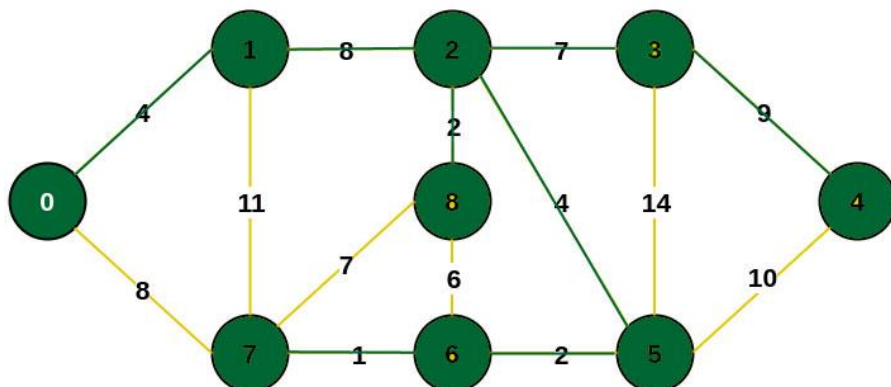
So the final MST will look like
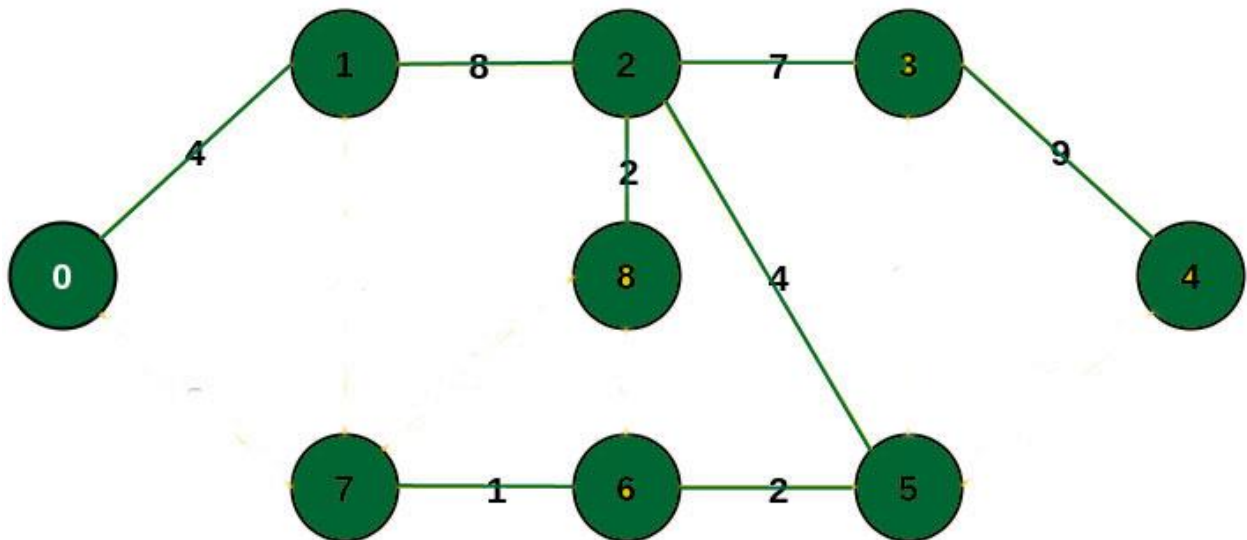
Alternative Spanning Tree would be


```
Prim(G, w, r)
{   for each u ∈ V                           initialize
    {   key[u] = +∞;
        color[u] = W;
    }
    key[r] = 0;                              start at root
    pred[r] = NIL;
    Q = new PriQueue(V);                     put vertices in Q
    while(Q is nonempty)                     until all vertices in MST
    {   u=Q.extraxtMin();                    lightest edge
        for each (v ∈ adj[u])
        {   if ((color[v] == W)&&(w[u, v] < key[v]))
            key[v] = w[u, v];                new lightest edge
            Q.decreaseKey(v, key[v]);
            pred[v] = u;
        }
        color[u] = B;
    }
}
```

Prim's Algorithm Runtime Analysis

- It takes $O(\log V)$ to extract a vertex from the priority queue.

