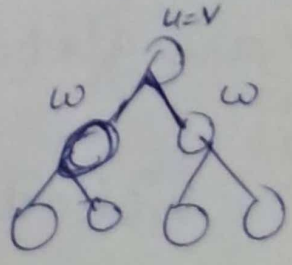
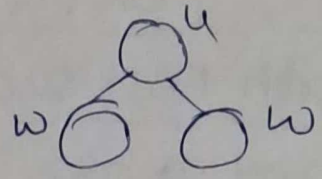
BFS(Breadth First search) :- In BFS start at the vertex V and mark it as visited. The vertex V is said to be unexplored at this time.
- The vertex is said to be explored by an algorithm when all adjacent vertices of that vertex are visited.
- All unvisited vertices adjacent from V are visited next. These are new unexplored vertices
- The newly visited vertices have not been explored and they are put on end of list of unexplored vertices. The first vertex on this list is next to be explored.
- Exploration continues until no unexplored vertex is left.

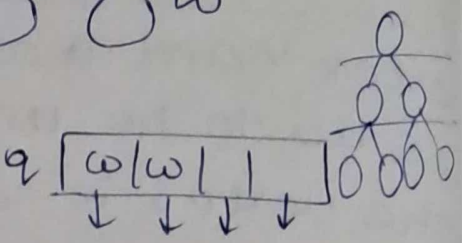
Algorithm BFS(v)
// A BFS of G is carried out beginning at vertex v. For any node i, visited[i]=1 if i has already been visited. The Graph G and array visited[] are global. visted[]=0

```
{   ui:=v;                    u
    visited[v]:=1;            | 1 |
    repeat;
    {
      for all vertices 'w' adjacent from u do
      {
        if(visited[w]=0) then
```



```
        {
            Add w to q;
            visited[w]:= 1;
        }
    }
    if q is empty then return;
    Delete next element u from q;
    }
    until (false);
}
```





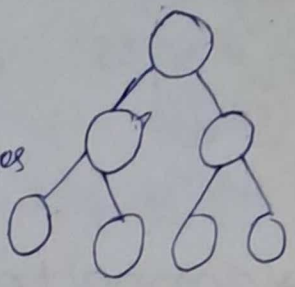
$O(n+E)$

→ DFS (Depth First Search) :-

In Depth first search the ~~expression~~ exploration of a vertex v is suspended as soon as the new vertex is reached. At this time, the exploration of new vertex u begins. When this new vertex has been explored, the exploration of v continues. The search terminates when all reached vertices have been fully explored.

```
Algorithm DFS (v)
// Given an undirected graph G(V, E) with
// n-vertices and an array visited[]
initially set to zero, This algorithm visits
// all vertices reachable from v. G and visited[]
// are global.
{
    visited[v] := 1;
```

for each vertex w adjacent from v do
{ if (visited [w]=0) then DFS(w);
}
}



for both DFS/BFS →
* Time complexity → O(n+E)
E → no of edges
n → no of vertices

Spanning Trees :-

A spanning Tree of Graph G is a sub graph which is basically a tree and it contains all the vertices of G containing no circuit.

Minimum Spanning Tree →

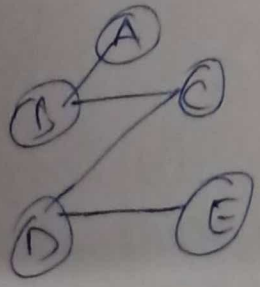
A minimum spanning Tree of a weighted connected Graph G is a spanning tree with minimum (or) smallest weight.

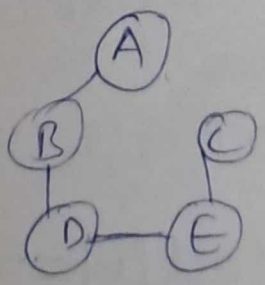
Weight of the Tree :-

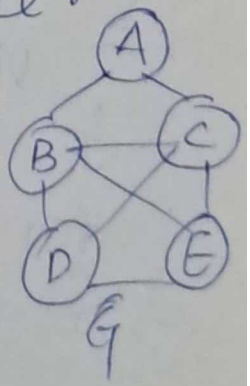
A weight of tree is defined as the sum of weights of all its edges.

