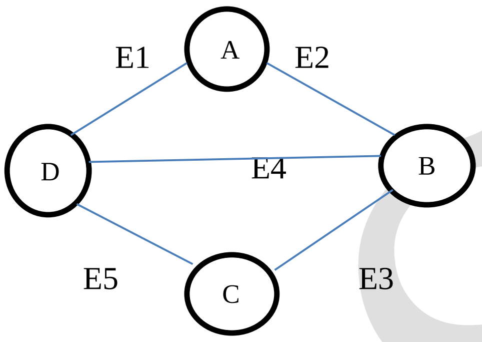
GRAPHS

DEFINATION

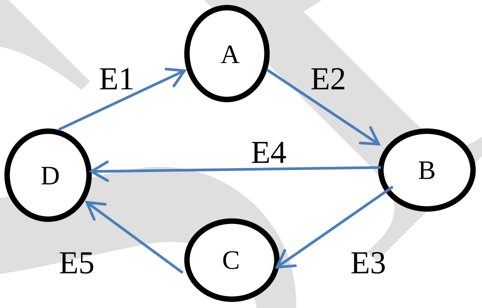
A Graph G is a set of vertices (V) and set of edges(E).The set of V is finite, nonempty set of vertices. The set E is a set of pair of vertices representing edges.
Any graph is denoted by G= (V,E)



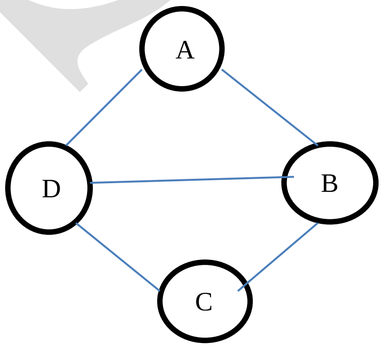
V= {A, B, C, D} E={E1,E2,E3,E4,E5}
G= {{A, B, C, D},{E1,E2,E3,E4,E5}}

TYPES OF GRAPH

1) **Directed Graph:** In directed graph the direction are shown on the edges.

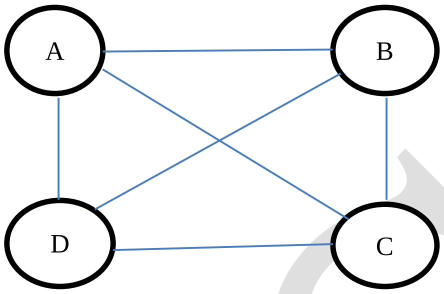


2) **Undirected Graph:** In the undirected graph edges are not in ordered.

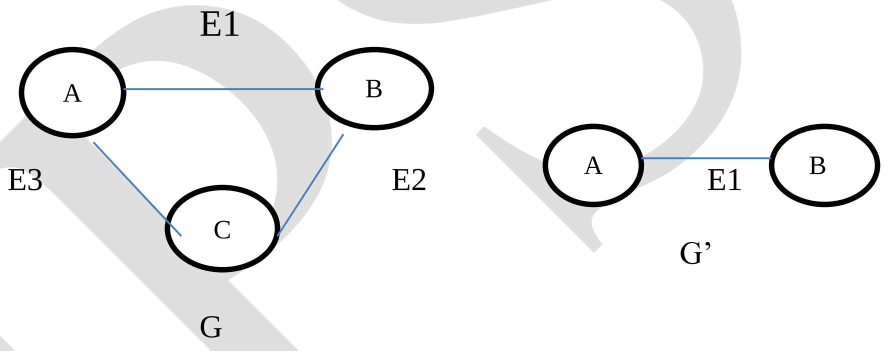


PROPERTIES OF GRAPH:

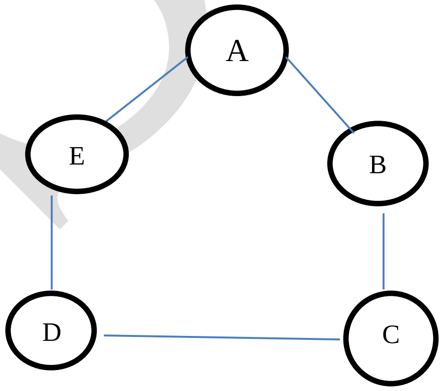
1) **Complete graph:** In an undirected graph of n vertices consist of n(n-1)/2 edges then it is called as complete graph.



