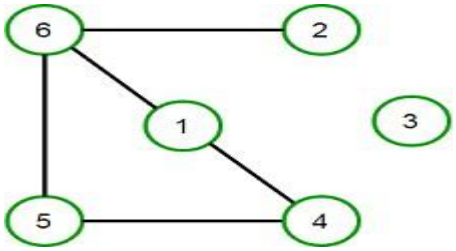
MODULE-5

Graphs

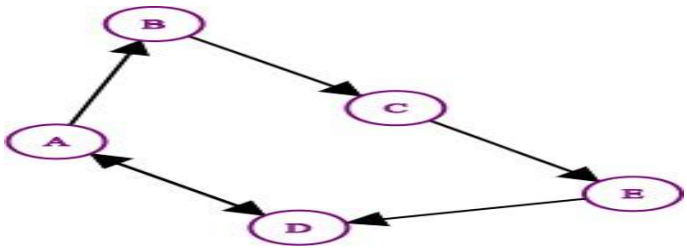
Graphs are primal data structures used in computers & scientific discipline to represent a wide range of relationships and connections between objects. Graph terminology belong of various terms and concepts used to describe and analyze graphs. Here are some key graph terminology and definitions:

1. A Graph is an Order pair Graph = (VS, ES) comprises a set VS of nodes or vertices and a set of edges ES. An edge has starting node and ending node from VS.

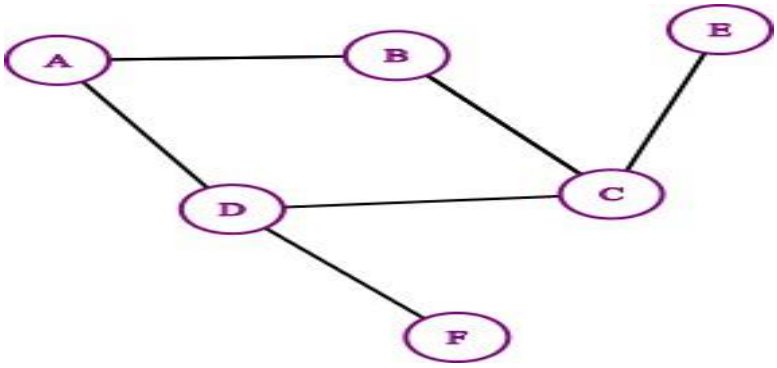


V = { 1, 2, 3, 4, 5, 6 }
E = { (1, 4), (1, 6), (2, 6), (4, 5), (5, 6) }

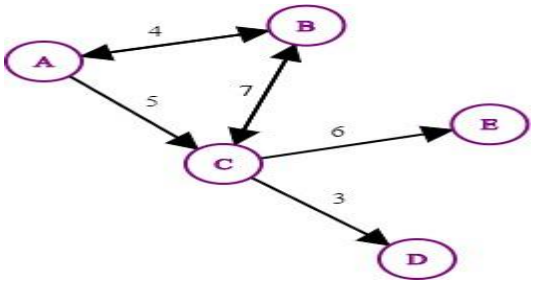
2. Vertex (Node): A vertex, often referred to as a node, is a fundamental unit of a graph. It represents an object or entity. In a social network, for example, each person can be represented as a vertex.
3. Edge (Arc): An edge is a connectivity between 2 vertices. It represents a relationship or connection between the corresponding objects. In a social network, an edge between two vertices (persons) may indicate a friendship.
4. Directed Graph (Digraph): In a directed graph, each edge has a direction, meaning it goes from one vertex (the source) to another vertex (the target). These are often used to model situations where the relationship between vertices is asymmetrical.

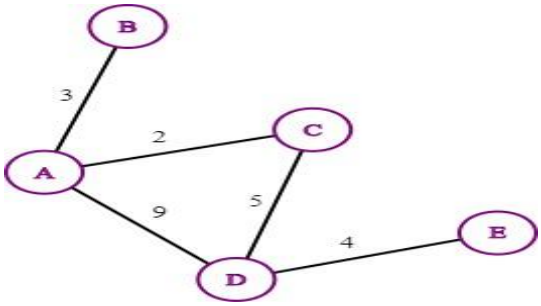


5. Undirected Graph: In an undirected graph, edges have no direction. They simply represent a mutual relationship between two vertices. If there's an edge between vertex A and vertex B, it implies that B is also connected to A.



6. Weighted Graph: If edges of a graph are associated with weights, then it is referred as a weighted. These weights can refers to various properties such as distance, cost, or capacity. Weighted graphs are often used in algorithms like Dijkstras shortest path algorithm. The weighted edge graph may be directed or undirected.

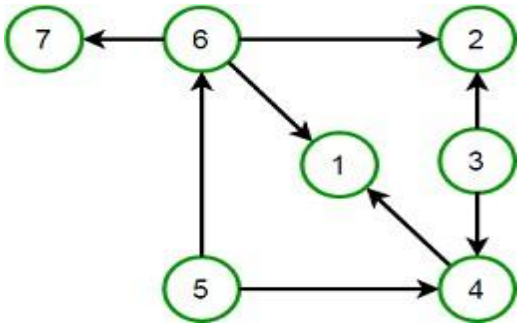




Some interpretations of a weight on an edge:

- Weight may refer to the distance between two nodes.
- Weight may refer to the time required to travel between two nodes.

7. Directed-Acyclic-Graph(DAG):
   A Directed-Acyclic-Graph (DAG) is a oriented graph that consists no cycles.

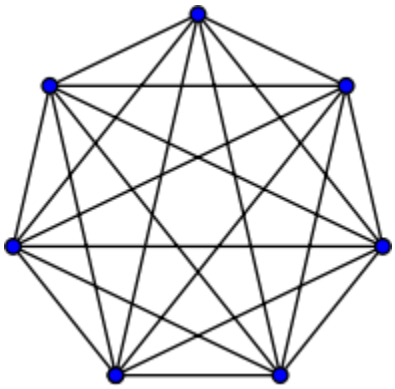


8. Multi graph

A Multi-graph is an undirected graph in which multiple edges (and sometimes loops) are allowed. Multiple edges join same two nodes. A loop is an edge line that joins a node to itself.