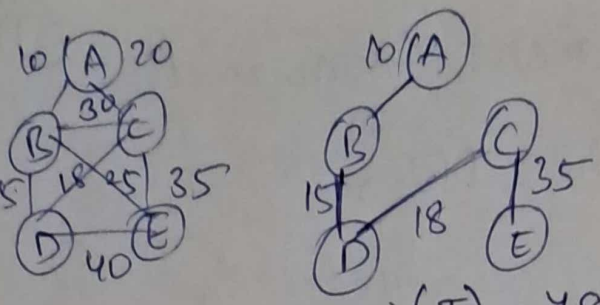
Ex- If a spanning tree obtained by BFS then it is BFS spanning tree.

If a spanning tree is obtained from DFS then it is Depth first spanning tree.







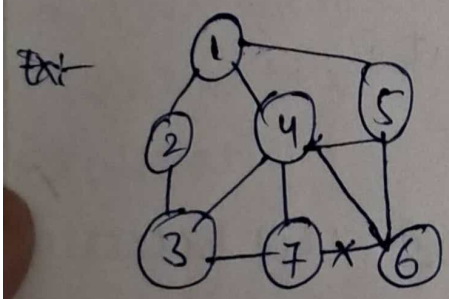


$w(T) = 78$.

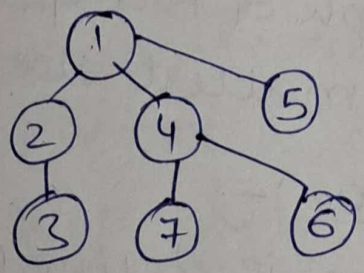
→ connected components :-

A Graph is said to be connected if there exist a path b/w 2 nodes (or) vertices.

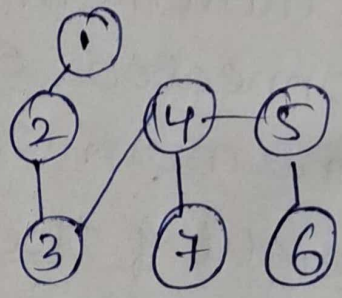
a) If Graph G is connected undirected graph then, can visit all the vertices of graph in first call of DFS (or) BFS. The sub graph which obtained after the traversing using BFS (or) DFS represents the connected component of graph



Graph (G)



Connected component by BFS.



Connected component by DFS.

```
Algorithm components (G, n)
{
    for (i ← 1 to n)
    {
        visited[i] ← 0
    }
    for (i ← 1 to n)
    {
        if (visited[i] == 0)
            DFS(i);
```

output newly visited vertices with adjacent edges

}

}

The time complexity of the above algorithm is $O(n+e)$ (n: no of vertices, e: no of edges).

→ Bi-connected components :-

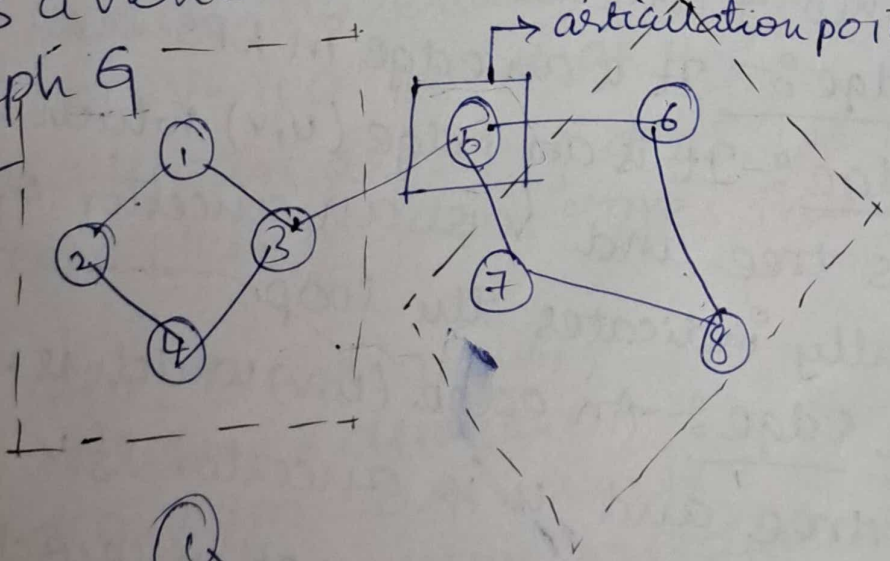
A Graph G is said to be biconnected if it contains no articulation points.

