UNIT 4

GREEDY METHOD.

*Greedy method:

*Comparison of divide and conquer and greedy method:

→ Divide and conquer is applicable for the problems which can be decomposible into sub problems. where as greedy method is applicable for the problems which can't be decomposible.

→ All of these problems have 'n' inputs and require is to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called as a feasible solution.

→ We need to find a feasible solution that either maximize and minimize objective function.

→ A feasible condition or solution that does this called an optimal solution

Ex: Function $Z = 4x+3y$

Subject to the constraints $-1 \le x \le 1$

$-1 \le y \le 1$

Find maximum value of z (objective function)

input set (x, y)	$-1 \le x \le 1$	$-1 \le y \le 1$
	$-1, 0, 1$	$-1, 0, 1$

$(-1, -1)$ $z = 4x-1 + 3x-1 = -7$

$(-1, 0)$ $z = 4x-1 + 3x0 = -4$

$(-1, 1)$ $z = 4x-1 + 3x1 = -1$

$(0, -1)$ $z = 4x0 + 3x-1 = -3$

$(0, 0)$ $z = 4x0 + 3x0 = 0$

$(0, 1)$ $z = 4x0 + 3x1 = 3$

$(1, -1)$ $z = 4x1 + 3x-1 = 1$

$(1, 0)$ $z = 4x1 + 3x0 = 4$

$(1, 1)$ $z = 4x1 + 3x1 = (7)$

Feasible solution

Optimal solution.

→ Optimal solution is $z = 7$ for the input values $(x, y) = (1, 1)$.

→ In greedy method, an algorithm works in stages, considering one input at a time.

→ At each stage, a decision is made regarding whether a particular ~~line~~ input is ~~in~~ an optimal solution.

→ This can be done by considering inputs in an order determined by some selection procedure.

*Control abstraction / General method of greedy method:

```
Algorithm Greedy (a,n)
// a[1...n] contains n inputs
{
 Solution =0;
 for:=1 to n do
  x:= Select(a);
  if Feasible (Solution, x) then
     Solution:= Union (Solution,x);
 }
 return Solution;
 }
```

→ In this control abstraction, the function select() selects an input from array a[] and removes it. The selected input value is assigned to x.

→ Feasible is a boolean valued function that determines whether input value x satisfying the given constraint or not. If the input 'x' is not satisfying the given constraint then eliminate from the input set. If the input 'x' is satisfying the given constraint then find feasible solution.

→ The function union combines 'x' with the solution and updates the objective function.

* Applications:

* 0/1 - Knapsack problem:

Ex: $n = 3$

$m = 20$

$(P_1, P_2, P_3) = (25, 24, 15)$

$(w_1, w_2, w_3) = (18, 15, 10)$

→ In knapsack problem we have 'n' items and a knapsack (empty bag). Each item has a weight (w_i) and profit (P_i) and knapsack has a capacity of 'm' units.

→ The objective of the problem is to obtain a filling of the knapsack that maximize the total profit.

→ ~~Beside~~ Decide which item are to be placed in the bag so that we get maximum profit.

→ Generally, the problem can be stated as

maximize profit $\sum_{1 \le i \le n} P_i x_i$

Subject to constraint $\sum_{1 \le i \le n} w_i x_i$

and $0 \le x_i \le 1$, $1 \le i \le n$.

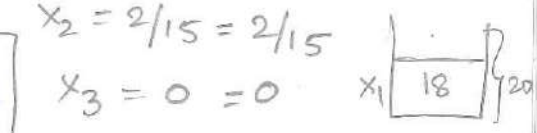
→ Consider the following instance of knapsack

problem. $n=3, m=20$, $(P_1, P_2, P_3) = (25, 24, 15)$, $(w_1, w_2, w_3) = (18, 15, 10)$. Find optimal solution.

Sol:- Case(i): Maximum profit

Select the item which has maximum profit and place it in the bag.

$x_1 = 18/18 = 1$
$x_2 = 2/15 = 2/15$
$x_3 = 0 = 0$



$$\sum_{1 \le i \le n} P_i x_i = P_1 x_1 + P_2 x_2 + P_3 x_3$$

$$= 25 \times 1 + 24 \times 2/15 + 15 \times 0$$

$$= 25 + 3.2 + 0$$

profit = 28.2.

→ Now a bag consists of two units of space and so second item can be placed partially

$x_2 = 2/15 = 2/15$ $[(x_1, x_2, x_3) = (1, 2/15, 0)]$

→ Now the bag is full so x_3 item cannot be placed in the bag

→ Profit = 28.2

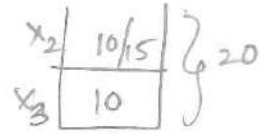
Case(ii): Minimum ~~profit~~ weight

→ Place the items which has minimum weight into the bag. First place item x_3 into the bag completely.

$x_3 = 10/10 = 1$.

→ Now bag consists of 10 units of space so

item x_2 is placed partially.



$x_2 = 10/15 = 2/3$

→ Now bag is full, item x_1 cannot be placed in bag.

x_1 = Not placed = 0 $\quad (x_1, x_2, x_3) = (0, 2/3, 1)$

$$\sum P_i x_i = P_1x_1 + P_2x_2 + P_3x_3$$

$$= 25 \times 0 + 24 \times 2/3 + 15 \times 1$$

profit = 0 + 16 + 15 = 31

case(iii): Maximum profit per unit weight.

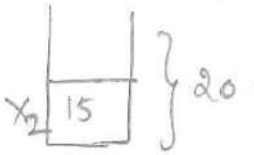
$$x_1 = \frac{P_1}{w_1} = \frac{25}{18} = 1.4$$

$$x_2 = \frac{P_2}{w_2} = \frac{24}{15} = 1.6$$

$$x_3 = \frac{P_3}{w_3} = \frac{15}{10} = 1.5$$

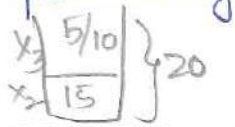
→ place item x_2 in the bag because it has maximum profit per unit weight.

$$x_2 = \frac{15}{15} = 1$$



→ Now bag consists of 5 units of space so item x_3 is placed partially

$$x_3 = 5/10 = 1/2$$



→ Now bag is full, item x_1 cannot be placed in bag.

$x_1 = 0$.

$(x_1, x_2, x_3) = (0, 1, 1/2)$

$\Sigma P_i x_i = P_1x_1 + P_2x_2 + P_3x_3$

$= 25 \times 0 + 24 \times 1 + 15 \times 1/2$

$= 0 + 24 + 7.5$

profit $= 31.5$

∴ Optimal solution is $31.5 = \Sigma P_i x_i$ for the input values of $(x_1, x_2, x_3) = (0, 1, 1/2)$

2) construct the following instance of 0/1 knapsack problem $n=4$, $m=15$

$(P_1, P_2, P_3, P_4) = (18, 16, 4, 9)$

$(w_1, w_2, w_3, w_4) = (6, 4, 5, 10)$. Find maximum profit.

Sol:- case(i): Maximum profit.

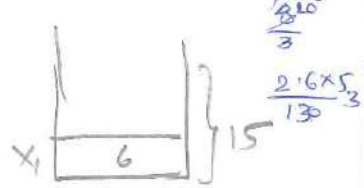
$x_1 = 10/10 = 1$ → Place an item which has maximum profit in the bag (knapsack)

$x_2 = 5/6$

$x_3 = 0$

$x_4 = 0$ → place first item x_1

$x_1 = 6/6 = 1$



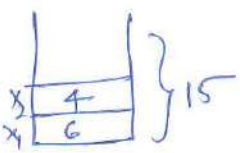
$\Sigma P_i x_i = 18 \times 1 + 16 \times 5/6 + 0 + 0$

$= 18 + 13 = 31$

x_1 is placed completely.

