

Unit 6

Graph



CHAPTER-6

GRAPH



Basic Terminologies and Representations

Definitions: Vertices, edges, paths, etc

Representations: Adjacency list and adjacency matrix





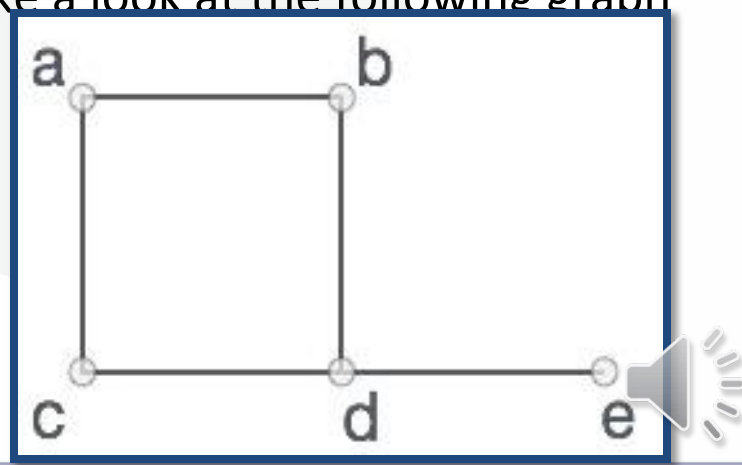
Definitions: Graph, Vertices, Edges

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph –

$V = \{a, b, c, d, e\}$

$E = \{ab, ac, bd, cd, de\}$





Graph data Structure

Vertex – Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.

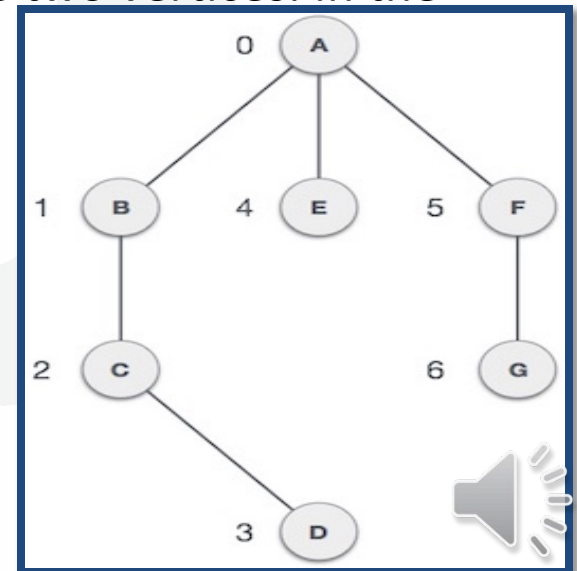
Edge – Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.



Basic Terminologies and Representations

Adjacency – Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.

Path – Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



Basic Operations

Add Vertex – Adds a vertex to the graph.

Add Edge – Adds an edge between the two vertices of the graph.

Display Vertex – Displays a vertex of the graph.





Motivation

Many algorithms use a graph representation to represent data or the problem to be solved

Examples:

- Cities with distances between
- Roads with distances between intersection points
- Course prerequisites
- Network
- Social networks
- Program call graph and variable dependency graph



Graph Classifications

There are several common kinds of graphs

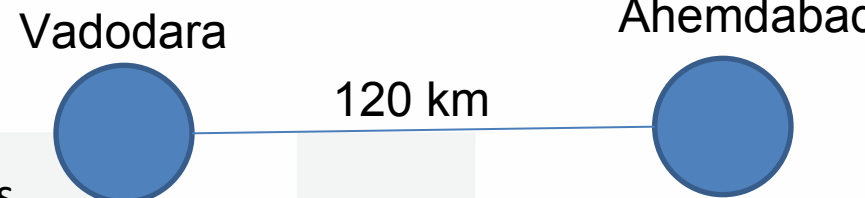
- Weighted or Unweighted
- Directed or Undirected
- Cyclic or Acyclic



Weighted Graph & Unweighted Graph

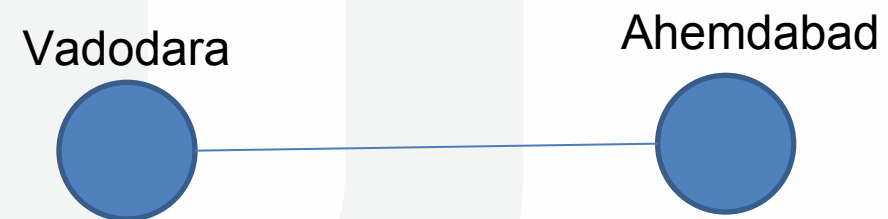
Weighted graph: edges have a weight

Weight typically shows cost of traversing
Example: weights are distances between cities



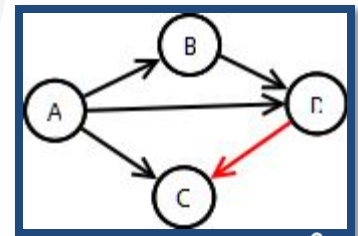
Unweighted graph: edges have no weight

Edges simply show connections
Example: link between two cities



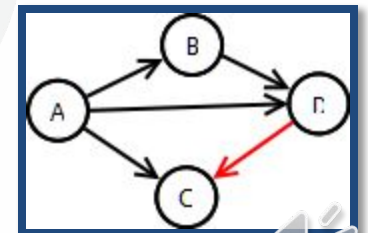
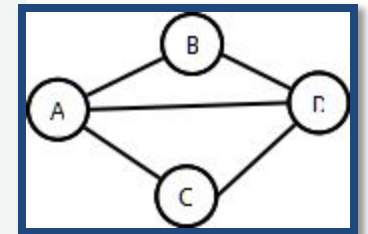
Directed & Undirected

- **Undirected Graphs:** each edge can be traversed in **either direction**
- **Directed Graphs:** each edge can be traversed **only in a specified direction**
 - Edge drawn as arrow
 - Edge can only be traversed in direction of arrow
 - Example: $E = \{(A,B), (A,C), (A,D), (B,D), (D,C)\}$
 - Thus: (u,v) and (v,u) are not the same edge
 - In the example, $(D,C) \in E$, $(C,D) \notin E$
 - What would edge (B,A) look like? Remember $(A,B) \neq (B,A)$
 - A node can have an edge to itself (eg (A,A) is valid)



Cyclic & Acyclic

- A **Cyclic** graph contains cycles
Example: roads (normally)
- An **acyclic** graph contains no cycles
Example: Course prerequisite