→ Articulation point :-

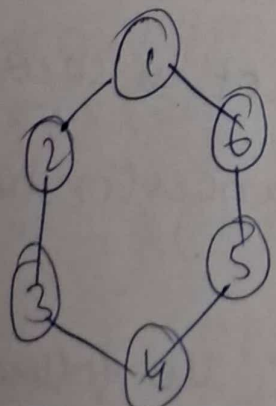
let G=(V, E) be a connected undirected graph then an articulation point of Graph G is a vertex whose removal disconnects Graph G

Ex:

1)



2)



Biconnected.

NOTE:- If there exists any articulation point in the given graph then it is an undesirable feature of the graph

(SAQ)

Identification of articulation point:-

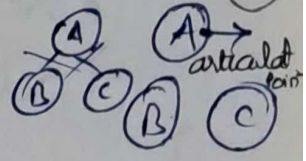
1) The easiest method is to remove a vertex and its corresponding edges one by one from the graph G & test whether the resulting graph is still connected or not.
2) Another method is to use DFS in order to find the articulation point. After performing DFS on a given graph will get DFS tree.
3) While building the DFS tree give the number outside of each vertex. These number indicates the order in which a DFS visits the vertices. These numbers are called DFS numbers [DFN] of the corresponding vertex.

→ While building the DFS tree, the graph is classified into 4 categories

1) Tree edge :- It is an edge in DFS tree.
2) Back edge :- It is an Edge (u,v) which is not in DFS tree and v is an ancestor of u. It is basically indicates the loop.
3) Forward edge :- An edge (u,v) which is not in search tree and u is ancestor of v.
4) Cross edge :- An edge u,v not in search tree and v is neither an ancestor nor a descendent of u.

.To identify articulation points the following observations can be made.

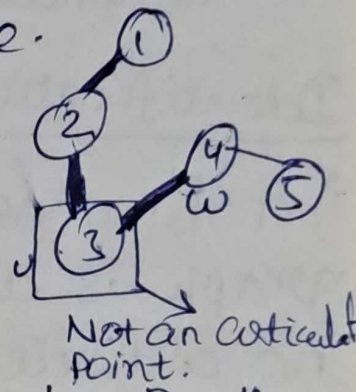
1) The root of the DFS tree is an articulation point if it has 2 (or more children.



2) A leaf node of DFS tree is not an articulation point

3) If u is any internal node then it is not an articulation point if and only if from every child w of u it is possible to reach an ancestor of u using only a part made up of descendents of w and back edge.

[only 1 path should be present from arti point descen to all ancestors]



Algorithm DFS_Art(u, v) [2M, 3M]

// The vertex u - is starting vertex for Depth
// First search traversal
// In DFT V - is a parent of u.
// Initially an array of dfn[ ] is initialized
// to zero [10], The dfn stores the dfs numbers

```
{   dfn[u] = dfn_num;
    Low[u] = dfn_num;
    dfn_num = dfn_num + 1;
    for (each vertex w adjacent to u) do
    {   if (dfn[w] = 0) then
        {   DFS_Art(w, u);
            Low[u] = min(low[u], low[w]);
        }
```

```
        else if(w != u) then
        low[u] = min(low[u], dfn[w]);
    }
}
```

The time complexity of above algorithm is $O(n+e)$ (n: no of nodes, e: no of edges)

Identification of Bi-connected components:-

1) A Biconnected graph G(V, E) is a connected graph is a connected graph which has no articulation points.
2) Biconnected component of Graph G is maximal biconnected subgraph.
3) Two diff biconnected components shouldn't have any common edges.
4) 2 diff biconnected compon can have comma vertex.
5) The common vertex which is attaching 2 (or) more biconnected components must be an articulation point of G.

