MODULE -4

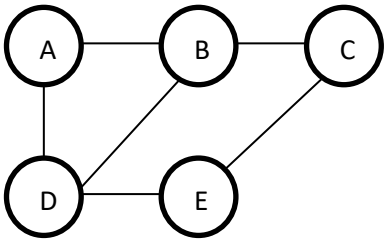
Non Linear Data Structures – Graphs: Graph Terminologies - Vertex, Edge, Adjacent vertices, Self-loop, Parallel edges, Isolated vertex, Degree of vertex, Pendant vertex, Subgraph, Paths and Cycles, - Types of Graphs - Directed, Undirected, Simple, Complete, Cyclic, Acyclic, Bipartite, Complete Bipartite, Connected, Disconnected and Regular, - Representation of Graphs - Set – Linked – Matrix - Graph Traversals - Warshall's Shortest path algorithm.

GRAPH

Graph is basically a collection of vertices (also called nodes) and edges that connect these vertices.

Definition: A Graph G is defined as an ordered set (V, E), where V(G) represents the set of vertices and E(G) represents the edges that connect these vertices.

For example, consider the following graph:



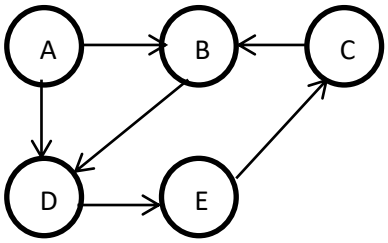
V(G) ={A, B, C, D, E}

E(G) = {(A,B), (B,C), (D,E), (A,D), (B,D), (C,E)}

In this graph there are 5 vertices and 6 edges. A graph can be directed or undirected. In an undirected graph, edges do not have any direction associated with them. That means, if an edge is drawn between nodes A and B, then the nodes can be traversed from A to B as well as from B to A. The above graph is an undirected graph.

In a directed graph, edges form an ordered pair. If there is an edge from A to B, then there is a path from A to B but not from B to A.

For example, consider the following graph:

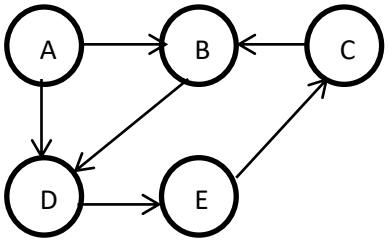


V(G) ={A, B, C, D, E}

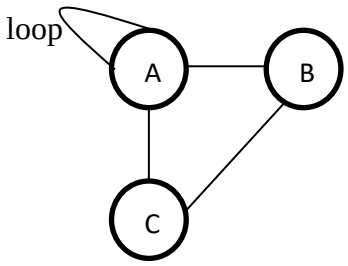
E(G) = {(A,B), (C,B), (D,E), (A,D), (B,D), (E,C)}

Graph Terminologies

1. **Vertex**: Every individual data element is called a vertex or a node. Vertices are the fundamental units of the graph. Every node/vertex can be labelled or unlabelled.
2. **Edge**: It is a connecting link between two nodes or vertices. Each edge has two ends and is represented as (startingVertex, endingVertex). Sometimes, edges are also known as arcs. Every edge can be labelled/unlabelled.
3. **Adjacent nodes or neighbours**: For every edge, e=(u,v) that connects nodes u and v, the nodes u and v are the end points of that edge and are said to be the adjacent nodes or neighbours.
4. **Parallel Edges**: In a graph, if a pair of vertices is connected by more than one edge, then those edges are called parallel edges
5. **Degree of a node**: It is the total number of edges connecting the node.
6. **Isolated Vertex**: If degree of a node is zero, then it is called an **isolated node**.
7. **Pendent Vertex:** A vertex with degree one is called pendent vertex
8. **Path**: A path is defined as a sequence of nodes v_0, v_1, ……, v_n.
9. **Closed Path or Cycle**: If a path has same end points then it is a closed path or cycle. ie, if a path starts and ends at the same vertex then it is called a closed path. For example, in the following graph, BDECB is a closed path.



10. **Simple Path:** If all nodes in a path are distinct except first and last node then the path is simple. If it is closed then it is called closed simple path or simple cycle. In the above graph BDECB is a closed simple path.
11. **Labelled Graph or Weighted Graph:** If every edge is assigned some data or weight then it is called weighted graph or labelled graph. The weight may be the cost, distance etc.
12. **Subgraph:** A graph G1 = (V1, E1) is called a subgraph of a graph G(V, E) if V1(G) is a subset of V(G) and E1(G) is a subset of E(G) such that each edge of G1 has same end vertices as in G.
13. **Self Loop:** If an edge has same end points then it is loop.