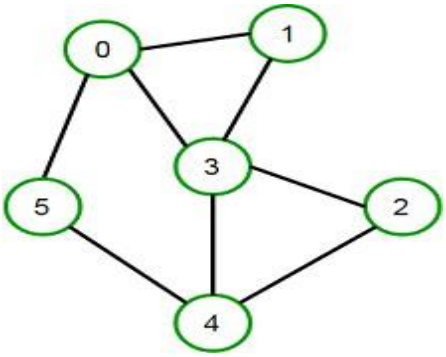
9. Complete graph

Complete graph is a one in which every pair of node links are adjacent

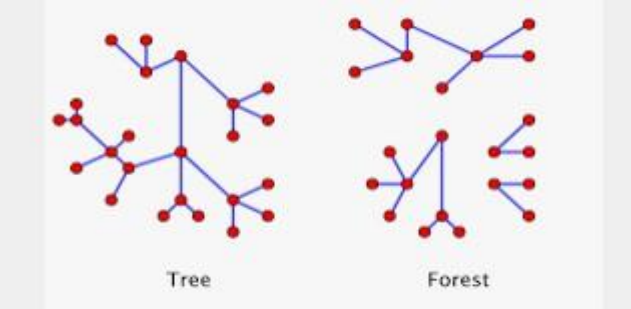


10. Connected graph

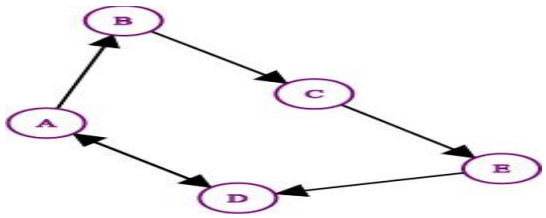
A Connected graph has a path between every couple of nodes. In other words, there are no unreachable nodes in it. A disconnected graph is a graph that is not connected.



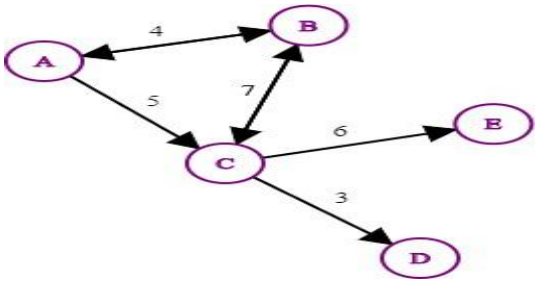
11. A graph is Strongly Connected if there exists a directed path from A to B for every pair of nodes A, B.
12. Degree of a Vertex: The amount of edges connecting to a node determines its degree. In an undirected graph, it is simply the count of adjacent edges. In a directed graph, there are 2 degrees: in-degree (measure of incoming edges) and out-degree (measure of outgoing edges) for each vertex.
13. Path: In a graph, a path line is a sequence of nodes where each adjacent pairs are connected by an edge. The measure of edges in a path is referred as path length.
14. Cycle: If first vertex and last vertex are same in a path, then it is called as a cycle. Cycles can exist in both directed and undirected graphs.
15. Disconnected Graph: A graph is disconnected if it is not connected. This means there are at least two separate groups of vertices that have no path between them.
16. A bridge is an edge whose removal would disconnect the graph.
17. A forest is a graph that lacks cycles. A connected graph without any cycles is referred to as a tree. In a forest, each connected component is a tree.



18. If we remove all the cycles from a Graph, it becomes a tree, and if we remove any edge in the tree, it will become a forest.
19. Subgraph: A subgraph is a graph formed by selecting a subset of vertices and a subset of edges from a larger graph. It preserves the relationships between the selected vertices.
20. Adjacency Matrix (AM): AM is a square matrix used for representing a graph. It is a n by n square matrix, where n represents the number of nodes. The cell value at ith row and jth columnj shows whether an edge between ith node and jth node exists or not. For every pair of nodes, AM keeps a value **1/0/edge-weight** to specify whether the edge exists or not. It requires n^2 space. They can be efficiently used only when the graph is dense.

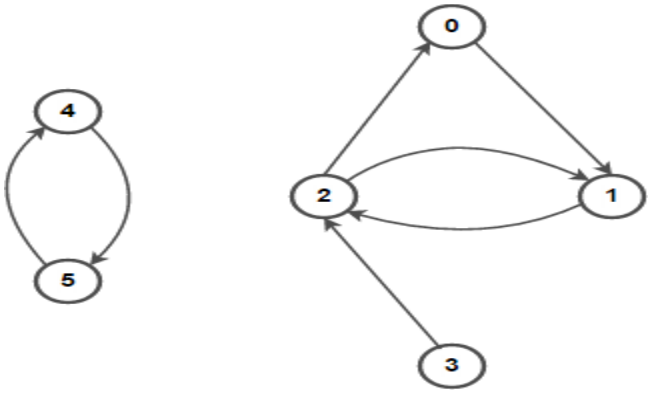


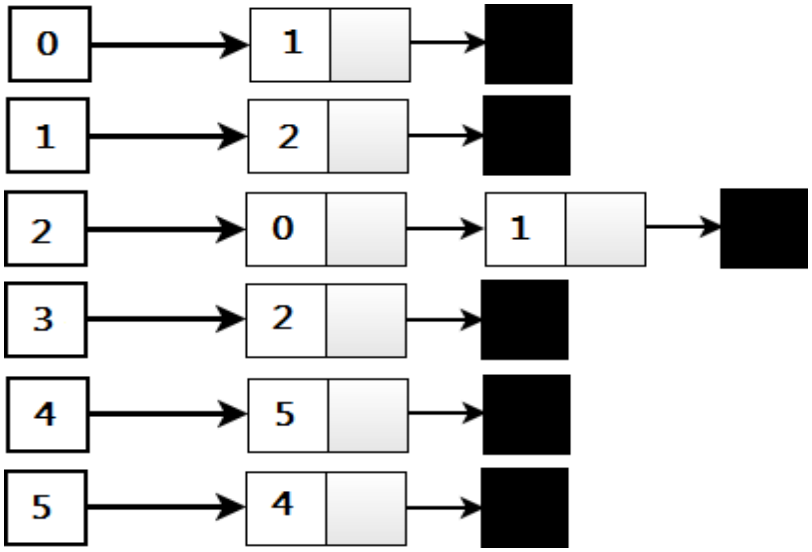
	A	B	C	D	E
A	0	1	0	1	0
B	0	0	1	0	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0