- For each incident edge, we spend potentially $O(\log V)$ time decreasing the key of the neighboring vertex.

- Thus the total time is $O(\log V + \deg(u) \log V)$.

- The other steps of update are constant time.

Prim's Algorithm Runtime Analysis

So the overall running time is

$$T(V, E) = \sum_{u \in V} (\log V + \deg(u) \log V)$$

$$= \log V \sum_{u \in V} (1 + \deg(u))$$

$$= (\log V)(V + 2E)$$

$$= \Theta((V + E) \log V)$$

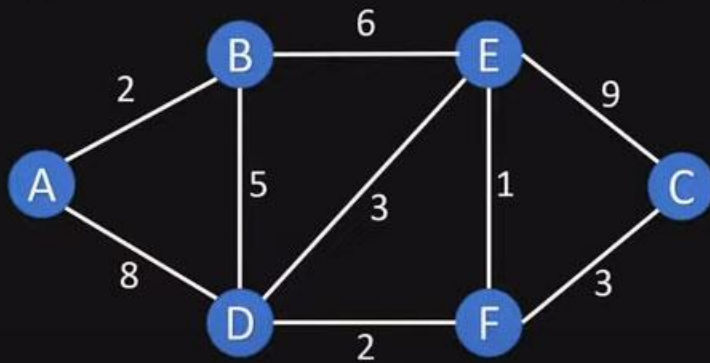Since G is connected, V is asymptotically

Prim's Algorithm Runtime Analysis

$$= \log V \sum_{u \in V} (1 + \deg(u))$$

$$= (\log V)(V + 2E)$$

$$= \Theta((V + E) \log V)$$

Since G is connected, V is asymptotically no greater than E so this is $\Theta(E \log V)$, same as Kruskal's algorithm.

Dijkstra's Shortest Path Algorithm

- Shortest path from a fixed node to every other node

- e.g. Cities and routes between them

- Calculate the shortest path from A to C

Step-1: Keep track of two list

- o List of vertices unvisited
- o List of vertices visited

Right now we haven't visited any vertex so
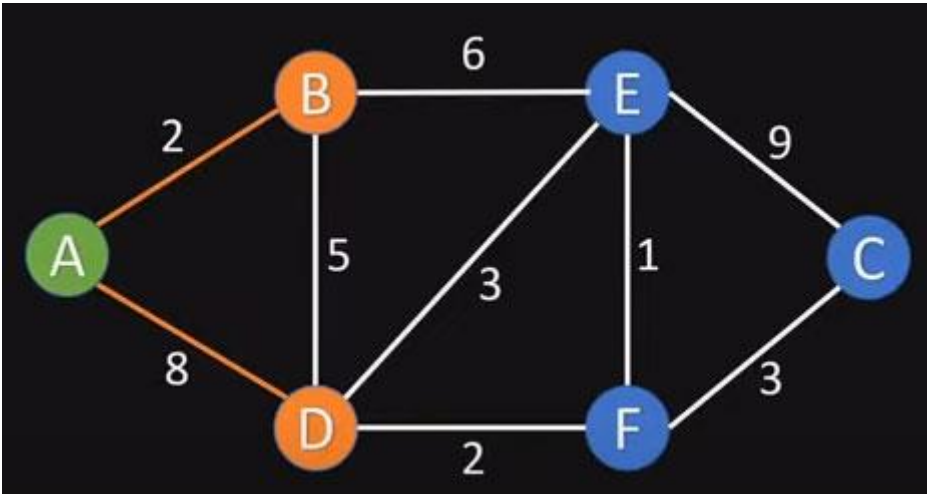
Visited Vertex [ ]   Unvisited Vertex [A, B, C, D, E, F]

Step-2: Assign to all vertices a tentative distance value, for that create a table and initialize the shortest distance with the ∞. The shortest distance of source vertex i.e. A would be zero (0).