→ place x_2 item $x_2 = 4/4 = 1$

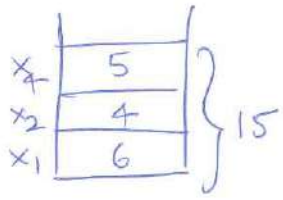
x_2 is placed completely.



x_2 is placed completely.

→ Now bag consists of 5 units of space x_4 item is placed partially.

$x_4 = 5/10 = 1/2$



→ Now knapsack is full so x_3 item cannot be inserted.

$$\text{Profit} = \sum_{i \le 1} P_i x_i = P_1x_1 + P_2x_2 + P_3x_3 + P_4x_4$$

$$= 18 \times 1 + 16 \times 1 + 4 \times 0 + 9 \times 1/2$$

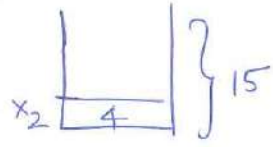
$$= 38.5$$

case(ii): Minimum weight.

→ place an item which has minimum weight into bag.

→ place x_2 item

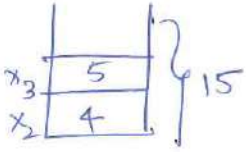
$x_2 = \frac{4}{4} = 1$



x_2 is placed completely.

→ place x_3 item

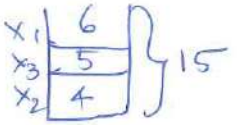
$x_3 = \frac{5}{5} = 1$



x_3 is placed completely.

→ Now bag consits of 6 units of space x_1 item is placed completely

$x_1 = \frac{6}{6} = 1$



→ Now bag is full so x_4 item cannot be inserted

$$\text{profit} = \sum_{i \le 1} p_i x_i = 18 \times 1 + 16 \times 1 + 4 \times 1 + 9 \times 0$$
$$= 38.$$

Case(iii): Maximum profit per unit weight.

→ place an item which has maximum profit per unit weight.

→ ~~place~~ $x_1 = \frac{p_1}{w_1} = \frac{18}{6} = 3$

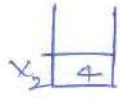
$$x_2 = \frac{p_2}{w_2} = \frac{16}{4} = 4$$

$$x_3 = \frac{p_3}{w_3} = \frac{4}{5} = 0.8$$

$$x_4 = \frac{p_4}{w_4} = \frac{9}{10} = 0.9$$

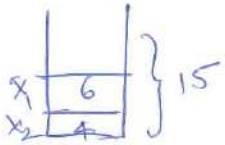
→ place an item x_2. x_2 is placed completely.

$$x_2 = \frac{4}{4} = 1$$



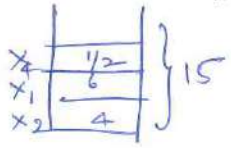
→ place an item x_1. x_1 is placed completely

$$x_1 = \frac{6}{6} = 1$$



→ place an x_4 item is placed partially completely

$$x_4 = \frac{5}{10} = 1/2$$



→ Now bag is full so x_3 cannot be inserted.

$$x_3 = 0.$$

$$\sum p_i x_i = 18 + 16 + 0 + 4.5$$
$$= 38.5$$

Optimal solution = 38.5 for input values

$(x_1\ x_2\ x_3\ x_4) = (1, 1, 0, 1/2) = 38.5$.

Feasible solution

S/No	Input $(x_1\ x_2\ x_3\ x_4)$	feasible solution
1	(1, 1, 0, 1/2)	38.5
2	(1, 1, 1, 0)	38
3	(1, 1, 0, 1/2)	38.5

optimal solution

* Pseudo code for 0/1 knapsack problem:

```
Greedy-knapsack (M,n)
// w[1:n] - Array of weights
// p[1:n] - Array of profits
// M - capacity of knapsack
// x[1:n] is solution
{
for i:=1 to n do
 x[i]=0
 cu:=m;
for i:=1 to n do
if (w[i] >cu) then
break;
x[i]:=1;
cu = cu - w[i];
}
```

```
if(i<=n) then
x[i] = cu/w[i];
}
```

→ Time complexity for knapsack problem is O(n).

* Job sequencing with deadlines:

→ Job sequencing deals with a set of 'N' jobs, associated with each job 'i' is an integer deadline $d_i \geq 0$ and a profit $P_i \geq 0$. For any job 'i' the profit 'P_i' is earned if it's job is completed by its deadline. 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 a feasible solution 'J' is the sum of profits of the job in 'J'. An optimal solution is the feasible solution with maximum value.

Ex: Let $N=4$, $(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$. Solve the problem of job sequencing with deadlines using greedy method.

Sol:- Since the maximum deadline is 2 units, maximum of two jobs will form the feasible solution.

S.No	Solution	Sequence	Profit
1.	{1}	(1)	100
2.	{1,2}	(2,1)	100+10=110
3.	{1,3}	(1,3),(3,1)	100+15=115
4.	{1,4}	(4,1)	100+27=127 optimal solution
5.	{2,3}	(2,3)	10+15=25
6.	{2,4}	(2,4),(4,2)	10+27=37
7.	{3,4}	(4,3)	15+27=42
8.	{2}	(2)	10
9.	{3}	(3)	15
10.	{4}	(4)	27

→ Serial no - 4 or solution 4 is optimal solution. In this solution only jobs 1 and 4 are processed and the profit is 127. These jobs must be processed in the order, job 4 followed by job 1. Thus, the processing of job 4 begins at time zero and that of job 1 is completed at time two.

Ex: 2) $N = 5$, $(P_1, P_2, P_3, P_4, P_5) = (20, 15, 10, 5, 1)$,

$(d_1, d_2, d_3, d_4, d_5) = (2, 2, 1, 3, 3)$.

Sol:-

S.No	feasible solution	processing sequence	profit	
1.	{1}	(1)	20	
2.	{1,2}	(1,2),(2,1)	35	
3.	{1,3}	(3,1)	30	
4	{1,4}	(1,4)	25	
5	{1,5}	(1,5)	21	
6	{2,3}	(3,2)	25	
7.	{2,4}	(2,4)	20	
8	{2,5}	(2,5)	16	
9	{3,4}	(3,4)	15	
10	{3,5}	(3,5)	11	
11.	{4,5}	(4,5)(5,4)	6	
12	{2}	(2)	15	
13	{3}	(3)	10	
14	{4}	(4)	5	
15	{5}	(5)	1	
16	{1,2,3}	(3,2,1)	45	optimal solution
17	{1,2,4}	(1,2,4)	40	

18	{1,2,5}	(1,2,5)	36
19	{1,3,4}	(3,1,4)	35
20	{1,3,5}	(3,1,5)	31
21	{1,4,5}	(1,4,5)	26
22	{2,3,4}	(3,2,4)	30
23	{2,3,5}	(3,2,5)	26
24	{2,4,5}	(2,4,5)	21
25	{3,4,5}	(5,4,3)	16.