	A	B	C	D	E
A	0	4	5	0	0
B	4	0	7	0	0
C	0	7	0	3	6
D	0	0	0	0	0
E	0	0	0	0	0

21. Adjacency List: An adjacency list is a data structure that represents a graph by storing a list of neighbors for each vertex. It's often more memory-efficient than an adjacency matrix, especially for sparse graphs. Each vertex in the graph is linked to a set of its neighboring vertices or edges, meaning that each vertex maintains a list of adjacent vertices. The specific representation of the adjacency list can vary depending on the implementation.





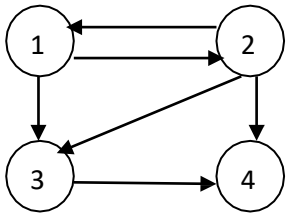
22. Operations on Graph Data Structure

Following are the basic graph operations in graph data structure:

- Add Node, Remove Node – Insert or Delete a node in a graph. (Insert/Delete Vertex)
- Add edge, Remove Edge – Insert or Delete an edge between two nodes. (Insert/Delete Edge)
- Check if the graph contains a given value (Search or Lookup)
- Find path – Discover a path to a destination node from source node. (Find Path)

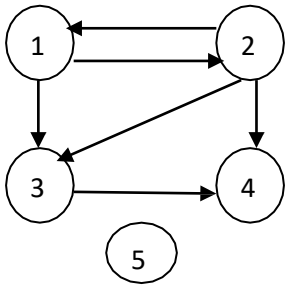
Inserting a Vertex into a Graph:

Example Graph (G) and its Adjacency Matrix: Call this graph as original graph.



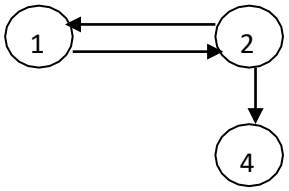
Adjacency Matrix				
	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	0	0	0	1
4	0	0	0	0

Insert Vertex 5 in to graph (G) and show the changes to Adjacency Matrix (addVertex(V_n)):



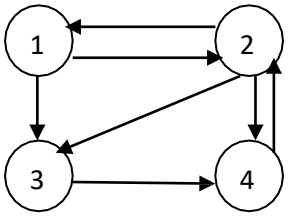
Adjacency Matrix					
	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	1	0
3	0	0	0	1	0
4	0	0	0	0	0
5	0	0	0	0	0

Delete Vertex 3 in original graph (G) and show the changes to Adjacency Matrix (deleteVertex(V_n)):



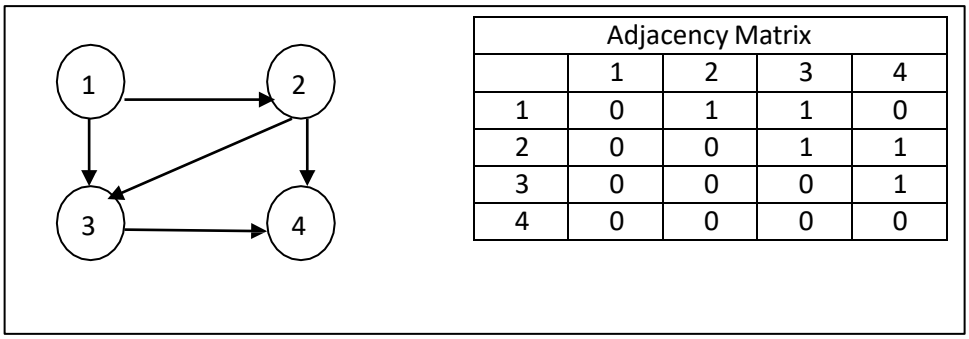
Adjacency Matrix			
	1	2	4
1	0	1	0
2	1	0	1
4	0	0	0

Insert Edge from Vertex 4 to Vertex 2 into original graph (G) and show the changes to Adjacency Matrix (addEdge(V_s, V_e)):



Adjacency Matrix				
	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	0	0	0	1
4	0	1	0	0

Delete Edge from Vertex 2 to Vertex 1 in original graph (G) and show the changes to Adjacency Matrix (deleteEdge(V_s, V_e)):



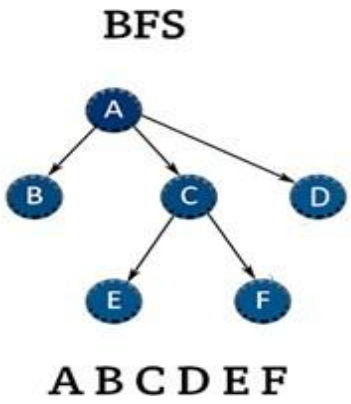
Adjacency Matrix				
	1	2	3	4
1	0	1	1	0
2	0	0	1	1
3	0	0	0	1
4	0	0	0	0

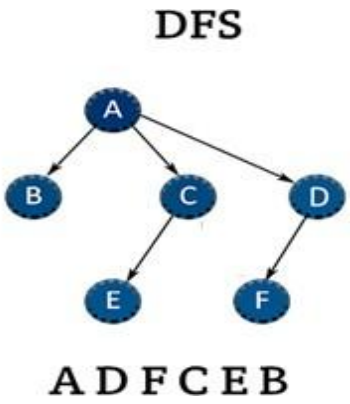
23. **Graph Traversal in Data Structure**

Graph traversal is visiting or updating each node in a graph. It is classified based on the order visiting the nodes. Below are the the 2 traversal techniques:

1. Breadth First Search (BFS) – It is a traversal operation that horizontally traverses the graph. It traverses all nodes at a single level before moving to next level. It begins at the graph's root and traverses all nodes at single depth level before moving on to next level.