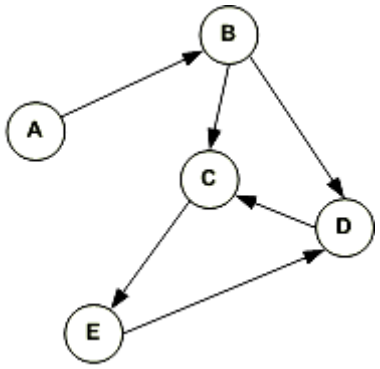


Terminologies of a Directed Graph

1. **Out-degree of a node:** It is the number of edges originates at that node. In the above directed graph, out-degree of B is 1.
2. **In-degree of a node:** It is the number of edges terminates at that node. In the above directed graph, in-degree of B is 2
3. **Degree of a node:** It is the sum of in-degree and out-degree.

TYPES OF GRAPHS

Directed Graph :A graph in which edges have a direction, i.e., the edges have arrows indicating the direction of traversal. Directed graph is also known as **digraphs**.

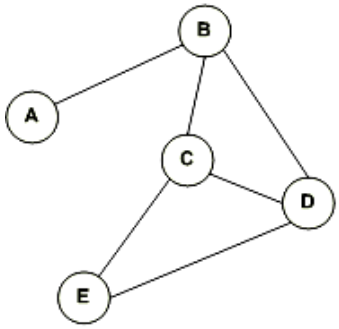


Terminologies of a Directed Graph

1. **Out-degree of a node:** It is the number of edges originates at that node. In the above directed graph, out-degree of B is 1.
2. **In-degree of a node:** It is the number of edges terminates at that node. In the above directed graph, in-degree of B is 2
3. **Degree of a node:** It is the sum of in-degree and out-degree.

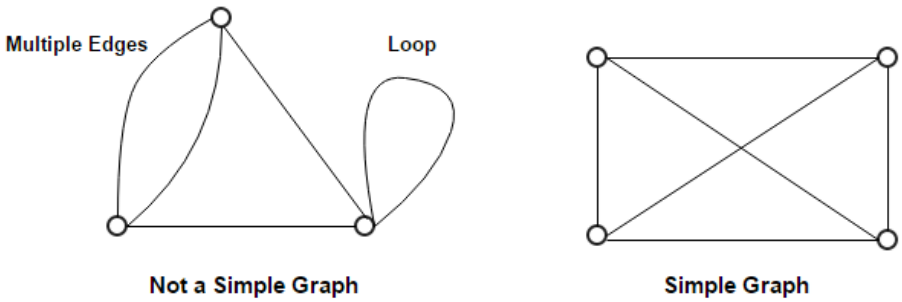
Undirected Graph

A graph in which edges have no direction, i.e., the edges do not have arrows indicating the direction of traversal.



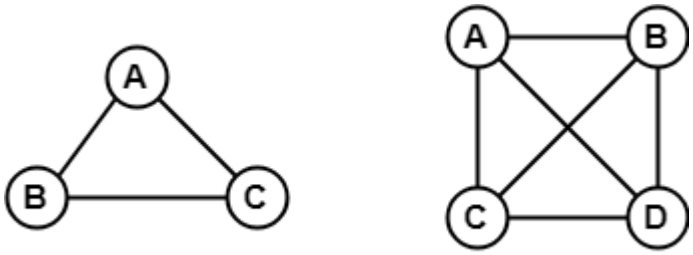
Simple Graph

A **simple graph** is the undirected graph with **no parallel edges** and **no loops**. A simple graph which has n vertices, the degree of every vertex is at most n -1.



Complete Graph

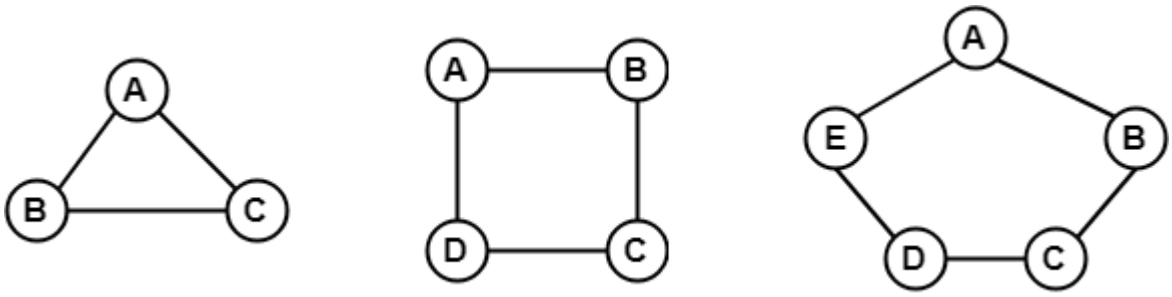
A graph in which every pair of vertices is joined by exactly one edge is called **complete graph**. It contains all possible edges. A complete graph with n vertices contains exactly nC2 edges and is represented by Kn.
Example



