01/07/20

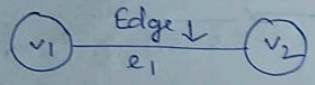
Graph.

Graph is an important data structure in computer science due to its innumerable applications in real life scenarios. We unknowingly deal with them in our ~~daily~~ day to day life.

Graph : Represents pair-wise relationship between a set of objects. Component of Graph are →

1. Vertex (nodes)
2. Edges (arc)

The edges determine the relationship between pair of vertices.



A basic graph with two vertices v_1, v_2 and an edge e_1 between them.

A Graph can be of two types :

1. **Directed graph (di-graph)** : have pair of ordered vertices (u, v)

i.e. (u, v) is not equal to (v, u)

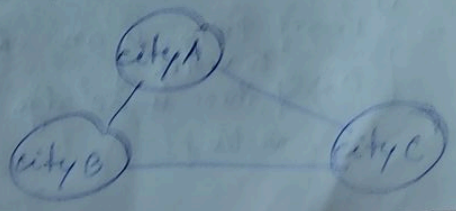
2. Un-directed Graph : Have pair of unordered vertices. (u,v) and (v,u) are same.

Graph applications :

1. Social Networks - Facebook, LinkedIn etc.
2. City - Road network.
3. Precedence constraints.

Graphs are used to represent a social network. In these networks each person is represented with a node. These node contains information like person id, name, his gender, his location etc and the edges are used to represent a relationship or a friendship.

Graphs are also used to represent a road network. Consider a network of cities. If we say in terms of graph theory these cities can be seen as vertices of a graph and the road used to connect them can be the edges.



Precedence constraints problems refers to situation where in there are certain pre-requisites that need to be followed. An example for it could be in your college, your college may issue constraints that in order to have course B, you need to pass course A making it a pre-requisite for course B.

Graph Representation.

There are generally two ways to represent a graph data structure —

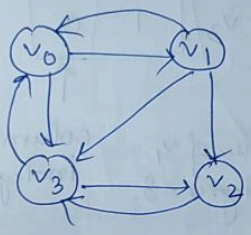
1. Adjacency matrix.
2. Adjacency list.

Adjacency matrix :

Adjacency matrix is a matrix that maintains the information of adjacent vertices. The entries of this adjacency matrix are filled using this definition -

$$A(i,j) = \begin{cases} 1 \Rightarrow \text{if there is an edge from vertex } i \text{ to vertex } j \\ 0 \Rightarrow \text{if there is no edge from vertex } i \text{ to vertex } j \end{cases}$$

Hence all the entries of this matrix are either 1 or 0. Let us take a directed graph and write the adjacency matrix for it.



(a) Directed graph.

	v_0	v_1	v_2	v_3
v_0	0	1	0	1
v_1	1	0	1	1
v_2	0	0	0	1
v_3	1	0	1	0

(b) Adjacency matrix for graph (a).

Here the matrix entry $A(0,1) = 1$, which means that there is an edge in the graph from vertex v_0 to vertex v_1.

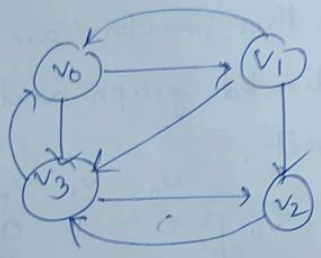
Similarly $A(2,0) = 0$, which means that there is no edge from vertex v_2 to vertex v_0.

—x—

Outdegree and Indegree:

In the adjacency matrix of a directed graph, rowsum represents the outdegree and columnsum represents the indegree of that vertex.

eg. from the above matrix, we can see that the rowsum of vertex v_1 is 3 which is its outgegree and columnsum is 1 which is its indegree.



	v_0	v_1	v_2	v_3
v_0	0	1	0	1
v_1	1	0	1	1
v_2	0	0	0	1
v_3	1	0	1	0

row sum of v_1 outdegree = 3.

column sum of v_1 indegree = 1.

—x—

Adjacency list :

In adjacency list representation of graph, we will maintain two lists.