2. Depth-First Search (DFS): This is another traversal operation that traverses the graph vertically. It starts with the root node of the graph and investigates each branch as far as feasible before backtracking.





// Algorithm for BFS():

Step1. Initialize all the nodes to ready state (set STATUS = 1)

Step2. Add the starting node into QUEUE. The change the status of that node to waiting (set STATUS = 2)

Step 3: Repeat Step 4 and Step 5 until QUEUE is EMPTY

Step 4: Remove the front node from QUEUE, Process the node, Change its status to processed state (set STATUS = 3)

Step 5: ADD all neighbors of the node which are in the ready state (STATUS = 1) to the REAR of the QUEUE and change status of those nodes to waiting state (set STATUS = 2)

Step 6: Quit.

Implementation of BFS using C:

```
//#include <stdio.h>//
//#include <stdlib.h>//
//#include <stdbool.h>//
//#define// MAXIMUM_NODES 100
// Queue data structure for BFS
struct Queue
{
   int items[MAXIMUM _NODES];
   int f, r;
};
struct Queue* createQueue() {
   struct Queue* q = (struct Queue*)malloc(sizeof(struct Queue));
   q->f = -1;
   q->r = -1;
   return q;
}
bool isEmpty(struct Queue* q) {
   return q->f == -1;
}
void enqueue(struct Queue* q, int item) {
   if (q->r == MAXIMUM _NODES - 1) {
      printf("Queue is full.\n");
      return;
   }
   if (isEmpty(q))
      q->f = 0;
   q->r++;
   q->items[q->r] = item;
```

```
}
int dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return -1;
    }
    int item = q->items[queue->front];
    q->f++;
    if (q->f > q->r) {
        q->f = q->r = -1;
    }
    return item;
}
// Graph data structure
struct Graph {
    int V; // Number of vertices
    int** adjMatrix; // Adjacency matrix
};
struct Graph* createGraph(int V) {
    struct Graph* g = (struct Graph*)malloc(sizeof(struct Graph));
    g->V = V;
    g->adjMatrix = (int**)malloc(V * sizeof(int*));
    for (int m = 0; m < V; m++) {
        g->adjMatrix[m] = (int*)malloc(V * sizeof(int));
        for (int n = 0; n < V; n++) {
            g->adjMatrix[m][n] = 0;
        }
    }
    return g;
}
void edgeAdd(struct Graph* g, int s, int d) {
    g->adjMatrix[s][d] = 1;
    g->adjMatrix[s][d] = 1;
}
void BFS(struct Graph* g, int startVertex) {
    bool* visited = (bool*)malloc(g->V * sizeof(bool));
    for (int m = 0; m < g->V; m++) {
        visited[m] = false;
    }
    struct Queue* queue = createQueue();
    visited[startVertex] = true;
    printf("Breadth First Traversal starting from vertex %d:\n", startVertex);
    printf("%d ", startVertex);
    enqueue(queue, startVertex);
    while (!isEmpty(queue)) {
        int currentVertex = dequeue(queue);
```

```
        for (int m = 0; m < g->V; m++) {
            if (g->adjMatrix[currentVertex][m] == 1 && !visited[m]) {
                printf("%d ", i);
                enqueue(queue, i);
                visited[i] = true;
            }
        }
    }
    printf("\n");
}
int main() {
    int k = 7;
    struct Graph* g = createGraph(k);
    edgeAdd(g, 0, 1);
    edgeAdd(g, 0, 2);
    edgeAdd(g, 1, 3);
    edgeAdd(g, 1, 4);
    edgeAdd(g, 2, 5);
    edgeAdd(g, 2, 6);
    BFS(g, 0);
    return 0;
}
```

Explanation:

1. The program starts by defining a structure for a queue that will be used for the BFS traversal. The queue is implemented as an array-based data structure.
2. A structure for the graph is defined, including the number of vertices and an adjacency matrix to represent the graph.
3. Functions for creating a graph, adding edges, and performing BFS are defined.
4. In the BFS function, we use a visited array to keep track of visited vertices. A queue is used to keep track of the vertices to be explored. We start with the startVertex and enqueue it. Then, we enter a loop where we dequeue a vertex, mark it as visited, and enqueue its unvisited neighbors. This process continues until the queue is empty.
5. In the main function, a sample graph is created and edges are added to it. The BFS traversal is initiated with a starting vertex (in this case, vertex 0).
6. The BFS traversal is performed and the nodes are printed in the order they were visited, demonstrating the breadth-first exploration of the graph.

//Algorithm for DFS()

Step1. Initialise all the links to ready state (set STATUS = 1)

Step2. Put opening node onto STACK and modify its status as waiting (set STATUS = 2)

Step 3: Repeat Step 4 and Step 5 till STACK becomes BLANK

Step 4: Remove topmost node on STACK and Process the node. Then modify its state as processed (set STATUS = 3)

Step 5: Add all neighbors of the node which are in ready state (i.e., STATUS = 1) to the STACK and modify their state to waiting (set STATUS = 2)

Step 6: Quit.