In the above example, since each vertex in the graph is connected with all the remaining vertices through exactly one edge therefore, both graphs are complete graph.

Cyclic Graph

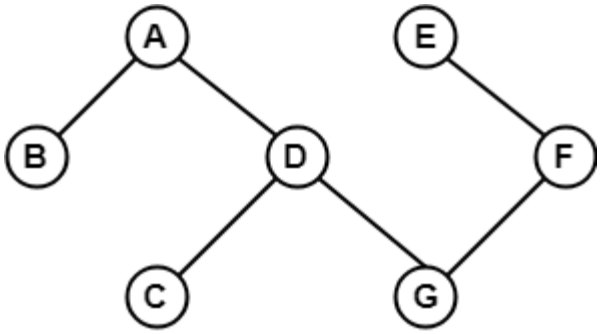
A graph with 'n' vertices (where, n>=3) and 'n' edges forming a cycle of 'n' with all its edges is known as **cycle graph**. A graph containing at least one cycle in it is known as a **cyclic graph**. In the cycle graph, degree of each vertex is 2. The cycle graph which has n vertices is denoted by Cn.
Example 1



In the above example, all the vertices have degree 2. Therefore they all are cyclic graphs.

Acyclic Graph

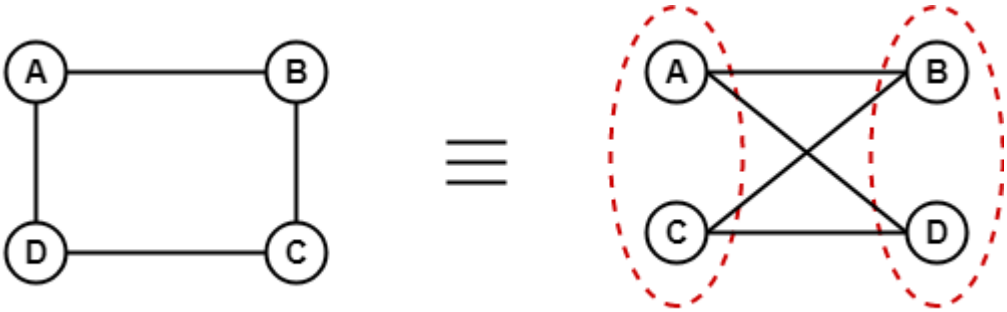
A graph which does not contain any cycle in it is called as an **acyclic graph**.
Example



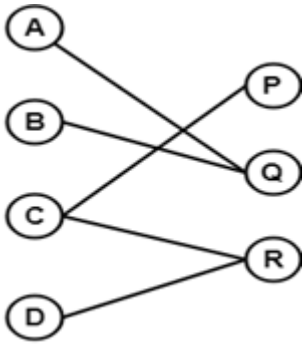
Since, the above graph does not contain any cycle in it therefore, it is an acyclic graph.

Bipartite Graph

A **bipartite graph** is a graph in which the vertex set can be partitioned into two sets such that edges only go between sets, not within them.
A graph G (V, E) is called bipartite graph if its vertex-set V(G) can be decomposed into two non-empty disjoint subsets V1(G) and V2(G) in such a way that each edge e ∈ E(G) has its one end point in V1(G) and other end point in V2(G).
The partition V = V1 ∪ V2 is known as bipartition of G.
Example 1



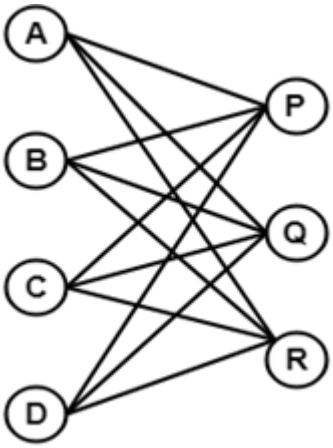
Example 2



Complete Bipartite Graph

A **complete bipartite graph** is a bipartite graph in which each vertex in the first set is joined to each vertex in the second set by exactly one edge. A complete bipartite graph is a bipartite graph which is complete.