Algorithm BIConnect(u,v) (2)

```
// The vertex u is a starting vertex for depth first
//traversal.
// In aft V -is a parent if u initially an array
//dfn[] is initialized to 0
{
// put the dfn number for u.
    dfn[u] ← dfn_num;
    Low[u] ← dfn_num;
    dfn_num ← dfn_num+1
    for(each vertex w adjacent to u)do
    {// w_is child of u, if w_is not visited
    if((v!=w)and(dfn[w] < dfn[u])) then
      push(u,w)on to the stack st;
    if(dfn[w]=0)then
    { if(Low[w]>=dfn[u])then
      {
       write("obtained articulation point");
       write("The new bi-connected compo
                                    -nent");
       repeat
       {
         edge(x,y) ← pop edge from the
         top of stack.
         write(x,y);
       } until((x,y)=(u,w)) OR ((x,y)=(w,u));
    }
```

```
    //if w is unvisited then
    Bi-connected(w,u)
    //update(w,u)
    Low[u] ← min(Low[u], Low[w])
    }
    else if(w!=u) then
    Low[u] ← min(Low[u], dfn[w]);
  }
}
```

→ Time complexity :- $O(n+E)$

- **Greedy Method :-**

In this method the decision of the solution is taken based on the information available.

* The Greedy method is a straight forward method. This method is a popular for obtaining the optimized solutions.
* In Greedy technique the solution is constructed through a sequence of steps expanding a partially constructed each solution obtained so far, until a complete solution to the problem is reached.

→ At each step, the choice made should be
- → Feasible : It should satisfy the problem constraints
- → Locally optimal : Among all feasible

solutions the best choice is to be made

→ Irrevocable : Once the particular choice is made then it should not get changed on subsequent steps.

General method (Greedy):- (2M)

Algorithm:

```
Algorithm Greedy(D,n)
//D is a domain, from which the solution
// is to be obtained of size n.
{
    Solution ← 0
    for i←1 to n do
    {
        S ← select(D)
        if(Feasible(solution,S)) then
        Solution ← union(solution,S);
    }
    return solution
}
```

(3)

→ In Greedy method, the following activities are performed.

1) First select some solution from input domain
2) Then check whether the solution is feasible (or) not.
3) From the set of feasible solutions, particular solution that satisfies (or) nearly satisfies the objective of the function such solution is called optimal solution.

4) As Greedy method works in stages at each stage, only one input is considered at each time. Based on this input, it is decided whether particular input gives optimal solution (or) not.

→ Applications of Greedy method:-

Various problems that can be solved using greedy approach

1) Optimal storage ~~and~~ on tapes
2) job sequencing with dead lines
3) Knap sack problem
4) Minimum cost spanning trees.
5) Single source shortest path problem.

→ Optimal storage on tapes:- length Prog.

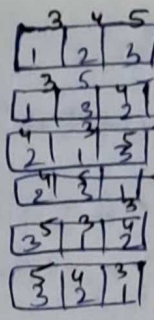
There are n programs that are to be stored on a computer tape of length l. Associated with each program i is a length l_i, $1 \le i \le n$. All programs can be stored on the tape if and only if the sum of the lengths of the programs is atmost l.

2) whenever a program is to be retrieved from this tape, the tape is initially positioned at the front. Hence if the programs are stored in order $I = I_1, I_2, \ldots, I_n$, the time t_j needed to retrieve the program i_j is proportional to $\sum_{1 \le k \le j} l_{ik}$

ex: $n = 1, 2, 3$ [No of pro $= 3$]

$l = (3, 4, 5)$

$\Rightarrow$ No of prog $= (3)! = 6$.



