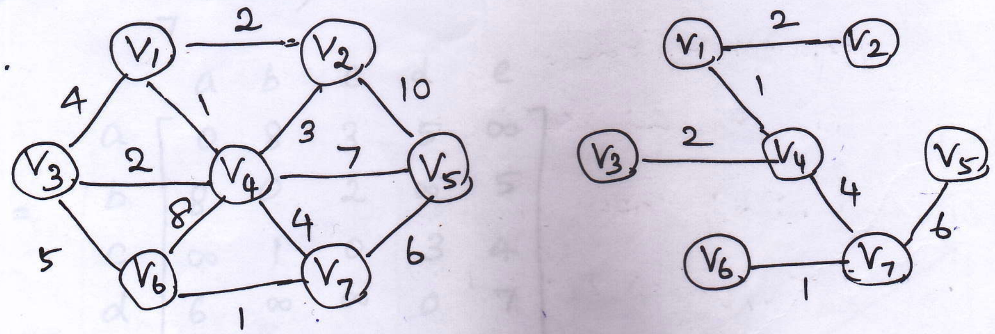
MINIMUM SPANNING TREE:

* A minimum spanning tree of an undirected graph G is a tree formed from graph edges that connects all the vertices of G at lowest total cost. The number of edges in the minimum spanning tree is V-1.

eg.



* Two algorithms are used to construct minimum spanning tree.

1. Prim's Algorithm
2. Kruskal's Algorithm.

1. Prim's Algorithm:

* It constructs tree in successive stages. In each stage, one node is picked as root and add an edge and thus an associated vertex to the tree.

* The algorithm finds at each stage, a new vertex to add to the tree by choosing an edge (u,v) such that the cost of (u,v) is smallest among all the edges where u is vertex in tree and v is not.

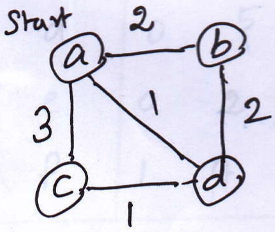
* Prim's algorithm is identical to Dijkstra's Alg. For each vertex, it keeps values d_v, P_v and an indication of whether it is known or unknown.

* After a vertex v is selected, for each

Algorithm:

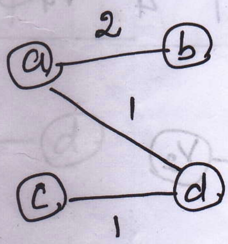
```
void prim (Table T)
{
  vertex v, w;
  for ( ;  ; )
    v = Smallest distance unknown vertex;
    T[v]. known = True;
    for each w adjacent to v
      if (!T[w]. known)
        if ( $C_{v,w}$ < T[w].dist )
        {
          T[w]. Dist = ~~min~~ $C_{v,w}$ ;
          T[w]-path = v
        }
  }
```

Example 1:



vertex	k	d_v	p_v	k	d_v	p_v	k	d_v	p_v	k	d_v	p_v
a	0	0	0	1	0	0	1	0	0	1	0	0
b	0	∞	0	0	2	a	0	2	a	0	2	a
c	0	∞	0	0	3	a	0	1	d	1	1	d
d	0	∞	0	0	1	a	1	1	a	0	1	a

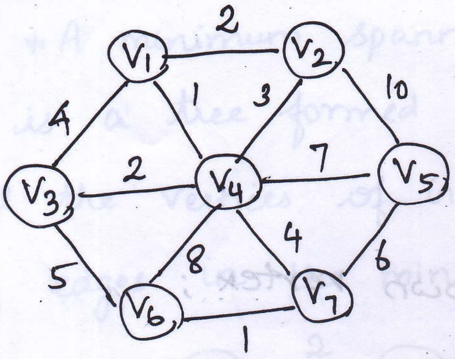
vertex	k	d_v	p_v
a	1	0	0
b	1	2	a
c	1	1	d
d	1	1	a



Minimum cost of Spanning tree is

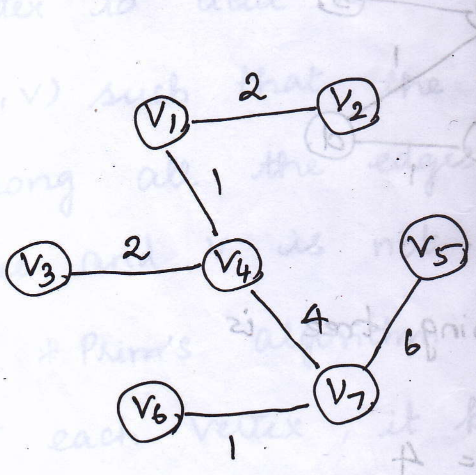
$$C_{a,b} + C_{a,d} + C_{c,d} = 4$$

Example 2 :