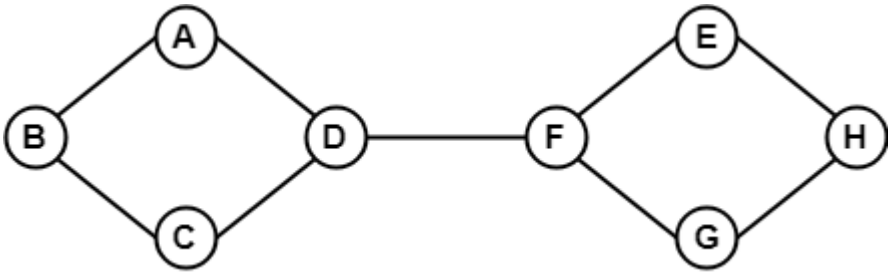
1. Complete Bipartite graph = Bipartite graph + Complete graph
   Example



The above graph is known as $K_{4,3}$.

Connected Graph

A **connected graph** is a graph in which we can visit from any one vertex to any other vertex. In a connected graph, at least one edge or path exists between every pair of vertices.
Example

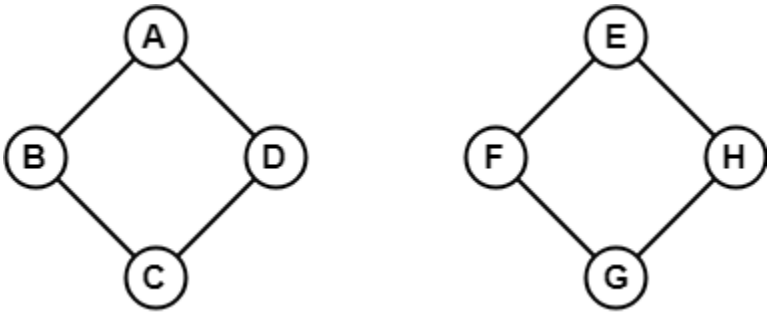


In the above example, we can traverse from any one vertex to any other vertex. It means there exists at least one path between every pair of vertices therefore, it a connected graph.

Disconnected Graph

A **disconnected graph** is a graph in which any path does not exist between every pair of vertices.

Example

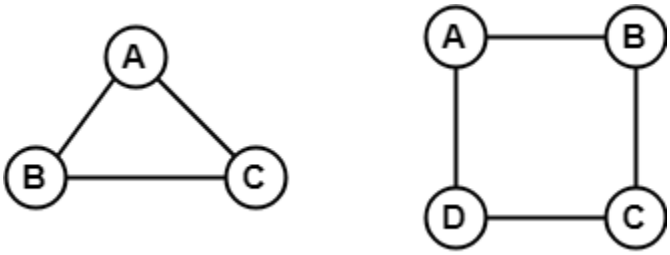


The above graph consists of two independent components which are disconnected. Since it is not possible to visit from the vertices of one component to the vertices of other components therefore, it is a disconnected graph.

Regular Graph

A **Regular graph** is a graph in which degree of all the vertices is same. If the degree of all the vertices is k, then it is called k-regular graph.

Example



In the above example, all the vertices have degree 2. Therefore they are called 2- **Regular graph**.

GRAPH REPRESENTATIONS

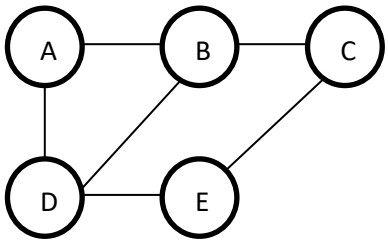
The two basic representations of graphs are

a. Adjacency Matrix Representation and
b. Adjacency List Representation
c. Adjacency Set Representation

Adjacency Matrix Representation

An adjacency matrix is used to represent which nodes are adjacent to one another. If there are 'n' nodes in a graph, then the adjacency matrix is a square matrix, say A of size nxn. An entry A_{ij} will contain 1 if there is an edge connecting v_i and v_j. Otherwise, A_{ij} will contain 0. Adjacency matrices will contain only 1's and 0's. So it is also called a bit matrix or a boolean matrix.