3) $N=4$, $(P_1, P_2, P_3, P_4) = (3, 5, 20, 18)$, $(d_1, d_2, d_3, d_4) = (1, 3, 2, 3)$

Sol:- According to greedy method, arrange the jobs in decreasing order of their profits

$(P_3, P_4, P_2, P_1) = (20, 18, 5, 3)$

$(d_3, d_4, d_2, d_1) = (2, 3, 3, 1)$

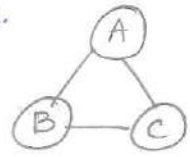
S.No	feasible solution	processing sequence	profit	
1.	{3}	(3)	20	
2.	{3,4}	(4,3)	38	
3.	{3,4,2}	(3,4,2),(3,2,4)	43	maximum profit is optimal solution
4.	{3,4,1}	(1,3,4)	41	
5.	{4,2,1}			

* Spanning tree:

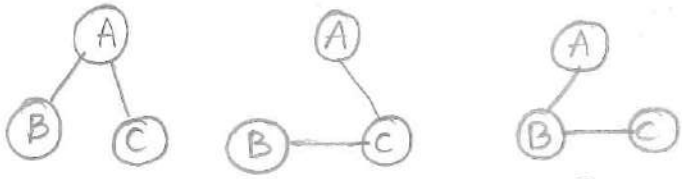
Spanning tree is a sub graph of a given graph G. It satisfies the following properties.

1. It contains all vertices of a graph.
2. It does not contains cycles or loops.
3. It contains (n-1) edges where n is number of vertices.

Ex: 1.



Spanning trees of given graph



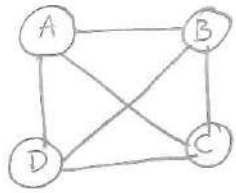
(a) (b) (c)

→ we found ~~formed~~ three spanning trees of given complete graph. A complete undirected graph can have maximum "n^{n-2}" number of spanning trees where n is number of vertices.

→ In above example,

$n^{n-2} = 3^{3-2} = 3$ ~~are~~ spanning trees are possible.

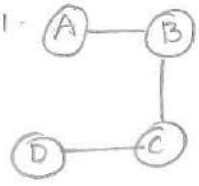
Ex: 2.

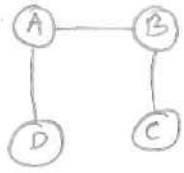


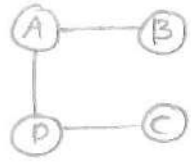
$n = 4$

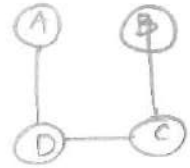
$n^{n-2} = 4^{4-2} = 4^2 = 16$

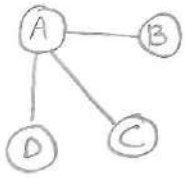
Maximum no-of spanning trees are 16.

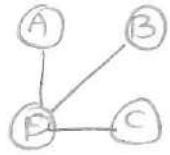
1. 

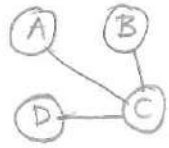
2. 

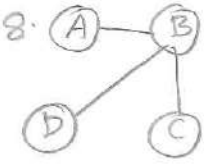
3. 

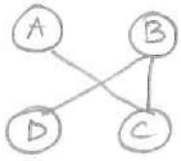
4. 

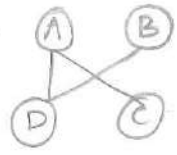
5. 

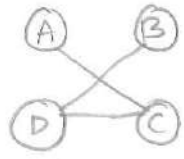
6. 

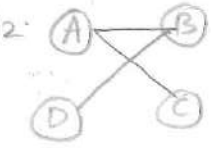
7. 

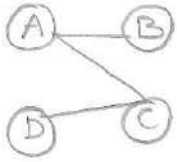
8. 

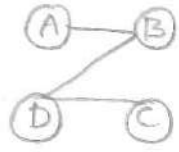
9. 

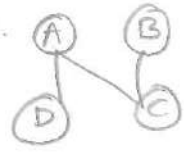
10. 

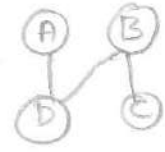
11. 

12. 

13. 

14. 

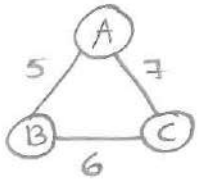
15. 

16. 

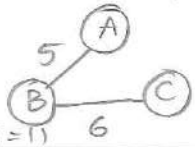
* Minimum cost spanning tree:

→ Spanning tree of graph with minimum cost is known as minimum cost spanning tree.

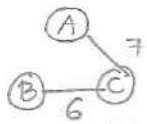
Ex:



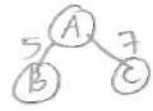
Spanning trees of the given graph



cost = 5+6 = 11



cost = 6+7 = 13



cost = 5+7 = 12

minimum cost spanning tree is 5+6 = 11

* Minimum cost spanning tree applications:

→ Minimum cost spanning tree is used in design of networks. including computer networks, telecommuni -cation networks, transportation networks, water supply network and electrical grides.

→ Constructing trees for broadcasting in computer networks.

→ Handwriting recognition of mathematical expression.

→ Image registration and segmentation.

→ Minimum cost spanning tree can be used to solve the travelling salesperson problem.

→ There are two algorithms to find minimum cost spanning tree.

1. Prims algorithm
2. krushkal's algorithm.

1. Prims algorithm:

→ The prims algorithm starts with a tree that includes only a minimum cost edge of graph 'G'.

→ Then edges are added to this tree one by one.

→ Add an edge (i,j) with minimum cost to the tree.

→ The next edge (i,j) to be added in such a way that 'i' is a vertex already included in tree, 'j' is a vertex not yet included and the cost of (i,j) is minimum among all edges (k,l) such that vertex 'k' is in the tree and vertex 'l' is not in the tree.

→ To determine this edge (i,j), we associate with each vertex 'j' not yet included in the tree. A value ~~near of j~~ near[j] is a vertex in the tree such that cost[j, near[j]] is minimum among all choices for near[j].

→ we define near[j] = 0 for all vertices 'j' that are already in the tree.

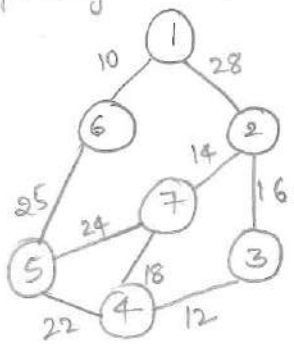
→ The next edge to include is defined by vertex 'j' such that near[j] ≠ 0 and cost[j, near[j]] is minimum.