Vertex	Shortest Distance	Previous Vertex
A	0	
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	

Now start calculating from the source vertex i.e. A

Check the distance of neighbor vertex of source A, in the above graph A's unvisited neighbor vertices are B and D.

Update the shortest distance, if new distance is shorter than old distance

| Vertex	Shortest Distance	Previous Vertex
A	0	
B	**2**	**A**
C	∞	
D	**8**	**A**
E	∞	
F	∞	

Now we have visited the vertex A, so update in the tracking lists

Visited Vertex [A]   Unvisited Vertex [B, C, D, E, F]

Step-3: To choose the next vertex from the unvisited having a minimum cost/weightage, according to above table vertex B has '2' weightage, so next visited vertex would be 'B'.
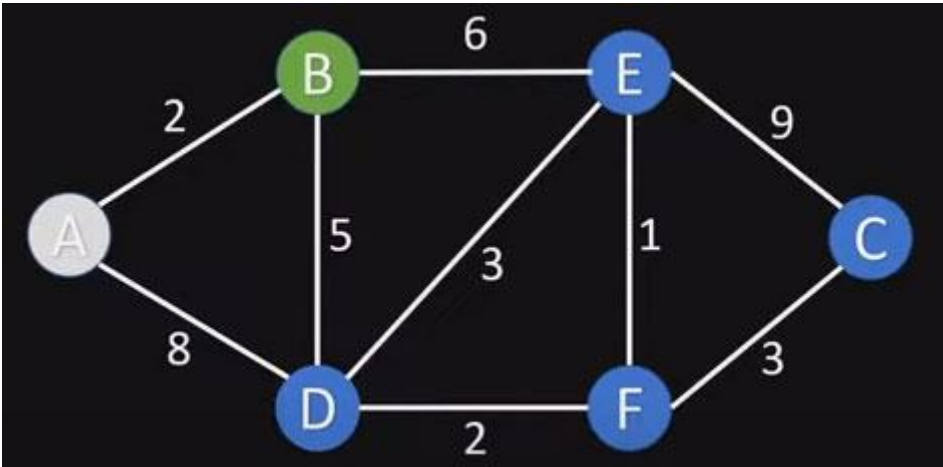

Current visited vertex is 'B' and its unvisited neighbors are 'E' and 'D'.
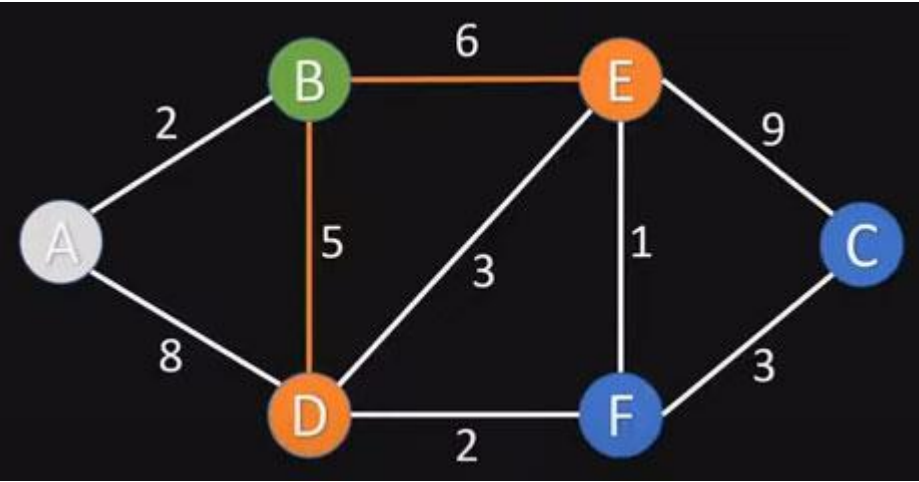
Step-4: Now calculate the distance for the above neighbors' i.e. 'E' and 'D', remember we add the weightage of current visited vertex with each neighbor vertices to get the total shortage distance between two vertices.