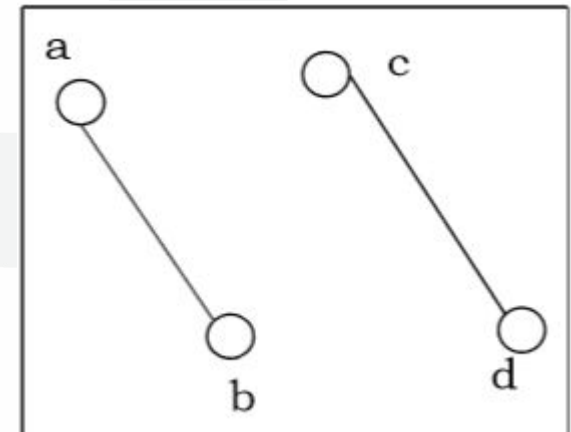
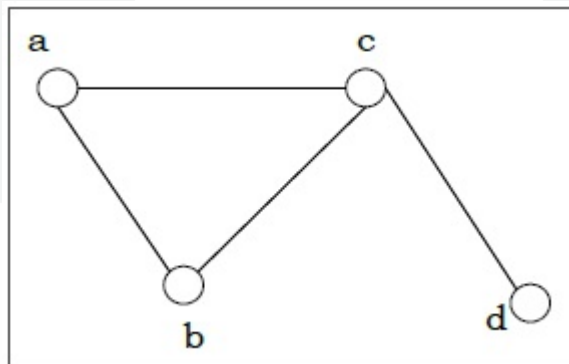


Connected and Unconnected Graphs

- An *undirected* graph is **connected** if every pair of vertices has a path between it

Otherwise it is unconnected

- An unconnected graph can be broken in to **connected components**
- A *directed* graph is **strongly connected** if every pair of vertices has a path between them, in **both directions**



Trees and Minimum Spanning Trees

Tree: undirected, connected graph with no cycles

Example ...

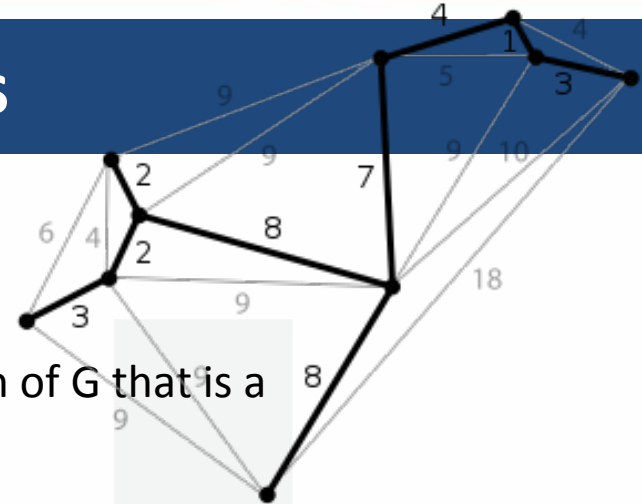
Spanning tree: a **spanning tree** of G is a connected subgraph of G that is a tree

Example ...

Minimum spanning tree (MST): a spanning tree with minimum weight

Example ...

Spanning trees and minimum spanning tree are not necessarily unique





Graphs: Terminology Involving Paths

Path: sequence of vertices in which each pair of successive vertices is connected by an edge

Cycle: a path that starts and ends on the same vertex

Simple path: a path that does not cross itself

That is, no vertex is repeated (except first and last)

Simple paths cannot contain cycles

Length of a path: Number of edges in the path

Sometimes the sum of the weights of the edges

Examples

A sequence of vertices: (A, B, C, D) [Is this path, simple path, cycle?]

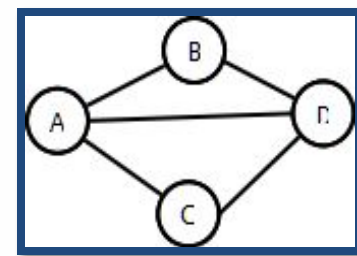
(A, B, D, A, C) [path, simple path, cycle?]

(A, B, D, A, C) [path, simple path, cycle?]

Cycle: ?

Simple Cycle: ?

Lengths?





Data Structures for Representing Graphs

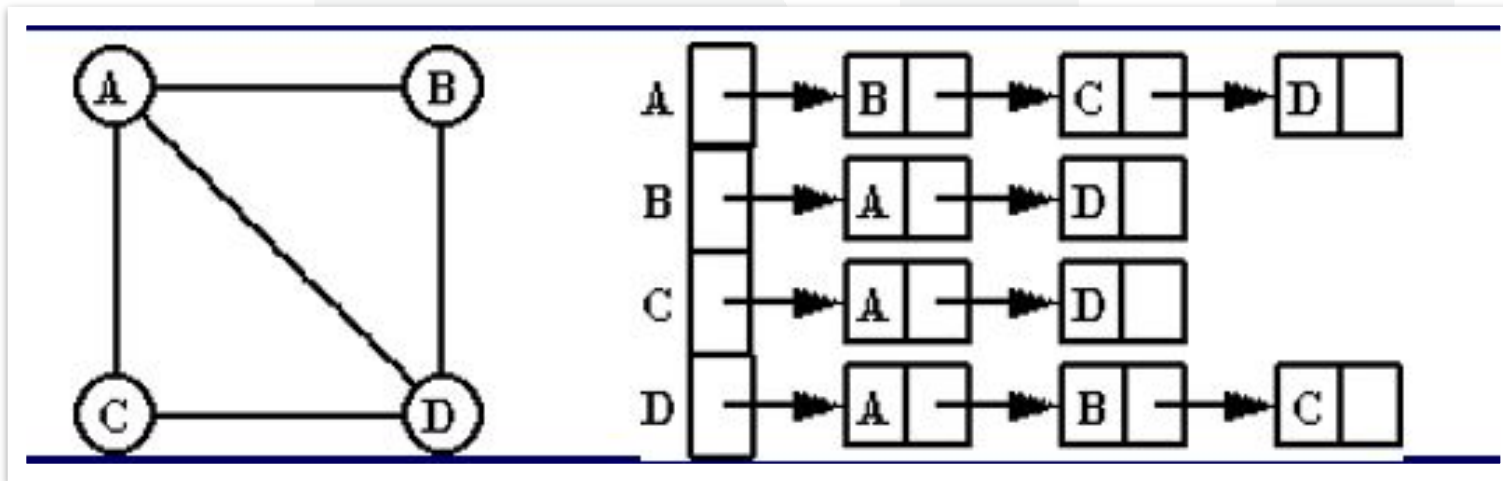
Two common data structures for representing graphs:

- Adjacency lists
- Adjacency matrix



Adjacency List Representation

A representation of the graph consisting of a list of nodes, with each node containing a list of its neighboring nodes.



This representation takes $O(|V| + |E|)$ space.



Adjacency List Representation

Each node has a list of adjacent nodes:

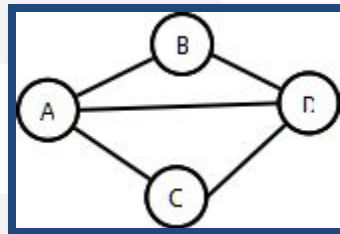
Example (undirected graph):

A: B, C, D

B: A, D

C: A, D

D: A, B, C



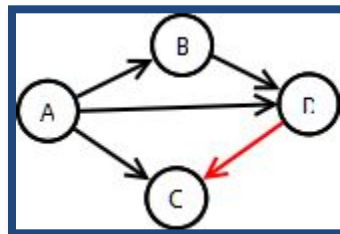
Example (directed graph):

A: B, C, D

B: D

C: Nil

D: C



Adjacency List Representation

Time:

To visit each node that is adjacent to node u : $\Theta(\text{degree}(u))$

To determine if node u is adjacent to node v : $\Theta(\text{degree}(u))$





Adjacency Matrix Representation

Graphs $G = (V, E)$ can be represented by adjacency matrices $G[v_1..v_{|V|}, v_1..v_{|V|}]$, where the rows and columns are indexed by the nodes, and the entries $G[v_i, v_j]$ represent the edges. In the case of unlabeled graphs, the entries are just Boolean values.