1, 2, 3 = 3 + (3+4) + (3+4+5)

1, 3, 2 = 3 + (3+5) + (3+5+4)

2, 1, 3 = 4 + (4+3) + (4+3+5)

2, 3, 1 = 4 + (4+5) + (4+5+3)

3, 1, 2 = (5) + (5+3) + (5+3+4)

3, 2, 1 = 5 + (5+4) + (5+4+3)

feasible soln

- If all programs are retrieved equally then the expected time (or) the mean retrieval time is $\frac{1}{n}\left(\sum_{1 \le j \le n} t_j\right)$

$\frac{1}{n}$ — no of pro; $\sum_{1 \le j \le n} t_j$ — Total time taken

- In optimal storage ~~and~~ (or) tape problem it is required to find a permutation for the n programs, so that when they are stored on tape in this order the mean retrieval time is minimized. (MRT).

Minimizing the MRT (Mean Retrieval time) is minimizing the

$$d(I) = \sum_{1 \le j \le n} \sum_{1 \le k \le j} l_{ik}$$

Ex (2) $n = 3$, $l = (l_1, l_2, l_3) = (5, 10, 3)$

A:- There are $n! = 3! = 6$ feasible solutions are there.

→ These ordering and respective d values a

ordered I	d(I)	t_j
1, 2, 3	5 + (5+10) + (5+10+3)	38
1, 3, 2	5 + (5+3) + (5+3+10)	31
2, 1, 3	10 + (10+5) + (10+5+3)	43

ordered I	d(I)	tj
2,3,1	10+(10+3)+(10+5+3)	41
3,1,2	3+(3+5)+(3+5+10)	29 (minimal)
3,2,1	3+(3+10)+(3+10+5)	34

5,10,3

∴The optimal soln is 3,1,2

→Algorithm store(n,m)
//n is the number of programs,
//m is the number of tapes

O(n)

```
{
    j:=0; // Next tape to store on
    for i:=1 to n do
    {
        write("append program",i,"to permutation
        for tape",j);
        j:=(j+1) mod m;
    }
}
```

→Time complexity = O(n);

Job sequencing with dead lines: (JPd) Profit deadline

There are a set of n jobs, associated with job i is an integer deadline
di>0 and a profit Pi>0 for any job i
profit Pi is n if and only if the job is completed by its deadline.
2) To complete a job one has to process the job on a machine for one unit of time.

only one machine is available for processing jobs.

→ A feasible solution for this problem is a subset j of jobs such that each job in this subset can be completed by its deadline.

→ The value of feasible solution j is the sum of profits of the jobs in j that is

$$\sum_{i \in j} P_i.$$

→ An optimal solution is a feasible solution with max value.

Ex:- Let $n = 4$, $(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$.

① ② ③ ④

A:- The feasible solutions are

	feasible soln	processing seq	value	
1.	(1, 2)	(2, 1)	110	
2.	(1, 3)	1, 3 (or) 3, 1	115	
3.	(1, 4)	4, 1	127	← (maximal) optimal soln
4.	(2, 3)	2, 3	25	
5.	(3, 4)	4, 3	42	
6.	1	1	100	
7.	2	2	10	
8	3	3	15	
9.	4	4	27	

```
Algorithm Job-seq(D, J, n)
{
// Problem Description : This algorithm is for
//job sequencing Greedy method.
```

```
// D[i] denotes ith deadline, 1 ≤ i ≤ n
// J[i] denotes ith job
// D[J[i]] ≤ D[J[i+1]]

    D[0] ← 0;
    J[0] ← 0;
    J[1] ← 1;
    count ← 1;
    for i ← 2 to n do
    {
      t ← count;
      while (D[J[t]] > D[i]) AND D[J[t]] != t) do
            t ← t-1;
      if ((D[J[t]] ≤ D[i]) AND (D[i] > t)) then
      { // insertion of ith feasible seq into J array
        for s ← count to (t+1) step -1 do
        J[s+1] ← J[s];
        J[t+1] ← i;
        count ← count+1;
      } // end if
    } // end for
    return count;
}
```

$O(n^2)$

→ The sequence of J will be inserted if and only if D[J[t]] ≠ t
i.e the job J is processed

if it is within the deadline

→ The time complexity of this algor = $O(n^2)$

• Knapsack problem :- (maximization) weight(w_i) Profit (P_i)

The knapsack problem can be stated as If there are n objects, $i = 1, 2, 3 \ldots n$ each obj i has some positive weight w_i, and some profit value is associated with each obj which is denoted as P_i. This knapsack carry out the maximum weight W.

→ while solving the knapsack problem, capacity is the constraint. Using Greedy method,

1) choose only those objects that give maximum profit.

2) The total weight of selected objects should be $\leq W$

maximized $\sum_{1}^{n} P_i X_i$ subject to $\sum_{1}^{n} w_i x_i < W$.

where, the knapsack can carry the fraction x_i of an object i, such that $0 \leq x_i \leq 1$, & $1 \leq i \leq n$.

Ex:- consider that there are 3 items weights and profit value of each item is P/ feasible solutions are

i	w_i	P_i
1	18	30
2	15	21
3	10	18

↓ Jobs

x_1	x_2	x_3	→ obj
$1/2$	$1/3$	$1/4$	
1	$2/15$	0	
0	$2/3$	1	
0	1	$1/2$	

Also $W = 20$

$\sum w_i x_i$ = ① $18 \times \frac{1}{2} + 15 \times \frac{1}{3} + 10 \times \frac{1}{4}$

$9 + 5 + 2.5 = 16.5$

② $18 \times 1 + 15 \times \frac{2}{15} + 10 \times 0$

$18 + 2 = 20.$

③ $18 \times 0 + 15 \times \frac{2}{3} + 10 \times 1$

$10 + 10 = 20$

④ $18 \times 0 + 15 \times 1 + 10 \times \frac{1}{2}$

$15 + 5 = 20$

$\sum P_i x_i$ ① $30 \times \frac{1}{2} + 21 \times \frac{1}{3.4} + 18 \times \frac{1}{4} = 15 + 7 + 4.5 = 20.5$

② $30 \times 1 + 21 \times \frac{2}{15} + 18 \times 0 = 30 + 2.8 = 32.8$

③ $30 \times 0 + 21 \times \frac{2}{3} + 18 \times 1 = 14 + 18 = 32$

④ $30 \times 0 + 21 \times 1 + 18 \times \frac{1}{2} = 21 + 9 = 30$

→ ↘	$\sum w_i x_i$	$\sum P_i x_i$
1	16.5	20.5
2	20	32.8 → optimal soln
3	20	32
4	20	30

O(n)

Algorithm KnapSack_Greedy(w,n)

{

// P[i] contains the profits of 'i' items

//such that $1 \le i \le n$, w[i] contains weights

//of 'i' items.

//x[i] is solution vector

// w is the max size of knap sack

for i=1 to n do

{

```
if (w[i] < w) then
{ x[i] := 1.0;
   w = w - w[i];
}
}
if (1 <= n) then x[i] := w/w[i] ;
}
```

→ Time complexity = O(n);

② $w_1 = 18$, $w_2 = 15$, $w_3 = 10$

$W = 20$

feasible solutions:-

		x_1	x_2	x_3
w_1, w_2, w_3	1	1	2/15	0
w_1, w_3, w_2	2	1	0	2/10 = 1/5
w_2, w_1, w_3	3	5/18	1	0
w_2, w_3, w_1	4	0	1	5/10 = 1/2
w_3, w_1, w_2	5	10/18 = 5/9	0	1
w_3, w_2, w_1	6 ✓	0	10/15 = 2/3	1

$w_1 = \frac{18}{18} = 1 \quad w_2 = \frac{20-18}{15} = \frac{2}{15} \quad w_3 = 0.$

$w_1 = \frac{18}{18} = 1 \quad w_2 = 0 \quad w_3 = \frac{20-18}{10} = \frac{2}{10} = \frac{1}{5}$

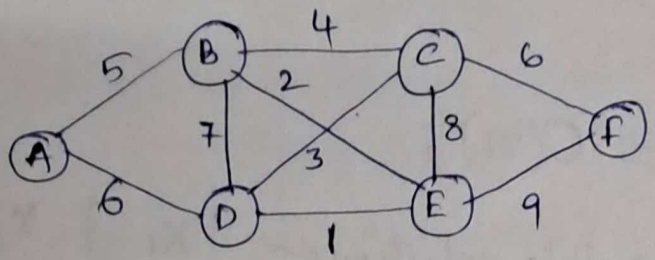
→ Minimum cost spanning trees:-

Minimum cost spanning tree of weighted connected graph G is a spanning tree with min. (or) smallest cost.

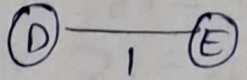
weight

Prim's algorithm:-

Select an edge with minimum weight and proceeds by selecting adjacent edges with minimum weight. care should be taken for not forming a circuit (or cycle.

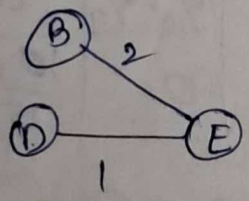


Step 1: minimum wt (D, E) = 1



Step 2: Adjacent edges for (D, E) are (D, B) = 7
(D, C) = 3
(E, B) = 2 ✓
(E, C) = 8
(E, F) = 9

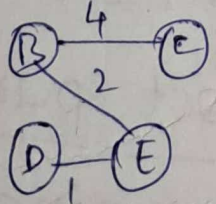
Minimum cot (E, B) = 2.



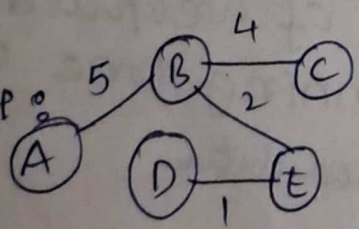
Step 3: Adjacent edges for D, E, B

(D, B) = 7 (E, C) = 8 (B, C) = 4 (E, F) = 9 (B, C) = 4

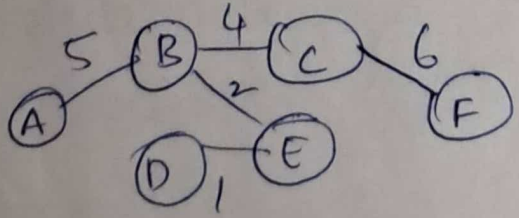
(B, C) = 4 ✓



Step 4:



Step 5:



Total Minimum cost of spanning tree = 5+2+4+6
= 18

```
Algorithm Prim (G[0....size-1, 0..... size-1], nodes)
//problem Description: This algorithm for prim's
//algorithm for (finding spanning tree.
//Input: Weighted Graph G.
//Output: Spanning tree gets printed.
total ← 0;
for i ← 0 to nodes - 1 do
  //tree[i] ← 0;
    tree[i] ← 0;
    tree[0] ← 1;
for k ← 1 to nodes do
{ min_dist ← ∞
 for i ← 0 to nodes-1 do
 { for j ← 0 to nodes-1 do
    { if (G[i,j] AND ((tree[i] AND ! tree[j]) OR
      (!tree[i] AND tree[j])) then
      { if (G[i,j] < min_dist) then
        { min_dist ← G[i];
          v1 ← i;
          v2 ← j;
        }
      }
    }
 }
write (v1, v2, min_dist);
 tree[v1] ← tree[v2] ← 1
 total ← total + min_dist;
} write ("total Path length is = ", total);
```