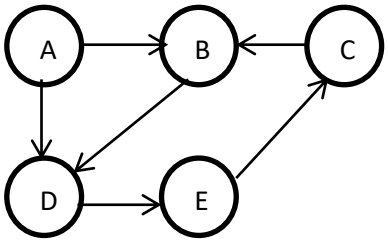
Example: Consider the following undirected graph



Its adjacency matrix will be:

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Consider the following directed graph:



Its adjacency matrix will be:

	A	B	C	D	E
A	0	1	0	1	0
B	0	0	0	1	0
C	0	1	0	0	0
D	0	0	0	0	1
E	0	0	1	0	0

Properties of an Adjacency Matrix:

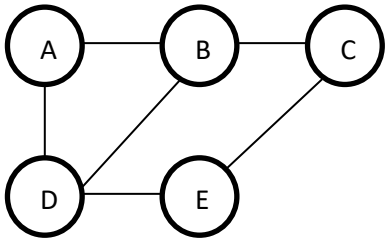
1. For a graph that has no loops the adjacency matrix has 0's on its diagonal.
2. The adjacency matrix of an undirected graph is symmetric.
3. In a directed graph the number of 1's in the adjacency matrix is equal to the edges in the graph, whereas in an undirected graph, it is equal to twice the number of edges in the graph.

Adjacency List Representation

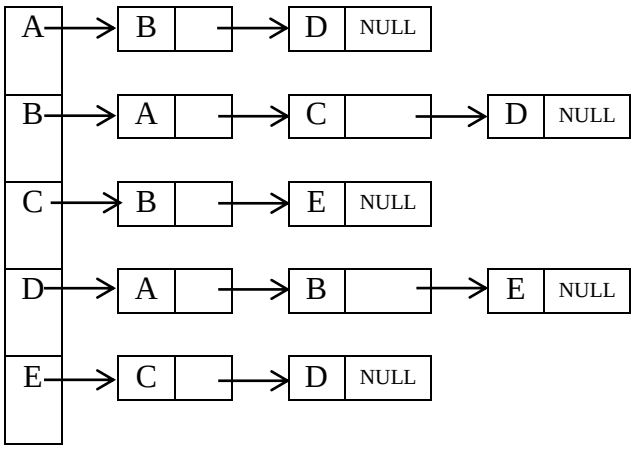
It consists of a list of all nodes in the graph and every node is in turn linked to its own linked list that contains all nodes that are adjacent to it. For a directed graph, total number of nodes

in the list is equal to the number of edges in the graph. For an undirected graph, total number of edges is equal to twice the number of edges in the graph.

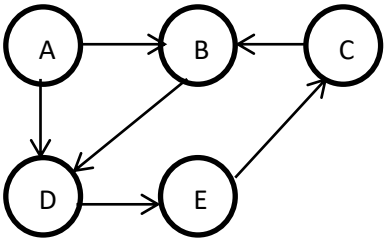
Example: Consider the following undirected graph



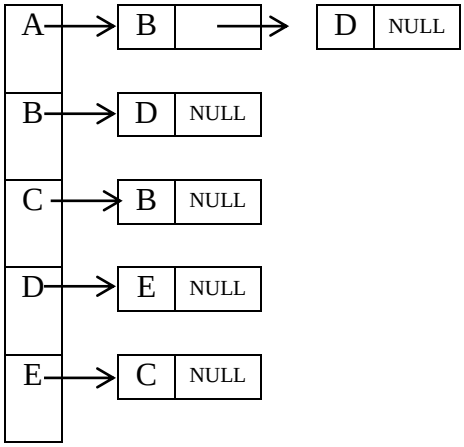
Its adjacency list will be:



Consider the following directed graph:



Its adjacency list will be:



Adjacency Set Representation

Adjacency set is quite similar to adjacency list except for the difference that instead of a linked list; a set of adjacent vertices is provided. Adjacency list and set are often used for sparse graphs with few connections between nodes. Contrarily, adjacency matrix works well for well-connected graphs comprising many nodes.

GRAPH TRAVERSAL ALGORITHMS

By traversing a graph we mean the method of examining nodes and edges of the graph exactly once. There are two standard algorithms for graph traversal.

1. Breadth First Search (BFS)
2. Depth First Search(DFS)

Breadth First Search (BFS) Algorithm

We use queue data structure to implement BFS traversal.

Algorithm

Step 1: Define a queue of size : total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and insert it into the queue.

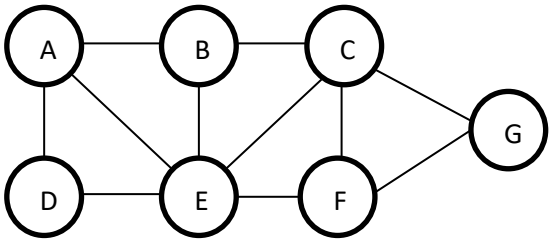
Step 3: Visit all non-visited adjacent vertices of the vertex which is at front of the queue and insert them into the queue.

Step 4: Delete the vertex which is at front of the queue.

Step 5: Repeat steps 3 and 4 until queue becomes empty.

Step 6: Stop.

Example: Consider the following graph.



Step 1: Select the vertex A as starting point, Visit A and insert A into the queue.

Queue :

A						

Step 2: Visit all non-visited adjacent vertices of A : (B, D, E), insert them into the queue and delete A from the queue.

Queue :

	B	D	E			

Step 3: Visit all non-visited adjacent vertices of B : (C), insert them into the queue and delete B from the queue.

Queue :

		D	E	C		

Step 4: Visit all non-visited adjacent vertices of D : (there is no such vertex), insert them into the queue and delete D from the queue.

Queue :

			E	C		

Step 5: Visit all non-visited adjacent vertices of E : (F), insert them into the queue and delete E from the queue.

Queue :

				C	F	

Step 6: Visit all non-visited adjacent vertices of C : (G), insert them into the queue and delete C from the queue.

Queue :

					F	G

Step 7: Visit all non-visited adjacent vertices of F : (there is no such vertex), insert them into the queue and delete F from the queue.

Queue :

						G

Step 8: Visit all non-visited adjacent vertices of G : (there is no such vertex), insert them into the queue and delete G from the queue.

Queue :

Now the queue becomes empty. So stop the BFS process. Final result of BFS is: ABDECFG

Depth First Search (DFS) Algorithm

We use stack data structure to implement DFS traversal.

Algorithm

Step 1: Define a stack of size : total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and push it on to the stack.

Step 3: Visit any one of non-visited adjacent vertex of a vertex which is at the top of stack and push it on to the stack.

Step 4: Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of stack.

Step 5: Pop one vertex from the stack.

Step 6: Repeat steps 3, 4 and 5 until stack becomes empty.

Step 7: Stop.

Example: Consider the above graph for DFS traversal.

Step 1: Select vertex A as the starting point, Visit A and push A on to the stack.

Stack:

A						

Step 2: Visit any non-visited adjacent vertex of A (say B) and push it on to the stack.

Stack:

A	B					

Step 3: Visit any non-visited adjacent vertex of B (say C) and push it on to the stack.

Stack:

A	B	C				

Step 4: Visit any non-visited adjacent vertex of C (say E) and push it on to the stack.

Stack:

A	B	C	E			

Step 5: Visit any non-visited adjacent vertex of E (say D) and push it on to the stack.

Stack:

A	B	C	E	D		

Step 6: Visit any non-visited adjacent vertex of D, since there is no such vertex, pop D from the stack.

Stack:

A	B	C	E			

Step 7: Visit any non-visited adjacent vertex of E (say F) and push it on to the stack.

Stack:

A	B	C	E	F		

Step 8: Visit any non-visited adjacent vertex of F(say G) and push it on to the stack.

Stack:

A	B	C	E	F	G	

Step 9: Visit any non-visited adjacent vertex of G, since there is no such vertex, pop G from the stack.

Stack:

A	B	C	E	F		

Step 10: Visit any non-visited adjacent vertex of F, since there is no such vertex, pop F from the stack.

Stack:

A	B	C	E			

Step 11: Visit any non-visited adjacent vertex of E, since there is no such vertex, pop E from the stack.

Stack:

A	B	C				

Step 12: Visit any non-visited adjacent vertex of C, since there is no such vertex, pop C from the stack.

Stack:

A	B					

Step 13: Visit any non-visited adjacent vertex of B, since there is no such vertex, pop B from the stack.

Stack:

A						

Step 14: Visit any non-visited adjacent vertex of A, since there is no such vertex, pop A from the stack.

Stack:

Now the stack became empty and the DFS stops. The DFS is : ABCEDFG

Program

```
#include<stdio.h>
#include<stdlib.h>
int n,a[20][20],front=-1,rear=-1,v[20],q[20],s[20],top=-1;
void create();
void display();
void insert(int);
void delfront();
void BFS();
void DFS();
void push(int);
void pop();
void create()
{
        int ch,i,j;
        printf("Enter the no. of vertices:");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
                for(j=i;j<=n;j++)
                {
                        printf("\nIs there any edge from %d to %d  (Enter 0/1): ",i,j);
                        scanf("%d",&ch);
```

```
                        if(ch==0)
                                a[i][j]=a[j][i]=0;
                        else
                                a[i][j]=a[j][i]=1;
                }
}
void display()
{
        int i,j;
        printf("\nThe adjascency matrix is:");
        for(i=1;i<=n;i++)
        {
                printf("\n");
                for(j=1;j<=n;j++)
                        printf("%d ",a[i][j]);
        }
}
void BFS()
{
        int i,j;
        for(i=1;i<=n;i++)
                v[i]=0;
        front=rear=-1;
        insert(1);
        v[1]=1;
        do
        {
                i=q[front];
                for(j=1;j<=n;j++)
                {
                        if((a[i][j]==1)&&(v[j]==0))
                        {
                                insert(j);
                                v[j]=1;
                        }
                }
                printf("  %d",q[front]);
                delfront();
        }while(front!=-1);
}
void delfront()
{
         if(front==rear)
                front=-1;
```

```
        else
                front++;
}
void insert(int i)
{
        rear++;
        q[rear]=i;
        if(front==-1)
                front++;
}
void DFS()
{
       int i,j;
        for(i=1;i<=n;i++)
                v[i]=0;
        v[1]=1;
        push(1);
        printf("1”);
        do
        {
                i=s[top];
                for(j=1;j<=n;j++)
                {
                        if((a[i][j]==1)&&(v[j]==0))
                        {
                                v[j]=1;
                                push(j);
                                printf("  %d",j);
                                break;
                        }
                }
                if(j==n+1)
                        pop();
        }while(top!=-1);
}
void push(int i)
{
        top++;
        s[top]=i;
}
void pop()
{
       top--;
}
```

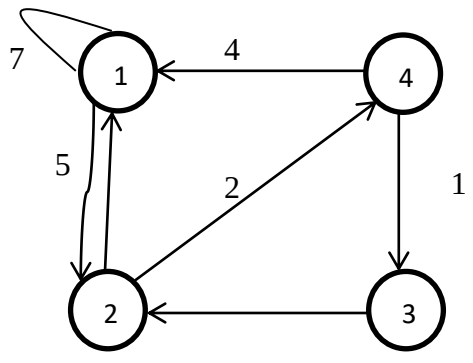
```
void main()
{
        int ch,c;
        create();
        display();
        do
        {
                printf("1.BFS      2.DFS");
                printf("\nEnter the choice:");
                scanf("%d",&ch);
                if(ch==1)
                {
                        printf("\nBFS is:");
                        BFS();
                }
               else if(ch==2)
               {
                        printf("\nDFS is:");
                        DFS();
               }
               else
                       printf("\nInvalid choice");
               printf("\nDo u want to continue (Enter 0/1): ");
               scanf("%d",&c);
        }while(c==1);
}
```

ALL PAIRS SHORTEST PATH ALGORITHM

The all pairs shortest path problem is to determine the shortest path distances between every pair of vertices in a given graph. The algorithm works in the case of a weighted graph. In a weighted graph each edge 'e' is assigned a non-negative value w(e) called weight of edge e. In this case, the adjacency matrix is called a weight matrix or cost adjacency matrix and is defined as:

$$A(i,j) = \begin{cases} w(e), if\ there\ is\ an\ edge\ e\ from\ i\ to\ j \\ \infty, if\ there\ is\ no\ edge\ from\ i\ to\ j \end{cases}$$

For example, consider the following directed weighted graph.



7

3

Its cost adjacency matrix is:

A0	1	2	3	4
1	7	5	∞	∞
2	7	∞	∞	2
3	∞	3	∞	∞
4	4	∞	1	∞

In all pairs shortest path algorithm we find a matrix called shortest path matrix. It is a square matrix consisting of costs of shortest paths between every pair of vertices.

There are many algorithms to find all pairs shortest path. One of them is Floyd-Warshall algorithm.

Floyd-Warshall Algorithm

In this algorithm, we initialize the shortest path matrix as the input cost adjacency matrix. Then we update the matrix by considering all vertices as intermediates in shortest paths. The idea is to pick all vertices one by one and update all shortest paths by including the picked vertex as an intermediate vertex.