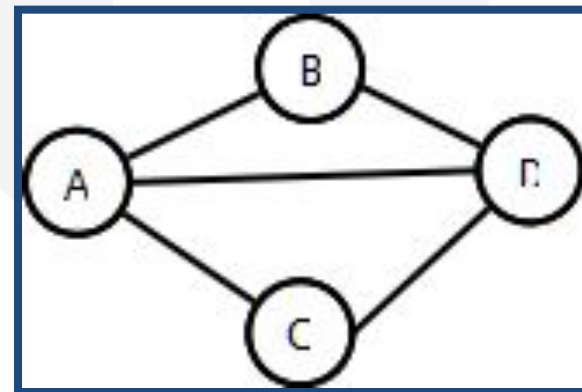
Adjacency Matrix Representation

Adjacency Matrix: 2D array containing weights on edges

- Row for each vertex
- Column for each vertex
- Entries contain weight of edge from row vertex to column vertex
- Entries contain ∞ last if no edge from row vertex to column vertex
- Entries contain 0 on diagonal (if self edges not allowed)

Example undirected graph (assume self-edges not allowed):

| | A | B | C | D |
|---|---|----------|----------|---|
| A | 0 | 1 | 1 | 1 |
| B | 1 | 0 | ∞ | 1 |
| C | 1 | ∞ | 0 | 1 |
| D | 1 | 1 | 1 | 0 |



Adjacency Matrix Representation

Example directed graph (assume self-edges allowed):

| | A | B | C | D |
|---|----------|----------|----------|----------|
| A | ∞ | 1 | 1 | 1 |
| B | ∞ | ∞ | ∞ | 1 |
| C | ∞ | ∞ | ∞ | ∞ |
| D | ∞ | ∞ | 1 | ∞ |

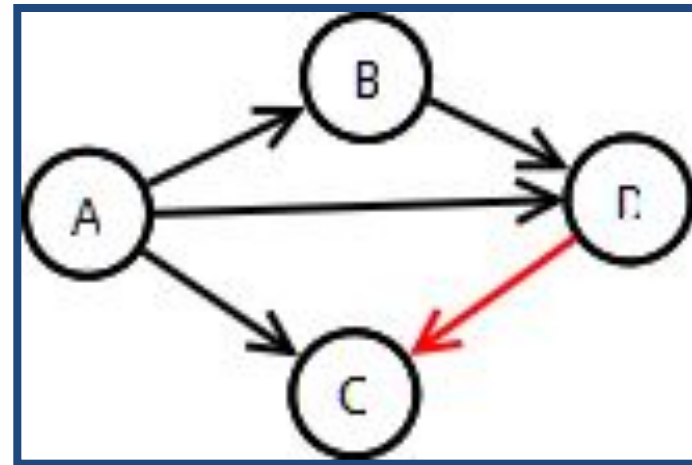
Can store weights in cells

Space: $\Theta(V^2)$

Time:

To visit each node that is adjacent to node u : $\Theta(V)$

To determine if node u is adjacent to node v : $\Theta(1)$





Graphs Search and Traversal Algorithms

Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

- ***Depth-First Traversal***
- ***Breadth-First Traversal***



DFS (Depth First Search)

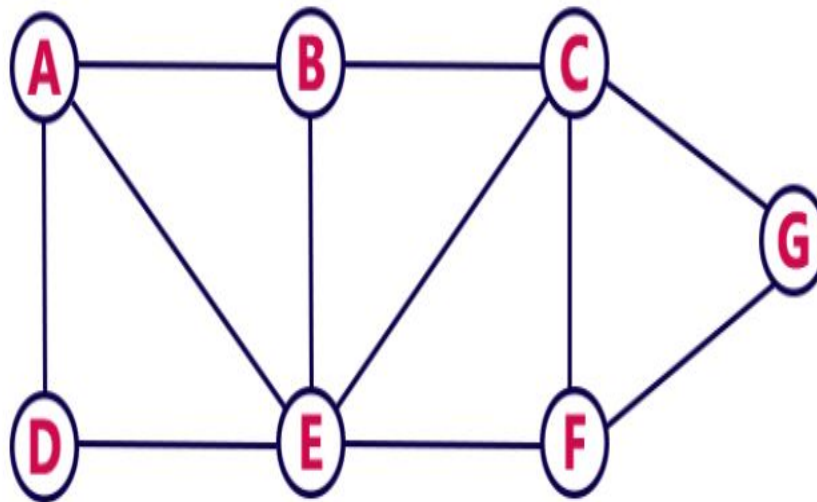
DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal.

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 non-visited **adjacent** vertices 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 the stack.
- Step 5** - When there is no new vertex to 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

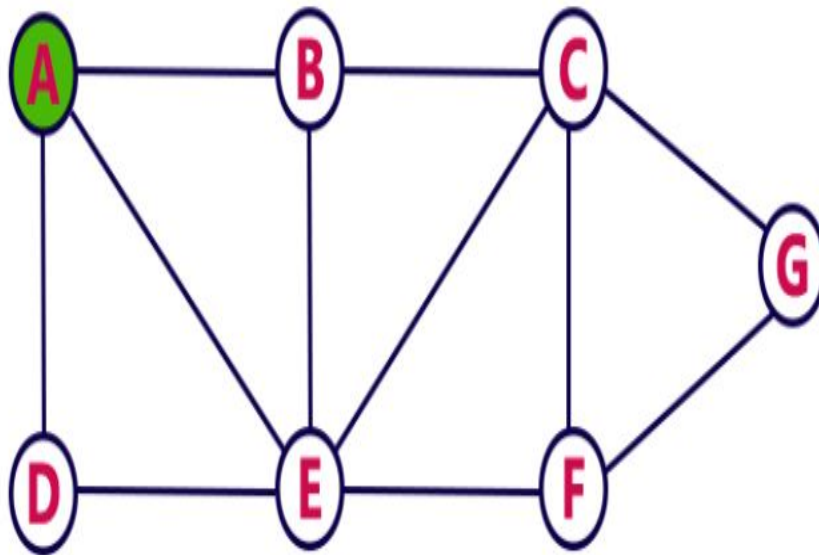
Example

Consider the following example graph to perform DFS traversal



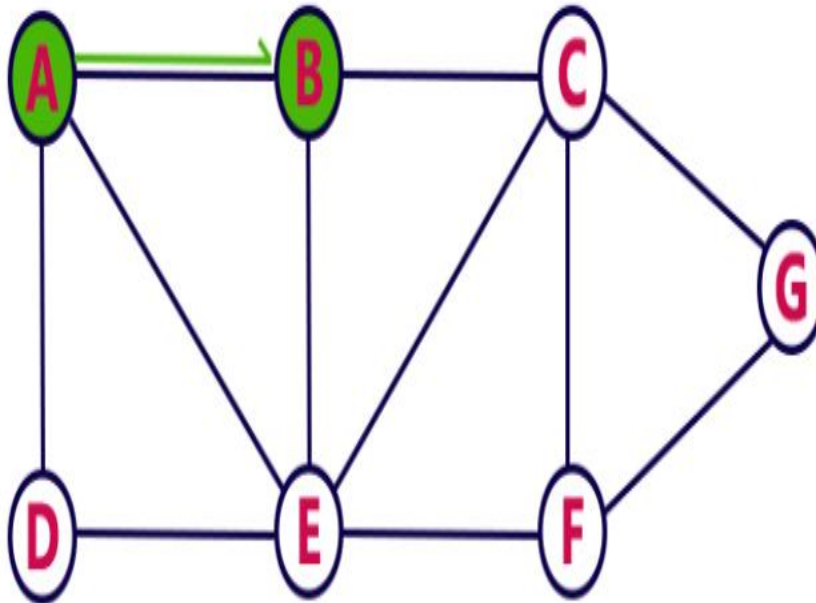
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