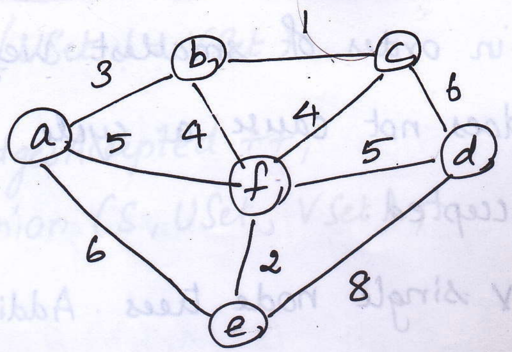


Vertex	k	dv	pv	k	dv	pv	k	dv	pv	k	dv	pv	k	dv	pv
V1	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0
V2	0	∞	0	0	2	V1	0	2	V1	1	2	V1	1	2	V1
V3	0	∞	0	0	4	V1	0	2	V4	0	2	V4	1	2	V4
V4	0	∞	0	0	1	V1	1	1	V1	1	1	V1	1	1	V1
V5	0	∞	0	0	∞	0	0	7	V4	0	7	V4	0	7	V4
V6	0	∞	0	0	∞	0	0	8	V4	0	8	V4	0	5	V3
V7	0	∞	0	0	∞	0	0	4	V4	0	4	V4	0	4	V4

vertex	k	dv	pv	k	dv	pv	k	dv	pv
V1	1	0	0	1	0	0	1	0	0
V2	1	2	V1	1	2	V1	1	2	V1
V3	1	2	V4	1	2	V4	1	2	V4
V4	1	1	V1	1	1	V1	1	1	V1
V5	0	6	V7	0	6	V7	0	6	V7
V6	0	1	V7	1	1	V7	1	1	V7
V7	1	4	V4	1	4	V4	1	4	V4

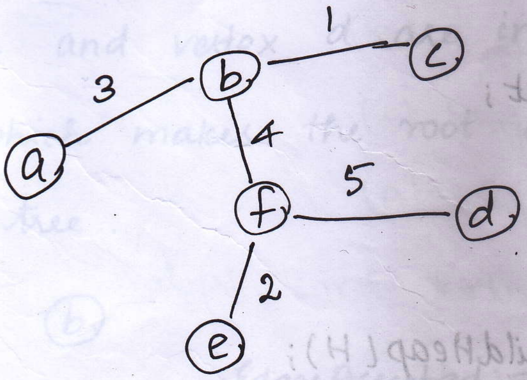


example 3 :



Vertex	k	d_v	p_v	k	d_v	p_v	k	d_v	p_v	k	d_v	p_v
a	0	0	0	1	0	0	1	0	0	1	0	0
b	0	∞	0	0	3	a	1	3	a	1	3	a
c	0	∞	0	0	∞	0	0	1	b	1	1	b
d	0	∞	0	0	∞	0	0	∞	0	0	6	c
e	0	∞	0	0	6	a	0	6	a	0	6	a
f	0	∞	0	0	5	a	0	4	b	0	4	b

Vertex	k	d_v	p_v	k	d_v	p_v	k	d_v	p_v
a	1	0	0	1	0	0	1	0	0
b	1	3	a	1	3	a	1	3	a
c	1	1	b	1	1	b	1	1	b
d	0	5	f	0	5	f	1	5	f
e	0	2	f	1	2	f	1	2	f
f	1	4	b	1	4	b	1	4	b



2) Kruskal's Algorithm :

* The algorithm uses a greedy technique to compute minimum Spanning tree.

* It selects the edges in order of smallest weight and accept an edge if it does not cause a cycle. It terminates if enough edges are accepted.

* Initially, there are V single node trees. Adding an edge merges 2 trees into one. When algorithm terminates, there is only one tree, which is called minimum Spanning tree

* The Algorithm uses 2 data structures.

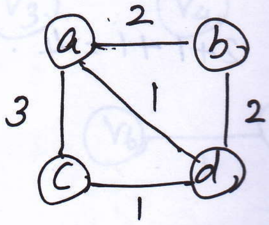
(i) Find (U) - It returns the root of the tree that contains vertex U.

(ii) Union (S, U, V) - It merges 2 trees by making root pointer of one node point to root node of other tree.

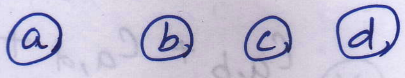
Algorithm:

```
void Kruskal (Graph G)
{
    int EdgesAccepted = 0;
    Disjoint_Set S;
    Heap H;
    SetType USet, VSet;
    Edge E;
    Vertex V, W;
    Initialize (S); BuildHeap (H);
    while (EdgesAccepted < NumofVertex - 1)
    {
        E = DeleteMin (H);
        USet = Find (U, S);
        VSet = Find (V, S);
    if (USet ! = VSet)
    {
      EdgesAccepted ++;
      Union (S, USet, VSet);
     }
   }
 }
```

Example 1:



Step 1: Initialize (S):



Step 2: BuildHeap (H).