* Prims algorithm

```
Algorithm prim (E, cost, n, t)
// E is set of edges in G
// cost [1:n, 1:n] is the cost adjacency matrix
// ∞ - if no edge (i,j) exists
// t [1:n; 1:2] - min cost spanning tree
```

```
{
mincost = 0;
for i = 2 to n do
  near[i] := 1;
near[1] = 0;
for i = 1 to n-1 do
{
let j be an index such that near[j] ≠ 0 and
cost[j, near[j]) is minimum;
t[i,1] = j;
t[i,2] = near[j];
mincost = mincost + cost[j, near[j]];
near[j] = 0;
for k := 1 to n do
if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k,j]))
then
near[k] = j;
}
return mincost;
}
```

Spanning tree ex:

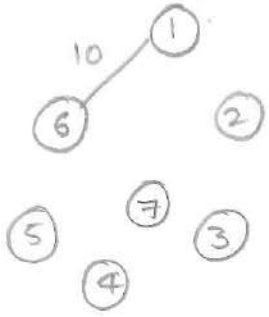


Cost Adjacent matrix

	1	2	3	4	5	6	7
1	∞	28	∞	∞	∞	10	∞
2	28	∞	16	∞	∞	∞	14
3	∞	16	∞	12	∞	∞	∞
4	∞	∞	12	∞	22	∞	18
5	∞	∞	∞	22	∞	25	24
6	10	∞	∞	∞	25	∞	∞
7	∞	14	∞	18	24	∞	∞

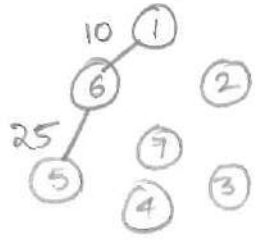
Step-1:

→ Start with a source vertex '1' find nearest vertex of '1' (i.e) vertex '6' because cost of (1,6) is 10(i.e) minimum. Now add an edge 1,6 to the minimum cost spanning tree



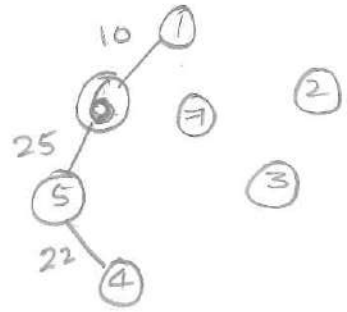
Step-2:

→ Find out nearest vertex of vertices 1,6. Nearest vertex of 1 is 2 and cost is 28. Nearest vertex of 6 is 5 and cost is 25. Select vertex 5 and add to minimum cost spanning tree.



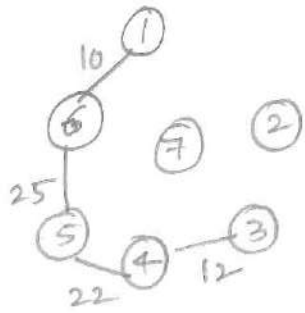
Step-3:

Find nearest vertex of vertices 1, 6, 5. Nearest vertex of 5 is 4 and cost is 22. Add this vertex to minimum cost spanning tree.



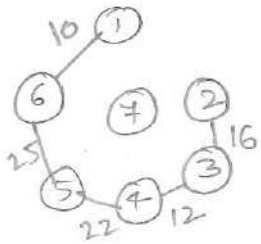
Step-4:

Find nearest vertex of vertices 1, 6, 5, 4. Nearest vertex of 4 is 3 and cost is 12. Add this vertex to minimum cost spanning tree.



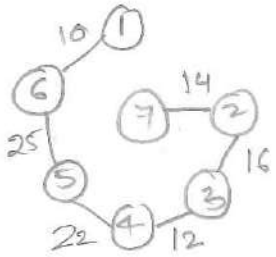
Step-5:

Find nearest vertex of vertices 1, 6, 5, 4, 3. Nearest vertex of 3 is 2 and cost is 16. Add this vertex to minimum cost spanning tree.



Step-6:

Find nearest vertex of vertices is 1, 6, 5, 4, 3, 2. Nearest vertex of 2 is 7 and cost is 14. Add this vertex to minimum cost spanning tree.



Minimum cost spanning tree is $10+25+22+12+16+14 = 99$

2. Krushkal's algorithm:

→ Kruskal's algorithm is another method to find minimum cost spanning tree. It follows the following steps to find minimum cost spanning tree

Step-1:

Arrange the edges in increasing order of cost.

Step-2:

Select an edge with minimum cost and add to the minimum cost spanning tree

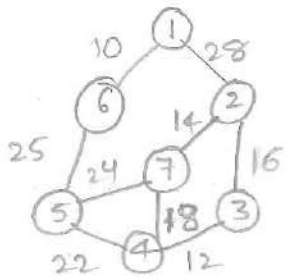
Step-3:

If an edge forms a cycle then eliminate that edge from the spanning tree of minimum cost.

Step-4:

Repeat Step 2 and Step 3 until all vertices are placed in minimum cost spanning tree.

Ex:



Step-1:

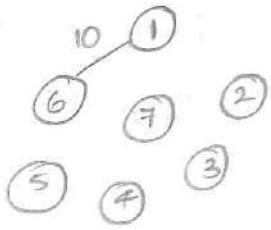
Arrange the edges in increasing order of cost

edges	cost
(1,6)	10
(3,4)	12
(2,7)	14
(2,3)	16
(4,7)	18
(4,5)	22
(5,7)	24
(5,6)	25
(1,2)	28

Step-2:

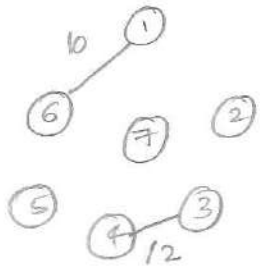
Select an edge with minimum cost (i.e) (1,6) edge.

add this edge to minimum cost spanning-tree



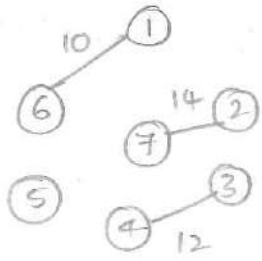
Step-3:

→ Select next minimum cost edge (i.e) (3,4) edge. and insert into MCSP



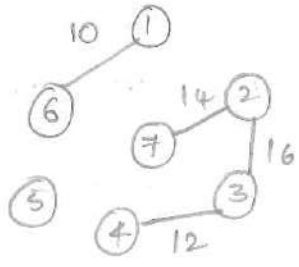
Step-4:

→ Select next minimum cost edge (2,7) add to MCSP



Step-5:

→ Select next minimum cost edge (2,3) add to MCSP

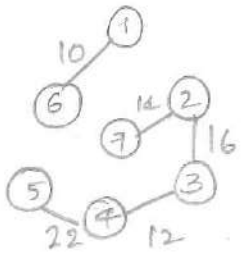


Step-6:

→ Select next minimum cost edge (4,7). This edge will form a cycle when it is placed in MC spanning tree so eliminate this edge.

Step-7:

Select next minimum cost edge (4,5) add to MCST.

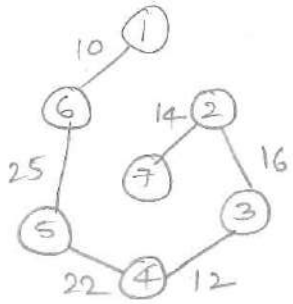


Step-8:

Select next minimum cost edge (5,7) This edge will form a cycle so it is eliminated.

Step-9:

Select next minimum cost edge (5,6) add to MCST.



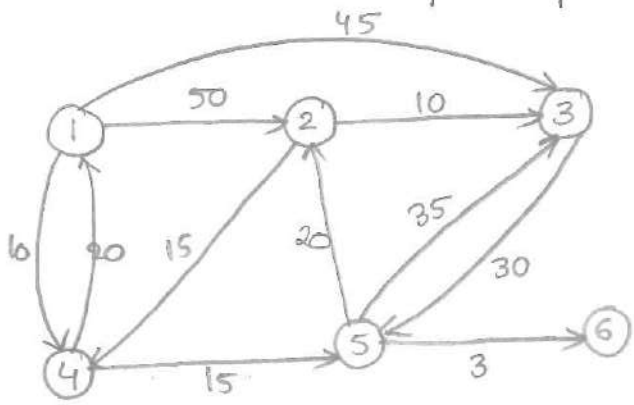
Step-10:

Select next minimum cost edge (1,2). This edge will form a cycle so it is eliminated.