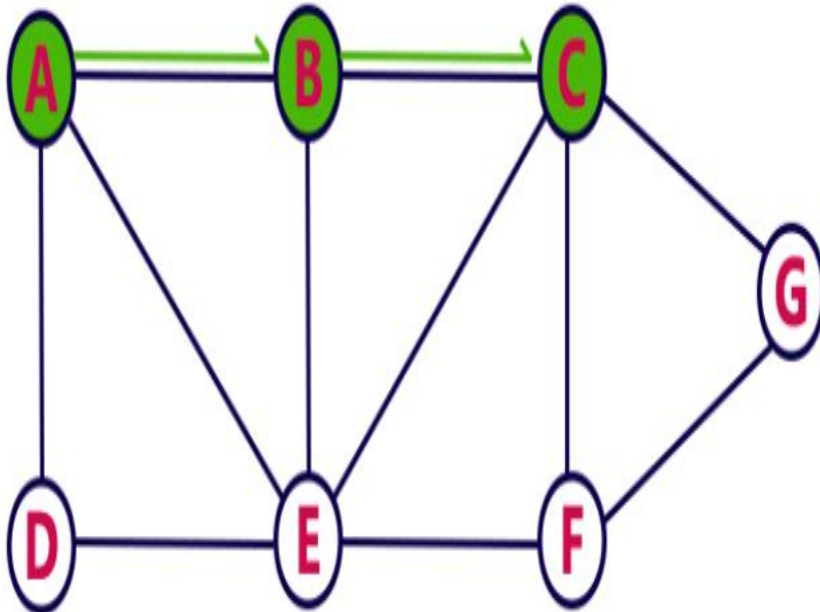
Step 2:

- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



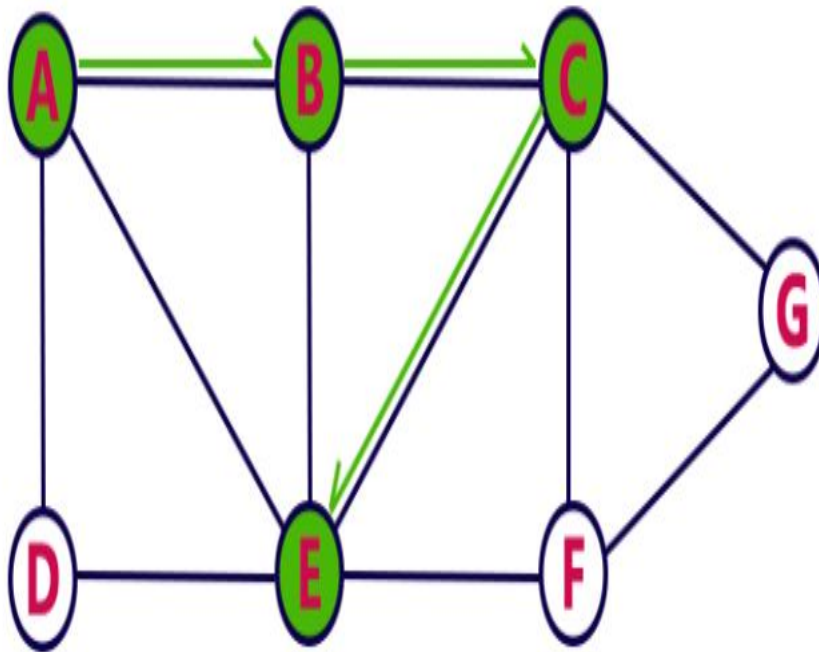
Step 3:

- Visit any adjacent vertex of **B** which is not visited (**C**).
- Push C on to the Stack.



Step 4:

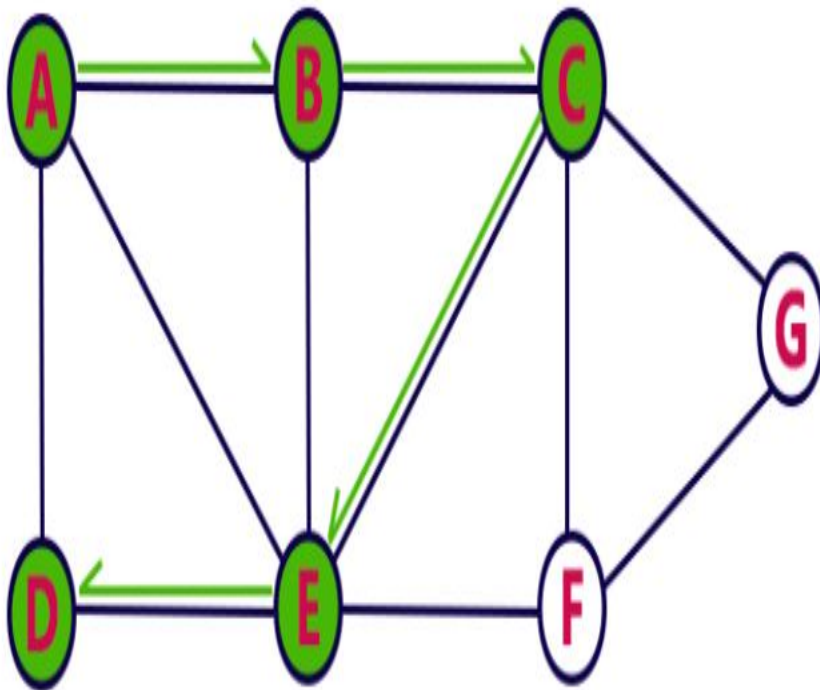
- Visit any adjacent vertex of **C** which is not visited (**E**).
- Push E on to the Stack