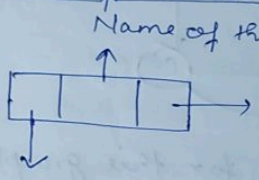
First list will keep track of all the nodes in the graph.

Second list will maintain a list of adjacent nodes for each node.

Suppose there are n nodes then we will create one list which will keep information of all nodes in the graph and after that we will create n lists, where each list will keep information of all adjacent nodes of that particular node.

Each list has a header node, which will be the corresponding node in the first list.

Structure of header node.



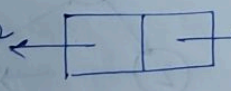
Pointer which points to the first node adjacent to header node
(edge * adj)

Pointer which points to the next node in header node list
(node * next)

```
Struct node {
        struct node * next;
        char name;
        struct edge * adj;
};
```

Structure of edge.

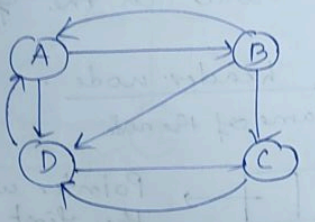
Name of the destination node of the edge
(char dest)



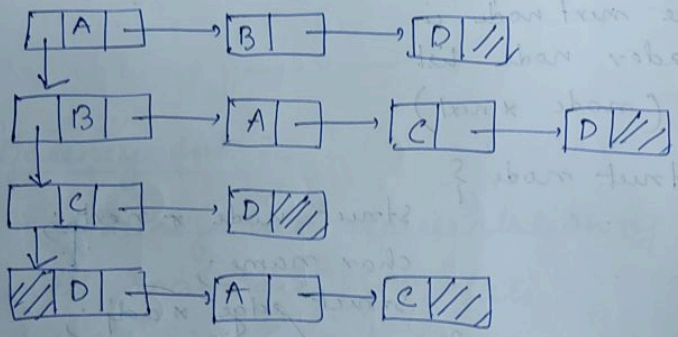
Pointer which points to the next adjacent node of header node
(edge * link).

```
Struct edge {
        char dest;
        struct edge * link;
};
```

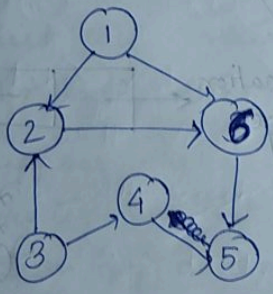
Let us take the same graph.



The adjacency list for this graph will be as.



H.W.



Find out the adjacency matrix and adjacency list.

19/7/20

(Graph Traversal & Tree Traversal) Difference

1. There is no first vertex or root vertex in a graph, hence the traversal can start from any vertex.

2. In tree or list, when we start traversing from the first vertex, all the elements are visited but in graph only those vertices will be visited which are reachable from the starting vertex.

3. In tree or list while traversing, we never encounter a vertex more than once while in graph we may reach a vertex more than once. So to ensure that each vertex is visited only once, we have to keep the status of each vertex whether it has been visited or not.

4. In tree or list we have unique traversal like inorder, preorder and postorder. But in graph, there is no any order of traversal. There are two techniques of traversal of graph

   (1) BFS (Breadth first search)

   (2) DFS (Depth First search)

DFS. (Depth First Search)

→ DFS stands for Depth First Search.

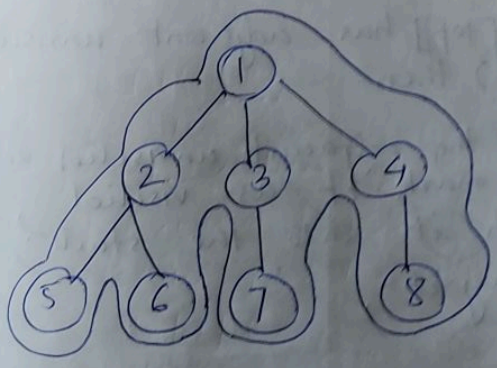
→ DFS algorithm traverses the graph in a Depthward motion.

→ Stack data structure is used to implement DFS technique.

→ DFS produces Non-optimal solution.