Our current vertex is 'B' and according to the above table 'B' has shortage distance value of '2', now add it to each neighbor's weightage.
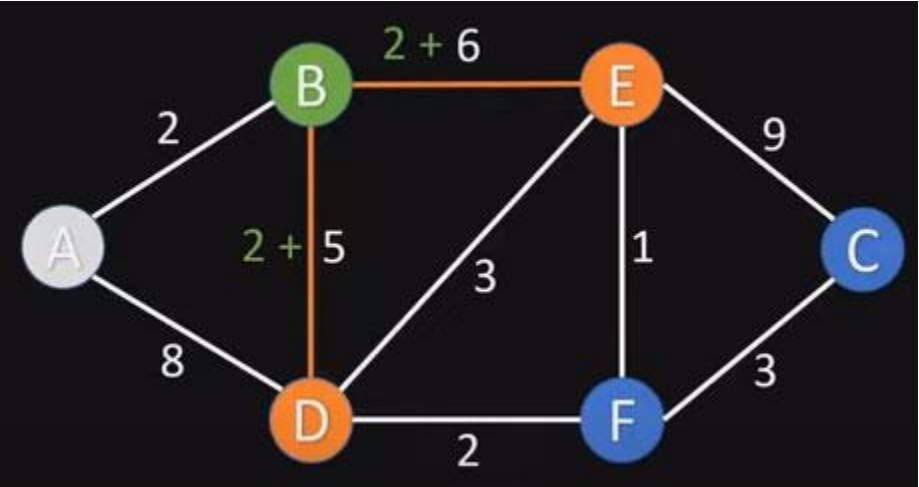
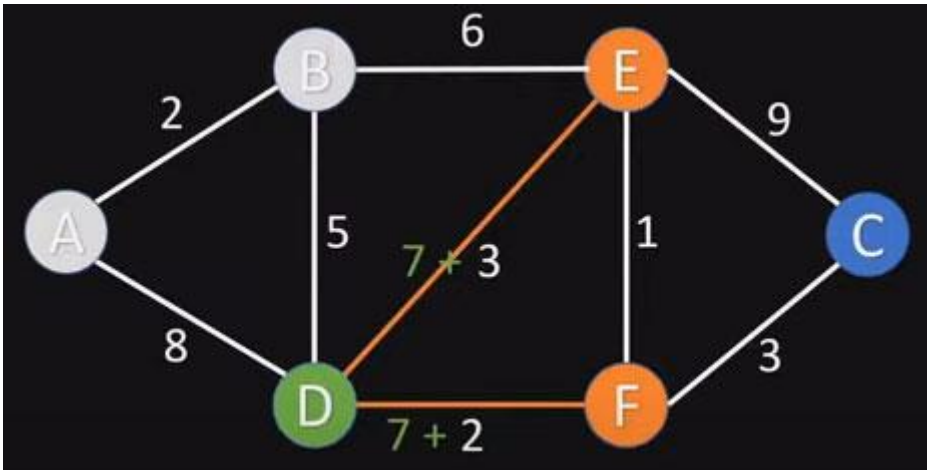Update the table and tracking list accordingly, we get.

Vertex	Shortest Distance	Previous Vertex
A	0	
B	2	A
C	∞	
D	7	B
E	8	B
F	∞	

If you observe that previously vertex 'D' had value of '8' now we updated with the new value i.e. '7', which is minimum from the previous value and also updated previous vertex with 'B'.

Repeat the step-3 to step-4, for the next vertex

Step-3: To choose the next vertex from the unvisited having a minimum cost/weightage, according to above table vertex D has '7' weightage, so next visited vertex would be 'D'.

Step-4: Now calculate the distance for the above neighbors' i.e. 'E' and 'F', remember we add the weightage of current visited vertex with each neighbor vertices to get the total shortage distance between two vertices.