Stack

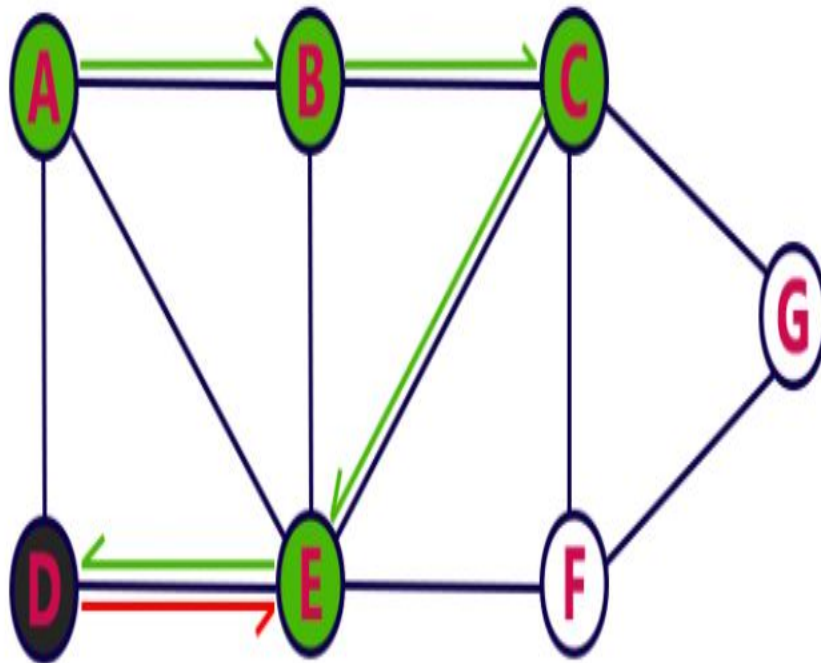
Step 5:

- Visit any adjacent vertex of **E** which is not visited (**D**).
- Push D on to the Stack



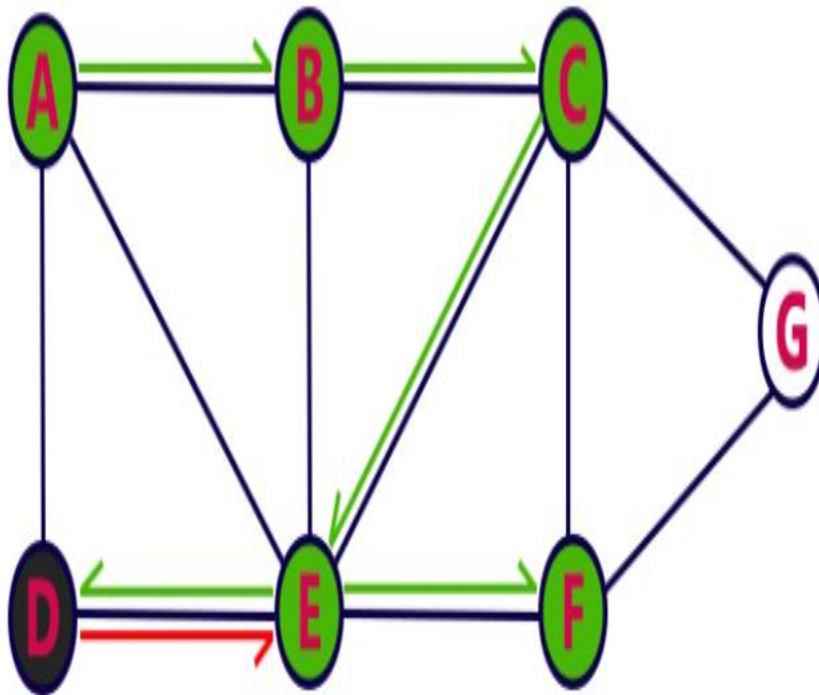
Step 6:

- There is no new vertex to be visited from D. So use back track.
- Pop D from the Stack.



Step 7:

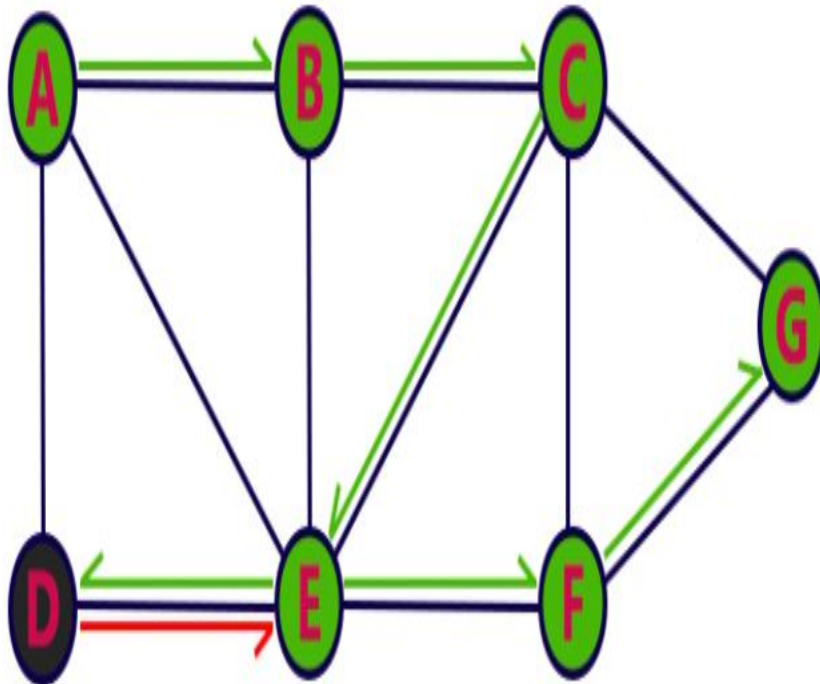
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.



Stack

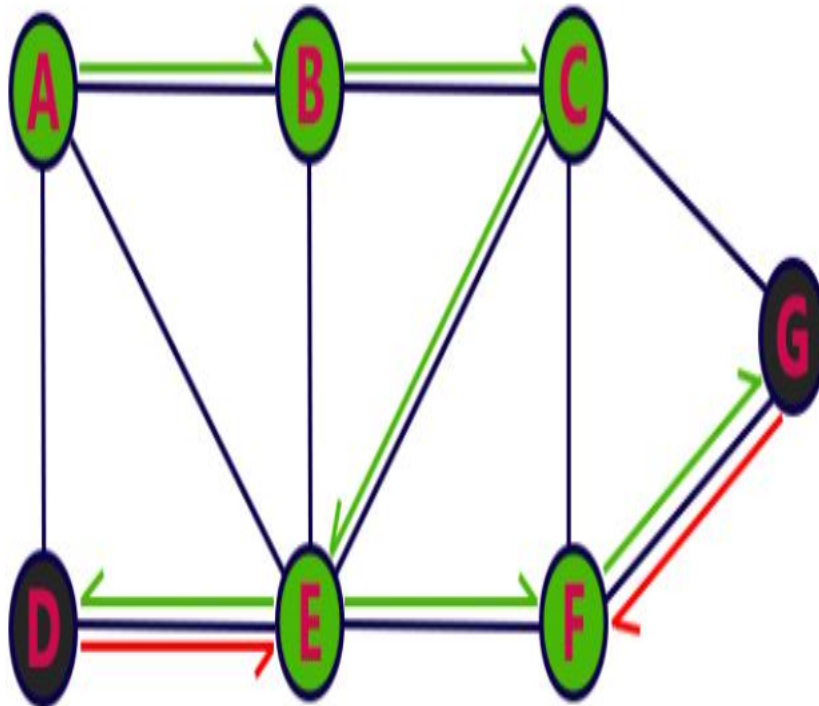
Step 8:

- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.



Step 9:

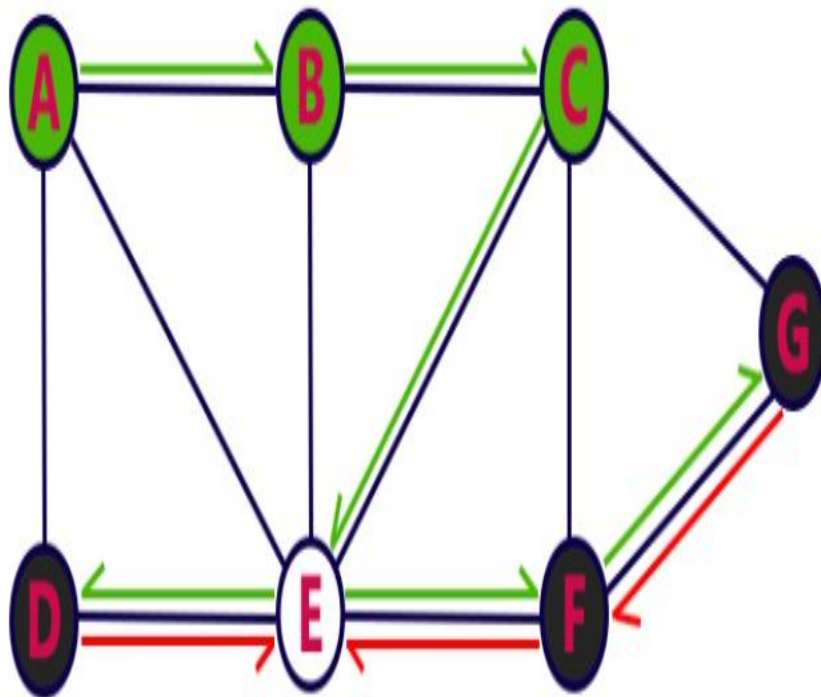
- There is no new vertex to be visited from G. So use back track.
- Pop G from the Stack.



Stack

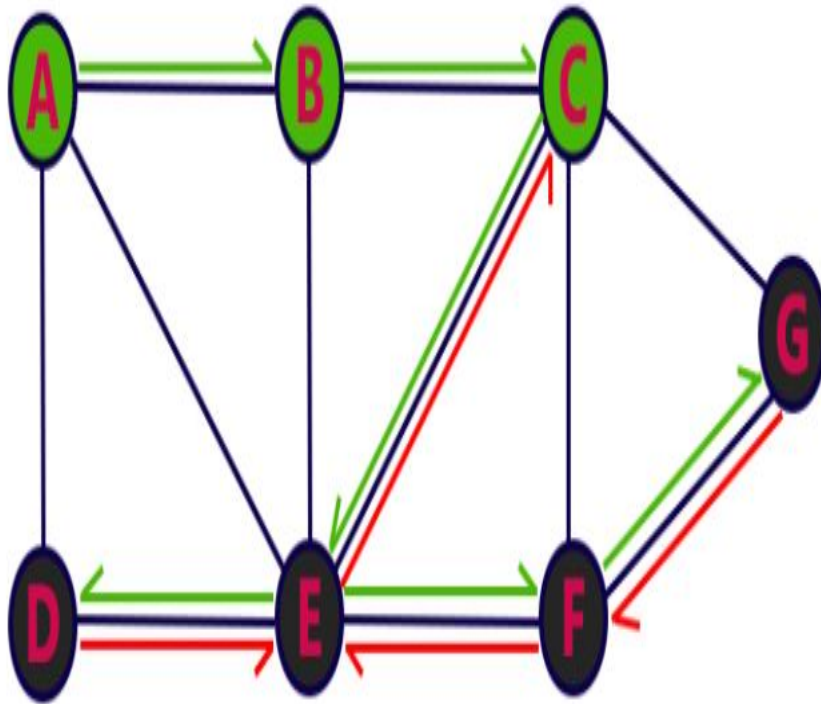
Step 10:

- There is no new vertex to be visited from F. So use back track.
- Pop F from the Stack.



Step 11:

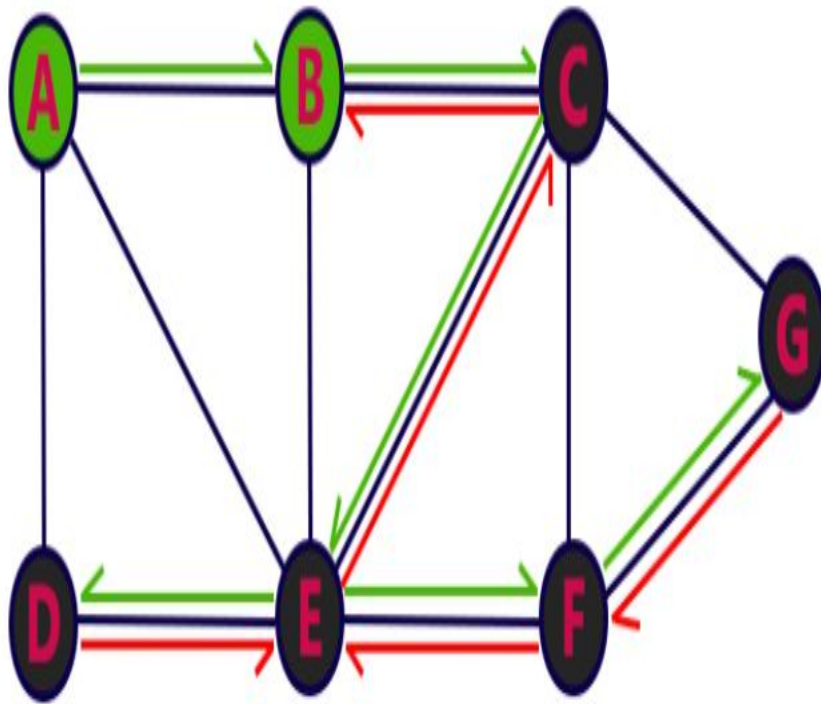
- There is no new vertex to be visited from E. So use back track.
- Pop E from the Stack.



Stack

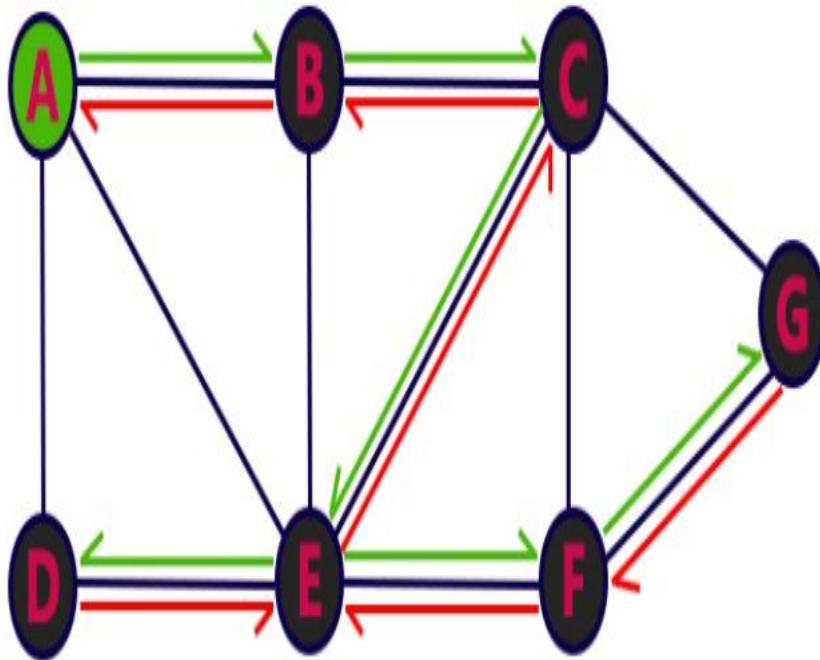
Step 12:

- There is no new vertex to be visited from C. So use back track.
- Pop C from the Stack.



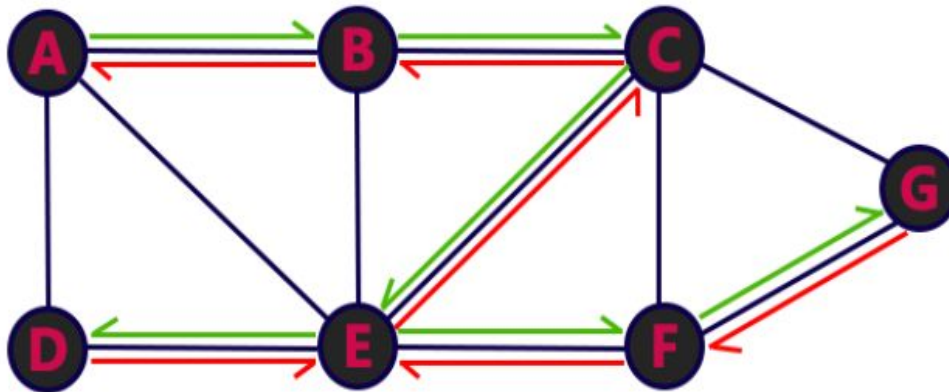
Step 13:

- There is no new vertex to be visited from B. So use back track.
- Pop B from the Stack.

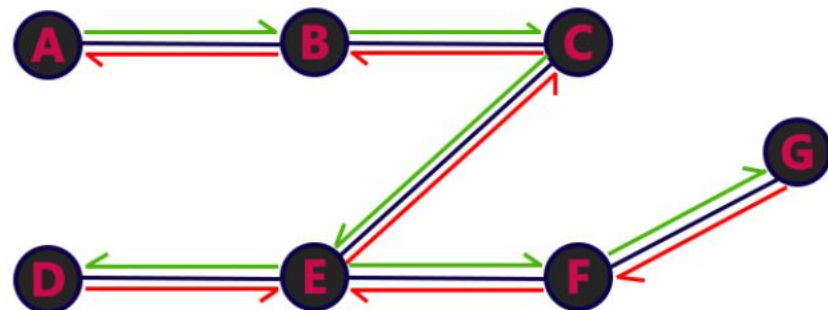


Step 14:

- There is no new vertex to be visited from A. So use back track.
- Pop A from the Stack.



- Stack became Empty. So stop DFS Traversal.
- Final result of DFS traversal is following spanning tree.



BFS (Breadth First Search)

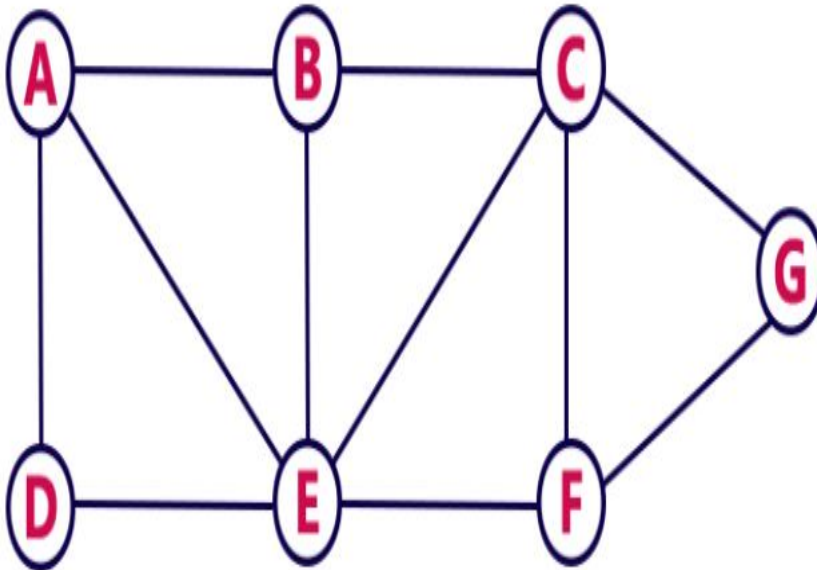
BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

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 non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.
- Step 4** - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
- Step 5** - Repeat steps 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