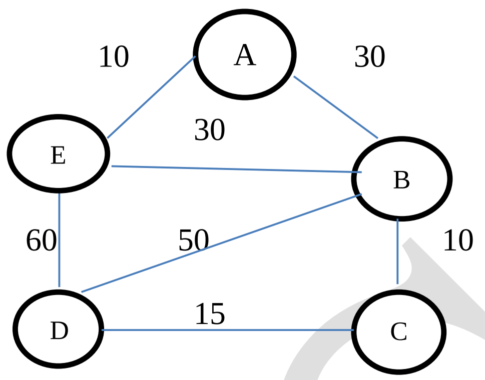
2) **Sub graph:** A graph G’ of graph G is a graph such that the set of vertices & set of edges of G’ are proper subset of the set of the edges of G.



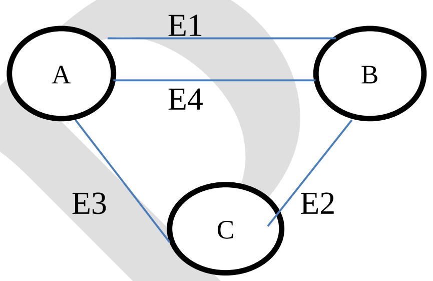
3) **Connected graph:** When undirected graph is said to be connected if for every pair of distinct vertices Vi & Vj in V(G) there is a graph from Vi to Vj in G.



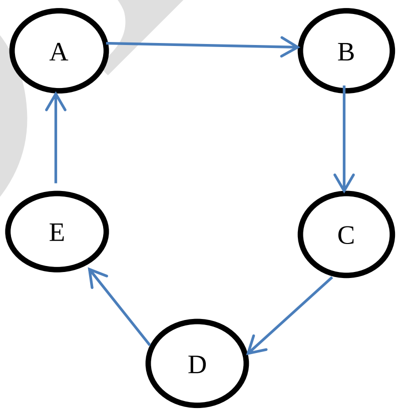
4) **Weighted graph:** A weighted graph is a graph which consist of weights along with its edges.



5) **Multiple edges:** A graph G contains multiple edges if pair of vertices is connected by more than one edges.

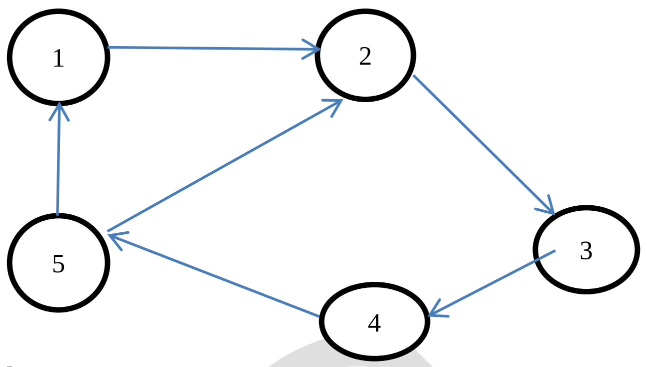


6) **Path:** A path is denoted using sequence of vertices & there exists on edge from one vertex to next vertex.



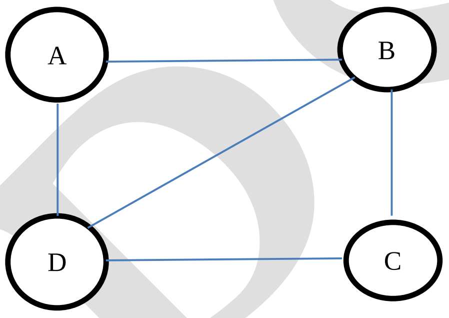
7) **Loop:** An edge E is called loop if it has identical end points that is e=(V1,V1)

8) **Cycle:** A closed walked loop the graph with repeated vertices that is same starting & ending vertex is called a cycle.



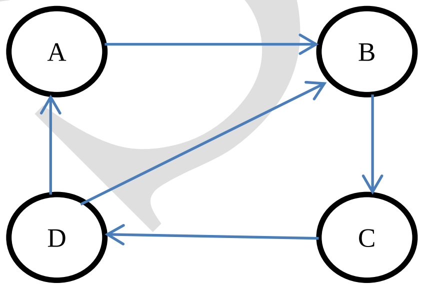
Cycle 2,3,4,5,1, 2 and 2,3,4,5, 2

9) **Degree of the vertex:** In a given graph G degree of the vertex is defined as the number of edges incident on V.



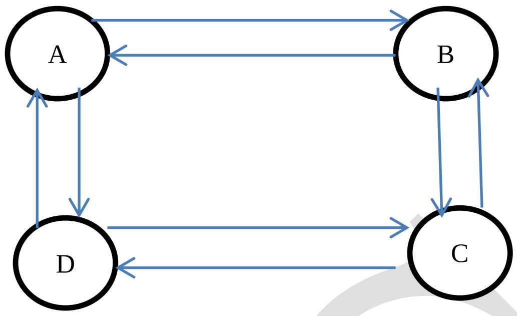
A=2,B=3,C=2,D=3

10) **Indegree & outdegree:** Indegree of the vertex is the number of edges that incident to the vertex & outdegree of vertex is total number of edges that are going away from the vertex.



Nodes	**Indegree**	**Outdegree**
A	1	1
B	2	1
C	1	1
D	1	2

11) **Strongly connected graph:** A graph G is strongly connected if for A & B there exist a path from A to B & from B to A.



Depth First Search (DFS):-

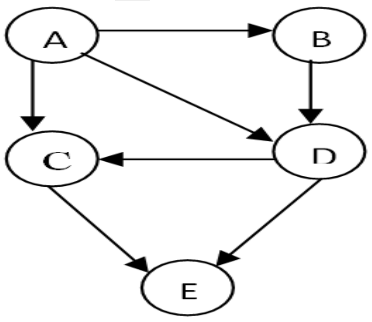
DFS is traversal technique. It is like preorder traversal of tree. Traversal can start from any vertex, say V_i. V_i is visited and then all vertices adjacent to V_i are traversed recursively using DFS.

DFS process start with node A as start node and mark and unmark adjacent node of A is selected and mark to become new start node. The traversal continue in the graph until current path ends at the node with outdegree 0 or at the node with all adjacent node which still has not mark adjacent not mark and continues marking until all nodes are mark. It required stack to be used.

In DFS following steps are follows:-

1. START
2. Initialized all node to the ready state
3. Push starting node A on to the stack and change the status to the waiting state
4. Repeat step 5 & step 6 until stack is empty
5. Pop stack element N(If N is not Process) & process N and change its status to the process state
6. Push all the neighbors of N into stack and change its status to waiting state
7. STOP

Example:-



DFS Traversal: A-B-D-C-E

DFS

```
#include<stdio.h>
#define MAX 5

void dfs(int[MAX][MAX],int[MAX],int);

void main()
{
	int adj[MAX][MAX];
	int visited[MAX]={0},i,j;

	printf("\nenter the adjacency matrix:\n");
	for(i=0;i<MAX;i++)
	{
		for(j=0;j<MAX;j++)
		{
			scanf("%d",&adj[i][j]);
		}
	}

	printf("\nDFS Traversal is ");
	dfs(adj,visited,0);
	printf("\n");

}

void dfs(int adj[][MAX],int visited[],int start)
{
	int stack[MAX];
	int tos=-1,i;
	printf("%c ",start+65);
	visited[start]=1;
	stack[++tos]=start;

	while(tos!=-1)
	{
		start=stack[tos];
		for(i=0;i<MAX;i++)
		{
			if(adj[start][i]==1 && visited[i]==0)
			{
				stack[++tos]=i;
				printf("%c ",i+65);
				visited[i]=1;
				break;
			}
		}
```

```
                    if(i==MAX)
                              tos--;
          }

}
```

Breadth First Search (BFS):

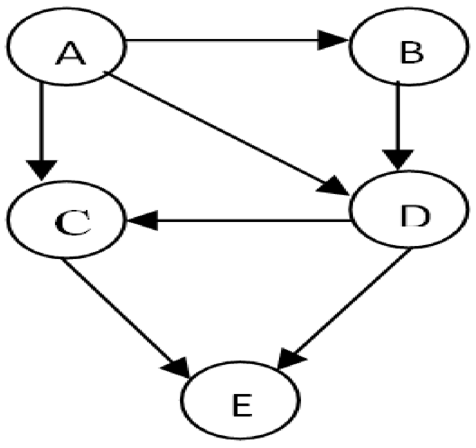
It is another popular approach used for visiting the vertices of a graph. Traversing using BFS for graph is used to find shortest distance between starting node and any node which is called as destination.

Stating at node ‘v’ and distance is calculated by examine all incident edges to node ‘v’ and then removing on to an adjacent node ‘w’ and repeating process. The traversal continues until all nodes in the graph have been examined. BFS technique used queue data structure.

