

Graph

GRAPH DATA STRUCTURE & GRAPH TRANSVERSAL ALGORITHM



December 13, 2023

Fobs AIAA

**INTRODUCTION TO GRAPHS**

Graph is a nonlinear data structure. A map is a well-known example of a graph. In a map various connections are made between the cities. The cities are connected via roads, railway lines and aerial network. We can assume that the graph is the interconnection of cities by roads.

A graph contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices.

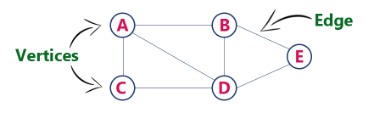
**A graph is defined as** Graph is a collection of vertices and arcs which connects vertices in the graph. A graph G is represented as

G = (V , E),

Where V is set of vertices and E is set of edges.

Example: graph G can be defined as G = ( V , E ) Where V = {A,B,C,D,E} and

E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}. This is a graph with 5 vertices and 6 edges.



*Graph Terminology*

1. **Vertex :**

An individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.

1. **Edge :**

An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (starting Vertex, ending Vertex). An edge is uniquely identified by its 2 end points,

In above graph, the link between vertices A and B is represented as (A,B).

Edges are three types:

1. **Undirected Edge**

An undirected edge is a bidirectional edge. If there is an undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).

1. **Directed Edge**

A directed edge is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).

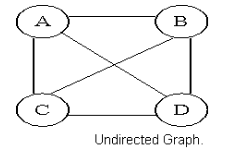
1. **Weighted Edge**

A weighted edge is an edge with cost on it.

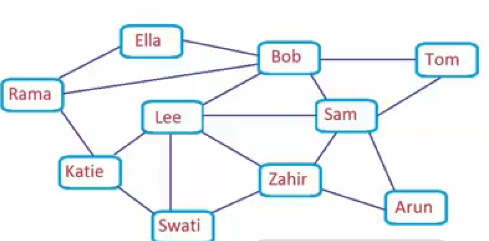
***Types of Graphs***

1. *Undirected Graph*

A graph with only undirected edges is said to be undirected graph



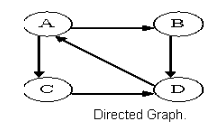
Let’s consider the graph bellow for the interconnection of friends in a social media platform like Facebook



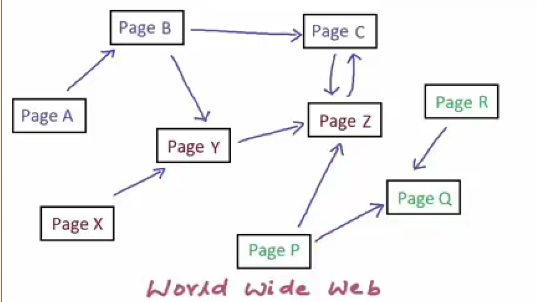
We can see that Rama is friend with Ella, Bob, and Katie. Since the graph is undirected Rama can communicate with Bob and Bob also can communicate with Rama which is the same case for every interconnected individual in the network. We see that for a social network like Facebook, the friends proposed to Rama will be the friend of his friends so the friend of Bob which are {Tom Sam and Lee}, the friends of Katie which are {Lee and Swati} and also the friends of Ella which is Bob.

1. *.Directed Graph*

A graph with only directed edges is said to be directed graph.



Consider the graph bellow for the interconnection of web pages in the World Wide Web

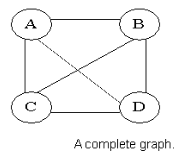


Here we see that page A has a link to page B but at page B you cannot go back to page A since the movement is directed from page A to page B.

But we also see that page C has a link to page Z and page Z also have a link to page C so the movement can be in the two direction and can be seen as an undirected path

1. *Complete Graph*

A graph in which any V node is adjacent to all other nodes present in the graph is known as a complete graph. An undirected graph contains the edges that are equal to edges = n(n-1)/2 where n is the number of vertices present in the graph. The following figure shows a complete graph. This type of graph can be implemented in mesh topology for interconnection of many computers and devices

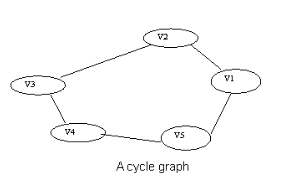


1. *Regular Graph*

Regular graph is the graph in which nodes are adjacent to each other, i.e., each node is accessible from any other node.