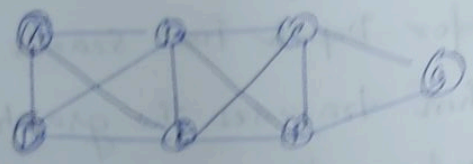
→ Time complexity of DFS is O (V+E)

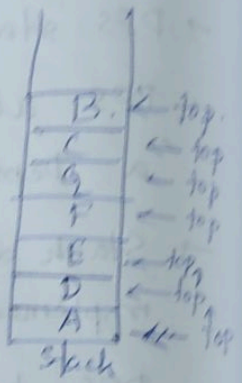
→ Applications : ① Strongly connected graph.
② Acyclic graph.
③ Topological order.



DFS : 1 2 5 6 3 7 4 8

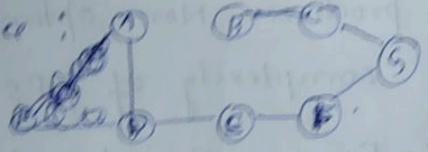
Apply DFS algorithm for the following graph.





DFS(G): A D E F G C B

DFS spanning tree:



Algorithm:

step1: Put the starting vertex into the stack.
Mark it as visited.
→ Display it

step2: If (stack[top] has adjacent unvisited vertex) then.
{
→ visit the adjacent unvisited vertex.
and mark it as visited.
→ push it into the stack;
→ Display it;
}
else
{
→ pop the top element from the stack;
}
→ repeat step2 until stack is empty.

BFS :

→ BFS stands for Breadth First search

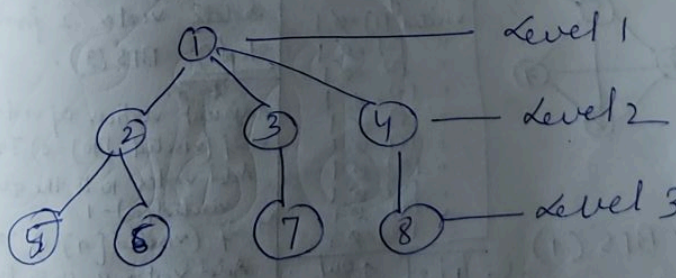
→ BFS algorithm traverses the graph in a Breadthward motion.

→ Queue Data Structure is used to implement BFS technique.

→ BFS produces optimal solution.

→ Time complexity of BFS is O(V+E)

→ Applications : (1) Bipartite Graph
(2) Connected components
(3) Single source shortest path problem.

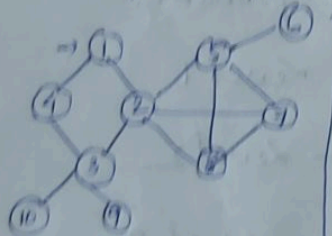


BFS : 1 2 3 4 5 6 7 8.

Algorithm for Breadth first search (BF) with example.

Algorithm BFS(v)
{
u := v; // Insert v into the queue
visited[v] = 1; // mark vertex v is visited.
do
{
for all vertices w adjacent from u do
{
if (visited[w] = 0) then
{
Add w to queue;
visited[w] = 1;
}
}
if (q is empty) then return;
Delete u from queue;
} while (false);
}

Example: Find out the BFS visiting order sequence of vertices for the following graph.



Initially:
visited[1] = 0 / 1
" [2] = 0 / 1
" [3] = 0 / 1
" [4] = 0 / 1
" [5] = 0 / 1
" [6] = 0 / 1
" [7] = 0 / 1
" [8] = 0 / 1
" [9] = 0 / 1
" [10] = 0 / 1

step 1: BFS(1)
u = 1
In: 1 | 2 → out

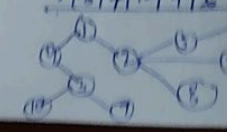
Insert vertex 1 into the queue.
visited[v] = 1.
Adjacent vertices of 1 are 4 and 2.
If (visited[4] == 0) then
Insert 4 into the queue.
visited[4] = 1
If (visited[2] == 0) then
Insert 2 into the queue.
visited[2] = 1
Delete 1 from queue.