Our current vertex is 'D' and according to the above table 'D' has shortage distance value of '7', now add it to each neighbor's weightage.
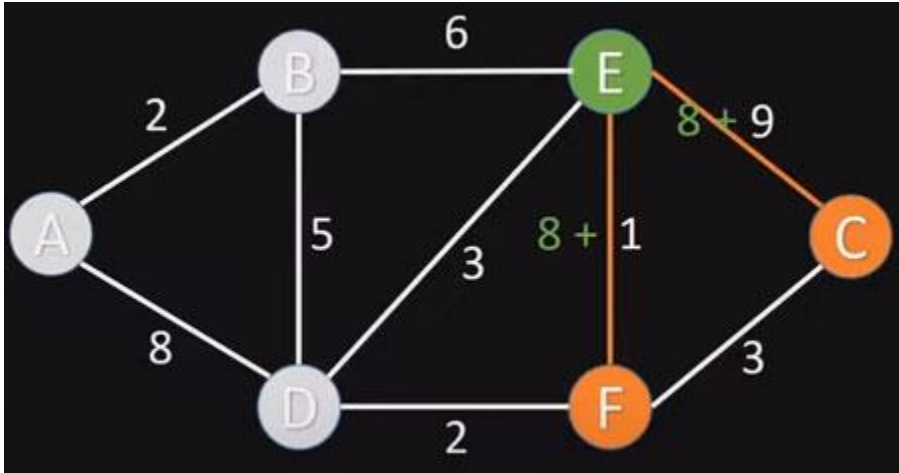
Update the table and tracking list accordingly, we get.

Vertex	Shortest Distance	Previous Vertex
A	0	
B	2	A
C	∞	
D	7	B
E	8	B
F	9	D

If you observe that vertex 'E' has value of '8' and current calculated distance is '10' which is greater than the previous value so we not update the value of 'E'.

Visited Vertex $[A, B, D]$ **Unvisited Vertex** $[C, E, F]$

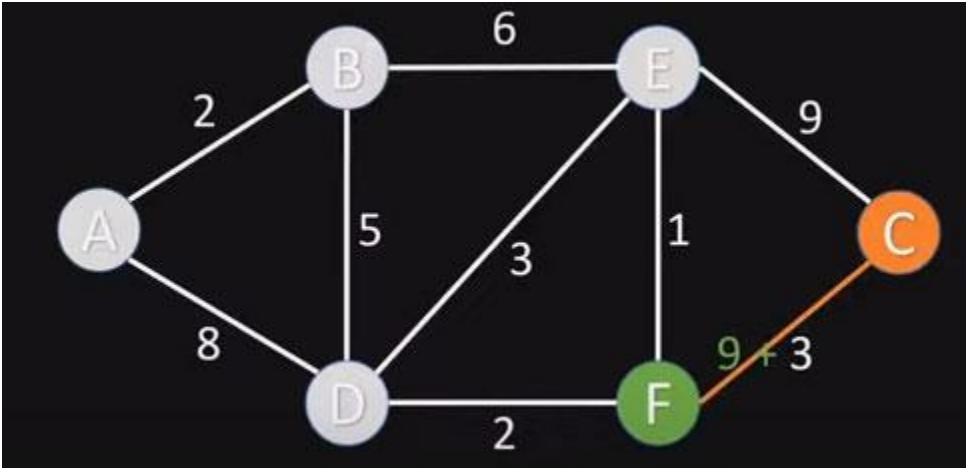Repeat the step-3 to step-4, for the next vertex

Update the table and tracking list accordingly, we get.

Vertex	Shortest Distance	Previous Vertex
A	0	
B	2	A
C	17	E
D	7	B
E	8	B
F	9	D

If you observe that vertex 'F' has the same value of '8' already so we don't update the value of 'F'.

Visited Vertex $[A, B, D, E]$ **Unvisited Vertex** $[C, F]$

Repeat the step-3 to step-4, for the next vertex

We have only one distance vertex i.e. 'C' and it has value of 17, but from vertex 'F' to 'C' its distance calculation give us value of '12', so we update the value of vertex 'C' with '12'.

| Vertex	Shortest Distance	Previous Vertex
A	0	
B	2	A
C	12	F
D	7	B
E	8	B
F	9	D

Visited Vertex [A, B, D, E, F]   **Unvisited Vertex** [C]