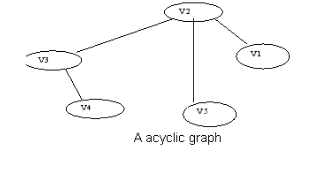
1. *Cycle Graph*

A graph having cycle is called cycle graph. In this case the first and last nodes are the same. A closed simple path is a cycle.



1. *Acyclic Graph*

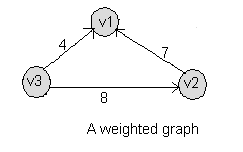
A graph without cycle is called acyclic graphs.



1. *Weighted Graph*

A graph is said to be weighted if there are some non-negative value assigned to each edges of the graph. The value is equal to the length between two vertices. Weighted graph is also called a network.

Intercity roads are example of weighted graph since each edge has a deferent magnitude.



An example of unweigted graph is the interconnection of users in a social media platform like Facebook

1. *Simple Graph*

A graph is said to be simple if there are no parallel and self-loop edges.

1. ***Self-loop***

An edge (undirected or directed) is a self-loop if its two endpoints coincide.

1. **Adjacent nodes**

When there is an edge from one node to another then these nodes are called adjacent nodes.

1. **Incidence**

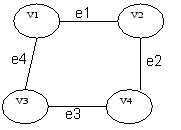
In an undirected graph the edge between v1 and v2 is incident on node v1 and v2.

1. **Walk**

A walk is defined as a finite alternating sequence of vertices and edges, beginning and ending with vertices, such that each edge is incident with the vertices preceding and following it.

1. **Closed walk**

A walk which is to begin and end at the same vertex is called close walk. Otherwise it is an open walk

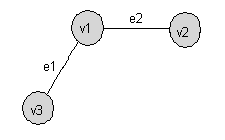


f e1,e2,e3,and e4 be the edges of pair of vertices (v1,v2),(v2,v4),(v4,v3) and (v3,v1) respectively ,then v1 e1 v2

e2 v4 e3 v3 e4 v1 be its closed walk or circuit.

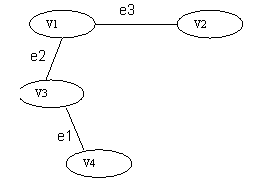
Path

An open walk in which no vertex appears more than once is called a path.



1. **Length of a path**

The number of edges in a path is called the length of that path. In the following, the length of the path is 3

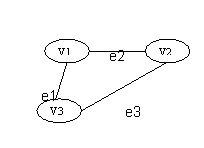


An open walk Graph

1. **Circuit**

A closed walk in which no vertex (except the initial and the final vertex) appears more than once is called a circuit.

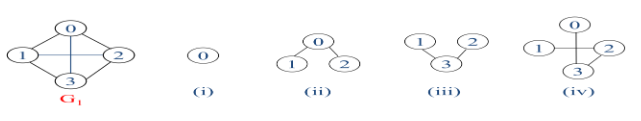
A circuit having three vertices and three edges.



1. **Sub Graph**

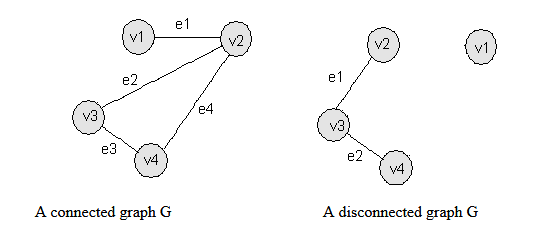
A graph S is said to be a sub graph of a graph G if all the vertices and all the edges of S are in G, and each edge of

S has the same end vertices in S as in G. A sub graph of G is a graph G’ such that V(G’) ⊆ V(G) and E(G’) ⊆ E(G)



1. **Connected Graph**

A graph G is said to be connected if there is at least one path between every pair of vertices in G. Otherwise is disconnected



1. **Degree**

The degree of a vertex is the number of edges incident to that vertex

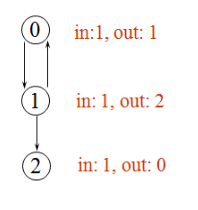
For directed graph,

– The in-degree of a vertex v is the number of edges that have v as the head

– The out-degree of a vertex v is the number of edges that have v as the tail

– If di is the degree of a vertex i in a graph G with n vertices and e edges, the number of edges is

Example:



*Graph Representations*

Graph data structure is represented using following representations

* Adjacency Matrix
* Adjacency List
* Adjacency Multilists

1. ***Adjacency Matrix***

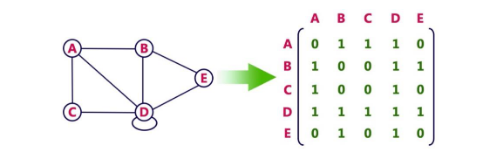
In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices; means if a graph with 4 vertices can be represented using a matrix of 4X4 size.

In this matrix, rows and columns both represent vertices.

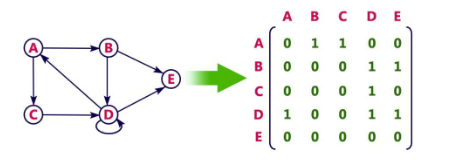
This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

Adjacency Matrix: let G = (V, E) with n vertices, n ≥1. The adjacency matrix of G is a 2-dimensional nxn matrix, A, A(i, j) = 1 if (vi, vj) belonging to E(G) ({vi, vj} for a diagraph), A(i, j) = 0 otherwise.

Example: for undirected graph



For a Directed graph



The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric.

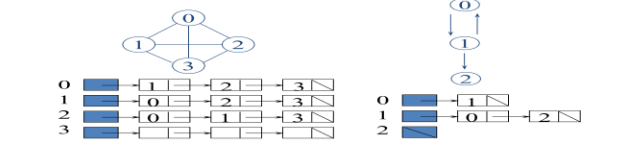
*Merits of Adjacency Matrix:*

* From the adjacency matrix, to determine the connection of vertices is easy
* The degree of a vertex is for a digraph, the row sum is the out degree, while the column sum is the in degree
* The space needed to represent a graph using adjacency matrix is n2 bits. To identify the edges in a graph, adjacency matrices will require at least O(n2) time.

1. ***Adjacency List***

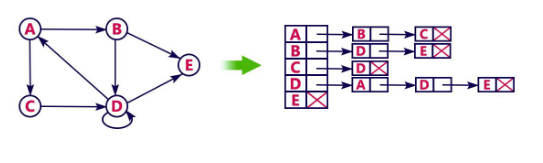
In this representation, every vertex of graph contains list of its adjacent vertices. The n rows of the adjacency matrix are represented as n chains. The nodes in chain I represent the vertices that are adjacent to vertex i.

It can be represented in two forms. In one form, array is used to store n vertices and chain is used to store its adjacencies. Example

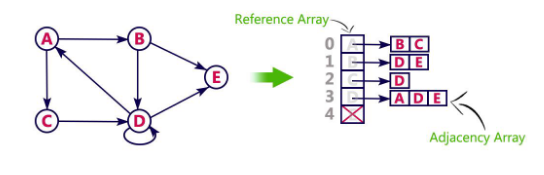


So that we can access the adjacency list for any vertex in O(1) time. Adjlist[i] is a pointer to to first node in the

Adjacency list for vertex i



This representation can also be implemented using array

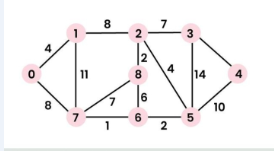


***Practical implementation of the graph control structure***

The graph data structure can be implemented to determine the shortest path from a give source to all other vertices in a given network using Dijkstra’s shortest path algorithm.

**Example**

*Input:* src = 0, the graph is given bellow



*Output*: 0 4 12 19 21 11 9 8 14

*Explanation*:

* The distance from 0 to 1 = 4
* The minimum distance from 0 to 2 = 12

Using path 0-1-2

* The minimum distance from 0 to 3 = 19

Using path 0-1-2-3

* The minimum distance from 0 to 4 = 21

Using path 0-7-6-5-4

* The minimum distance from 0 to 5 = 11

Using path 0-7-6-5

* The minimum distance from 0 to 6 = 9

Using path 0-7-6

* The minimum distance from 0 to 7 = 8

Using path 0-7

* The minimum distance from 0 to 8 = 14

Using path 0-1-2-8

**Dijkstra’s Algorithm using Adjacency Matrix**

**Algorithm:**

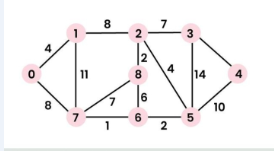
* Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
* Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign the distance value as 0 for the source vertex so that it is picked first.
* While sptSet doesn’t include all vertices
* Pick a vertex u that is not there in sptSet and has a minimum distance value.
* Include u to sptSet.
* Then update the distance value of all adjacent vertices of u.
* To update the distance values, iterate through all adjacent vertices.
* For every adjacent vertex v, if the sum of the distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

***Note:*** We use a Boolean array sptSet[] to represent the set of vertices included in SPT. If a value sptSet[v] is true, then vertex v is included in SPT, otherwise not. Array dist[] is used to store the shortest distance values of all vertices.

**Illustration of Dijkstra Algorithm**:

To understand the Dijkstra’s Algorithm let’s take a graph and find the shortest path from source to all nodes.

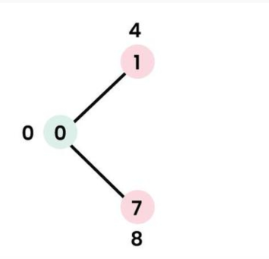
Consider below graph and src = 0



Step 1:

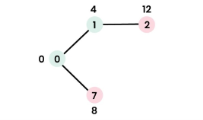
* The set sptSet is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite.
* Now pick the vertex with a minimum distance value. The vertex 0 is picked, include it in sptSet. So sptSet becomes {0}. After including 0 to sptSet, update distance values of its adjacent vertices.
* Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8.

The following sub graph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.



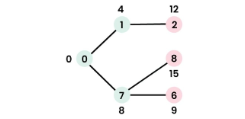
Step 2:

* Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet.
* So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1.
* The distance value of vertex 2 becomes 12.



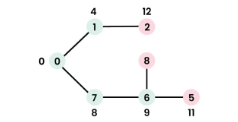
Step 3:

* Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}.
* Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).

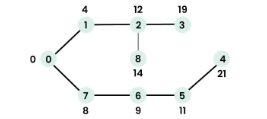


Step 4:

* Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}.
* Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until sptSet includes all vertices of the given graph. Finally, we get the following Shortest Path Tree (SPT).



This is the implementation of the above approach

#include <stdio.h>

#include <limits.h>

#define V 9

int minDistance(int dist[], int sptSet[]) {

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++)

if (sptSet[v] == 0 && dist[v] <= min)

min = dist[v], min\_index = v;

return min\_index;

}

void printSolution(int dist[]) {

printf("Vertex \t Distance from Source\n");

for (int i = 0; i < V; i++)

printf("%d \t %d\n", i, dist[i]);

}

void dijkstra(int graph[V][V], int src) {

int dist[V];

int sptSet[V];

for (int i = 0; i < V; i++)

dist[i] = INT\_MAX, sptSet[i] = 0;

dist[src] = 0;

for (int count = 0; count < V - 1; count++) {

int u = minDistance(dist, sptSet);

sptSet[u] = 1;

for (int v = 0; v < V; v++)

if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX

&& dist[u] + graph[u][v] < dist[v])

dist[v] = dist[u] + graph[u][v];

}

printSolution(dist);

}

int main() {

int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},

{4, 0, 8, 0, 0, 0, 0, 11, 0},

{0, 8, 0, 7, 0, 4, 0, 0, 2},

{0, 0, 7, 0, 9, 14, 0, 0, 0},

{0, 0, 0, 9, 0, 10, 0, 0, 0},

{0, 0, 4, 14, 10, 0, 2, 0, 0},

{0, 0, 0, 0, 0, 2, 0, 1, 6},

{8, 11, 0, 0, 0, 0, 1, 0, 7},

{0, 0, 2, 0, 0, 0, 6, 7, 0}

};

dijkstra(graph, 0);

return 0;

}

**Graph transversal algorithms**

The graph has two types of traversal algorithms. These are called the **Breadth First Search and Depth First Search.**

***Breadth-First Search***

In a breadth-first search, we begin by visiting the start vertex v. Next all unvisited vertices adjacent to v are visited. Unvisited vertices adjacent to these newly visited vertices are then visited and so on. Algorithm BFS

We use the following steps to implement BFS traversal...

*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 the adjacent vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.

*Step 4*: When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.

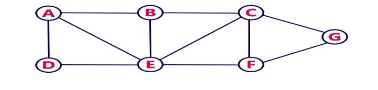
*Step 5*: Repeat step 3 and 4 until queue becomes empty.

*Step 6:* When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

**Analysis of BFS:**

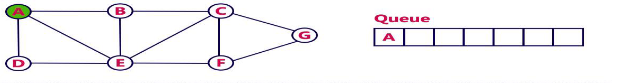
Each visited vertex enters the queue exactly once. So the while loop is iterated at most n times If an adjacency matrix is used the loop takes O(n) time for each vertex visited. The total time is therefore, O(n2). If adjacency lists are used the loop has a total cost of d0 + … + dn-1 = O(e), where d is the degree of vertex i. As in the case of DFS all visited vertices together with all edges incident to them, form a connected component of G.

Consider the following graph to perform BFS transversal



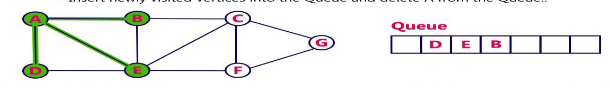
Step 1: -select the vertex A as starting point (visit A)

-Insert A into the queue



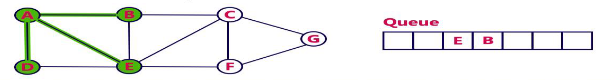
Step 2:- visit all adjecent vertices of A which are not visited (D,E,B)

-Insert newly visited vertices into the queue and delete A from the queue



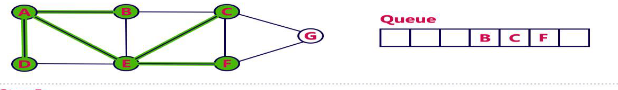
Step 3: -visit all adjecent vertices of D which are not visited (there is no vertex)

-Delete D from queue



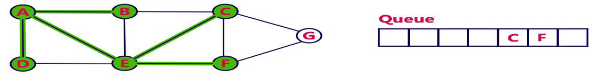
Step 4: -visit all adjecent vertices of E which are not visited (C,F)

-Insert newly visited vertices into the queue and delete E from the queue



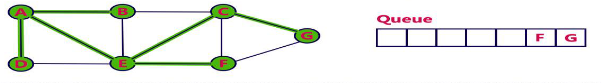
Step 5: -visit all adjecent vertices of B which are not visited (there is no vertex)

-Delete B from queue



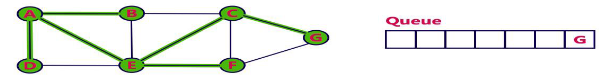
Step 6: -visit all adjecent vertices of C which are not visited (G)

-Insert newly visited vertices into the queue and delete C from the queue



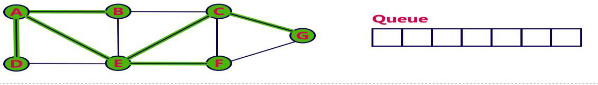
Step 7: -visit all adjecent vertices of F which are not visited (there is no vertex)

-Insert newly visited vertices into the queue and delete F from the queue



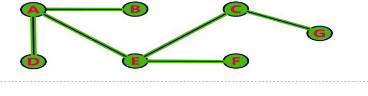
Step 8:- visit all adjecent vertices of G which are not visited (there is no vertex)

-Insert newly visited vertices into the queue and delete G from the queue



-Queue becomes empty, so stop the BFS process

-Final result of BFS is a spanning tree as shown bellow



***Depth-First Search***

DFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops.

We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal of a graph.

We use the following steps to implement DFS traversal...

*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 the adjacent vertex of the verex which is at top of the stack which is not visited and push it on to the stack.

*Step 4*: Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.

*Step 5*: When there is no new vertex to be visit then use back tracking and pop one vertex from the stack.

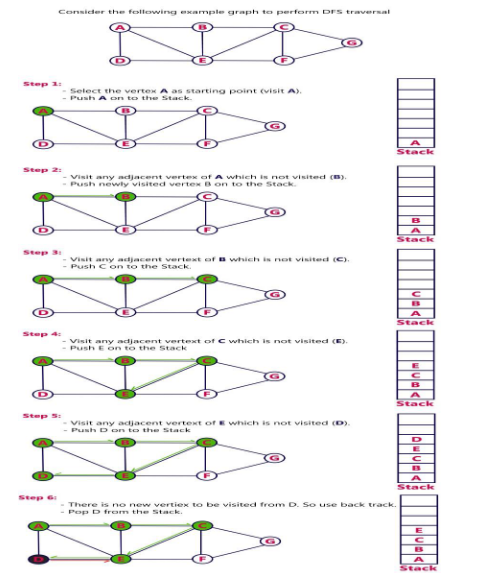
*Step 6*: Repeat steps 3, 4 and 5 until stack becomes Empty.

*Step 7:* When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

**Analysis for DFS:**

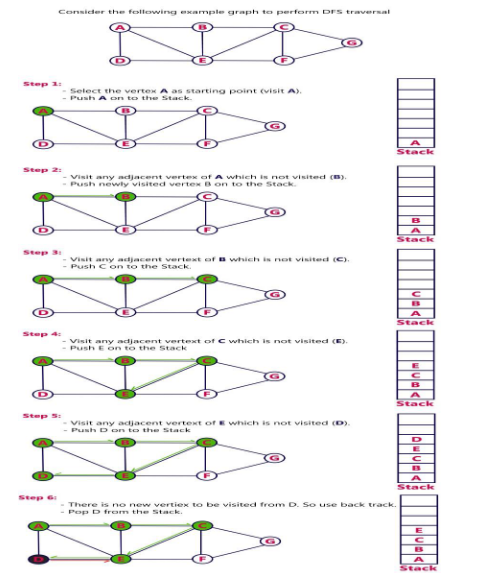
When G is represented by its adjacency lists, the vertices w adjacent to v can be determined by following a chain of links. Since DFS examines each node in the adjacency lists at most once and there are 2e list nodes the time to complete the search is O(e). If G is represented by its adjacency matrix then the time to determine all vertices adjacent to v is O(n). Since at most n vertices are visited the total time is O(n2).

Consider the following example graph to perform DFS transversal



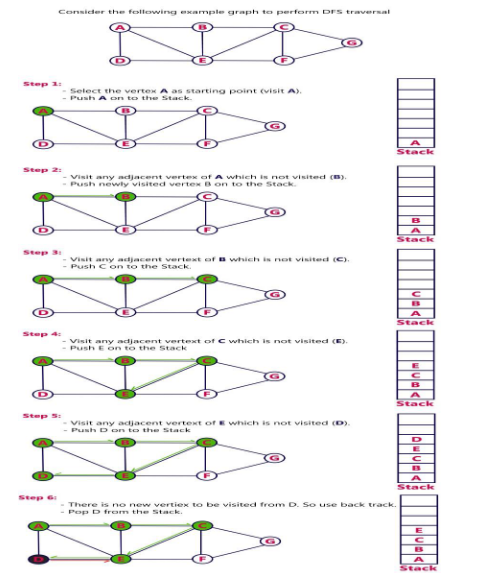
Step 1: -select the vertex A as a starting point (visit A)

-Push A onto the stack



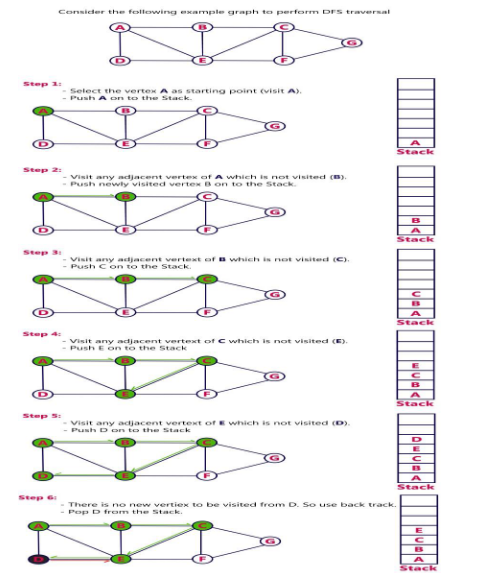
Step 2: -visit any adjecent vertex of A which is not visited (B)

-Push newly visited vertex B on the stack



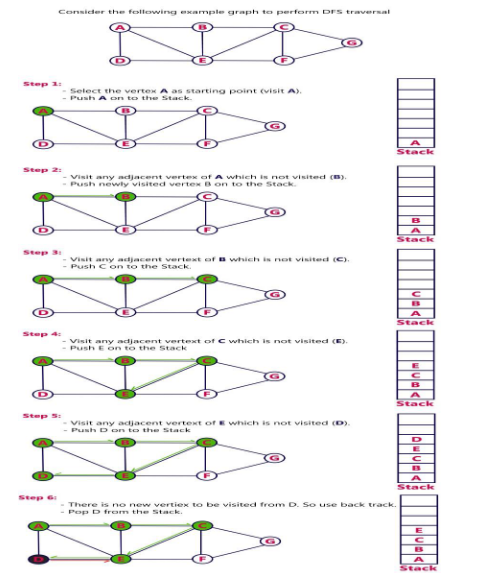
Step 3:- visit any adjecent vertex of B which is not visited (C)

-Push newly visited vertex C on the stack



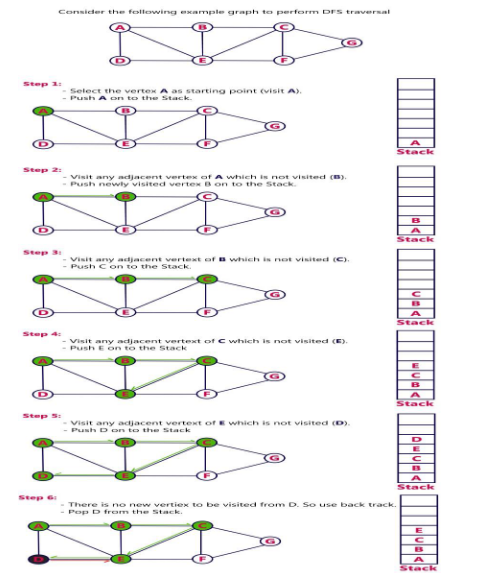
Step 4: -visit any adjecent vertex of C which is not visited (E)

-Push newly visited vertex E on the stack



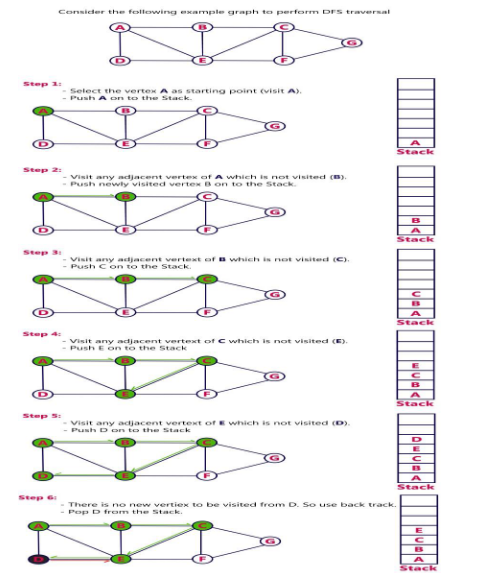
Step 5:- visit any adjecent vertex of E which is not visited (D)

-Push newly visited vertex D on the stack



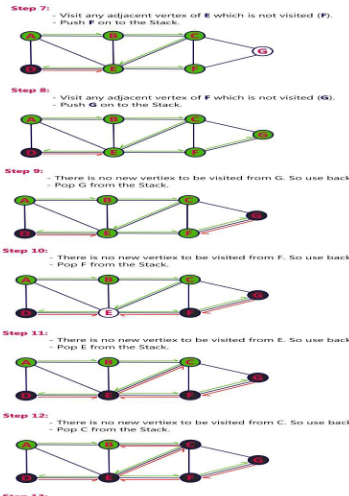
Step 6: -There is no new vertex to be visited from D.so use backtrack