In BFS following steps are follows:-

1. START
2. Initialized all node to the ready state
3. Place starting node ‘A’ in queue and change its status to the waiting state
4. Repeat step 5 & step 6 until queue is empty
5. Remove front node n from queue. Process n and change status of n to the process state
6. Add all neighbors of n to the rear of queue that are in the ready state and change their status to the waiting state.
7. STOP

Example:-



BFS Traversal: A-B-C-D-E

BFS

```
#include<stdio.h>

#define MAX 5

void bfs(int[MAX][MAX],int[MAX],int);

void main()
{
	int adj[MAX][MAX];
	int visited[MAX]={0},i,j;
	clrscr();
	printf("\nenter the adjacency matrix: ");
	for(i=0;i<MAX;i++)
	{
		for(j=0;j<MAX;j++)
		{
			scanf("%d",&adj[i][j]);
		}
	}

	printf("\nBFS Traversal is ");
	bfs(adj,visited,0);
	printf("\n");
	getch();
}

void bfs(int adj[][MAX],int visited[],int start)
{
	int queue[MAX];
	int rear=-1,front=-1,i;
	queue[++rear]=start;
	visited[start]=1;
	while(rear!=front)
	{
		start=queue[++front];
		printf("%c ",start+65);
		for(i=0;i<MAX;i++)
		{
			if(adj[start][i]==1 && visited[i]==0)
			{
				queue[++rear]=i;
				visited[i]=1;
			}
		}
	}
}
```

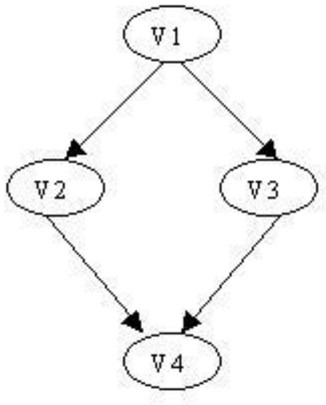
Topological sort:

An ordering of the vertices in a directed acyclic graph, such that: If there is a path from ***u*** to ***v***, then ***v*** appears after ***u*** in the ordering.

The graphs should be **directed:** otherwise for any edge (u,v) there would be a path from ***u*** to ***v*** and also from ***v*** to ***u***, and hence they cannot be ordered.

The graphs should be **acyclic**: otherwise for any two vertices ***u*** and ***v*** on a cycle
u would precede ***v*** and ***v*** would precede ***u.***

The ordering may not be unique:



V1, V2, V3, V4 and V1, V3, V2, V4 are legal orderings

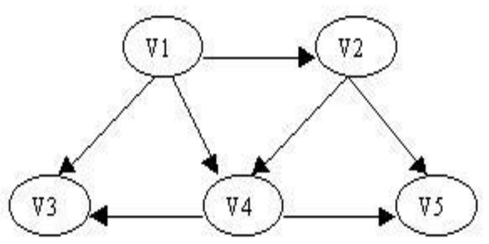
Degree of a vertex U: the number of edges (U,V) - outgoing edges
Indegree of a vertex U: the number of edges (V,U) - incoming edges

The algorithm for topological sort uses "indegrees" of vertices.

Algorithm

1. Compute the indegrees of all vertices
2. Find a vertex **U** with indegree 0 and print it (store it in the ordering)
   If there is no such vertex then there is a cycle and the vertices cannot be ordered. Stop.
3. Remove **U** and all its edges **(U, V)** from the graph.
4. Update the indegrees of the remaining vertices.
5. Repeat steps 2 through 4 while there are vertices to be processed.

Example



Compute the indegrees:

V1: 0
V2: 1
V3: 2
V4: 2
V5: 2

Find a vertex with indegree 0: V1

Output V1 , remove V1 and update the indegrees:

Sorted: V1
Remove edges: (V1,V2) , (V1, V3) and (V1,V4)
Updated indegrees:

V2: 0
V3: 1
V4: 1
V5: 2

The process is depicted in the following table:

	Indegree					
Sorted à		V1	V1,V2	V1,V2,V4	V1,V2,V4,V3	V1,V2,V4,V3,V5
V1	0					
V2	1	0				
V3	2	1	1	0		
V4	2	1	0			
V5	2	2	1	0	0	

One possible sorting: V1, V2, V4, V3, V5
Another sorting: V1, V2, V4, V5, V3