If the final vertex left in the unvisited vertex list, then the algorithm stops calculating and include the final vertex i.e. 'C' in the list of visited vertices.
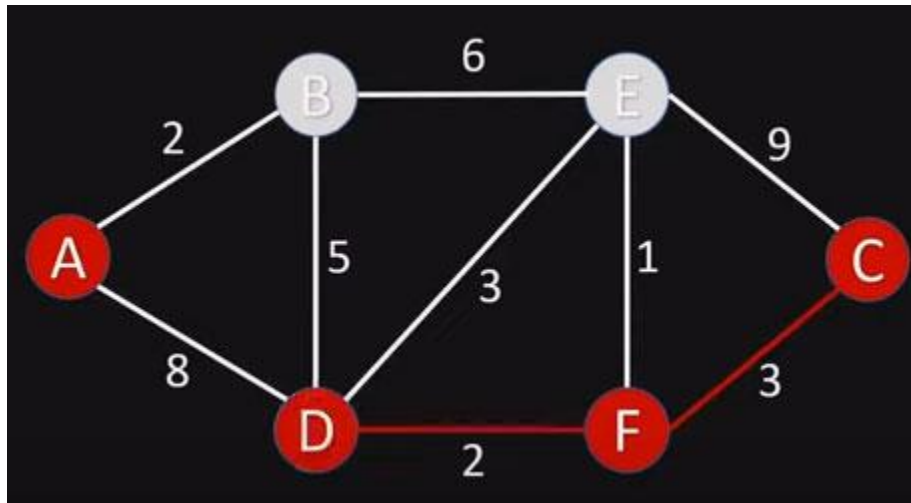
Visited Vertex [A, B, D, E, F, C]   **Unvisited Vertex** [ ]