(a,b) ⇒ 2

(a,c) ⇒ 3

(a,d) ⇒ 1

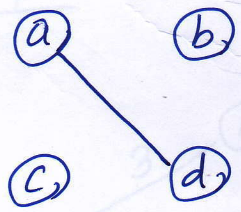
(c,d) ⇒ 1

(b,d) ⇒ 2

Step 3: Select an edge with minimum weight.

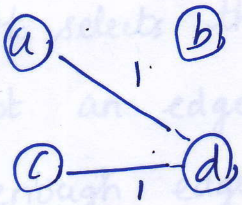
Edge (a,d) ⇒ 1 is choosen.

vertex a and vertex d are in different tree. So calles union which makes the root of 1 tree to subtree of another tree.



EdgesAccepted = 1.

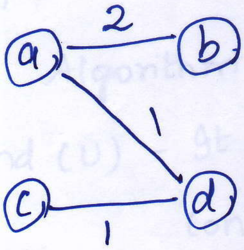
Step 4: Select next edge with minimum weight. So edge (c,d) ⇒ 1 is chosen. Check whether by including this edge, a cycle is formed. If cycle is formed,



Step 5: Repeat step 4 until all vertices are included in minimum Spanning tree (ie) EdgesAccepted = N-1

EdgesAccepted = 2

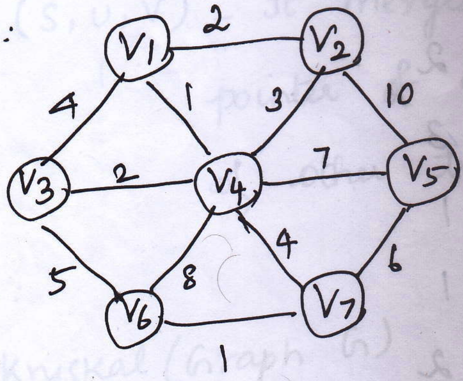
Next edge with min. cost is (a, b) = 2. It does not form cycle, so it is accepted.



Min. cost of spanning tree is 4.

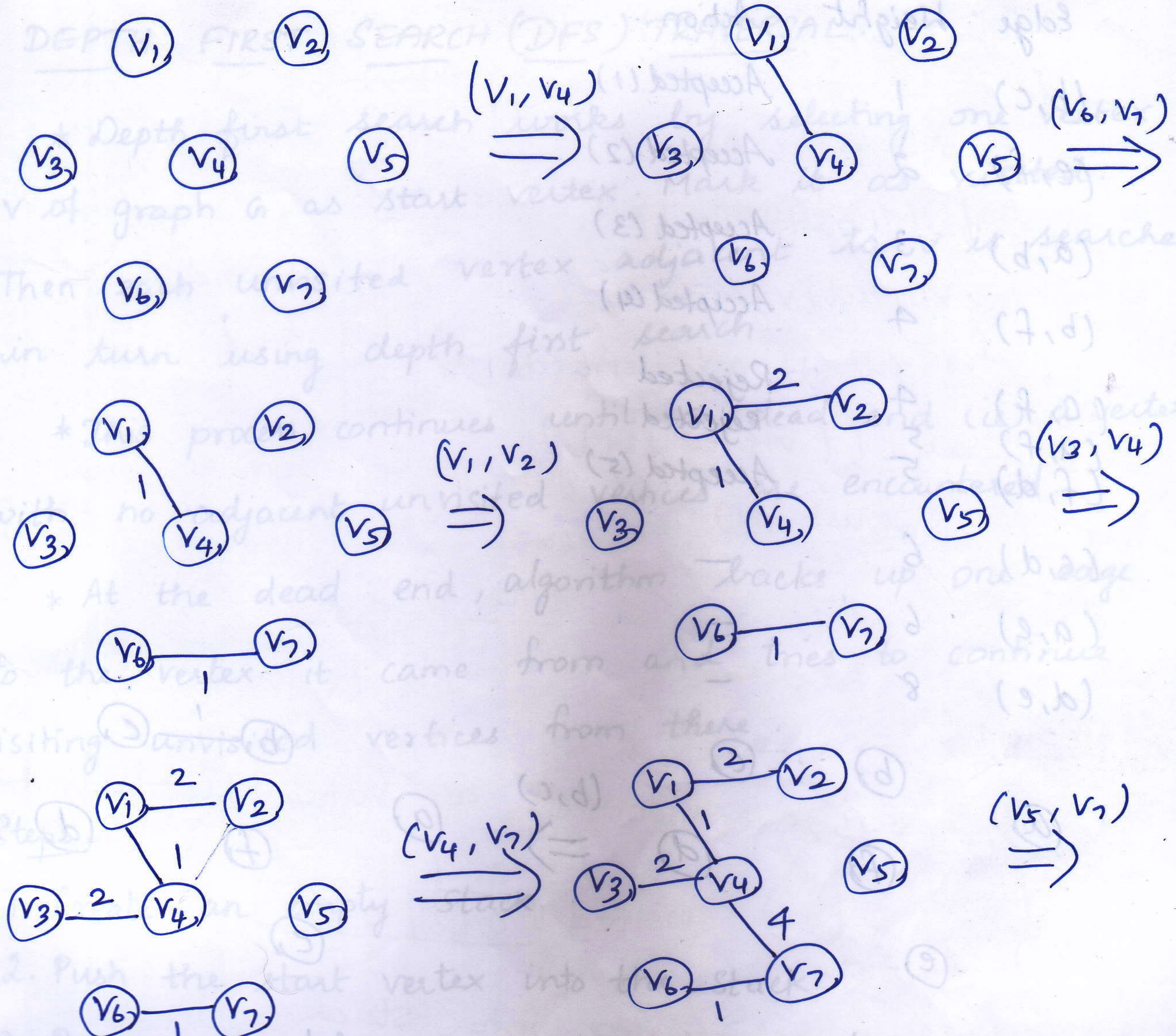
$C_{a,b} + C_{a,d} + C_{c,d} = 2+1+1 = 4.$