-Pop D from the stack



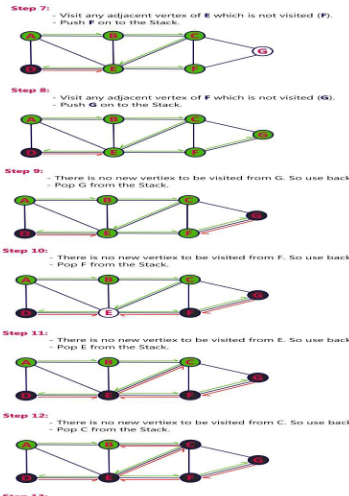
Step 7: -visit any adjecent vertex of E which is not visited (F)

-Push newly visited vertex F on the stack



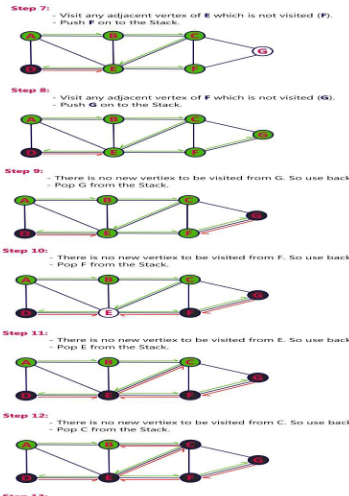
Step 8:- visit any adjecent vertex of F which is not visited (G)

-Push newly visited vertex G on the stack



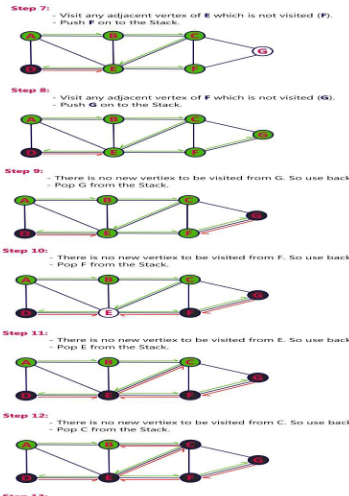
Step 9:- There is no new vertex to be visited from G. so use backtrack

-Pop G from the stack



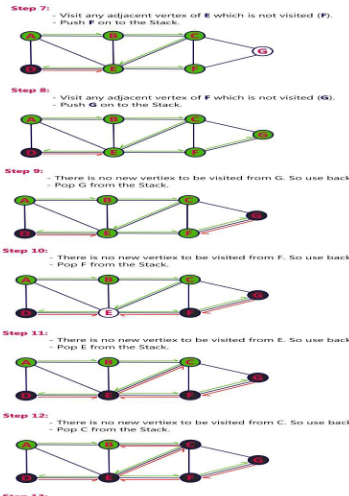
Step 10: -There is no new vertex to be visited from F.so use backtrack

-Pop F from the stack



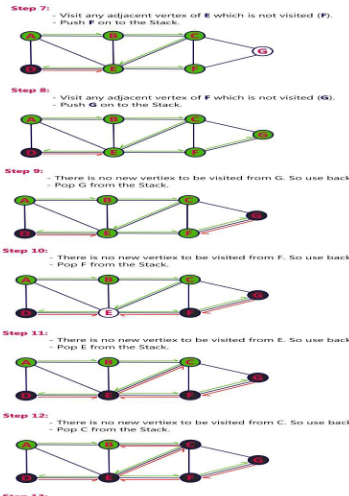
Step 11:- There is no new vertex to be visited from E.so use backtrack

-Pop E from the stack



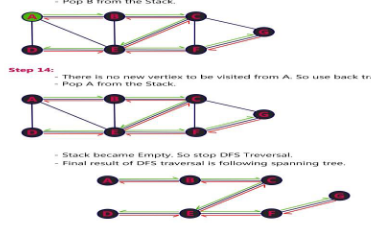
Step 12: -There is no new vertex to be visited from C.so use backtrack

-Pop C from the stack



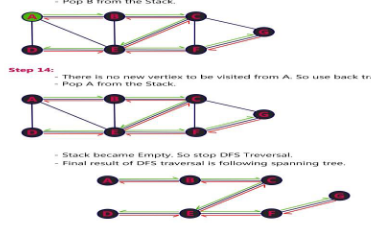
Step 13: -There is no new vertex to be visited from B.so use backtrack

-Pop B from the stack



Step 14: -There is no new vertex to be visited from A.so use backtrack

-Pop A from the stack



-Stack becomes empty so stop DFS transversal

-Final result of DFS is the following spanning tree