step(2): BFS(4)
u = 4
Adjacent vertices of vertex 4 is 3.
i.e. u = 4, i = 3.
If (visited[3] = 0) then
add vertex 3 into the queue.
visited[3] = 1.
Delete vertex 4 from the queue.
In: 4 | 2 | 3 → delete

step(3): BFS(2)
u = 2
Adjacent vertices of 2 are 5, 7, 8.
→ visited[5] = 0 (True)
Add 5 into the queue.
visited[5] = 1
→ visited[7] = 0 (True)
Add 7 into the queue.
visited[7] = 1
→ visited[8] = 0 (True).
Added 8 into the queue.
visited[8] = 1.
In: 2 | 3 | 5 | 7 | 8 → out
Delete vertex 2 from the queue.

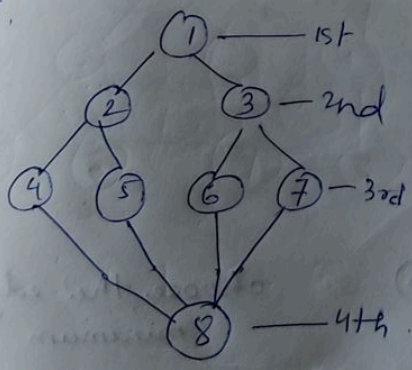
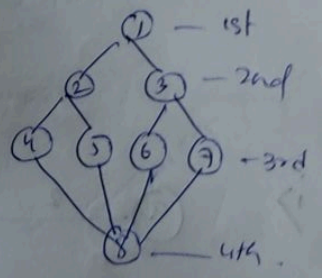
step(4): BFS(3).
u = 3.
Adjacent vertices of vertex 3 are 10, 9.
→ If (visited[10] = 0) True.
Add vertex 10 to the queue.
visited[10] = 1
→ If (visited[9] = 0) True
Add vertex 9 into the queue
visited[9] = 1 : 3 | 5 | 7 | 8 | 10 | 9
Delete vertex 3 from the queue.

step(5): BFS(5)
u = 5
Adjacent vertices of vertex 5 is 6.
If (visited[6] = 0) True
Add 6 to the queue.
visited[6] = 1
Delete 5 from the queue.
In: 5 | 7 | 8 | 10 | 9 | 6



spanning tree

Differences between BFS and DFS.

DFS.	BFS.
① The idea of DFS is proceeding to highes levels successively in the first opportunity. Later we backtrack and add the vertices which are not visited in the graph.	① The idea of BFS is to visit all the vertices sequentially on a given level before going on to the next level in the graph.
② DFS uses stack data structure to store the vertices in the graph.	② BFS uses queue data structure to store the vertices in the graph.
③ DFS is an Edge-based algorithm.	③ BFS is vertex based algorithm.
④ Ex.  1-2-4-8-5-6-3-7	ex. Ex  1-2-3-4-5-6-7-8

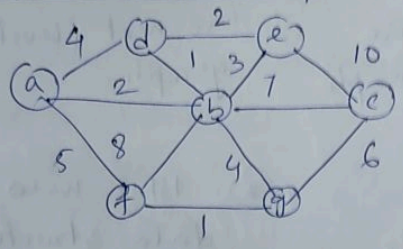
Minimum Spanning Tree.

Subgraph
- all vertices.
- connected
- No cycles.

Min ?

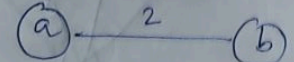
Σ wts edges is minimum.

Prim's Minimum Spanning Tree.

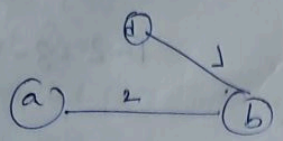


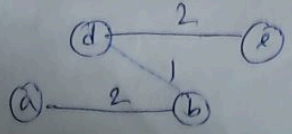
(1) Choose an arbitrary start vertex.

(2) Keep including connected min. edges provided that it does not form cycle.

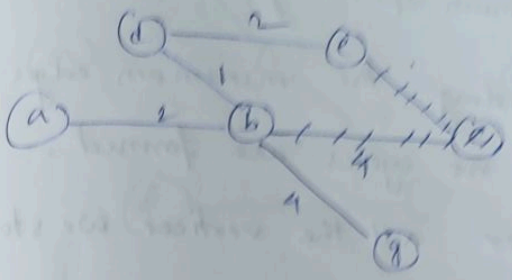
i) 

choose the edge which is minimum.