Implementation of DFS using C:

```
//#include <stdio.h>//
//#include <stdbool.h>//
//#include <stdlib.h>//

//#define// MAXIMUM _NODES 100

// Stack data structure for DFS
struct Stack
{
   int items[MAXIMUM_NODES];
   int t;
};

struct Stack* createStack() {
   struct Stack* s = (struct Stack*)malloc(sizeof(struct Stack));
   s->t = -1;
   return s;
}

bool isEmpty(struct Stack* s) {
   return s->t == -1;
}

void push(struct Stack* s, int item) {
   if (s->t == MAXIMUM _NODES - 1) {
      printf("Stack is full.\n");
      return;
   }
   s->items[++s->t] = item;
}
```

```
int pop(struct Stack* s) {
    if (isEmpty(s)) {
        printf("Stack is empty.\n");
        return -1;
    }
    return s->items[s->t--];
}

// Graph data structure
struct Graph {
    int V; // Number of vertices
    int** adjMatrix; // Adjacency matrix
};

    struct Graph* createGraph(int V) {
        struct Graph* g = (struct Graph*)malloc(sizeof(struct Graph));
        g->V = V;
        g->adjMatrix = (int**)malloc(V * sizeof(int*));
        for (int m = 0; m < V; m++) {
            g->adjMatrix[m] = (int*)malloc(V * sizeof(int));
            for (int n = 0; n < V; n++) {
                g->adjMatrix[m][n] = 0;
            }
        }
        return g;
    }
void edgeAdd(struct Graph* g, int s, int d) {
    g->adjMatrix[s][d] = 1;
    g->adjMatrix[d][s] = 1;
}

// Iterative DFS
void DepthFirstSearch(struct Graph* g, int startVertex) {
    bool visited[g->V];
    for (int m = 0; m < g->V; m++) {
        visited[m] = false;
    }

    struct Stack* s = createStack();

    visited[startVertex] = true;
    printf("Depth First Traversal starting from vertex %d:\n", startVertex);
    printf("%d ", startVertex);
    push(s, startVertex);

    while (!isEmpty(s)) {
```

```
        int currentVertex = s->items[s->top];

        int found = 0;
        for (int m = 0; m < g->V; m++) {
            if (g->adjMatrix[currentVertex][m] == 1 && !visited[m]) {
                printf("%d ", m);
                visited[m] = true;
                push(s, m);
                found = 1;
                break;
            }
        }

        if (!found) {
            pop(s);
        }
    }
}

int main() {
    int V = 7;
    struct Graph* g = createGraph(V);

    edgeAdd(g, 0, 1);
    edgeAdd(g, 0, 2);
    edgeAdd(g, 1, 3);
    edgeAdd(g, 1, 4);
    edgeAdd(g, 2, 5);
    edgeAdd(g, 2, 6);
    DepthFirstSearch(g, 0);
    return 0;}
```

Explanation:

1. The program starts by defining a structure for a stack, which is used for managing the depth-first traversal.
2. A structure for the graph is defined, including the number of vertices and an adjacency matrix to represent the graph.
3. Functions for creating a graph, adding edges, and performing DFS are defined.
4. The DFS function performs an iterative depth-first traversal. It uses a stack to manage the order of node exploration, starting from the specified source vertex (in this case, vertex 0).
5. The main function creates a sample graph, adds edges to it, and initiates the DFS traversal starting from vertex 0.
6. The DFS traversal is performed iteratively, and the nodes are printed in the order they are visited, demonstrating the depth-first exploration of the graph.

24. Shortest Paths and Minimum Spanning Trees

Spanning trees is a sub graph that connects all nodes of a given graph using minimum number of edges. It may or may not be weighted and does not have cycles.

Spanning trees of any connected undirected graph is a sub-graph, i.e., a tree that binds all nodes that contribute to minimization of amount of the weights of the edges.

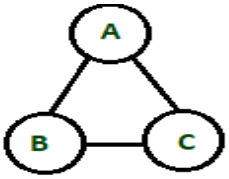
Let G be a graph with nodes set V and edges set E, i.e., G(V, E). Then G'(V', E') is a Spanning Tree of G if it obeys following constraints.

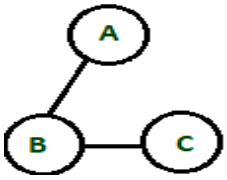
1. V' = V (number of nodes in G' = number of node in G)
2. E' = |V| - 1 (Number of edges G' = number of nodes in G minus 1)

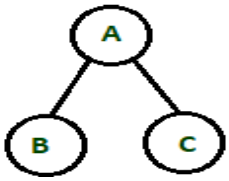
There might be many spanning trees possible for a given graph.

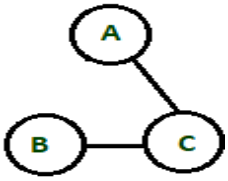
Properties:

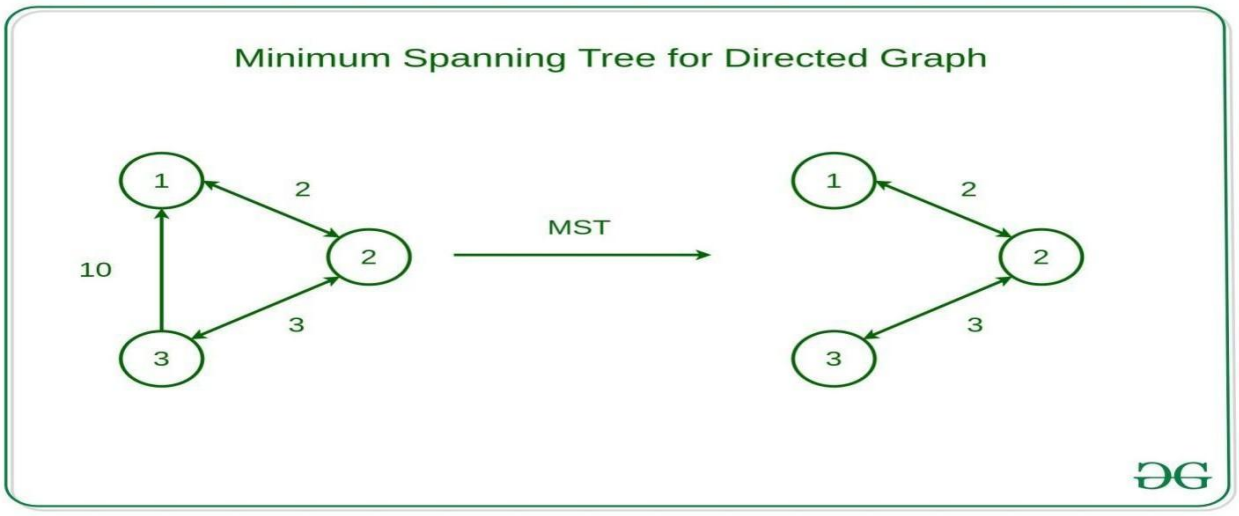
- A spanning trees can exist if a graph is connected. Otherwise many spanning trees called forest exists.
- The number of edges in a spanning tree = e-1, where e= number of nodes a given graph.
- **Cayleys formula**: measure of spanning trees in a complete undirected graph having n nodes K_n **= n^{n-2}.**
- For the K_3 graph, total number of spanning trees = $3^{3-2} = 3$.