⇒ The cost of minimum spanning tree is

$10+14+16+12+22+25=99$.

* Single source shortest paths problem:



→ Single source shortest paths problem is determined the shortest path from source vertex to all remaining vertices of graph 'G'.

S.No	Paths	Length	Shortest path	Length.
1.	1-2	50	1-4-5-2	45
	1-4-5-2	10+15+20=45		
	1-3-5-2	45+30+20=95		
2.	1-3	45	1-3	45
	1-2-3	60		
	1-4-5-3	60		
	1-4-5-2-3	55		
3.	1-4	10	1-4	10
	1-2-4	65		
	1-2-3-5-2-4	125		
4.	1-4-5	25	1-4-5	25
	1-2-4-5	80		
	1-2-3-5	90		
	1-3-5	75		
5.	1-4-5-6	28	1-4-5-6	28.
	1-2-3-5-6	93		
	1-3-5-6	78		
	1-2-4-5-6	83		

→ Shortest path from source vertex 1 to vertices 2, 3, 4, 5, 6.

Shortest path	length
1-4-5-2	45
1-3	45
1-4	10
1-4-5	25
1-4-5-6	28

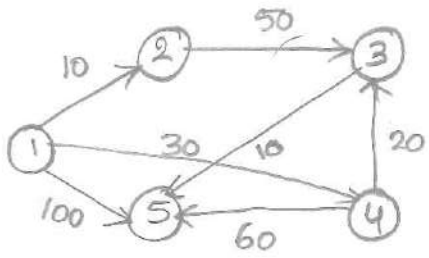
→ we can find solution to single source shortest path problem using "Dijkstra's algorithm".

```
Algorithm shortest paths (v, cost, dist, n)
//cost [1:n, 1:n] cost adjacency matrix
//dist [j], 1≤j≤n – length of shortest path from
// vertex v to vertex j.
{
for i=1 to n do
{
 s[i]=0;
 dist[i] = cost [v,i];
 }
 s[v]=1;
 dist [i]: cost [v,i];
 for num = 2 to n-1 do
```

```
{
//determine n-1 paths from v
choose u from among those vertices not in
's' such that dist[u] is minimum.
for(each w adjacent to u with s[w]=0) do
if(dist[w]>dist[u]+cost([u,w])) then
dist[w]=dist[u]+cost[u,w];
}
}
```

Algorithm:

```
{
S={1};
for(i=2;i<=n;i++)
D[i]=c[1,i];
for(i=1; i<=n-1;i++)
{
choose a vertex w∈V-S such that
D[w] is minimum
S=S∪{w};
for each vertex v∈V-S
} D[v]=min(D[v], D[w]+c[w,v])
}
```



Initially:

$S = \{1\}$

$D[2] = C[1,2] = 10$

$D[3] = C[1,3] = \infty$

$D[4] = C[1,4] = 30$

$D[5] = C[1,5] = 100$

Iteration 1: $i = 1$

$w \in V - S$

$w \in \{1,2,3,4,5\} - \{1\}$

$w \in \{2,3,4,5\}$

choose $w = 2$ because $D[2] = 10$ is minimum

$S = S \cup \{w\}$

$= \{1\} \cup \{2\}$

$S = \{1,2\}$

for each vertex $V \in V - S$

$V \in \{1,2,3,4,5\} - \{1,2\}$

$V \in \{3,4,5\}$

$V = 3$

$D[v] = \min(D[v], D[w] + C[w,v])$

$D[3] = \min(D[3], D[2] + C[2,3])$

$= \min(\infty, 10 + 50)$

$= \min(\infty, 60)$

$D[3] = 60.$

V=4

$D[4] = \min(D[4], D[2] + c[2,4])$

$= \min(30, 10+\infty)$

$= 30$

V=5

$D[5] = \min(D[5], D[2] + c[2,5])$

$= \min(100, 10+\infty)$

$= 100.$

Iteration 2: $i=2$

$w \in V-S$

$w \in \{1,2,3,4,5\} - \{1,2\}$

$w \in \{3,4,5\}$

choose $w=4$

$S = S \cup \{w\}$

$= \{1,2\} \cup \{4\} = \{1,2,4\}$

for each vertex $w \in V-S$

$w \in \{1,2,3,4,5\} - \{1,2,4\}$

$w \in \{3,5\}$

$D[3] = \min(D[3], D[4] + c[4,3])$

$= \min(60, 30+20)$

$= 50$

$D[5] = \min(D[5], D[4] + c[4,5])$

$= \min(100, 30+60)$

$= \min(100, 90)$

$= 90$.

Iteration 3: $i = 3$

$w \in V - S$

$w \in \{1,2,3,4,5\} - \{1,2,4\}$

$w \in \{3,5\}$

choose $w = 3$ because in iteration $D[3] = 50$ is min

$S = S \cup \{w\}$

$= \{1,2,4\} \cup \{3\}$

$S = \{1,2,4,3\}$

for each vertex $w \in V - S$

$w \in \{1,2,3,4,5\} - \{1,2,3,4\}$

$w \in \{5\}$

$D[5] = \min(D[5], D[3] + c[3,4])$

$= \min(90, 50+10)$

$= \min(90, 60)$

$= 60$

$\boxed{v = 5}$

Iteration 4 : $i = 4$:

Select $w = 5$ so that $S = \{1,2,3,4,5\}$

$D[2] = 10$

$D[3] = 50 \Rightarrow \boxed{i = 2}$ min

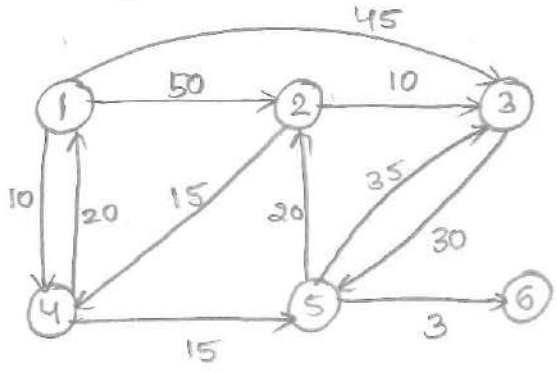
$D[4] = 30$

$D[5] = 60$

The time complexity of single source shortest path is $O(n^2)$.

* Alternate method:

i) Determine the shortest path from source vertex to remaining vertices.