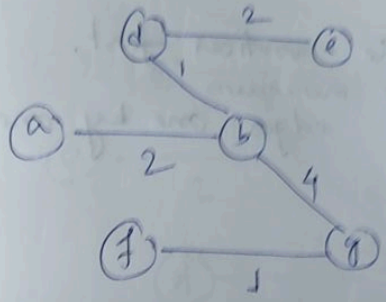
ii) 

iii) 

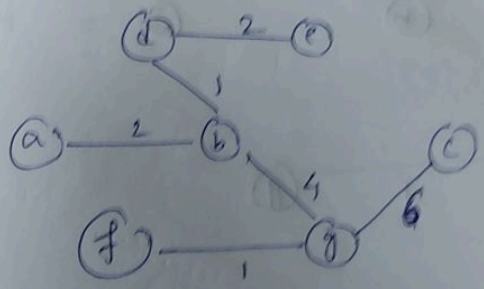
iv)



v)



vi)



cost = 2 + 1 + 2 + 4 + 1 + 6

= 16. which is a minimum spanning tree.

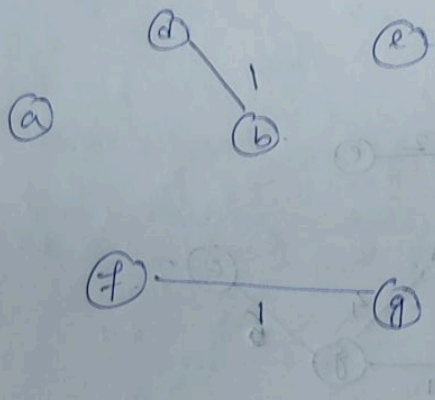
Kruskal- Minimum spanning tree.

① Keep including the minimum edges as long as no cycles are formed. Once we cover all the vertices we stop.

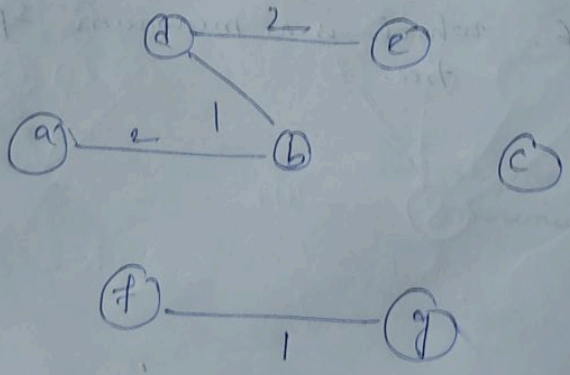
② Draw all the vertices first.

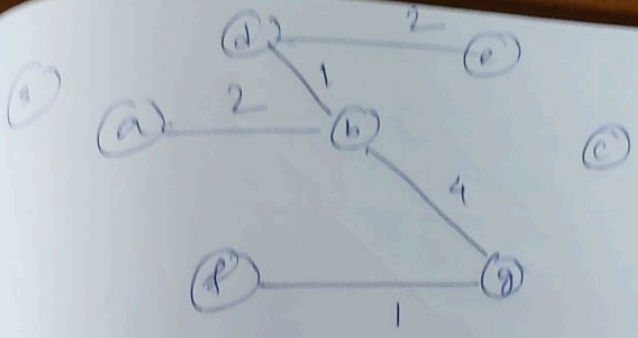
③ start including the minimum edges one by one.

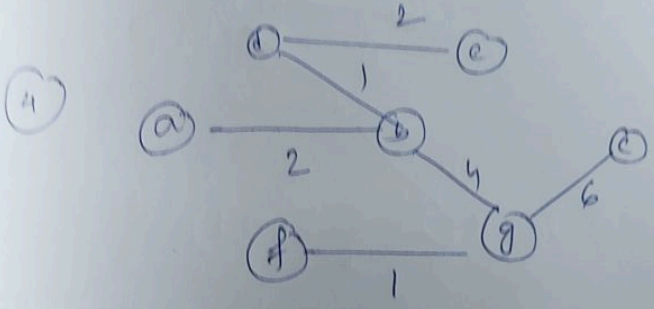
①.