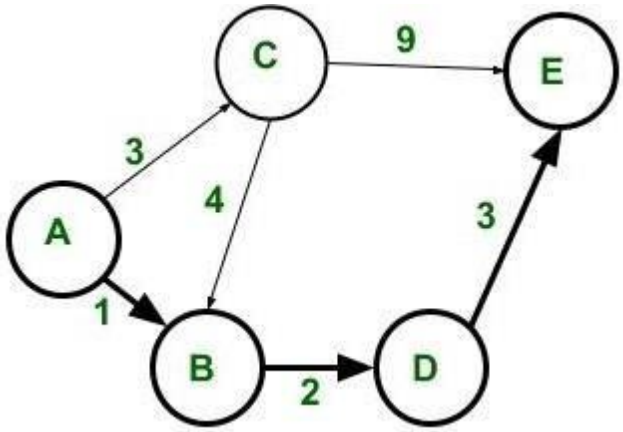








The Shortest path:



- Sum of the weights on edges in shortest path (i.e. ABDE) of any pair of vertices (say between A and E) is minimum among all possible paths between that pair of nodes. (i.e., A and E).
- Computing shortest path can be done for directed, undirected or mixed graphs.
- The problem for discovering a shortest path is classified as
    - Single-source the shortest path: Here the calculation of shortest path is done from given source node to every other node of the graph.

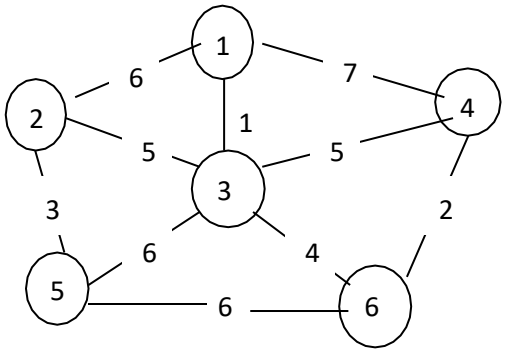
- Single-destination the shortest path: Here the computation of shortest path is done from all nodes of a graph to the given destination node.
- All pairs the shortest path: Here the calculation of shortest path is done for each pair of nodes.

Prim's Algorithm: This algorithm is to get least cost Spanning Tree for a undirected connected graph:

Let G1=(V1, E1) be an connected undirected graph and T1=(V1, E1') is sub-graph of G1 and is the spanning Tree for G1 if T1 is a Tree.

```
Prim(G)
    Begin
        E'= Φ;
        Choose a least cost edge (u1, v1) from E1;
        V1' = {u1};
        While V1' ≠ V1 do
            Let (u1, v1) be a least cost edge such that u1 is in V1' and v1 is in V1 –
            V1';
            Add edge (u1, v1) to set E1';
            Add v1 to set V1'
        End while
    End Prim
```

Example Graph:



Trace of Prims Algorithm:

V = {1, 2, 3, 4, 5, 6}

E = {(1,2), (1,3), (1,4), (2,3), (2,5), (3,4), (3,5), (3,6), (4,6), (5,6)}

E' = {}

Edge	Cost of the Edge	V’	E’
(1,3)	1	1	{}
(1,3)	1	{1, 3}	{(1,3)}
(3,6)	4	{1, 3, 6}	{(1,3), (3,6)}
(6,4)	2	{1, 3, 6, 4}	{(1,3), (3,6), (6,4)}
(3,2)	5	{1, 3, 6, 4, 2}	{(1,3), (3,6), (6,4),(3,2)}
(2,5)	3	{1, 3, 6, 4, 2, 5}	{(1,3), (3,6), (6,4),(3,2),2,5)}

Kruskal's Spanning Tree Algorithm

This algorithm is aimed to discover Minimum spanning Tree (MST) of a given weighted, connected, undirected graph.

In case, the graph is disconnected, on applying Kruskal's algorithm can find the MST of each connected component.

Spanning tree is a tree and also a sub-graph of connected, weighted and undirected graph that contains all the nodes. A minimum spanning tree corresponds to a spanning tree that has minimum total weight, where the overall weights of the spanning tree is equal to sum of the weights of the edges present in it.

Spanning Tree Basic properties:

- More than one spanning tree can exist to the connected graph.
- Spanning trees do not contain loops or cycles.
- As a spanning tree is loosely connected, if removing an edge from the tree will disconnect the graph.
- Addition of an edge to a spanning tree creates a loop because it is acyclic.
- Maximum number of spanning trees = $\mathbf{n^{n-2}}$. N= number of nodes in a complete graph
- In a spanning tree, total number of edges = n-1, where n = total number of nodes.

Steps in Kruskal's Algorithm:

Objective is to remove all parallel edges and loops in given weighted Graph G (V,E).

Step 1 - Initialize MST to null.

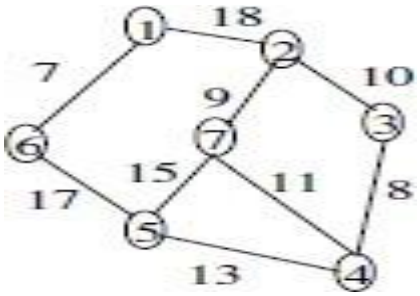
Step 2 − Order all edges of G in their ascending order of weights.