The final step is to re-construct the path according to the values given in the above table. We were finding the shortest path between vertex 'A' and 'C', now move backward from vertex 'C' to 'A'
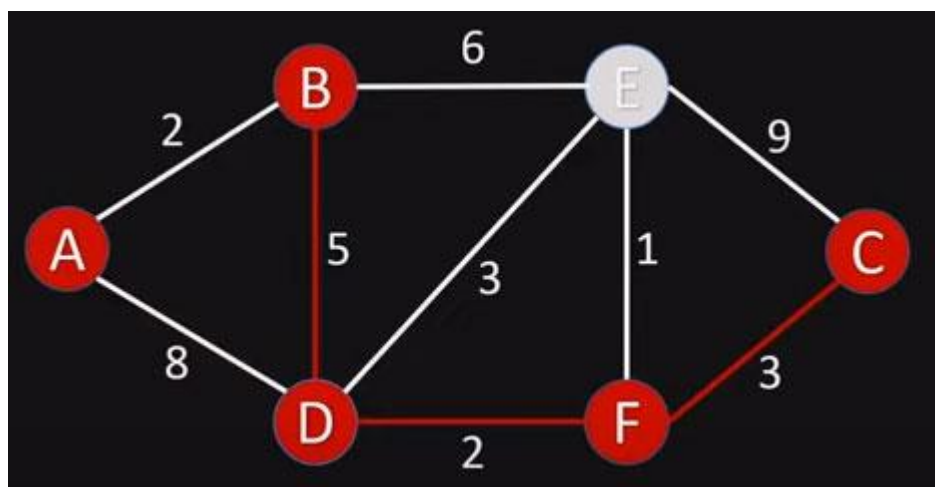
So starting from vertex 'C' and in the table its pervious vertex is 'F'
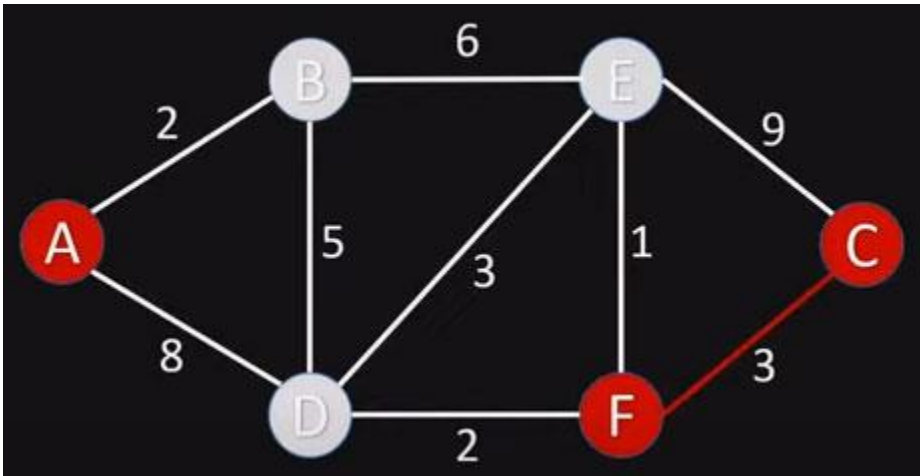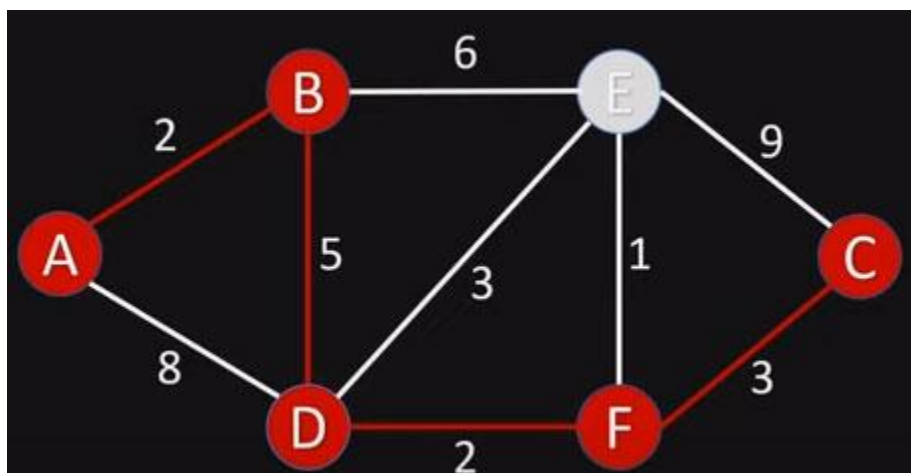
Now in the table vertex 'F' has previous vertex 'D'
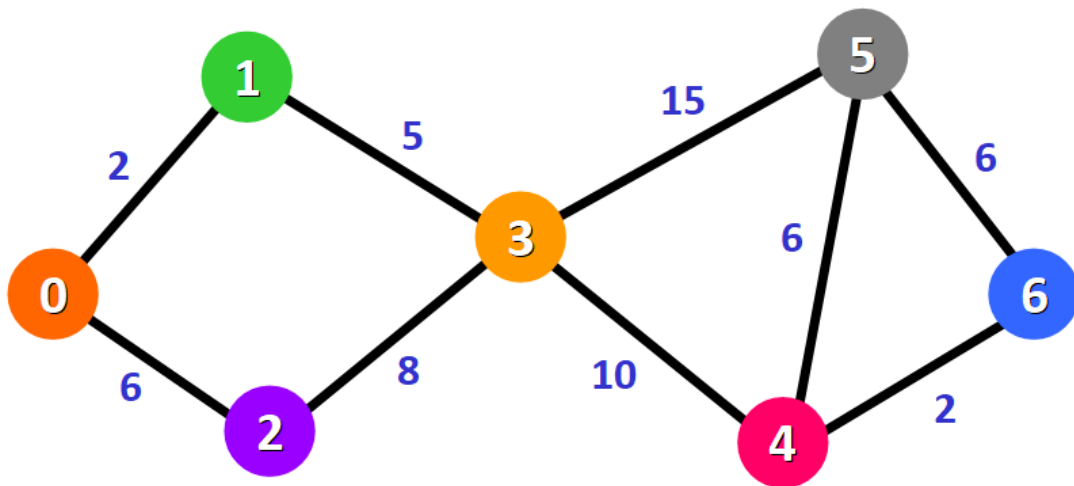

Vertex 'D' has previous vertex 'B'

Vertex 'B' has previous vertex 'A'

Hence, the shortest path weightage/cost from 'A' to 'C' is

$$\{2 + 5 + 2 + 3\} = 12$$

Let's take another example: