UNIT-IV

GRAPH AND ALGORITHMS

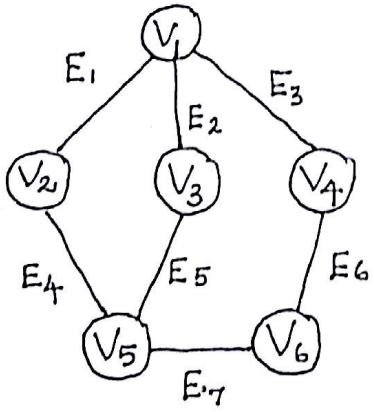
Graph - Non Linear Data Structure

Definition.

→ A graph $G_1 = (V, E)$ consists of a set of vertices, V and a set of edges, E

* Vertex - A node in the graph.
* Edge - Edge is a pair (v, w) where $v, w \in V$ sometimes refferred to as arcs.

Eg:



$G_1 = \{\{V_1, V_2, V_3, V_4, V_5, V_6\},$
$\{E_1, E_2, E_3, E_4, E_5, E_6, E_7\}\}$

Comparison between Graph and Trees

Graph	Tree
1) Graph is non-linear DS	1) Tree is a non linear DS
2) It is a collection of vertices/nodes and edges	2) It is a collection of nodes and edges
3) Each node can have any number of edges	3) A node can have any number of child nodes. Binary tree can have only 2 child nodes.

4) There is no unique node called root in graph.	4) There is a unique node called root in the tree
5) A cycle can be formed	5) There will not be any cycle.
6) Applications: For finding shortest path in networking graph is used.	6) Applications: For game tree decision trees, the tree is used.

Types of graphs:

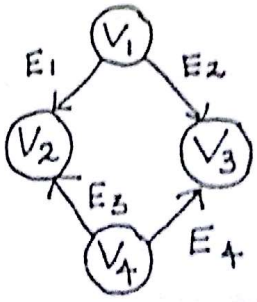
1) Directed graphs (digraphs)
2) Undirected graphs.

1) Directed graph

* In directed graph, directions are shown on the edges. The edges between the vertices are ordered.

E_1 - set of (V_1, V_2) not (V_2, V_1)

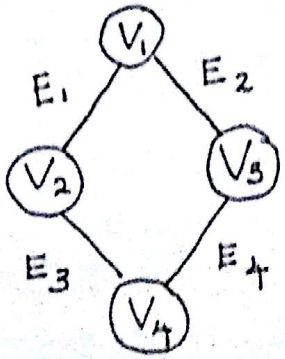
- V_1 - head
- V_2 - tail



2) Undirected graph

* In undirected graph, the edges are not ordered.

E_1 - set of (V_1, V_2) or (V_2, V_1)

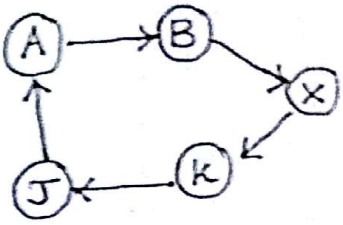


Path

A path is denoted using sequence of vertices $w_1, w_2, w_3 \dots w_N$ such that $(w_i, w_{i+1}) \in E$ for $1 \leq i < N$

Eg: Path from A to J

A-B-X-K-J



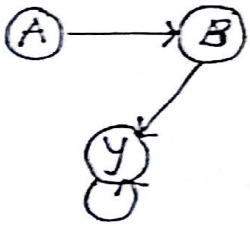
Length

The length of a path is the no. of edges on the path.

Eg: Length of A to J path is 4.

Loop

If the graph contains an edge (v,v) from a vertex to itself, then the path (v,v) is referred to as loop. (Simple graph → graph w/o loops)

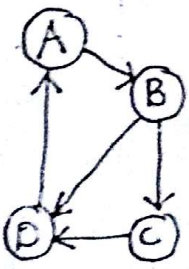


Simple path

A simple path is a path such that all vertices are distinct, except that the first and last could be same.

Cycle

A cycle in a directed graph is a path of length atleast 1 such that $w_1 = w_N$



cycle

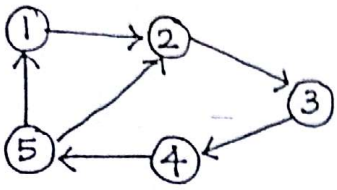
A-B-C-D-A

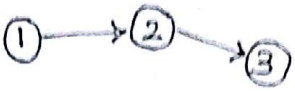
Acyclic

A directed graph is acyclic if it has no cycles.

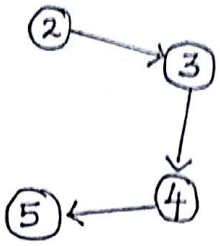
Component

* The maximal connected subgraph of a graph is called component of a graph.



1) 

2) 

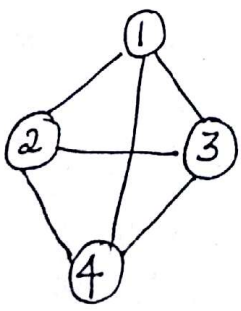
3) 

Connected graph

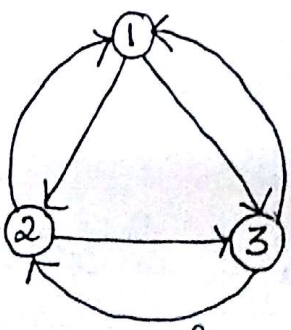
* An undirected graph is connected if there is a path from every vertex to every other vertex.

* A directed graph with this property is called strongly connected.

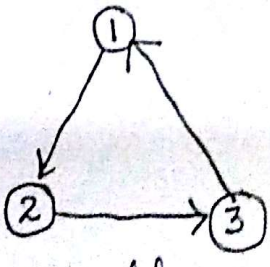
* If a directed graph is not strongly connected but the underlying graph (without direction to arcs) is connected, then the graph is said to be weakly connected.



connected



strongly connected



weakly connected

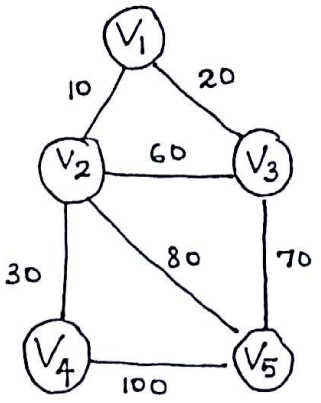
Complete graph

* If an undirected graph of 'n' vertices consists of $\frac{n(n-1)}{2}$ number of edges, then that graph is called a complete graph.

Weighted graph

* A graph in which weight, distance or cost is associated along every edge.

Eg:

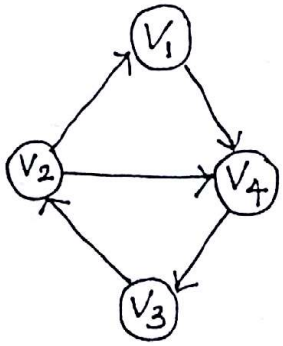


(DAG) Directed Acyclic Graph

* A directed graph does not contain cycle is called directed acyclic graph.

In-degree and Out-degree

* Degree - no. of edges associated with the vertex
* In degree - no. of edges incident to that vertex
* Out degree - no. of edges going away from the vertex



	Degree	In-degree	Out-degree
V_1	2	1	1
V_2	3	1	2
V_3	2	1	1
V_4	3	2	1

Applications:

1) Airport system modeling
2) Traffic flow modeling
3) Computer networks
4) Electrical circuits
5) Flow charts.

→ **Representations of Graph.**

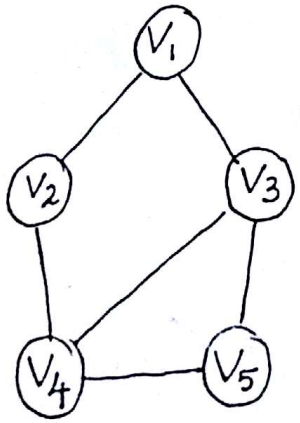
		space requirement
1) Adjacency Matrix	⟶	$O(\|V^2\|)$
2) Adjacency List	⟶	$O(\|E\|+\|V\|)$

1) Adjacency Matrix (static)

In this representation, a matrix or two - dimensional array is used.

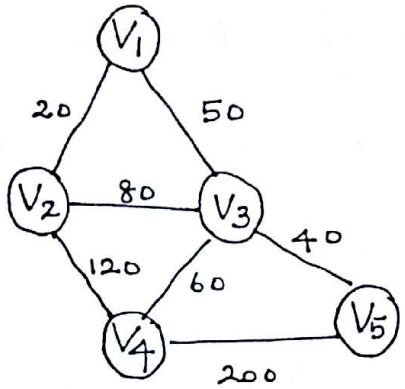
Consider a graph G of 'n' vertices. In a n×n matrix, M[i][j] = 1 if there is edge from V_i to V_j else M[i][j] = 0

Note: if M[i][j] = 1 M[j][i] also = 1 for undirected graph.



	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	1	0
3	1	0	0	1	1
4	0	1	1	0	1
5	0	0	1	1	0

For weight graph, M[i][j] = weight of edge b/w V_i & V_j

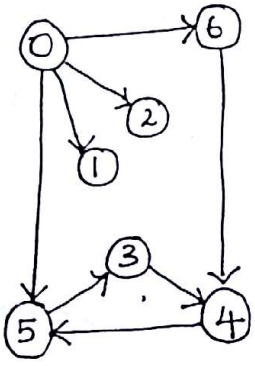


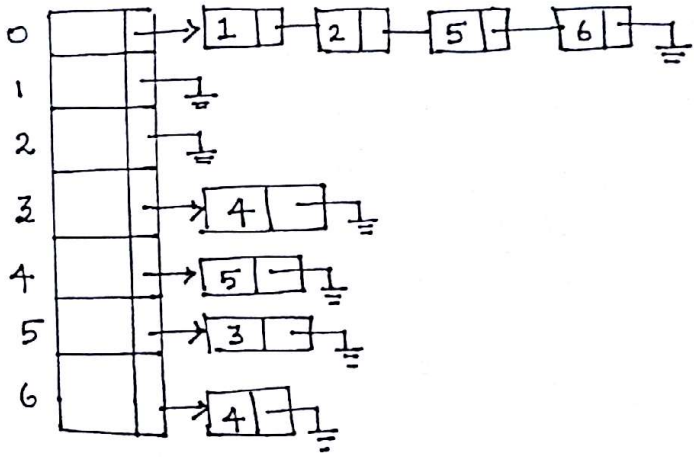
	1	2	3	4	5
1	0	20	50	0	0
2	20	0	80	120	0
3	50	80	0	60	40
4	0	120	60	0	200
5	0	0	40	200	0

2) Adjacency List: (Dynamic)

* In this representation, for each vertex, there is a linked list of vertices that are adjacent to it

Eg:



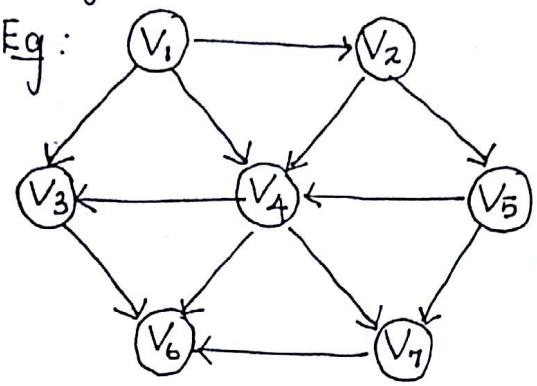


* If the edges have weights, then this additional information is also stored in the cells.

→ Topological Sorting. (determine an acceptable ordering by which to carry out task that depend on one another)

* A topological sort is an ordering of vertices in a directed acyclic graph such that if there is a path from v_i to v_j, then v_j appears after v_i in the ordering.

Eg:



Topological ordering.

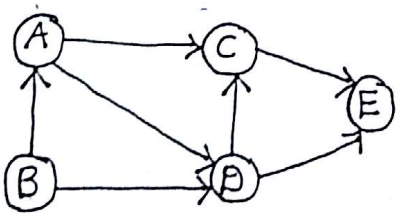
1) $V_1, V_2, V_5, V_4, V_3, V_7, V_6$

2) $V_1, V_2, V_5, V_4, V_7, V_3, V_6$

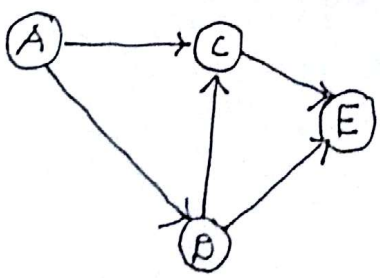
Strategy.

1) Find any vertex with no incoming edges.
2) Print this vertex, and remove it along with its edges from the graph.
3) Repeat the steps to the rest of the graph.

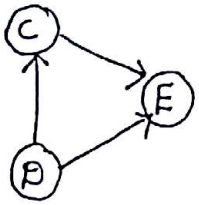
Example.



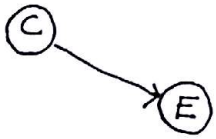
1) Delete 'B' - Print B



2) Delete 'A' - Print 'A'



3) Delete 'D', Print 'D'



4) Delete 'C', Print 'C'

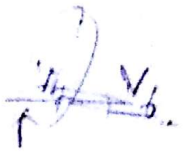
5) Delete 'E', Print E

Topological Ordering B-A-D-C-E

Pseudocode.

```
void Topsort(Graph G)
{
    Queue Q;
    int counter=0;
    Vertex V, W;
    Q = CreateQueue(NumVertex); MakeEmpty(Q);
    for each vertex V
        if (Indegree[V]==0)
            Enqueue(V,Q);
```

```
while ( !IsEmpty(Q))
  {
     V = Dequeue (Q);
     TopNum [ V] = ++Counter ;
     for each w adjacent to v
        if ( --Indegree[w] == 0)
           Enqueue (w, Q);
  }
if ( counter != NumVertex)
     Error("Graph has a cycle");
DisposeQueue (Q);
}
```



i) Indegree is completed for every vertex.

ii) All vertices of indegree 0 are placed on an initially empty queue.

iii) While the queue is not empty, a vertex v is removed and all edges adjacent to v have their indegrees decremented.

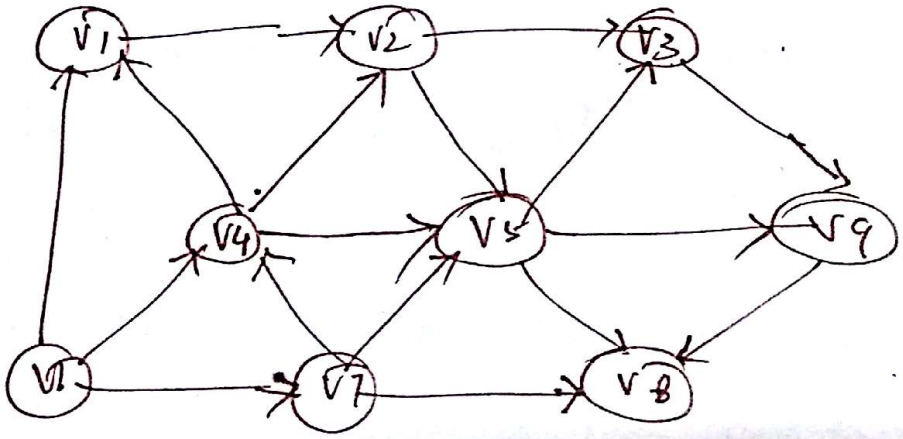
iv) A vertex is put on the queue as soon as the indegree falls at 0.

v) The topological ordering then is the order in which the vertices dequeue.

Vertex	Indegree before dequeue #						
	1	2	3	4	5	6	7
V_1	0	0	0	0	0	0	0
V_2	1	0	0	0	0	0	0
V_3	2	1	1	1	0	0	0
V_4	3	2	1	0	0	0	0
V_5	1	1	0	0	0	0	0
V_6	3	3	3	3	2	1	0
V_7	2	2	2	1	0	0	0
Enqueue	V_1	V_2	V_5	V_4	V_3, V_7		V_6
Dequeue	V_1	V_2	V_5	V_4	V_3	V_7	V_6

Topological sorting

$V_1 - V_2 - V_5 - V_4 - V_3 - V_7 - V_6$



→ Graph Traversals:

1) Depth-First Search (DFS)

2) Breadth-First Search (BFS)

1) Depth-First Search

* Strategy

* DFS starts visiting vertices of a graph at an arbitary vertex by marking it as haue been visited.

* On each iteration, the algorithm proceeds to an unvisited vertex that is adjacent to the one that is currently in.

* This process continues until a dead end - a vertex with no adjacent vertices is encountered.

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

* The algorithm eventually ends after backing up to the starting vertex, with the latter being a dead end.

* If unvisited vertex still remain, DFS must be restarted at any one of them.

Data Structure Used:

* Stack is used to trace the operation of DFS.

* Push a vertex onto the stack when the vertex is reached for the first time.

* Pop a vertex off the stack when it becomes a dead end.

Constructing a DFS Forest

* The traversal's starting vertex serves as the root of the first tree in such a forest.

* Whenever a new unvisited vertex is reached for the first time, it is attached as a child to the vertex from which it is being reached.

* Such an edge is called tree edge because the set of all such edges forms a forest.

Back edge- An edge leading to a previously visited vertex other than its immediate predecessor. It connects a vertex to its ancestor, other than the parent in the DFS forest.

DS
BQ

Algorithm DFS(G)

// Implements a depth-first search traversal of a given graph.

// Input : Graph $G = \{V, E\}$

// Output : Graph G with its vertices marked with consecutive integers in the order they have been first encountered by DFS traversal.

mark each vertex in V with 0 as a mark of being "unvisited"

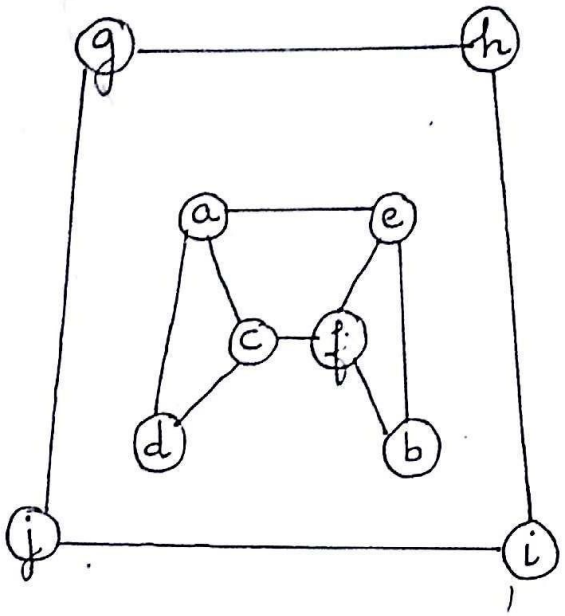
```
count ← 0
for each vertex v in V do
    if v is marked with 0
        dfs(v)
```

dfs(v)

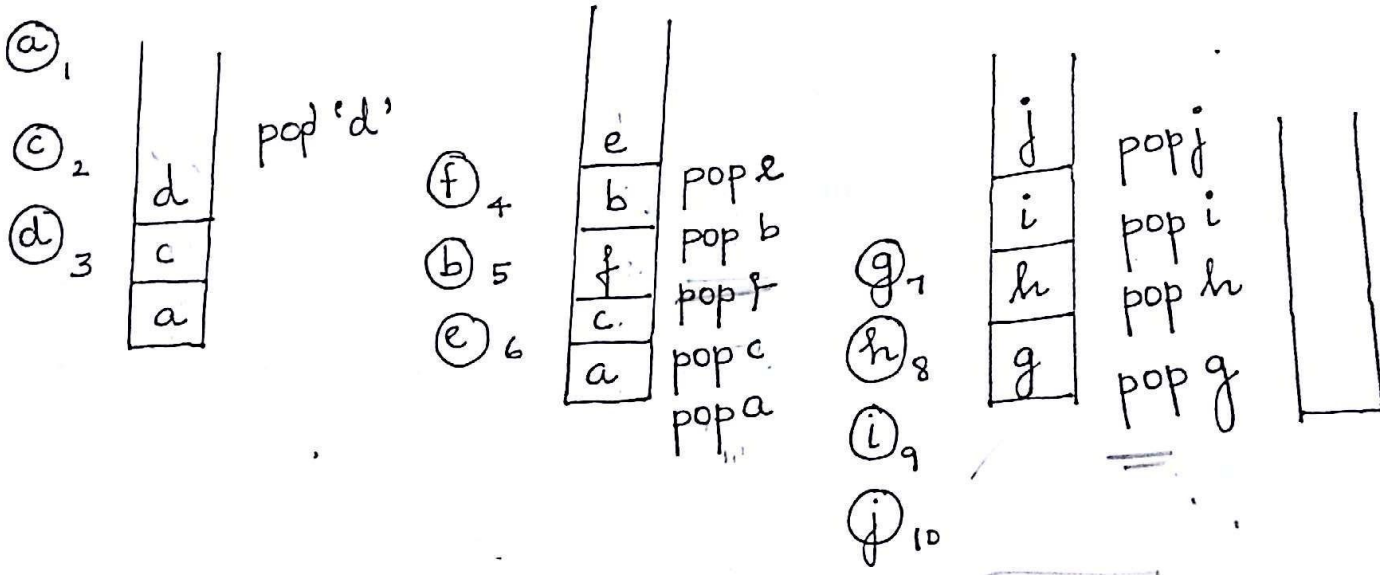
// visits recursively all unvisited vertices connected to vertex 'v' and assigns them the numbers in the order they are encountered via global variable count.

```
count ← count + 1;
mark v with count
for each vertex w in V adjacent to v do
    if w is marked with 0
        dfs(w)
```

Example:

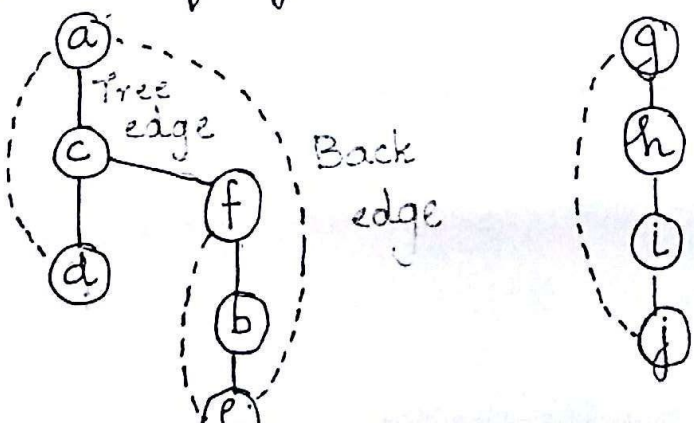


DFS using stack.



DFS traversal : a - c - d - f - b - e - g - h - i - j

DFS forest (ref pg no: 12)



Efficiency of DFS

1) Adjacency matrix representation } = $O(|V|^2)$

2) Adjacency list representation = $O(|V|+|E|)$

$|V|$ and $|E|$ are no. of vertices and edges.

Applications of DFS:

i) Checking connectivity of a graph.

ii) Checking acyclicity of a graph

iii) finding articulation points of a graph.

i) Checking Connectivity

* Start a DFS traversal at an arbitrary vertex and check, after the algorithm halts, whether all the graph's vertices have been visited.

* If they have, the graph is connected, otherwise it is not connected.

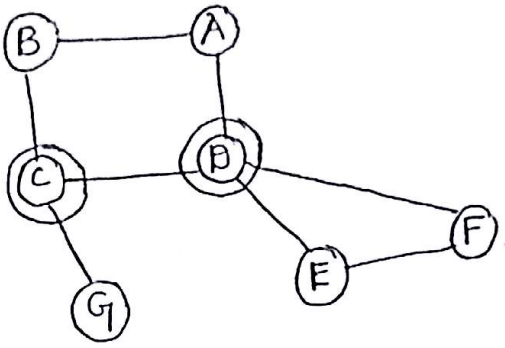
ii) Checking acyclicity:

* Construct a DFS forest for the given graph.

* If the DFS forest does not have back edge, the graph is acyclic.

iii) Finding Articulation points:

* A vertex of a connected graph is said to be its articulation point if its removal with all edges incident to it breaks the graph into disjoint pieces.



2. Breadth First Search:

Strategy:

* BFS visits first all vertices that are adjacent to a starting vertex then all unvisited vertices are two edges apart from it and so on.

* It proceeds until all the vertices in the same connected component as the starting vertex are visited.

* If unvisited vertices still remain, BFS must be restarted at any one of them.

Data structures used

* Queue is used to trace BFS operation

* Queue is initialized with traversal's starting vertex, which is marked as visited.

* On each iteration, all unvisited vertices adjacent to the front ~~view~~ vertex/ are added to queue marking them visited.

* After that the front vertex is removed from queue.

Constructing a BFS Forest.

* The traversal's starting vertex serves as the root of the first tree in forest.

* Whenever a new unvisited vertex is reached for the first time, the vertex is attached as a child to the vertex it is being reached from with an edge called tree edge.

* If an edge leading to a previously visited vertex other than its immediate predecessor is encountered, the edge is noted as CROSS edge.

Algorithm BFS(G)

// Implements a breadth-first search traversal of a given graph

// Input : Graph $G = \{V, E\}$

// Output : Graph G with its vertices marked with consecutive integers in the order they have been visited by the BFS traversal

mark each vertex in V with 0 as a mark of being "unvisited"

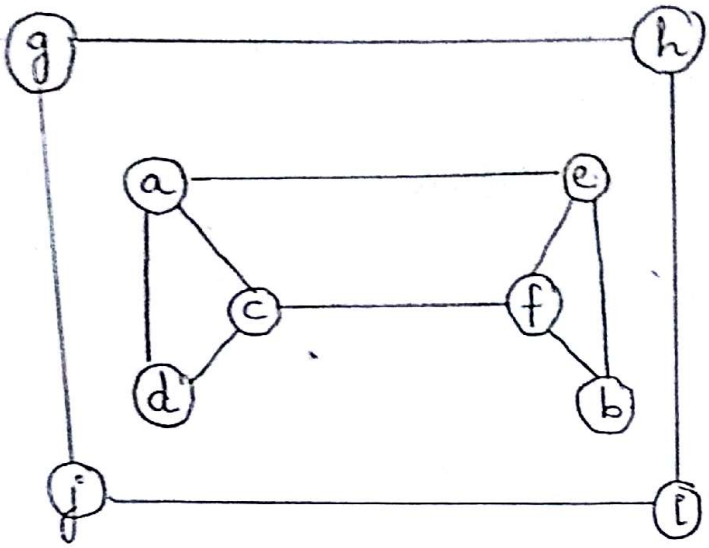
```
count ← 0
for each vertex v in V do
    if v is marked with 0
        bfs (v)
```

bfs(v)

// visits all the unvisited vertices connected to a vertex v and assigns them the numbers in the order they are visited via global variable count

```
count ← count + 1
mark 'v' with count and initialize a queue with v
while the queue is not empty do
    for each vertex w in V adjacent to the
                        front's vertex v do
        if w is marked with 0
            count ← count + 1
            mark w with count
            add w to the queue.
    remove vertex v from front of the queue.
```

Example.



BFS using queue

ⓐ$_1$

ⓒ$_2$

ⓓ$_3$ ⓔ$_4$

front

a	c	d	e

remove a front

ⓕ$_5$

front

c	d	e	f

remove c.

front

d	e	f

remove 'd'

ⓑ$_6$

front

e	f	b

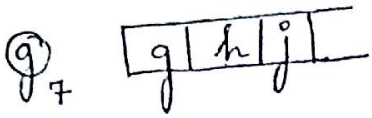
remove e

f	b

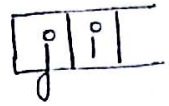
remove f

b

remove b

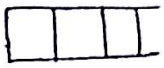


remove h



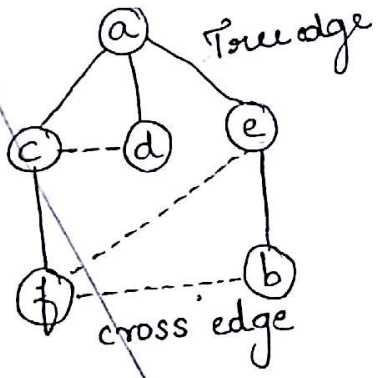
remove j

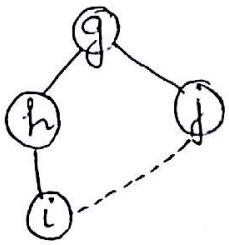
remove i



BFS traversal: a - c - d - e - f - b - g - h - j - i

BFS forest (ref pg no: 12)





Efficiency of BFS

① Adjacency Matrix Representation : $O(|V|^2)$

② Adjacency List Representation : $O(|V|+|E|)$

$|V|$ and $|E|$ are no. of vertices and edges

Applications of BFS

1) Checking connectivity of a graph
2) Checking acyclicity of a graph.
3) Finding a path with the fewest number of edges between two given vertices.

→ start a BFS traversal at one of the two vertices and stop as soon as the other vertex is reached. The simple path from the root of the BFS tree to the second vertex is the path.

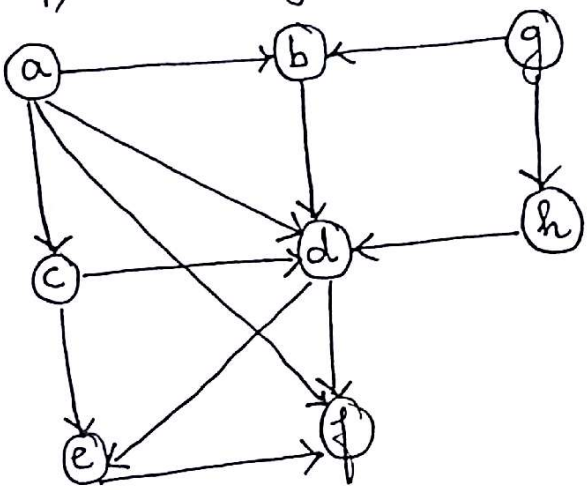
eg: 'a' to 'b' shortest path is a-e-b.

Comparison of BFS and DFS

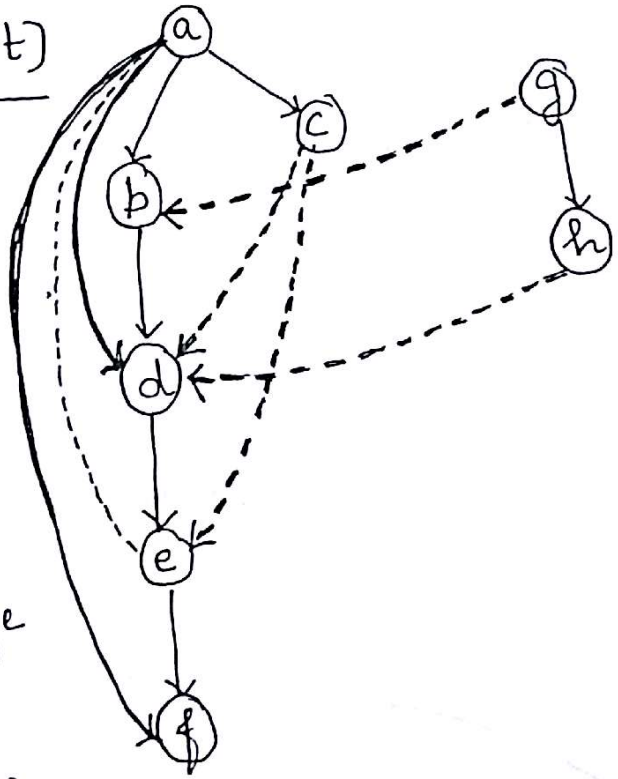
	DFS	BFS
* Data structure	stack	Queue
* No. of vertex ordering	2 ordering (Edge based)	1 ordering (vertex based)
* Edge type	tree edges and back edges	tree edges and cross edges
* Applications	Connectivity, acyclicity, articulation points	Connectivity, acyclicity, minim edges paths
* Efficiency Adj. matrix Adj. List	$O(\|V\|^2)$ $O(\|V\|+\|E\|)$	$O(\|V\|^2)$ $O(\|V\|+\|E\|)$

classification of edges:

1) Forward edge (ancestor to descendent)
2) Backedge (descendent to ancestor)
3) cross edge (no relationship)
   (between siblings, b/w subgraphs)
4) Tree edge



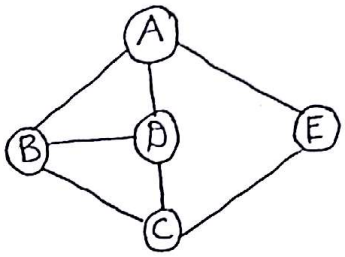
(DFS forest)



→ tree edge
⟶ forward edge
--> Back edge
----> cross edge

Biconnectivity (Application of DFS)

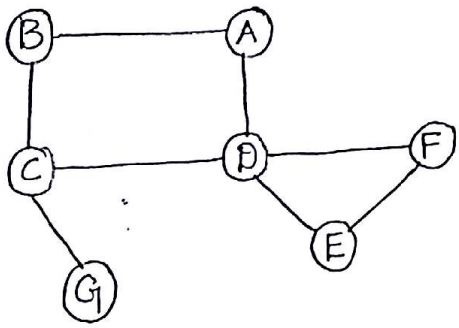
* A connected undirected graph is biconnected if there are no vertices whose removal disconnects the rest of the graph.



biconnected graph.

* If a graph is not biconnected, the vertices whose removal would disconnect the graph are as articulation points.

Example



C and D articulation

DFS traversal

* DFS can be used to find all articulation points in a connected graph

1) Num(v)

* starting at any vertex, perform DFS and number the nodes as they are visited. For each vertex v, we call this preorder number Num(v).

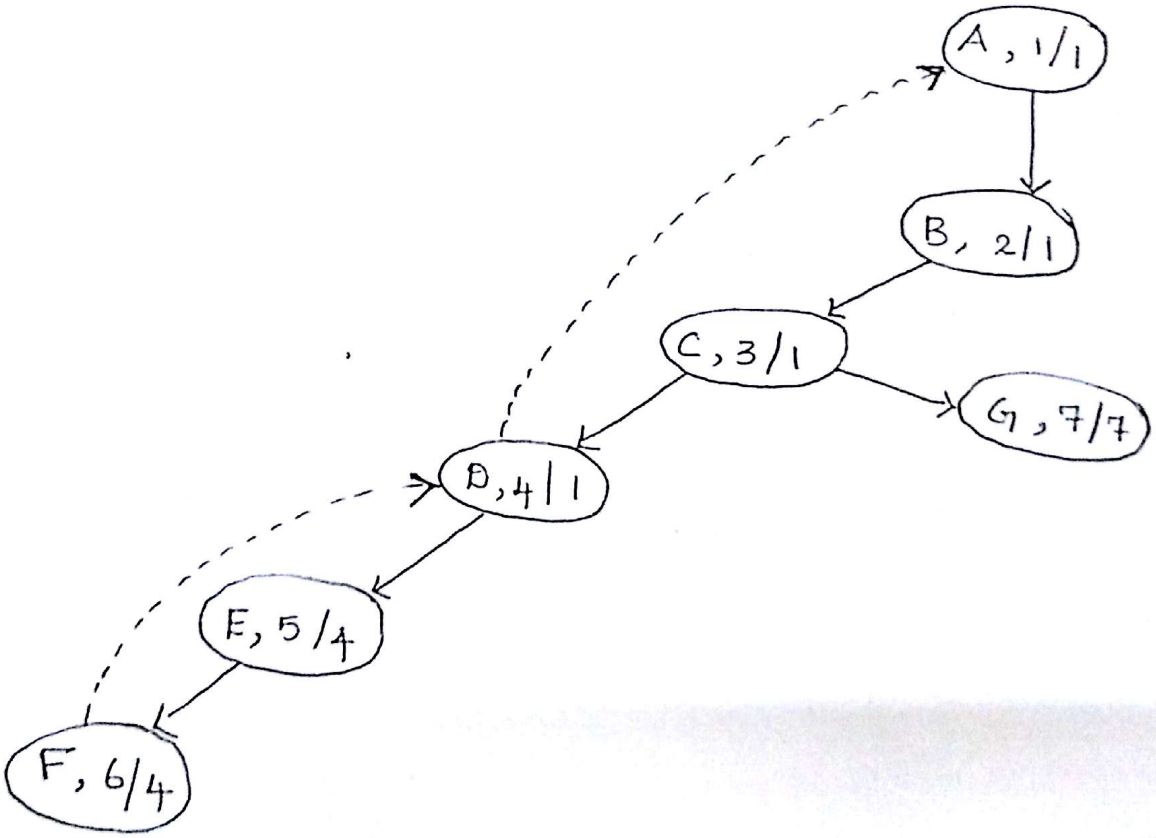
2) Low(v)

We compute the lowest-numbered vertex is reachable from 'v' by taking zero or more tree edges and then possibly one back edge.

// Low(v) is the minimum of

1. Num(v)
2. the lowest Num(w) among all back edges(v,w
3. the lowest Low(w) among all tree edges(v,w

Num and Low for each Vertex



Finding Articulation Points

1) Root is an articulation points if and only if it has more than one child

2) Any other vertex v, is an articulation point if and only if v has some child w such that Low(w) ≥ Num(v)

In above example,

B has a child E, Low(E) ≥ Num(B)

C has a child G, Low(G) ≥ Num(C)

∴ C and B are articulation points

Coding

1) Assign Num to vertices:

```
void AssignNum(Vertex v)
{
    vertex w,
    Num[v] = counter++;
    Visited[v] = True;
    for each w adjacent to v
        if(! visited[w])
        {
            parent[w] = v;
            AssignNum(w);
        }
}
```

2) Compute Low and test for articulation points

```
void AssignLow (Vertex V)
{
   vertex w;
   Low [v] = Num [v];
   for each w adjacent to v
    {
        if (Num [w] > Num [v])
          {
            AssignLow (w);
            if (Low [w] >= Num [v])
              printf ("%v is an articulation point \n",
            Low [v] = Min (Low [v], Low [w]);
          }
        else
          if (Parent [v] != w]
              Low [v] = Min (Low [v], Num [w]);
    }
}
```

Shortest Path Algorithms

1) Single-Source Shortest Path Algorithm:

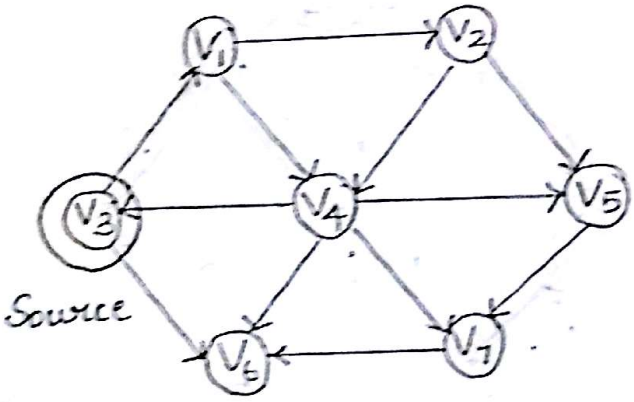
* Given as input a ~~weighted~~ graph, $G = (V, E)$, and a distinguished vertex s, find the shortest weighted path from s to every other vertex in G.

2) All pairs shortest Path Problem:

* To find the shortest paths between all pairs of vertices in the graph.

1) SINGLE SOURCE SHORTEST PATH ALGORITHM

a) UNWEIGHTED SHORTEST PATH



Initial Config of Table

V	known	d_v	P_v
V_1	0	∞	0
V_2	0	∞	0
V_3	0	∞	0
V_3	0	∞	0
V_4	0	∞	0
V_5	0	∞	0
V_6	0	∞	0
V_7	0	∞	0

strategy:

1) Create a queue, where the size of the queue is equivalent to the number of vertices of the graph.

2) Select the source vertex 's' and insert the source vertex 's' into the queue.

3) Find the unknown vertices adjacent to the source vertex 's' and update the distance d_v. To perform this, BFS is used with 3 parameters.

* known - to determine whether the shortest path from 's' to that vertex 'v' is identified or not.

d_v - distance of the path from s to v

P_v - vertex through which v may be reached.

4) Repeat steps until all the vertices in graph G are found to be known and get processed.

	Initial state			V_3 dequeued			V_1 dequeued			V_6 dequeued		
Vertex	known	d_v	P_v	known	d_v	P_v	known	d_v	P_v	known	d_v	P_v
V_1	0	∞	0	0	1	V_3	1	1	V_3	1	1	V_3
V_2	0	∞	0	0	∞	0	0	2	V_1	0	2	V_1
V_3	0	0	0	1	0	0	1	0	0	1	0	0
V_4	0	∞	0	0	∞	0	0	2	V_1	0	2	V_1
V_5	0	∞	0	0	∞	0	0	∞	0	0	∞	0
V_6	0	∞	0	0	1	V_3	0	1	V_3	1	1	V_3
V_7	0	∞	0	0	∞	0	0	∞	0	0	∞	0
Q	V_3			V_1, V_6			V_6, V_2, V_4			V_2, V_4		

	V_2 dequeued			V_4 dequeued			V_5 dequeued			V_7 dequeued		
Vertex	known	d_v	P_v	known	d_v	P_v	known	d_v	P_v	known	d_v	P_v
V_1	1	1	V_3	1	1	V_3	1	1	V_3	1	1	V_3
V_2	1	2	V_1	1	2	V_1	1	2	V_1	1	2	V_1
V_3	1	0	0	1	0	0	1	0	0	1	0	0
V_4	0	2	V_1	1	2	V_1	1	2	V_1	1	2	V_1
V_5	0	3	V_2	0	3	V_2	1	3	V_2	1	3	V_2
V_6	1	1	V_3	1	1	V_3	1	1	V_3	1	1	V_3
V_7	0	∞	0	0	3	V_4	0	3	V_4	1	3	V_2
Q	V_4, V_5			V_5, V_7			V_7			empty.		

Pseudocode

```
Void unweighted (Table T) /* Assume T is initialized
{
    Queue Q;
    Vertex V, W;
    Q = CreateQueue (NumVertex);
    MakeEmpty (Q);
    Enqueue (S, Q);
    while( ! IsEmpty (Q))
    {
        V = Dequeue (Q);
        T[V]. known = True;
        for each w adjacent to v
            if (T[W]. Dist == Infinity)
            {
                T[W]. Dist = T[V]. Dist + 1;
                T[W] .Path = V;
            } Enqueue (W, Q)
}
```

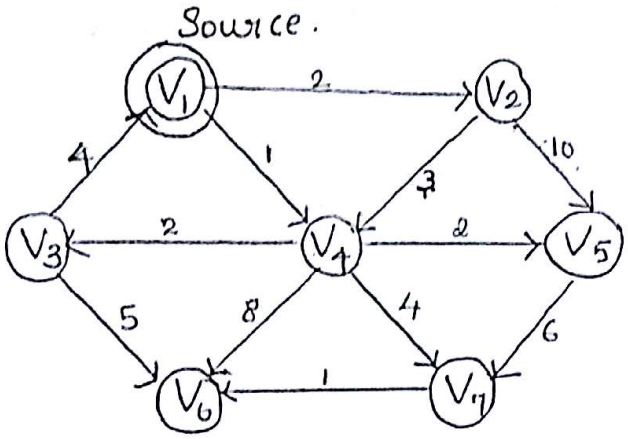
```
    DisposeQueue(Q);
}
```

Efficiency:

1) If adj. matrix is used : $O(|V|^2)$
2) If adj. list is used : $O(|E|+|V|)$

Weighted Shortest Path

DIJKSTRA's ALGORITHM: The method to solve the single-source shortest-path problem is known as Dijkstra's algorithm. It is a prime example of a greedy algorithm.



Strategy:

1. Create a table with known P_v and d_v parameters, where the size of the table is equivalent to no. of vertices.
2. Select a vertex v which has smallest d_v among all unknown vertices and set the shortest path from s to v as known.

3. The adjacent vertices w to v is located and d_w is set as $d_w = d_v + c_v$, if the new sum is less than the existing d_w. That is, in every stage d_w is updated only if there is an improvement in the path distance.

4. Repeat steps 2 and 3 until all vertices are classified under known.

rtex	Initial state			V_1 dequeued			V_4 dequeued			V_2 dequeued		
	known	d_v	P_v	known	d_v	P_v	known	d_v	P_v	known	d_v	P_v
V_1	0	0	0	1	0	0	1	0	0	1	0	0
V_2	0	∞	0	0	2	V_1	0	2	V_1	1	2	V_1
V_3	0	∞	0	0	∞	0	0	3	V_4	0	3	V_4
V_4	0	∞	0	0	1	V_1	1	1	V_1	1	1	V_1
V_5	0	∞	0	0	∞	0	0	3	V_4	0	3	V_4
V_6	0	∞	0	0	∞	0	0	9	V_4	0	9	V_4
V_7	0	∞	0	0	∞	0	0	5	V_4	0	5	V_4
Q	V_1			V_4, V_2			V_2, V_3, V_5, V_7, V_6			V_3, V_5, V_7, V_6		

Vertex	V_3 dequeued			V_5 dequeued			V_7 dequeued			V_6 dequ	
	known	dv	Pv	known	dv	Pv	known	dv	Pv	known	d
V_1	1	0	0	1	0	0	1	0	0	1	0
V_2	1	2	V_1	1	2	V_1	1	2	V_1	1	2
V_3	1	3	V_4	1	3	V_4	1	3	V_4	1	3
V_4	1	1	V_1	1	1	V_1	1	1	V_1	1	1
V_5	0	3	V_4	1	3	V_4	1	3	V_4	1	3
V_6	0	8	V_3	0	8	V_3	0	6	V_7	1	6
V_7	0	5	V_4	0	5	V_4	1	5	V_4	1	5
Q	V_5, V_7, V_6			V_7, V_6			V_6			Empt	

Pseudocode

1) Table Initialization Routine

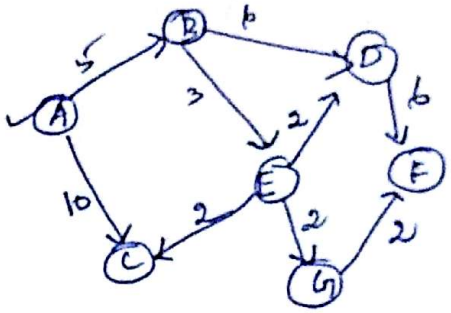
```
void initTable( Vertex Start, Graph G, Table T)
{
  int i;
  ReadGraph (G,T);
  for(i=0; i<NumVertex; i++)
    {
      T[i].known = False;
      T[i].Dist = Infinity;
      T[i].path = NotA Vertex;
    }
  T[start].dist=0;
}
```

2) Pseudocode for dijkstra's Algorithm.

```
void Dijkstra ( Table T)
{
    Vertex V, W;
    for ( ; ;)
    {
        V = smallest unknown distance vertex;
        if (V == NotA Vertex)
            break;
        T[V]. known = True;
        for each w adjacent to V;
            if ( ! T[W]. known)
                if (T[V]. Dist + CVW < T[W]. Dist)
                {
                    Decrease (T[W]. Dist to T[V]. Dist +
                                                CVW);
                    T[W]. path = V;
                }
    }
}
```

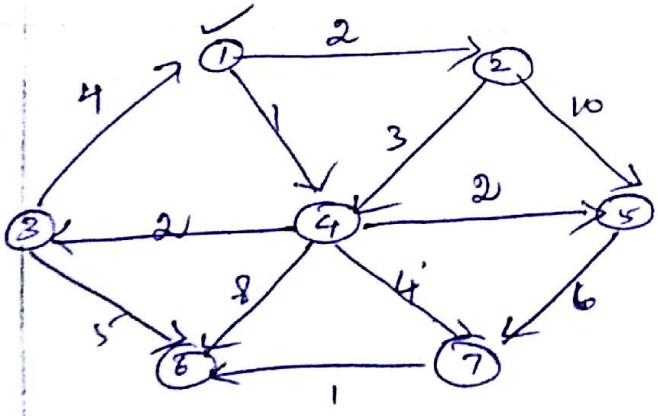
3) Routine to print actual Path

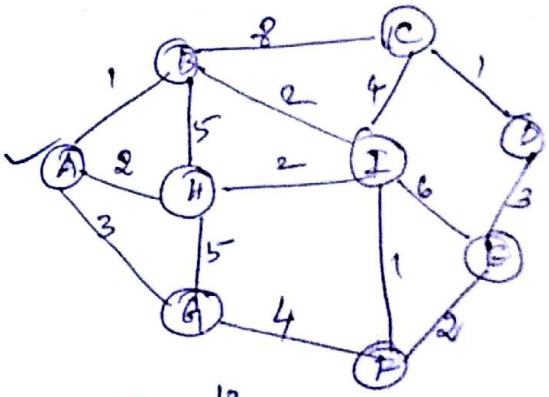
```
void PrintPath( Vertex V, Table T)
{
    if ( T[v]. Path != NotAVertex)
    {
        PrintPath (T[v].Path, T);
        printf ("to");
    }
    printf ("%v", V);
}
```

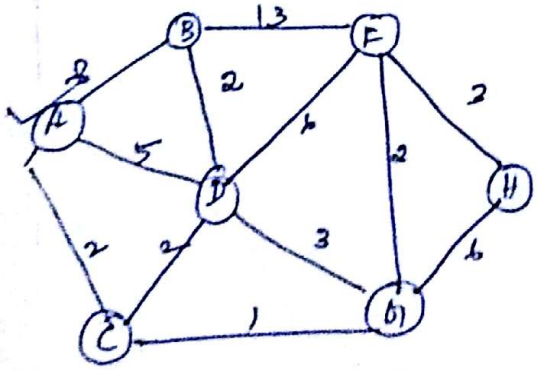


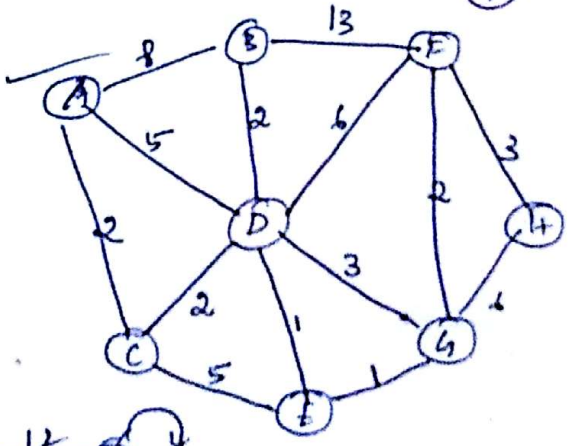
Efficiency of Algorithm

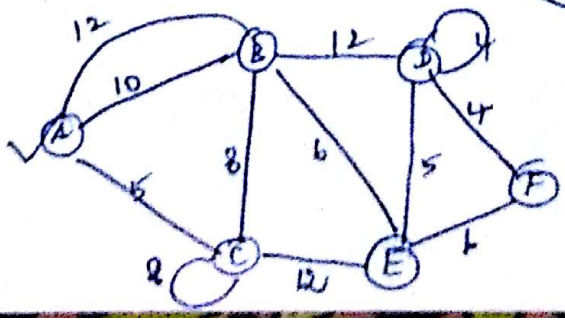
Running Time = $O(|E| + |V|^2) = O(|V|^2)$











Minimum Spanning Tree

Weight of tree : sum of weights of all its edges.

Spanning tree : Spanning tree of a graph is a tree with all vertices of the graph and a subset of edges connecting them without any cycles.

Minimum Spanning tree : of an undirected graph G is a tree formed from graph edges that connects all vertices G at lowest total cost

- Minimum Spanning tree exists if and only if G is connected.

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.

* A minimum spanning tree exists if and only if G is connected.

* No of edges in minimum spanning tree is

|V| - 1

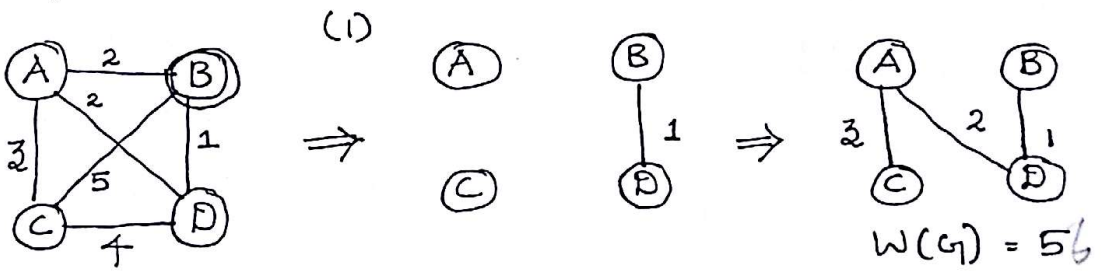
Minimum spanning tree is

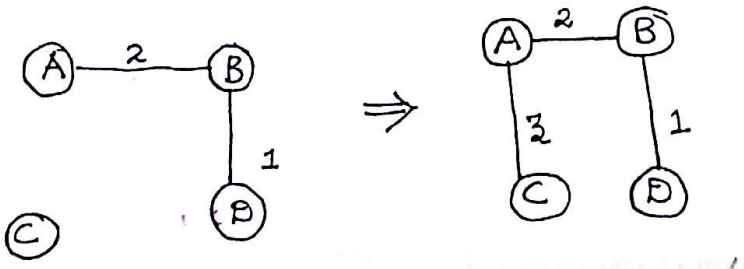
a tree - because it is acyclic

spanning - because it covers every vertex.

minimum - because it covers edges with lowest cost.

Example:





Applications of MST

1. Wiring a house with minimum cable.
2. Cheapest cost tour of a travelling salesman
3. Networking the PC's with low cost

Minimum Spanning tree Algorithm.

1) Prim's Algorithm
2) kruskal Algorithm

} GREEDY ALGORITHMS

At each step, grap the best alternative available yielding to global optimal solution.

1) PRIM'S ALGORITHM:

* In this method, minimum spanning tree is constructed in successive stages.

* One node is picked as the root and an edge is added (i.e) an associated vertex is added to the tree, until the vertices (all vertices) are present in the tree with |V|-1 edges.

strategy:

i) One node is picked as root node (u)

ii) At each stage, choose a new vertex v, by considering an edge (u,v) with minimum cost among all edges from u, where u is already in the tree and 'v' is not in the tree.

iii) Prim's algorithm table is constructed with three parameters

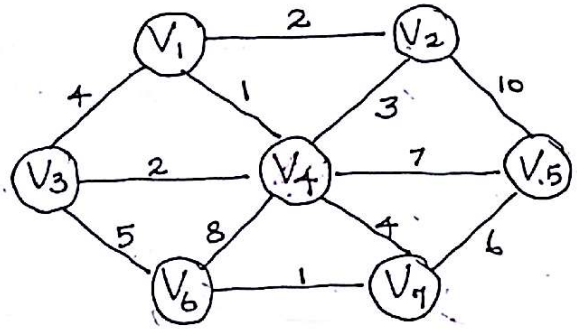
known - vertex is added in the tree or not

d_v - weight of the shortest arc connecting 'v' to a known vertex

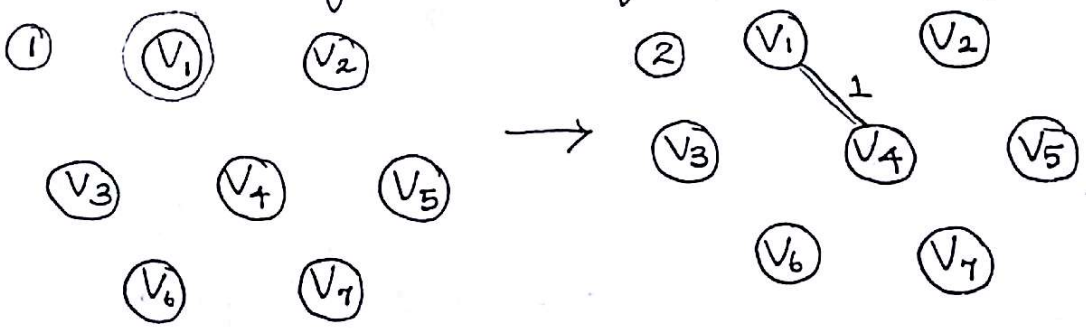
P_v - last vertex which causes a change in d_v

iv) After selecting the vertex v, the update rule is applied for each unknown w adjacent to v. The rule is $d_w = \min(d_w, c_{wv})$, that is if more than one path exist between v to w then d_w is updated with minimum cost.

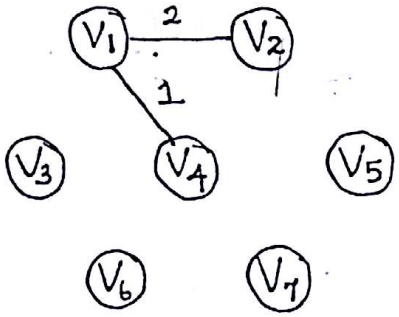
Example.,



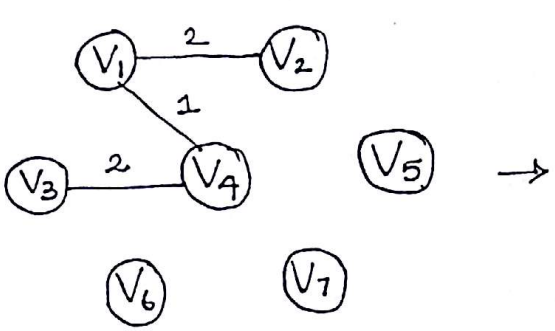
Prim's algorithm after each stage



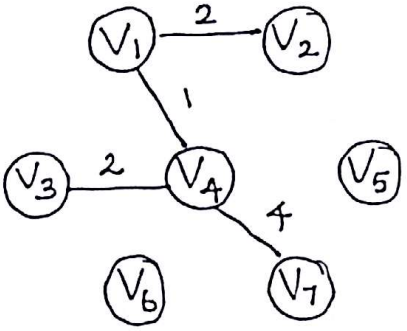
(3)

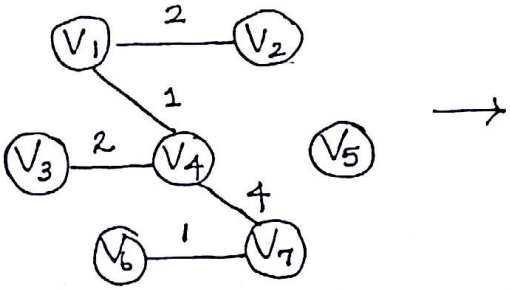


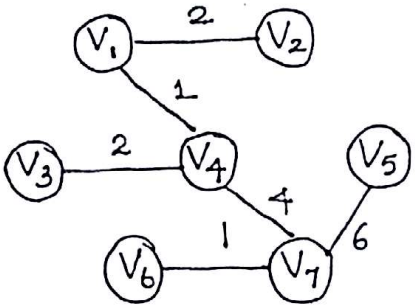
(4)



(5)







Vertex	Initial state			V_1 dequeued			V_4 dequeued			V_2 dequeued		
	known	d_v	P_v	known	d_v	P_v	known	d_v	P_v	known	d_v	P_v
V_1	0	0	0	1	0	0	1	0	0	1	0	0
V_2	0	∞	0	0	2	V_1	0	2	V_1	1	2	V_1
V_3	0	∞	0	0	4	V_1	0	2	V_4	0	2	V_4
V_4	0	∞	0	0	1	V_1	1	1	V_1	1	1	V_1
V_5	0	∞	0	0	∞	0	0	7	V_4	0	7	V_4
V_6	0	∞	0	0	∞	0	0	8	V_4	0	8	V_4
V_7	0	∞	0	0	∞	0	0	4	V_4	0	4	V_4
Q	V_1			V_4, V_2, V_3			V_2, V_3, V_1, V_5, V_6			V_3, V_7, V_5, V_6		

Vertex	V_3 dequeued known dv Pv	V_7 dequeued known dv Pv	V_5 dequeued known dv Pv	V_6 dequeued known dv Pv
V_1	1 0 0	1 0 0	1 0 0	1 0 0
V_2	1 2 V_1	1 2 V_1	1 2 V_1	1 2 V_1
V_3	1 2 V_4	1 2 V_4	1 2 V_4	1 2 V_4
V_4	1 1 V_1	1 1 V_1	1 1 V_1	1 1 V_1
V_5	0 7 V_4	0 6 V_7	0 6 V_7	1 6 V_7
V_6	0 5 V_3	0 1 V_7	1 1 V_7	1 1 V_7
V_7	0 4 V_4	1 4 V_4	1 4 V_4	1 4 V_4
Q	V_7, V_5, V_6	V_5, V_6	V_6	Empty

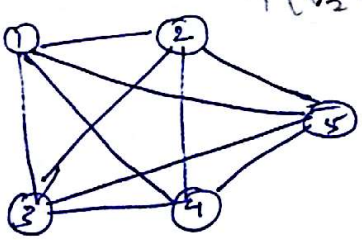
Total Cost of MST Constructed = 16

Efficiency:

Running time for dense graph : $O(|V|^2)$ → number of edges close to the maximal no of edges

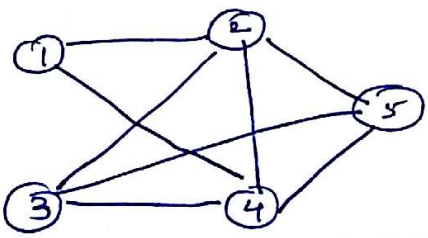
Running time for sparse graph : $O(|E|\log|V|)$ → number of edges close to the minimal no of edges

* A graph with relatively few edges is sparse and graph with many edges is dense.

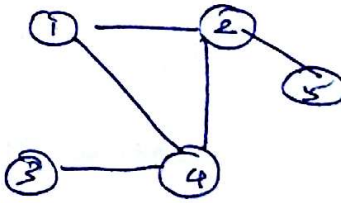


complete graph

$\frac{n(n-1)}{2}$ edges



Dense graph



Sparse Graph

2. KRUSKAL'S ALGORITHM:

* A greedy technique which continually select the edges in order of smallest weight and accept an edge if it does not cause cycle.

Forest

* Kruskal's Algorithm maintains a forest a collection of tree.
* Initially there are |V| single-node trees.
* Adding an edge merges two trees into one.
* When the algorithm terminates, there is only one tree, and this is the minimum spanning tree.

Strategy:

i) Sort the graph edges in non-decreasing order of their weights.

ii) Start with an empty subgraph.

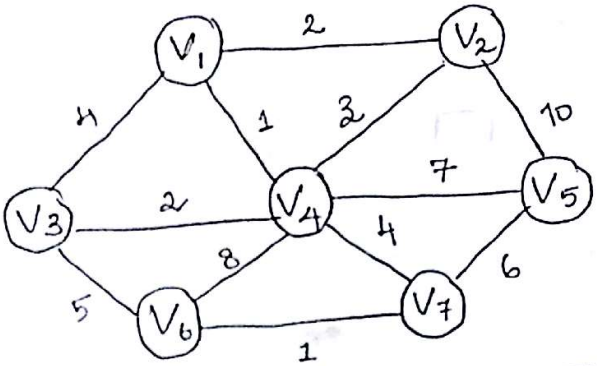
iii) Scan the sorted list adding the next stage on the list is the current subgraph if such inclusion does not create a cycle skip the edge otherwise.

Prim's Algorithm

* This algorithm is to obtain minimum spanning tree by selecting the adjacent vertices of already selected vertices.

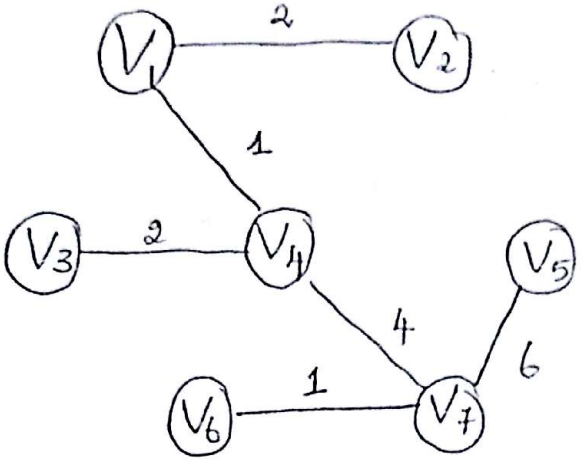
Kruskal's Algorithm

* This algorithm is to obtain minimum spanning tree but it is not necessary to choose adjacent vertices of already selected vertices.

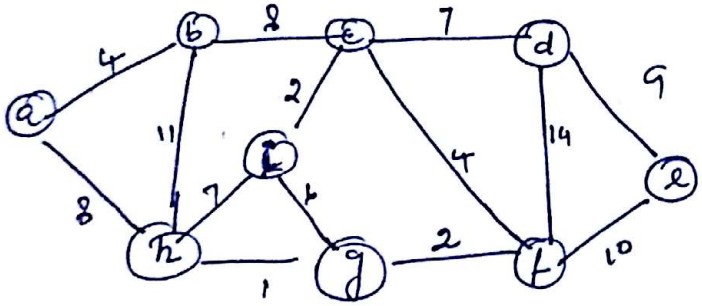


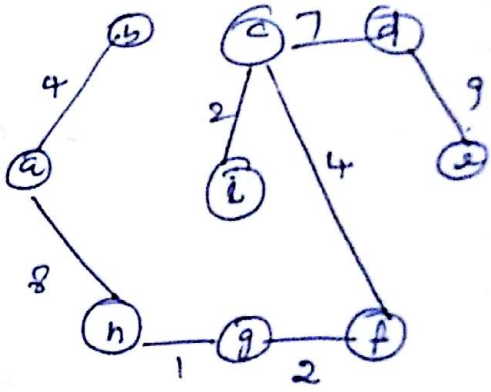
Selected tree Vertices	Cost Accepted / rejected.
V_1, V_4	1 (Accepted)
V_6, V_7	1 (Accepted)
V_1, V_2	2 (Accepted)
V_3, V_4	2 (Accepted)
V_2, V_4	3 (Rejected)
V_1, V_3	4 (Rejected)
V_4, V_7	4 (accepted)

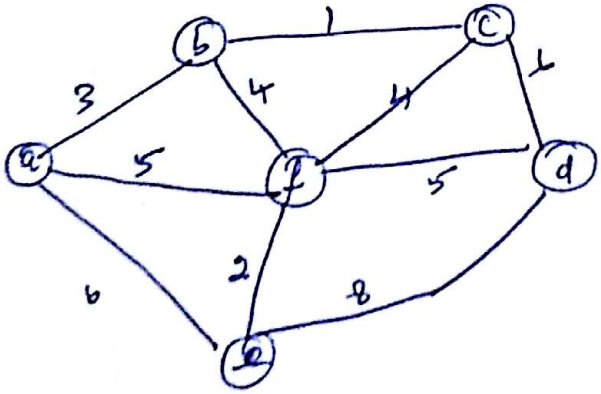
V_2, V_6	5 (Rejected)
(V_6, V_7)	6 (accepted)
V_4, V_5	7 (Rejected)
V_4 V_6	8 (Rejected)
V_2, V_5	10 (Rejected)

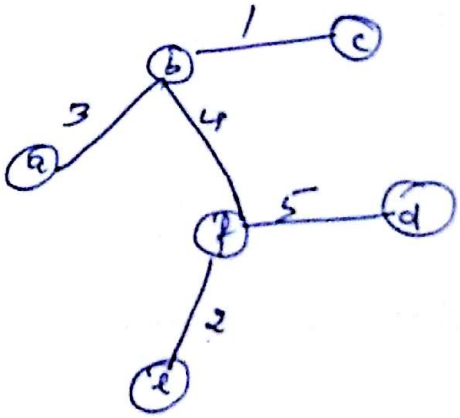


(Draw in successive stage









Algorithm kruskal(G_1)

// kruskal's algorithm for constructing a minimum spanning tree

// Input : A weighted graph $G_1 = \{V, E\}$

// Output : E_T, the set of edges composing a minimum spanning tree of G_1.

Sort E in non-decreasing order of the edges weights

$w(e_{i_1}) \leq \ldots w(e_{i|E|})$

$E_T \leftarrow \phi$; counter = 0 // initialize the set of tree edges and its size

$k \leftarrow 0$; // initialize the number of processed edges

while counter < |V| - 1

$k \leftarrow k+1$

if $E_T \cup \{e_{ik}\}$ is acyclic

$E_T \leftarrow E_T \cup \{e_{ik}\}$ counter $\leftarrow$ counter + 1

return E_T

Analysis

④ $(|E| \log |E|)$

E → total number of edges in the graph.

Efficiency

BFS: Adjacency Matrix representation $O(|V|^2)$

Adjacency list representation $O(|V|+|E|)$

BFS: Adjacency Matrix representation $O(|V|^2)$

Adjacency list representation $O(|V|+|E|)$

Unweighted shortest path:

Adjacency Matrix representation $O(|V|^2)$

Adjacency list representation $O(|V|+|E|)$

Dijkstra's Algorithm:

Running time $O(|V|^2)$

Prim's Algorithm:

Running time for dense graph $O(|V|^2)$

Running time for sparse graph $O(|E| \log$

Kouskal's Algorithm:

Running time $O(|E| \log |E|)$