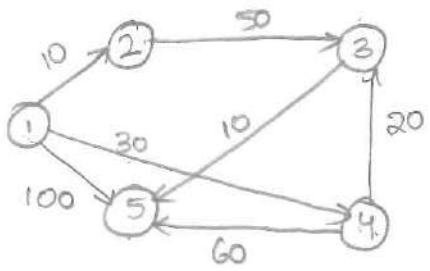


Sol:-

Iteration	S	Vertex Selected	Distance					
			1	2	3	4	5	6
Initial	{1}	-	∞	50	45	10	∞	∞
1	{1}	4	∞	50	45	10	25	∞
2	{1,4}	5	∞	45	45	10	25	28
3	{1,4,5}	6	∞	45	45	10	25	28
4	{1,4,5,6}	2	∞	45	45	10	25	28
5	{1,4,5,6,2}	3	∞	45	45	10	25	28

Path	length
1-4	10
1-4-5	25
1-4-5-6	28
1-4-5-2	45
1-3	45

2) Determine shortest path:



Iteration	S	Vertex Selected	Distance				
			1	2	3	4	5
Initial	{1}	-	∞	10	∞	30	100
1	{1,4}	2	∞	10	60	30	100
2	{1,2}	4	∞	10	50	30	90
3	{1,2,4}	3	∞	10	50	30	60
4	{1,2,4,3}	5	∞	10	50	30	60

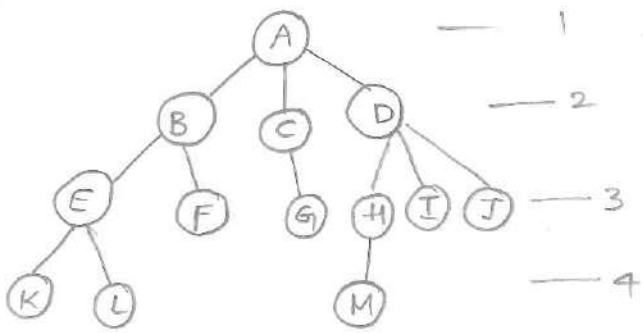
path	length
1-2	10
1-4	30
1-2-3	50
1-5	60

SEARCHING AND TRAVERSAL TECHNIQUE.

*Tree:

A tree is a finite set of one or more nodes such that there is a special node called the root node and the remaining nodes are partitioned into $n \geq 0$ disjoint sets. $T_1, T_2, \ldots T_n$. The sets $T_1, T_2, \ldots T_n$ are called the sub-trees of the root node.

Ex:



* Terminology:

*Degree: The number of subtrees of a node is called its degree.

Ex: The degree of root node 'A' is 3.

The degree of node 'C' is 1.

*Degree of a tree:

A degree of tree is the maximum degree of nodes

of the tree.

→ The degree of above tree is '3'.

* Terminal nodes or Leaf nodes:

→ Nodes that have degree '0' are called leaf or terminal nodes. (or)

→ Nodes that doesn't have child nodes called leaf or terminal nodes

→ In above tree, leaf nodes are K, L, F, G, M, I, J

* Non-terminal nodes:

→ Nodes that have one or more child nodes are called non-terminal nodes.

B, C, D .... are example of non-terminal nodes.

* Siblings:

→ Children of the same parent are said to be siblings

Ex: H, I, J are siblings of D.

* Ancestors:

→ The ancestors of nodes are all the nodes along the path from the root node to that node.

Ex: The ancestors of M are A, D, H.

* Level:

The level of node is defined by, Initially the root node will be at level 1. If a node is at level P then its child nodes are at level P+1.

* Height or depth of a tree:

Height or depth of a tree is defined to be the maximum level of any node in the tree.

Ex: The height of given tree is 4.

* Forest:

A forest is set of $n \geq 0$ disjoint trees

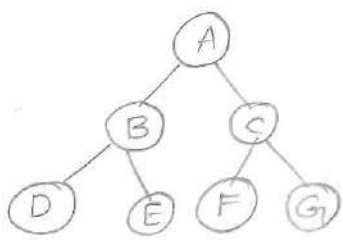
* Binary trees:

→ A Binary tree is a finite set of nodes that is either empty or contains root node and two disjoint binary trees called left and right subtrees.

(or)

→ Binary tree is a tree in which each node may have zero or one or two nodes exactly.

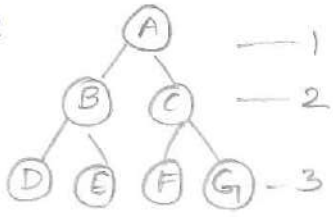
Ex:



* Properties:

→ The maximum no. of nodes on level i of a binary tree is 2^{i-1}

Ex:



At level i = 3, max no. of nodes $2^{i-1} = 2^{3-1} = 2^2 = 4$.

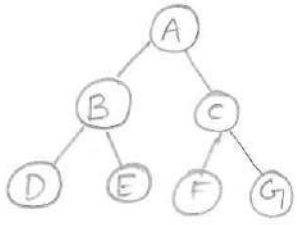
→ The maximum no. of nodes in a binary tree of depth k is $2^k - 1$

Ex: Maximum no. of nodes in a binary tree is k = 3

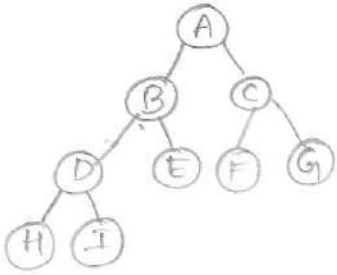
$2^k - 1 = 2^3 - 1 = 8 - 1 = 7$.

* Full binary tree:

→ The binary tree of depth 'k' that has exactly $2^k - 1$ nodes. is called full binary tree of depth 'k'.



It is a full binary tree

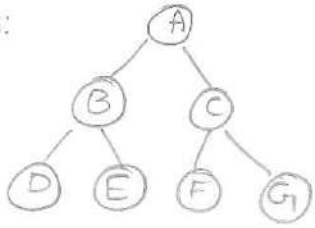


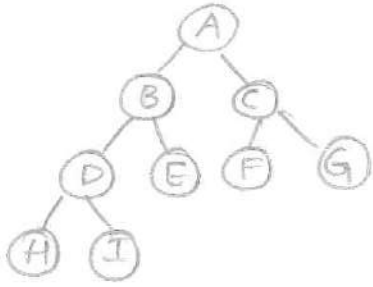
It is not full binary tree because it doesn't have $2^k - 1$ $2^4 - 1 = 15$ nodes.

* Complete binary tree:

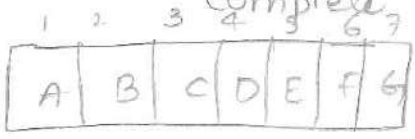
→ If a binary tree is represented using arrays and if it doesn't contain empty location then the tree is known as complete binary tree.

Ex:



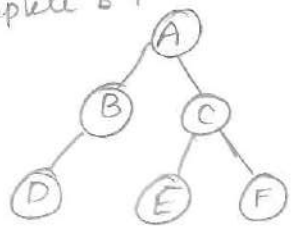


Complete binary tree

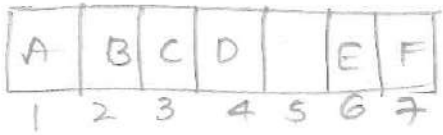


1. Full B.T
2. Complete B.T

1. Full B.T
2. Complete B.T



Not complete binary tree



1. Not full B.T
2. Not complete B.T

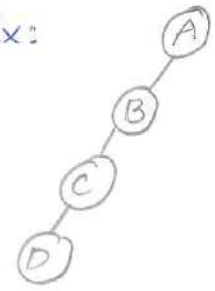
Note:

→ All full binary trees are complete binary trees but all complete binary trees are not full binary trees.

* Left skewed binary tree:

→ If nodes of a tree placed in left side to its parent then such type of tree is called left skewed binary tree.

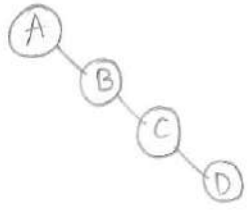
Ex:



* Right skewed binary tree:

→ If nodes of a tree placed in right side to its parent then such type of tree is called right skewed binary tree.

Ex:



* Tree traversal techniques (Binary tree):

→ Traversal is a process of visiting all the nodes of the tree.

→ There are three traversal techniques in binary tree.