→ Kruskal Algorithm :-

```
Algorithm kruskal(E, cost, n, t)
//E is the set of edges in G.
//G- has n_vertices cost[u,v] is the cost of edge (u,v).
//t is the set of edges in minimum cost tree
{
  construct a heap out if edge costs using heapify
  for i:=1 to n do parent[i]:=-1;
  i:=0; minimum:=0.0
  while((i<n-1) and (heap not empty)) do
  {
    Delete a minimum cost edge (u,v) from
    heap and reheapify using adjust:
    j:=Find(u); k:=Find(v);
    if (j ≠ k) then
    { i:=i+1;
      t[i,1]:=u; t[i,2]:=v;
      mincost:=mincost+cost(u,v);
      Union(j,k);
    }
  }
  if (i≠n-1) then write ("NO spanning tree");
  else return mincost;
}
```

no of edges in spanning tree

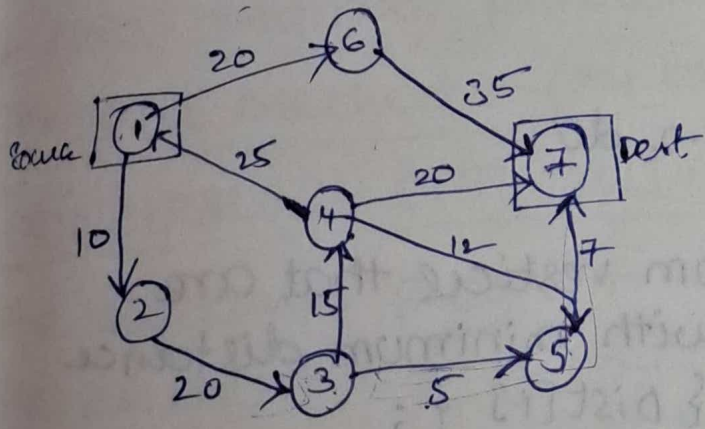
forsam ex:-
1
2
3
4
5
6
7
8
9

→ Single source shortest path algorithm :-

The length of the path is defined to be the sum of the edges on that path. The starting vertex of the path is referred to as the source & the last vertex is destination.

→ The graphs are digraphs to allow for one-way streets.

• In the problem, consider a directed graph G=(V,E, a weighing function cost for the edges of G & a source vertex V_0. The problem is to determine all the shortest paths from V_0 to all the remaining vertices of G.



Source vertex = 1
Destination vert = 7
S[1] = 1

[1,2] = 10 ✓
[1,3] = ∞
[1,4] = ∞
[1,5] = ∞
[1,6] = 20
[1,7] = ∞

[1,2,3] = 10+20 = 30 ✓
[1,2,4] = ∞
[1,2,5] = ∞
[1,2,6] = ∞
[1,2,7] = ∞

[1,2,3,5] = 35
[1,2,3,4] = 45
[1,2,3,6] = ∞
[1,2,3,7] = ∞

[1,2,3,5,7] = 35+7 = 42 ✓

[1,2,3,4,7] = 45+20 = 65

[1,6,7] = 55

∴ 1-2-3-5-7 = 42 is single source shortest path.

```
Algorithm Single-short_path (P, Cost, Dist, n)
{
// cost is an adjacency matrix string cost if each
// edge is i.e cost[1:0, 1:n]
// Dist is a set that stores the shortest path
//from the source vertex 'P' to any other vertex.
//'S' stores all visited vertices.
 for i←1 to n do
 {
    S[i]←0;
    Dist ←Cost[P,i];
 }
    S[P]←1;
    Dist[P]←0.0;
    for val ← 2 to n-2 do
    {
      choose 'q' from vertices that are
      not visited with minimum distance
      Dist[q] = min { Dist[i] };
      S[q]←1;
      for(all node 'v' adjacent to 'q' with
      S[v]= 0) do
      if (Dist[v] > Dist [q] + cost [P,q])) then
      Dist [r] ← Dist [q] + cost[P,q];
    }
}
```

Time Complexity = $O(n^2)$;