Step 3 – Select the edge that has the minimum or least weight. Add the selected edge to MST if its addition keeps the spanning tree properties remain intact. Otherwise ignore it.

Step 4 - Repeat step 3 till spanning tree is complete. i.e. all nodes are connected in MST.

.

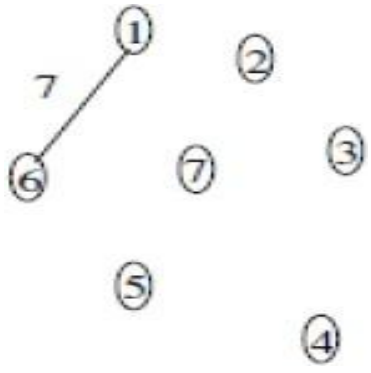
Example: Look at the following example to understand the algorithm of Krushkal.



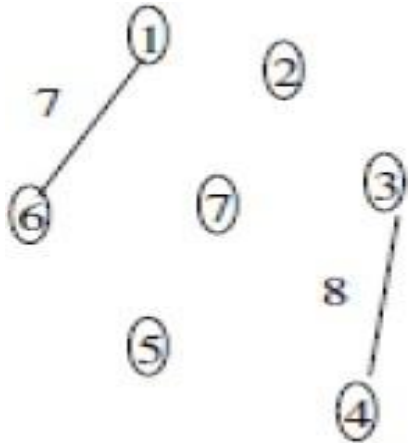
Edges of given graph with ascending order of weights:

Edge	(1, 6)	(4, 3)	(2, 7)	(2, 3)	(7, 4)	(4, 5)	(5, 7)	(5, 6)	(1, 2)
Weight	7	8	9	10	11	13	15	17	18

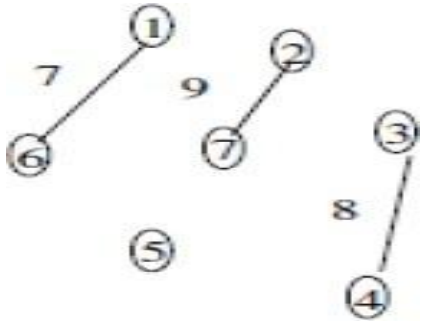
Take edge (1, 6): No loop is formed. Hence include it.



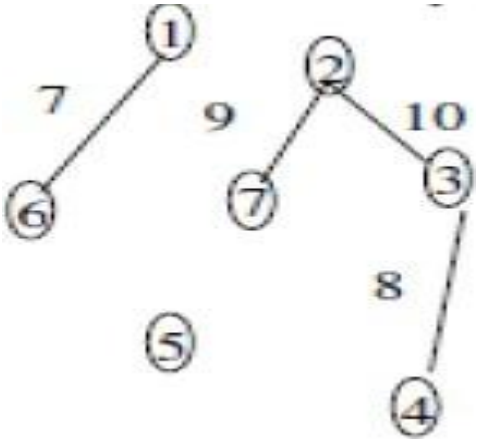
Take edge (4,3): No loop is formed. Hence include it.



Take edge (2,7): No loop is formed. Hence include it.

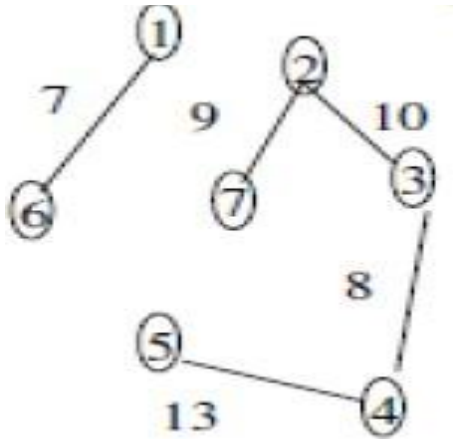


Take edge (2,3): No loop is formed. Hence include it.



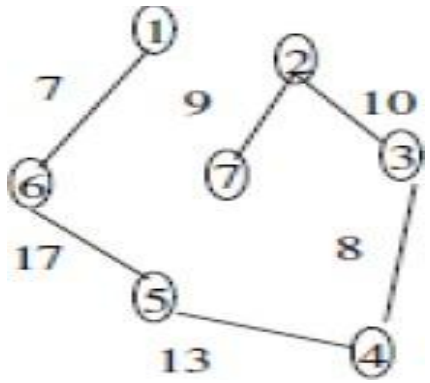
Take edge (7,4): Inclusion of this edge causes a loop. Hence discard it.

Take edge (4,5): No loop is formed. Hence include it.



Take edge (5,7): Inclusion of this edge causes a loop. Hence discard it.

Take edge (5,6): No loop is formed. Hence include it.



Total edges included = (n– 1). So the algorithm terminates here.

Floyd Warshall's algorithm:

- Floyd Warshall's algorithm is applied to discover shortest paths between every pairs of nodes in a given weighted graph.
- The complexity of this algorithm is $O(n^3)$, where n = the number of nodes in a weighted graph.
- This algorithm is also known as Floyd's algorithm, Roy-Floyd algorithm, or Roy Warshall algorithm.
- This algorithm uses dynamic programming paradigm to discover shortest paths.