1. In-order traversal
2. Pre-order traversal

3. Post order traversal.

1. In-order traversal:

→ In this traversal method, the following steps used to visit all the nodes of the binary tree.

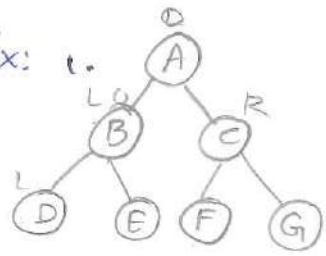
Algorithm:

Step 1: Traverse left subtree

Step 2: Visit root node

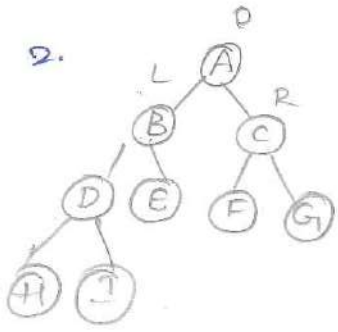
Step 3: Traverse Right Subtree.

Ex: 1.



D, B, E, A, F, C, G - Inorder.

2.



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

2. Pre-order traversal:

→ In this traversal method following steps are used to visit all the nodes of a binary tree.

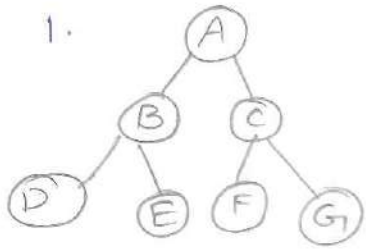
Algorithm:

Step 1: visit root node

Step 2: traverse left subtree

Step 3: traverse right subtree

Ex: 1.



A, B, D, E, C, F G.

3. Post-order traversal:

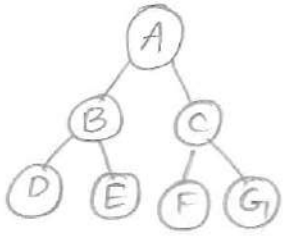
→ In this traversal method following steps are used to visit all the nodes of the binary tree.

Algorithm:

Step 1: traverse left subtree

Step 2: traverse right subtree

Step 3: visit root node.



D, E, B, F, G, C, A

6. Next vertex in the queue is 5 to be explored. But, all adjacent vertices of vertex '5' has already processed. So no exploration is required.

6	7	8

7. Next vertex in the queue is 6 to be explored. But all adjacent vertices of vertex '6' has already processed. So no exploration is required.

7	8

8. Next vertex in the queue is 7 to be explored. But all adjacent vertices of vertex '7' has already processed. So no exploration is required

8

9. Next vertex in the queue is 8 to be explored But all adjacent vertices of vertex '8' has already processed. So no exploration is required.

Connected graph:

→ A graph 'G' is said to be connected graph if their exists a path between every pair of vertices.

An undirected graph is said to be connected if and only if for every pair distinct vertices.(u,v) in graph G.

~~There if if is~~

*BFS algorithm:

```
Algorithm BFS(v)
// A breadth first search 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 ; visited[ ] is initialized to zero.
{
 u:=v; // q is a queue of unexplored vertices
 visited [v]:=1;
 repeat
 {
  for all vertices w adjacent from u do
  {
  if (visited [w] =0) then
  {
   Add w to q; // w is unexplored
    visited [w]:=1;
  }
  }
  if q is empty then return; // no unexplored vertex.
  delete u from q; // Get first unexplored vertex.
 }until (false);
}
```

2. Depth first search (DFS): Time complexity is $\theta(|V|+|E|)$

→ DFS is a graph traversal technique. DFS is an algorithm for traversing all vertices of the given graph or searching spanning tree one starts at the root (selecting some arbitary node as its root in the case of graph) and explores as far as possible along each branch before backtracking.

→ It employees the following rules

rule 1: Visit the adjacent unvisited vertex. Mark it as visited. display it. Push it in a stack.

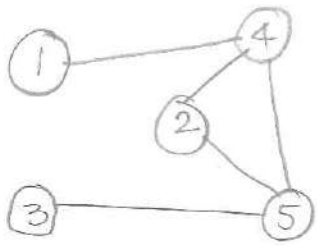
rule 2: If no adjacent vertex is found, pop up a vertex from a stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices)

rule 3: Repeat rule 1 and rule 2 until the stack is empty.

Algorithm for DFS:

```
Algorithm DFS(w)
{
  visited[v] := 1;
  for each vertex w adjacent from v do
  {
   if (visited[w]=0) then
    DFS(w);
  }
}
```

Ex: Traverse a graph shown below using DFS start from a vertex with number 1.



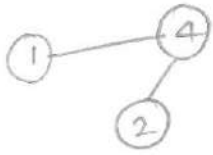
$2T(n-1)+1$

1. Mark a vertex 1 as visited

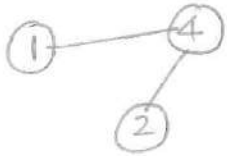
2. There is an edge (1,4) and vertex 4 is unvisited go there.

3. Mark the vertex 4 as visited

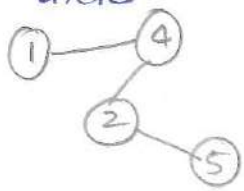
4. There is an edge (4,2) and vertex 2 is unvisited go there



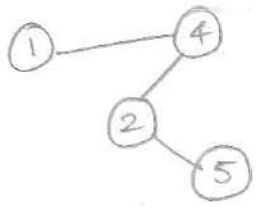
5. Mark the vertex 2 as visited.



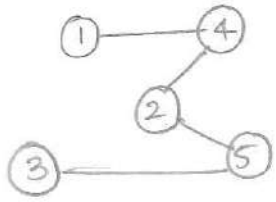
6. There is an edge (2,5) and vertex 5 is unvisited go there



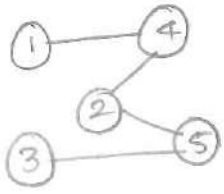
7. Mark the vertex 5 as visited



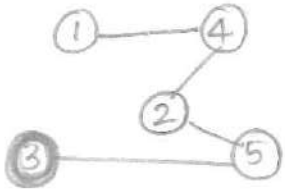
8. There is an edge (5,3) and a vertex 3 is unvisited go there



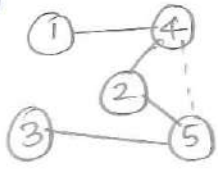
9. Mark the vertex 3 as visited.



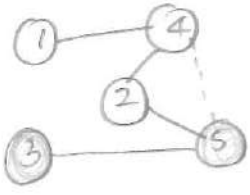
10. There are no ways to go from vertex 3. Mark it as black and backtrack to vertex 5.



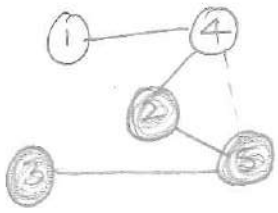
11. There is an edge (5,4) but the vertex 4 is already visited



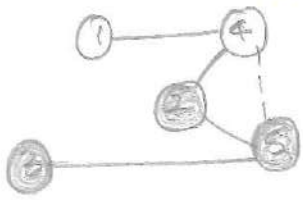
12. There are no ways to go from the vertex 5. Mark it as black and backtrack to vertex 2



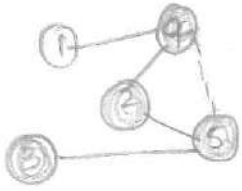
13. There are no more edges, adjacent to vertex 2. Mark it as black and backtrack to the vertex 4.