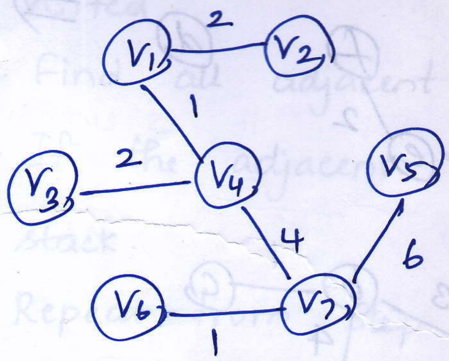
Example 2:



Arrange edges in increasing weight.

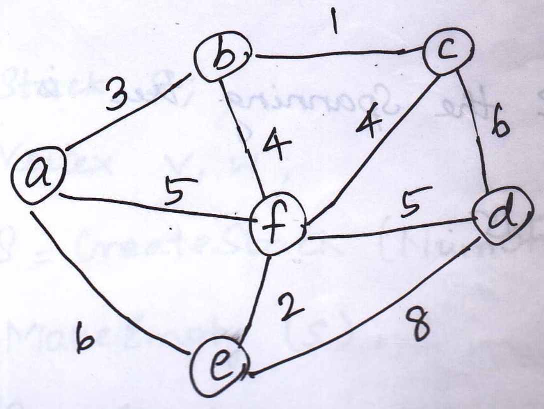
Edge	Weight	Action
(V_1, V_4)	1	Accepted (1)
(V_6, V_7)	1	Accepted (2)
(V_1, V_2)	2	Accepted (3)
(V_3, V_4)	2	Accepted (4)
(V_2, V_4)	3	Rejected. forms cycle.
(V_1, V_3)	4	Rejected
(V_4, V_7)	4	Accepted (5)
(V_3, V_6)	5	Rejected
(V_5, V_7)	6	Accepted (6)
(V_4, V_5)	7	–



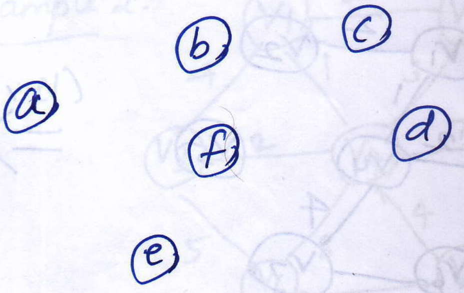


The minimum cost of spanning tree is 2+1+2+4+1+6 = 16.

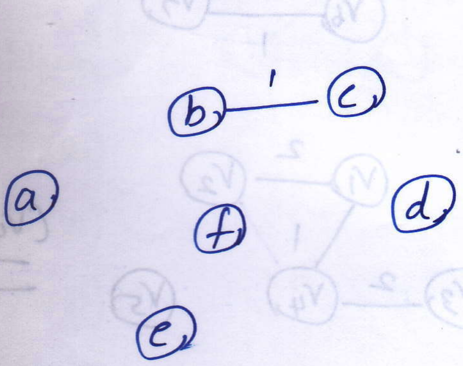
Example 3:



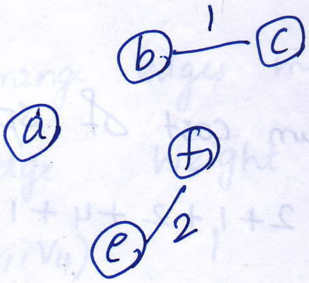
Edge	Weight	Action.
(b,c)	1	Accepted (1)
(e,f)	2	Accepted (2)
(a,b)	3	Accepted (3)
(b,f)	4	Accepted (4)
(c,f)	4	Rejected
(a,f)	5	Rejected
(f,d)	5	Accepted (5)
(c,d)	6	—
(a,e)	6	—
(d,e)	8	—