②.







wt = 16.

Minimum Cost Spanning Tree.

1) Kruskal's Algorithm.

The steps of the algorithm are —

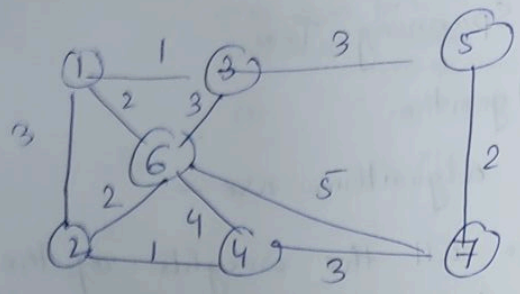
step(1). Arrange all the weights of the edges in increasing order of weights.

step(2) First select the minimum weight edge, and added to the spanning tree. ~~without forming~~

step(3). Next select the minimum weight edge and added to the spanning tree without forming a cycle.

step(4) Repeat step 3 for all the remaing edges until n-1 edges are present in the spanning tree.

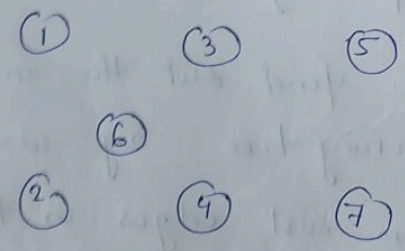
step(5). Then find out the minimum cost spanning tree by summing minimum cost edges in the spanning



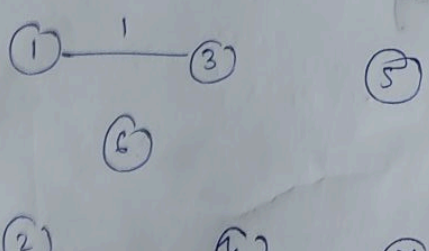
Arrange the weights of the edges in G in creasing order

1-3 = 1 Accepted
2-4 = 1 Accepted
1-6 = 2 "
2-6 = 2 "
5-7 = 2 "
1-2 = 3 "
3-6 = 3 "
4-7 = 3. "
3-5 = 3 "
6-4 = 4 "
6-7 = 5 "

step 1

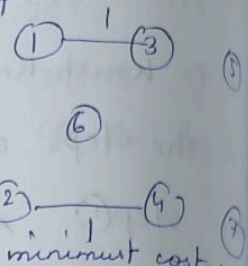


step 2.



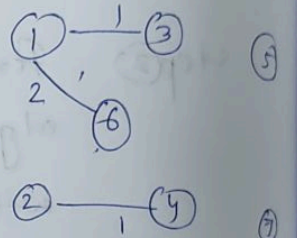
minimum cost = 1.

step 3



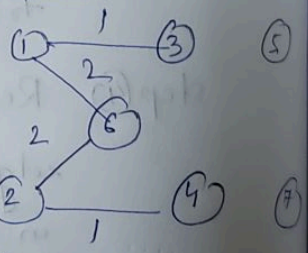
minimum cost = 1+1=

step 4



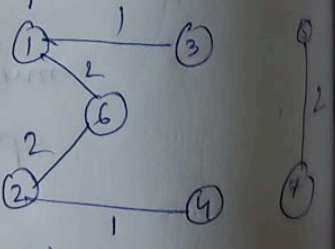
minimum cost = 1+2+1=4.

step 5



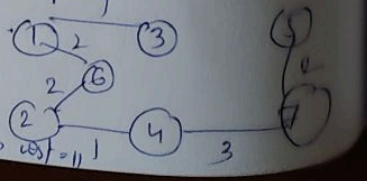
minimum cost = 1+2+2+1 = 6.

step 6



minimum cost 1+2+2+1+2 = 8.

step 7



minimum cost = 11

Prim's Algorithm.

The steps of the algorithm are -

(1) Initially the spanning tree is empty.