When we pick vertex k as an intermediate vertex, we already have considered vertices 0,1,2,…., k-1. For every pair (i,j) of vertices, there are two possible cases:

1. If k is not an intermediate vertex in path from i->j,
   we keep the value of A_{ij} as it is.
2. If k is an intermediate vertex in path from i->j,
   we update the value of A_{ij} as: $A_{ij} = \min\{ A_{ij}, A_{ik} + A_{kj} \}$

Equation 1

If there are n vertices then we define a sequence of matrices:

A0 (given cost adjacency matrix),

A1(considers paths where 1 is an intermediate vertex),

A2(considers paths where 1 and 2 are intermediate vertices),

.

.

.

An(considers paths where 1,2,3,….,n are intermediate vertices)

The final matrix An is the desired shortest path matrix.

Example: Consider the above graph.

A0- Its cost adjacency matrix.

A0	1	2	3	4
1	7	5	∞	∞
2	7	∞	∞	2
3	∞	3	∞	∞
4	4	∞	1	∞

A1 – update A0 by including 1 as intermediate vertex in all paths using equation 1

$2 \xrightarrow{7} 1 \xrightarrow{5} 2$

$2 \xrightarrow{7} 1 \xrightarrow{\infty} 3$

$2 \xrightarrow{7} 1 \xrightarrow{\infty} 4$

A1	1	2	3	4
1	7	5	∞	∞
2	7	**12**	∞	2
3	∞	3	∞	∞
4	4	**9**	1	∞

A2 – update A1 by including 2 as intermediate vertex in all paths using equation 1

$1 \xrightarrow{5} 2 \xrightarrow{7} 1$

$1 \xrightarrow{5} 2 \xrightarrow{\infty} 3$

$1 \xrightarrow{5} 2 \xrightarrow{2} 4$

$3 \xrightarrow{3} 2 \xrightarrow{7} 1$

$3 \xrightarrow{3} 2 \xrightarrow{\infty} 3$

$3 \xrightarrow{3} 2 \xrightarrow{2} 4$

$4 \xrightarrow{9} 2 \xrightarrow{7} 1$

$4 \xrightarrow{9} 2 \xrightarrow{\infty} 3$

$4 \xrightarrow{9} 2 \xrightarrow{2} 4$

A2	1	2	3	4
1	7	5	∞	**7**
2	7	12	∞	2
3	**10**	3	∞	**5**
4	4	9	1	**11**

A3 – update A2 by including 3 as intermediate vertex in all paths using equation 1

$1 \xrightarrow{\infty} 3 \xrightarrow{10} 1$

$1 \xrightarrow{\infty} 3 \xrightarrow{3} 2$

$1 \xrightarrow{\infty} 3 \xrightarrow{5} 4$

$2 \xrightarrow{\infty} 3 \xrightarrow{10} 1$

$2 \xrightarrow{\infty} 3 \xrightarrow{3} 2$

$2 \xrightarrow{\infty} 3 \xrightarrow{5} 4$

$4 \xrightarrow{1} 3 \xrightarrow{10} 1$

$4 \xrightarrow{1} 3 \xrightarrow{3} 2$

$4 \xrightarrow{1} 3 \xrightarrow{5} 4$

A3	1	2	3	4
1	7	5	∞	7
2	7	12	∞	2
3	10	3	∞	5
4	4	**4**	1	**6**

A4 – update A3 by including 4 as intermediate vertex in all paths using equation 1

$1 \xrightarrow{7} 4 \xrightarrow{4} 1$

$1 \xrightarrow{7} 4 \xrightarrow{4} 2$

$1 \xrightarrow{7} 4 \xrightarrow{1} 3$

$2 \xrightarrow{2} 4 \xrightarrow{4} 1$

$2 \xrightarrow{2} 4 \xrightarrow{4} 2$

$2 \xrightarrow{2} 4 \xrightarrow{1} 3$

$3 \xrightarrow{5} 4 \xrightarrow{4} 1$

$3 \xrightarrow{5} 4 \xrightarrow{4} 2$

$3 \xrightarrow{5} 4 \xrightarrow{1} 3$

A4	1	2	3	4
1	7	5	**8**	7
2	**6**	**6**	**3**	2
3	**9**	3	**6**	5
4	4	4	1	6

Here A4 is the final shortest path matrix for the above graph.

Algorithm

Step 1: Repeat for i,j=1,2,3,…..,n
Set A[i][j] = ∞ if there is no edge (i,j)
Otherwise Set A[i][j] = cost of the edge (i,j)
Step 2: for(k=1; k<=n; k++)
for(i=1; i<=n; i++)
for(j=1; j<=n; j++)
A[i][j] = min{ A[i][j], A[i][k] + A[k][j];
Step 3: Stop.