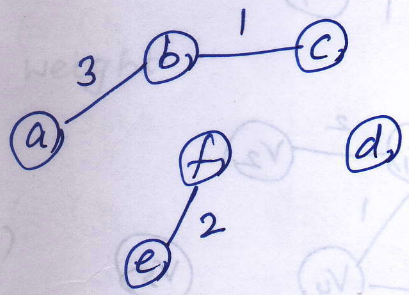
(b,c) ⇒



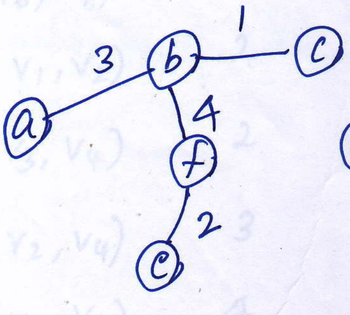
(e,f) ⇒



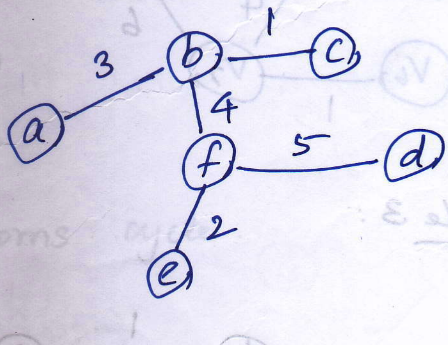
(a,b) ⇒



(b,f) ⇒



(f,d) ⇒



The minimum cost of the Spanning tree is

3+1+4+5+2 = 15.

DEPTH FIRST SEARCH (DFS) TRAVERSAL:

* Depth first search works by selecting one vertex v of graph G as start vertex. Mark it as visited. Then each unvisited vertex adjacent to v is searched in turn using depth first search.

* This process continues until a dead end (ie) a vertex with no adjacent unvisited vertices are encountered.

* At the dead end, algorithm backs up one edge to the vertex it came from and tries to continue visiting unvisited vertices from there.

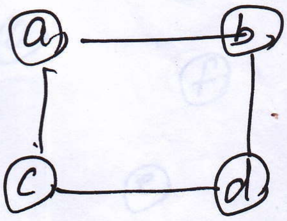
Steps:

1. Create an empty stack.
2. Push the start vertex into the stack.
3. Pop a vertex from stack. If it is unvisited mark it as visited.
4. Find all adjacent vertices w to the popped vertex v
5. If the adjacent vertex is unvisited, push it into the stack.
6. Repeat from Step 3 until stack becomes empty.

Algorithm:

```
void DFS (Graph G, start s)
{
 Stack S;
 Vertex v, w;
 S = CreateStack (NumOfVertex);
 MakeEmpty (S);
 S.push (s);
while (! IsEmpty (S))
{
   V = S. pop();
   if (! visited [V])
     visited [V] = 1;
   for all W adjacent to V
     if (! visited [W] )
       S.push (W);
 }
}
```

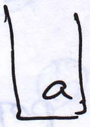
Example 1:



* Initially, all vertices are unvisited.

	a	b	c	d
visited	0	0	0	0

* Push the Start vertex into stack.



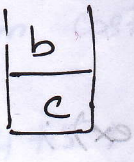
* Pop the vertex from stack. Since it is unvisited sets its visited to be true.

	a	b	c	d
visited	1	0	0	0

DFS

a,

* Find the adjacent vertices to a, (ie) W = b, c. Since the adjacent vertices are unvisited push it into stack.

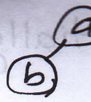


* Stack is not empty. Pop a vertex from stack. (ie) b is popped. Since unvisited, set it to be

	a	b	c	d
visited	1	1	0	0

DFS

a, b



* The adjacent vertices of b are a and d. a is already visited but d is unvisited. So push d into stack.

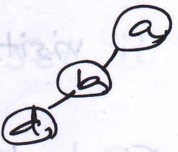
d
c

* Stack is not empty. Pop a vertex from stack (ie) d is popped. Since unvisited, set it to be visited.

	a	b	c	d
visited	1	1	0	1

DFS

a, b, d



* The adjacent vertices of d are c and b. b is already visited but c is unvisited. So push c into stack.

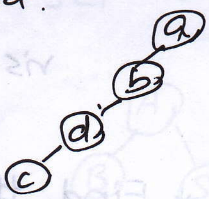
c
c

* Stack is not empty. Pop a vertex from stack (ie) c is popped. Since unvisited, set it to be visited.

	a	b	c	d
visited	1	1	1	1

DFS

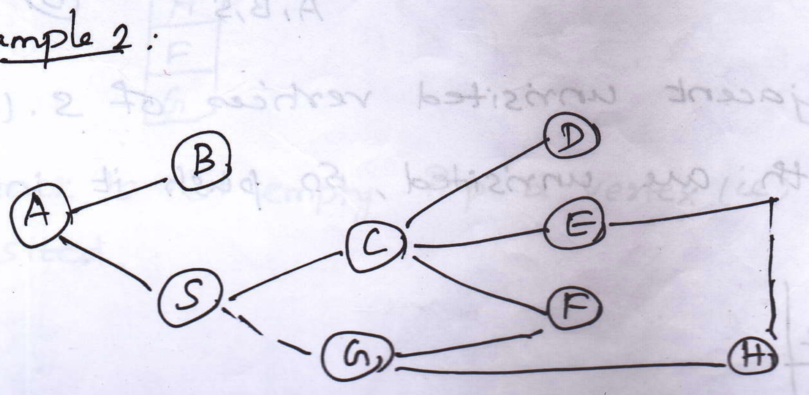
a, b, d, c



* The adjacent vertices of c are a and d. But they are already visited.