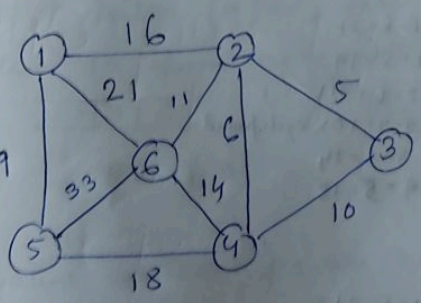
(2) Consider any vertex from v and added to the spanning tree.

(3) Find all the neighbouring adjacent vertices, consider all the edges of the neighbouring vertices and then add the minimum cost edge in the spanning tree without forming a cycle.

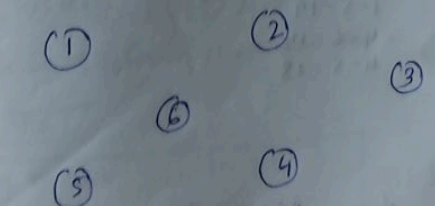
(4) Repeat step 3 until all the vertices are visited without forming a cycle.

(5) Find the sum of the costs of the edges in the spanning tree. Finally we get the minimum cost spanning tree.

ex.

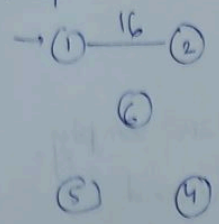


step(1).



minimum cost = 0.

Step2

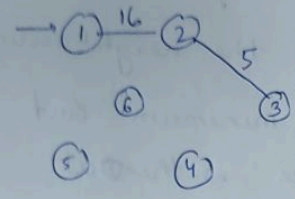


1-2 = 16 ✓
1-6 = 21
1-5 = 19

Select 1-2 edge with min. cost

min. cost = 16.

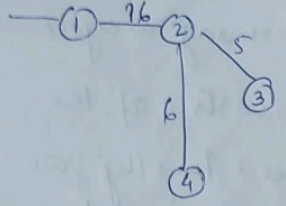
Step3



1-6 = 21
1-5 = 19
2-3 = 5 ✓
2-4 = 6
2-6 = 11

select 2-3 edge with min. cost 5 added to the spanning tree without forming a cycle.

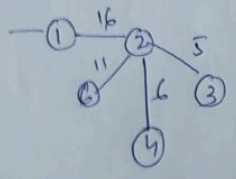
Step4



1-6 = 21
1-5 = 19
2-4 = 6 ✓
2-6 = 11
3-4 = 16

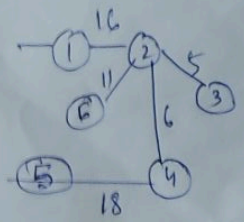
select 2-4 edge with min cost 6. added to the s.tree.

Step5



1-6 = 21
1-5 = 19
2-6 = 11 ✓
3-4 = 10 ✗ cycle formed.
4-6 = 14
4-5 = 18.

Step6



1-6 = 21
1-5 = 19
4-6 = 14
4-5 = 18.

Minimum spanning tree = 16 + 5 + 6 + 11 + 18
= 56.

Step 6: BFS (8)

u = 8

Adjacent vertices of 8 are 2, 5, 7.

If (visited [2] = 0) False.

" (" [5] = 0) "

" (" [7] = 0) "

Delete 8 from the queue.

7	10	9	6

Step 7: BFS (7)

u = 7

Adjacent vertices of 7 are 2, 5, 8.

If (visited [2] = 0) false.

" (" [5] = 0) "

" (" [8] = 0) "

Delete 7 from the queue.

10	9	6

Step 8: BFS (10).

u = 10

Adjacent vertices of 10 are 3.

If (visited [10] = 0) false.

Delete 10 from the queue.

9	6

Step 9: BFS (9)

u = 9

Adjacent vertices of 9 are 3.

If (visited [9] = 0) false.

Delete 9 from the queue.

6

Step 10: BFS (6)

u = 6

Adjacent vertices of 6 are 5

If (visited [5] = 0) false.

Delete 6 from the queue.

Adjacenty matrix =

Time complexity =

$O(n^2) + O(n) = O(n^2)$

BFS visiting order sequence

1-4-2-3-5-8-7-10-9-6.