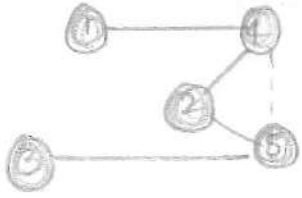
14. There is an edge (4,5) but the vertex 5 is black



15. There are no more edges, adjacent to the vertex 4. Mark it as black and backtrack to vertex '1'.



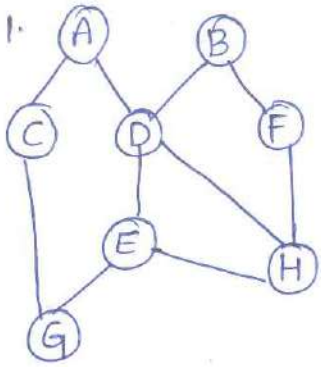
16. There are no more edges, adjacent to the vertex 1. Mark it as black. DFS is over.



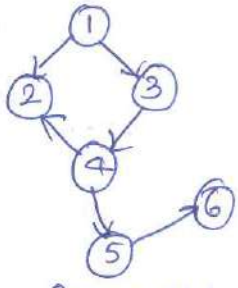
* Connected graph:

A graph is said to be connected iff there exists a simple path b/w every pair of vertices of given graph

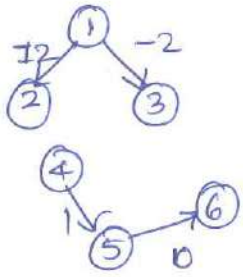
Ex: 1.



2.



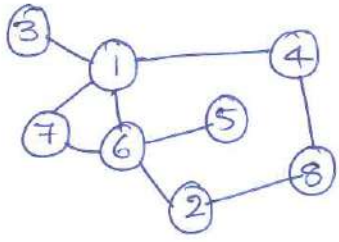
Connected

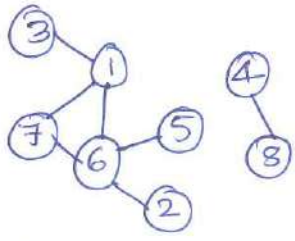


disconnected.

*Connected component:

Connected component of an undirected graph is a maximal connected subgraph.





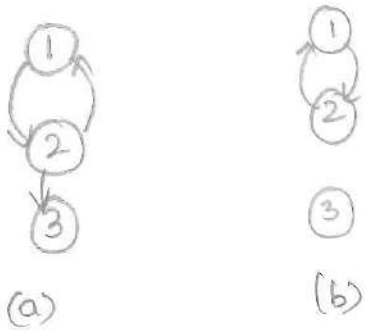
A graph that is connected and a graph that consists of two connected components.

* Strongly connected:

→ A directed graph G is said to be strongly connected iff for every pair of distinct vertices u and v in G there is a directed path from u to v and also from u to v.

→ A strongly connected component is a maximal subgraph that is strongly connected. The below figure (a) is not

strongly connected as there is no path from vertex from 3 to 2.



(a) (b)

A graph and its strongly connected components.

* Biconnected graph:

A graph is said to be biconnected if

1. It is connected i.e it is possible to reach every vertex from every other vertex, by a simple path.
2. Even after removing any vertex the graph remains connected.

→ A graph with no articulation point is called biconnected. In other words, a graph is biconnected if and only if any vertex is deleted, the graph remains connected.

* Articulation point:

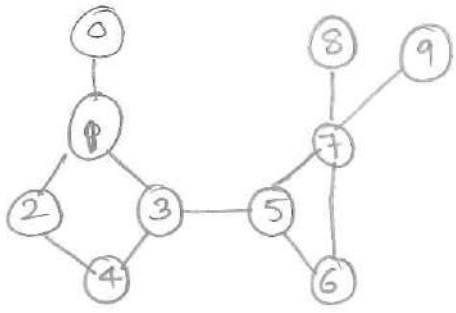
→ An articulation point in a connected graph is a vertex that, if delete, would break the graph into two or more pieces (connected component).

→ An articulation point is a vertex of G such that the deletion of v, together with all edges incident on v, produces a graph G1, that has atleast two

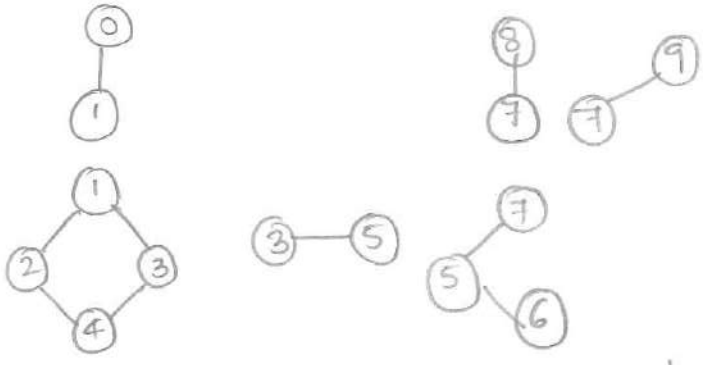
connected components. for ex, the connected graph of below figure has four articulation points, vertices 1,3,5,7.

*Biconnected component:

A biconnected component of graph is maximal biconnected subgraph.



(a) Connected graph

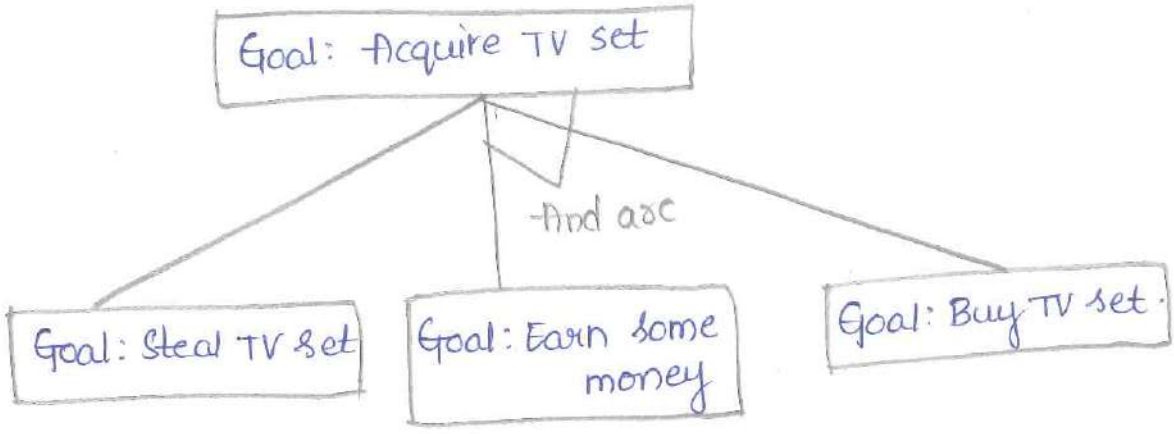


(b) Biconnected components.

*AND-OR Graph:

→ when a problem can be divided into a set of sub problems, where each sub problem can be solved separately and a combination of these will be a solution, AND-OR graphs or AND-OR trees are used for representing the solution. The decomposition of the problem or problem reduction generates AND arcs. One AND are may point to any no of successor nodes. All these must be solved so that the arc will rise to many arcs, indicating several possible solutions

Hence the graph is known as AND-OR instead of AND. figure shows an AND-OR graph.



Ex: AND-OR graph