* Stack is not empty. Pop a vertex from stack (ie) c is popped. But it is already visited.

Example 2 :



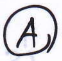
* Initially, all vertices are unvisited.

	A	B	C	D	E	F	G	H	S
visited	0	0	0	0	0	0	0	0	0

* Push the Start vertex into stack.

A

* Stack is not empty. Pop a vertex (ie) A. Since it is unvisited, set it to be visited.



	A	B	C	D	E	F	G	H	S
visited	1	0	0	0	0	0	0	0	0

DFS o/p:

A

* Find the adjacent unvisited vertices to A (ie) S and B. Since both are unvisited, push it to stack.

B

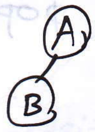
S

* Stack is not empty. Pop a vertex (ie) B. Since it is unvisited, set it to be visited.

	A	B	C	D	E	F	G	H	S
visited	1	1	0	0	0	0	0	0	0

DFS o/p:

A, B



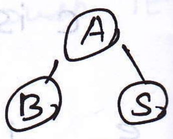
* Find the adjacent unvisited vertex of B. A is the only adjacent vertex but it is visited.

* Stack is not empty. Pop a vertex (ie) S. Since it is unvisited, set it to be visited.

	A	B	C	D	E	F	G	H	S
visited	1	1	0	0	0	0	0	0	1

DFS o/p:

A, B, S



* Find the adjacent unvisited vertices of S. (ie) C and G. Both are unvisited, so push it into stack.

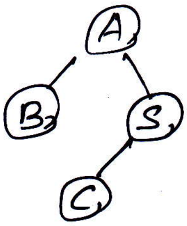
C

* Stack is not empty. Pop a vertex (ie) C. Since it is unvisited, set it to be visited.

	A	B	C	D	E	F	G	H	S
visited	1	1	1	0	0	0	0	0	1

DFS o/p:

A, B, S, C



* Find the adjacent unvisited vertices to C (ie) D, E and F. Push them into Stack.

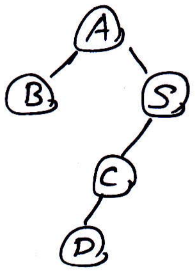
D
E
F
G

* Stack is not empty. Pop a vertex (ie) D. Set it to be visited.

	A	B	C	D	E	F	G	H	S
visited	1	1	1	1	0	0	0	0	1

DFS o/p:

A, B, S, C, D



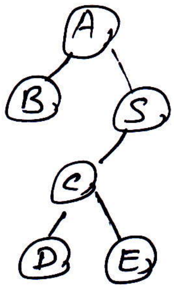
* Find the adjacent unvisited vertex to D. No adjacent unvisited vertex to D.

* Stack is not empty. Pop a vertex (ie) E. Set it to be visited.

	A	B	C	D	E	F	G	H	S
visited	1	1	1	1	1	0	0	0	1

DFS o/p:

A, B, S, C, D, E



* Find the adjacent unvisited vertex to E. H is the adjacent unvisited vertex to E. Push it into Stack

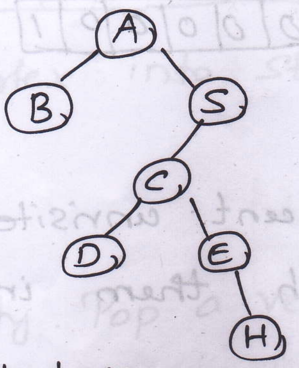
H
F
G

* Stack is not empty. Pop a vertex (ie) H. Mark it as visited.

	A	B	C	D	E	F	G	H	S
visited	1	1	1	1	1	0	0	1	1

DFS o/p:

A, B, S, C, D, E, H



* The adjacent unvisited vertex to H is G. Push it into stack.

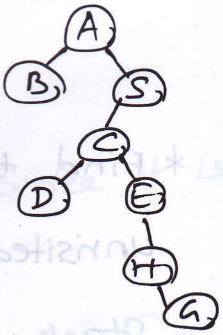
G.
F
G

* Stack is not empty. Pop a vertex (ie) G. Mark it to be visited.

	A	B	C	D	E	F	G	H	S
visited	1	1	1	1	1	0	1	1	1

DFS o/p:

A, B, S, C, D, E, H, G



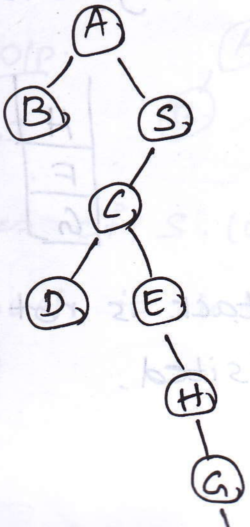
* The adjacent unvisited vertex to G is F. Push it into stack.

F
F
G

* Stack is not empty. Pop a vertex (ie) F. Mark it to be visited.

	A	B	C	D	E	F	G	H	S
visited	1	1	1	1	1	1	1	1	1

DFS o/p: A, B, S, C, D, E, H, G, F



* F has not adjacent unvisited vertex. So nothing to push.

* Stack is not empty. Pop a vertex (ie) F. But F is already visited.

* Stack is not empty. Pop a vertex (ie) G. But G is already visited.

* Stack is empty. So process gets completed.

APPLICATIONS OF DFS:

(i) To check whether undirected graph is connected or not.

(ii) To check whether connected ~~graph~~ undirected graph is Biconnected or not.

(iii) To check acyclicity of directed graph.

(i) TO CHECK _ UNDIRECTED GRAPH IS CONNECTED OR NOT.

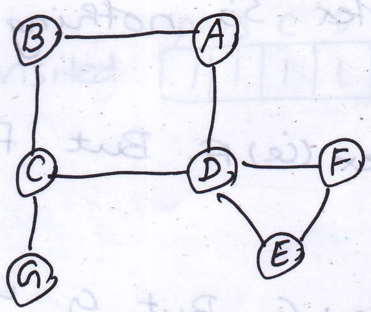
* An undirected graph is connected if and only if the depth first search starting from any node visits every node in the graph.

(ii) TO CHECK - CONNECTED UNDIRECTED GRAPH IS BICONNECTED OR NOT:

BICONNECTIVITY:

* A connected undirected graph is biconnected if there are vertices whose removal disconnects rest of the graph.

Articulation Points - It is a vertex whose removal disconnects graph.



In the eg, removal of C disconnects G from graph. Similarly, removal of D disconnects E and F.

So, C and D are articulation points.

Steps to find Articulation Points:

Step 1: Perform DFS, starting at any vertex.

Step 2: Number the vertex, as they are visited as Num(V).

Step 3: Compute the lowest numbered vertex for every vertex V in Depth first Spanning tree

By definition, low (V) is the minimum of

(i) Num (V)

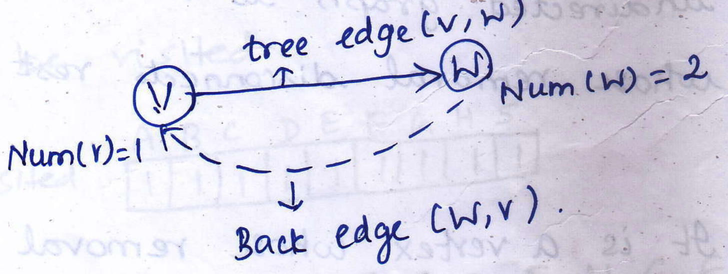
(ii) The lowest Num (W) among all the back edges (V, W)

(iii) The lowest Low (W) among all tree edges (V, W).

Step 4: (i) Root is an articulation point if and only if it has more than 2 children

(ii) Any vertex V other than root is an articulation point if and only if V has child W such that

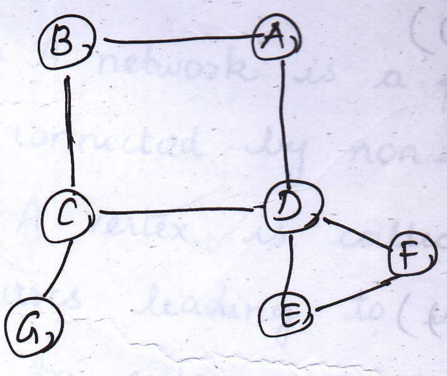
Low (W) ≥ Num (V)

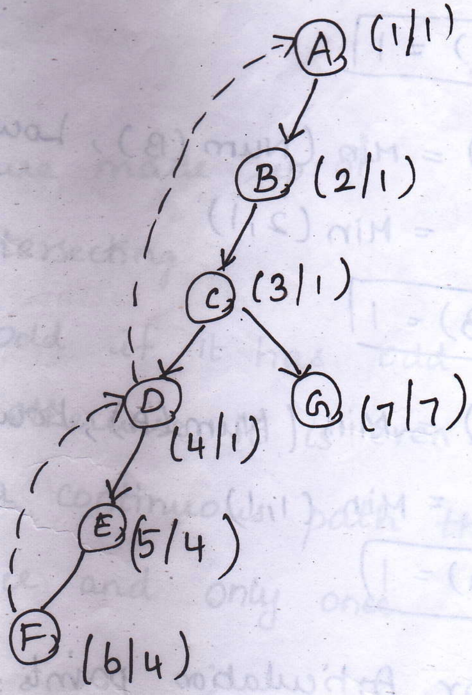


Routine to Compute Low & test for Articulation points:

```
void AssignLow (vertex V)
{
  vertex W;
Low [V] = Num [V] ; /* Rule 1 */
for each W adjacent to V
{
  if (Num [W] > Num [V])   /* tree edge */
  {
    AssignLow (W);
    if (Low [W] > = Num [V])
      printf ("%s is an articulation point", V);
    Low [V] = min (Low [V], Low [W]); /* Rule 3 */
  }
  else
    if (parent [V] != W)
      Low [V] = min (Low [V], Num [W]);  /* Rule 2 */
  }
}
```

Example:





* Low can be computed by performing postorder traversal of depth first spanning tree.

eg:

* Low (F) = Min (Num(F), Num (D)) /* No tree edge only back edge */

= Min (6,4)

$\boxed{\text{Low (F) = 4}}$

* Low (E) = Min (Num (E), Low (F))

= Min (5,4)

$\boxed{\text{Low(E) = 4}}$

* Low (D) = Min (Num (D), Low (E), Num (A))

= Min (4,4,1)

$\boxed{\text{Low (D) = 1}}$

* Low (G) = Min (Num (G))

$\boxed{\text{Low(G) = 7.}}$

* Low (C) = Min (Num (C), Low (D), Low (G))

= Min (3, 1, 7)

$\boxed{\text{Low(C) = 1}}$

* Low (B) = Min (Num (B), Low (C))

= Min (2,1)

$\boxed{\text{Low(B) = 1}}$

* Low (A) = Min (Num (A), Low (B))

= Min (1,1)

$\boxed{\text{Low(A) = 1}}$

Check for Articulation points:

* If (Low [W] $\geq$ Num [V]) then V is articulation point.

Low (G) $\geq$ Num (C)

7 $\geq$ 3 . So, C is an articulation point

IIIly, Low[E] = Num[D]

4 = 4.

So, D is an articulation point.

EULER'S CIRCUITS:

EULER PATH:

* A graph is said to be containing an Euler path if it can be traced in 1 swap without lifting pencil from paper and without tracing the same edge more than once.

* Vertices may be passed through more than once. Starting and ending points need not be the same.

EULER CIRCUIT:

* Similar to Euler path, except that the starting and ending points must be the same.

Definition:

(i) A network is a figure made up of connected by non-intersecting

(ii) A vertex is called odd if it has odd number of arcs leading to it, otherwise it is even.

(iii) An euler path is a continuous path that passes through every arc once and only once.

EULER THEOREM:

* If a network has 2 or 0 odd vertices, it has atleast one Euler path.

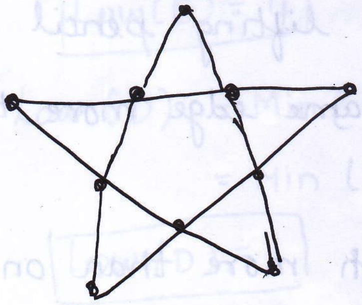
* If a network has exactly 2 odd vertices, then its Euler path can only start on one of the odd vertices

und end on the other.

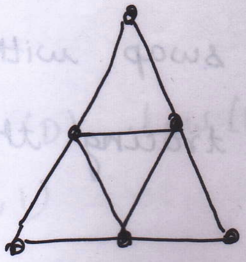
This type of Euler path is called an Euler Circuit.

Problem:

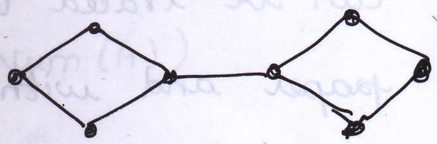
* For each network below, determine whether it has an Euler path.



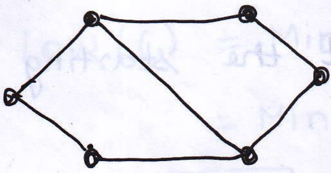
Graph 1



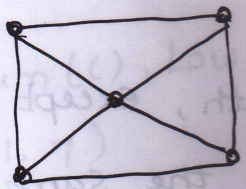
Graph 2



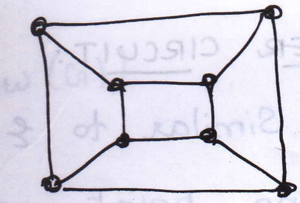
Graph 3



Graph 4



Graph 5



Graph 6

Graph	No. of odd vertices (vertices connected to odd no. of edges)	No. of even vertices (vertices connected to even no. of edges)	Euler path = P Euler circuit = C Neither = N
1	0	10	C
2	0	6	C
3	2	6	P
4	2	4	P
5	4	1	N
6	8	0	N