Algorithm in C:

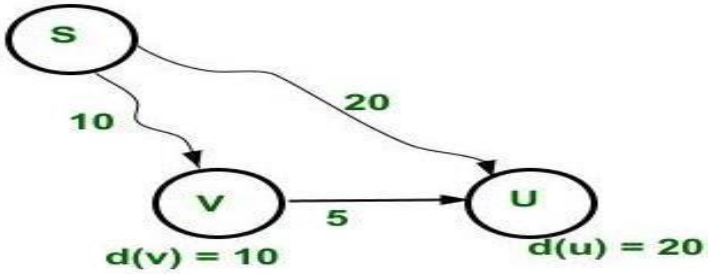
```
//#include<stdio.h>//
//#include<limits.h>//
//#define V 4//
voidfloydWarshall(int graph[V][V]) {
int distance[V][V];
int m, n, p;
for (m = 0; m < V; m++) {
for (n = 0; n < V; n++) {
          distance[m][n] = graph[m][n];
      }   }
for (p = 0; p < V; p++) {
for (m = 0; m < V; m++) {
for (n = 0; n < V; n++) {
if (dist[m][p] != INT_MAX && distance[p][n] != INT_MAX &&
                (distance[m][p] + distance[p][n] < distance[m][n])) {
                distance[m][n] = distance[m][p] + distance[p][n];
            }          }       }    }
printf("Shortest distances between all pairs of vertices:\n");
for (m = 0; m < V; m++) {
for (n = 0; n < V; n++) {
if (distance[m][n] == INT_MAX) {
printf("INF ");
          } else {
printf("From %d to %d: %d ", m, n, distance[m][n]);
          }       }    }}
Int main() {
int graph[V][V] = {
      {0, 3, INT_MAX, 7},
      {8, 0, 2, INT_MAX},
      {5, INT_MAX, 0, 1},
      {2, INT_MAX, INT_MAX, 0}
   };
   floydWarshall(graph);
return0;}
```

Dijkstra's Algorithm:

- This algorithm is developed to discover shortest path from one node to other nodes in a given graph. It can be used on directed graphs as well as undirected graphs.
- Because it calculates the shortest path from a source node to all other nodes in the given graph, this approach is also known as the single-source shortest path algorithm. This algorithm produces the shortest path spanning tree.
- The algorithm execution starts from source node. Graph G is the input to the algorithm. **Shortest path spanning tree** is the output of this algorithm.

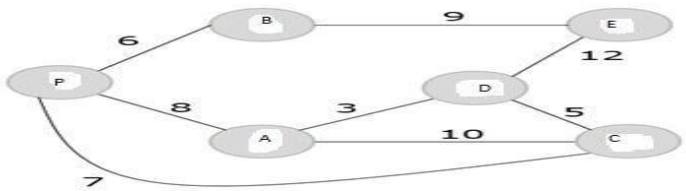
Dijkstra's Algorithm Steps:

1. Declare two vectors –
    - *Distance*[] for maintaining distances from the source node to all other nodes in the graph
    - *Visited*[] for maintaining the visited nodes.
2. Intialise distance[S] = '0' and Distance[v] = ∞, here v refers all the other nodes in the given graph.
3. Add source node S to *Visited*[] Vector. Then find the adjacent nodes of S and update *Distance*[] vector for adjacent nodes of S.
4. Select a node from (*Vertex*[] – *Visited*[]) that has shortest distance. Add this node to Visited. Find all the adjacent nodes to this node and perform relaxation using each of the adjacent nodes. Update the *Distance*[] vector accordingly.
5. Repeat 4th step until all the nodes are in *Visited*[] vector and the shortest path spanning tree is formed.
6. **Relaxation Property:**
   Modify distances for all the adjacent nodes of node, say u, using Relaxation rule discussed below using diagram. For changing distance values, iteratively run through all adjacent nodes of u. For each adjacent nodes, say v, if sum of the weight of edge (u, v) and the distance of u from source is smaller than v's distance form the source, then change the length of v from the source.



- Let dist(U) = distance of node U from source node S
- Cost(V, U) = distance from node V to node U.
- If dist(U) > dist(V) + Cost(V, U) then dist(U) = dist(V) + Cost(V, U). This is called Relaxation.
- As an example, 20>10+5, dist(U) = 10+5 = 15

Example: Tracing the algorithm using the following graph



Step 1: Initialization

Initialize distances to all nodes from source node, P, to ∞. Make P to P as 0.

Node	P	B	E	D	A	C
Distance	0	∞	∞	∞	∞	∞

Let the source node P be visited. Add P to visited Vector. visited = {P}

Step 2:

The Node P has 3 adjacent nodes, B, A, C, with different distances. Hence The distance of S to A, S to D and S to E will be changed in the distance vector.

P → B = 6
P → A = 8
P → C = 7

Node	P	B	E	D	A	C
Distance	0	6	∞	∞	8	7

Step 3: Repeat Step

- Select a node from (Vertex[] – Visited[]) that has shortest distance. Add this node to Visited. Find all the adjacent nodes to this node and perform relaxation for each of the adjacent nodes. Update the distance vector accordingly.

  The vertex having minimum distance among all the nodes in (Vertex[] – Visited[]) is B. Hence, B is added to Visited. The adjacent node(s) to B is E. Hence perform Relaxation and update the distance vector as follows.

Node	P	B	E	D	A	C
Distance	0	6	15	∞	8	7

Now, Visited = {P, B}

Exercise: Perform repeat step and discover shortest paths to all the nodes from the source node.

Requirements

1. Dijkstra's Algorithm is guaranteed to work properly if graphs have only **positive** weights. The reason for this is that the edge weights are to be added to discover shortest path.
2. The algorithm may not perform properly if any negative weight exists in graph. The movement a node is denoted as "visited", then the present path for that node can be marked as a shortest path. Existence of negative weights may alter this path if sum of weights gets reduced later.

25. **Applications of Graph Data Structure:**

Graph data structure has a variety of applications. Some of the most popular applications are:

- Graphs have been used for representing flow of computation in programs of software systems.
- Google Maps has been using for constructing digital transportation systems. The shortest path discovery between pair of vertices is being used by the Google.
- Linkedin and Facebook have been using for social networks analysis.
- Operating Systems use a Resource Allocation Graph where every process and resource acts as a node. While we draw edges from resources to the allocated process.
- Used in the world wide web where the web pages represent the nodes.
- Blockchains also use graphs. The nodes store many transactions while the edges connect subsequent blocks.
- Used in modelling data.
- Some other applications of graphs include:
    - Knowledge graphs
    - Biological networks
    - Neural networks
    - Product recommendation graphs

These are some of the fundamental terms and concepts used to describe and work with graphs in the field of data structures and algorithms. Graphs are incredibly versatile and find applications in various domains, including computer networks, social networks, recommendation systems, and route planning, among others.