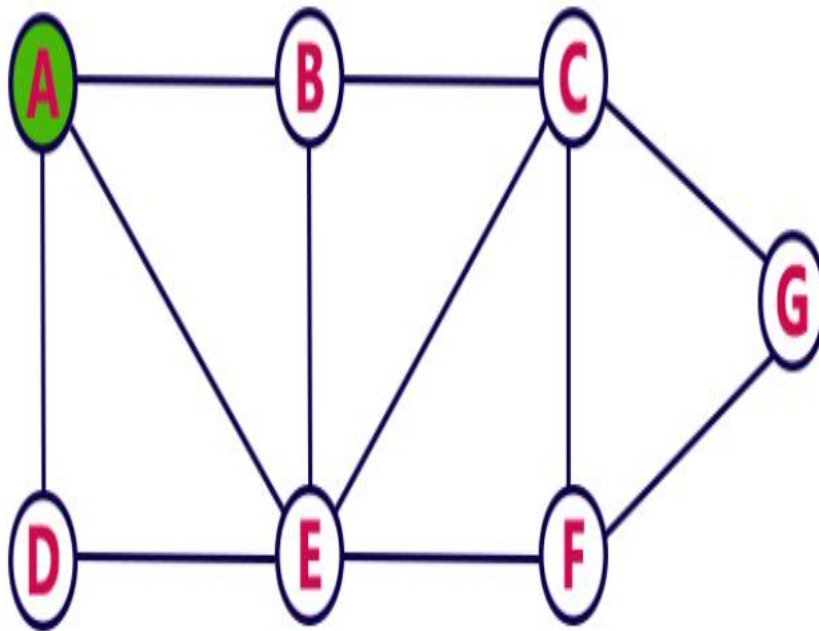
Example

Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

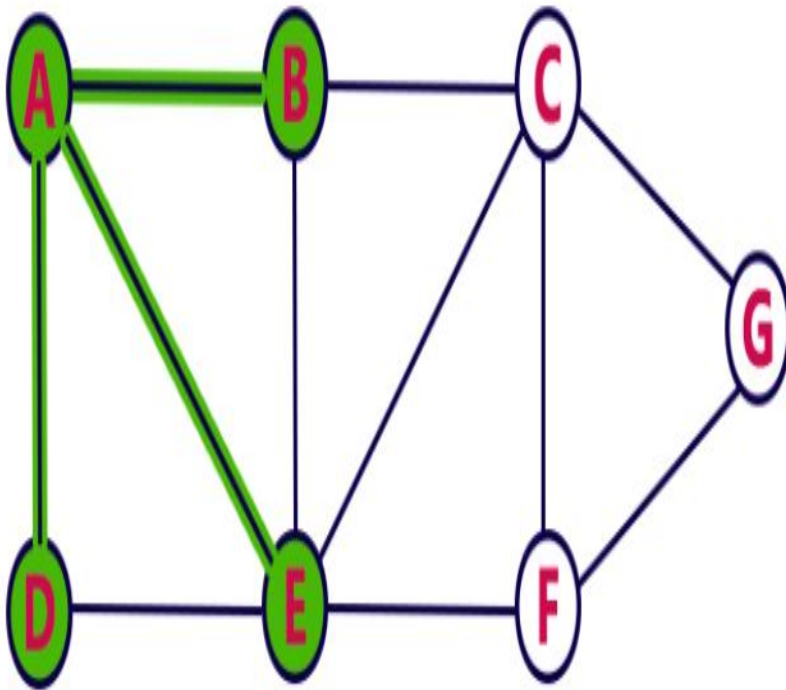


Queue



Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

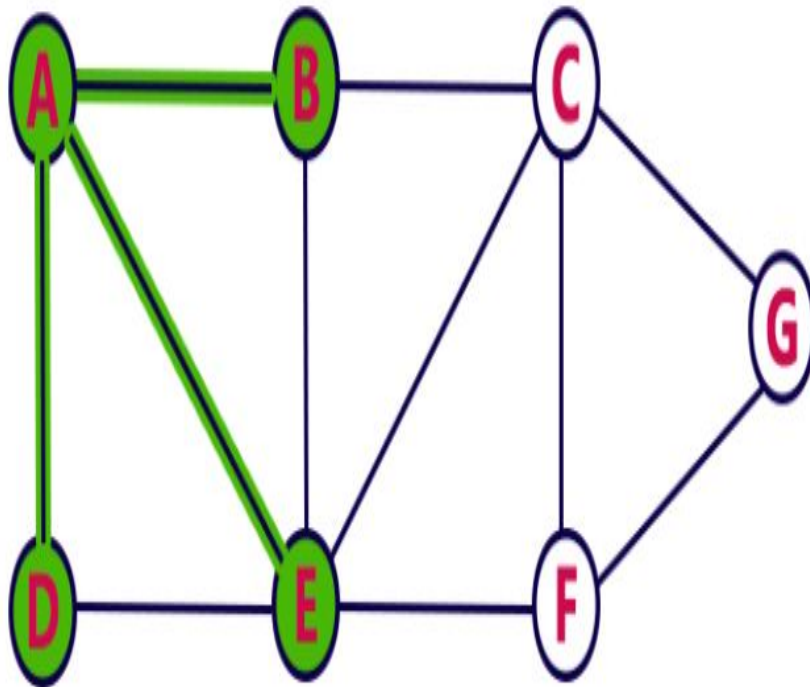


Queue

| | | | | | | |
|--|---|---|---|--|--|--|
| | D | E | B | | | |
|--|---|---|---|--|--|--|

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

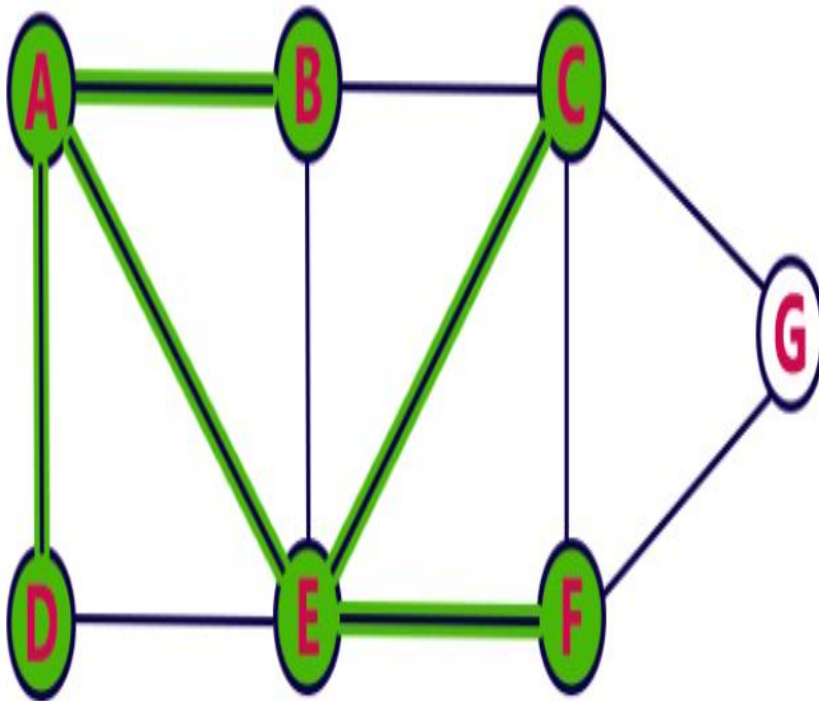


Queue

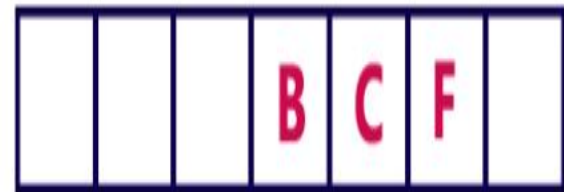
| | | | | | | |
|--|--|---|---|--|--|--|
| | | E | B | | | |
|--|--|---|---|--|--|--|

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

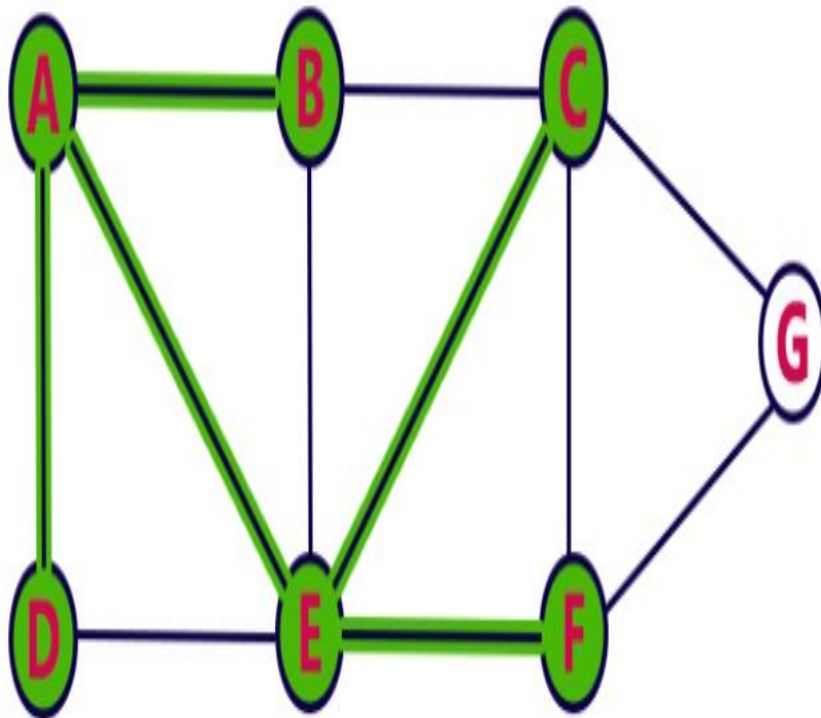


Queue



Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

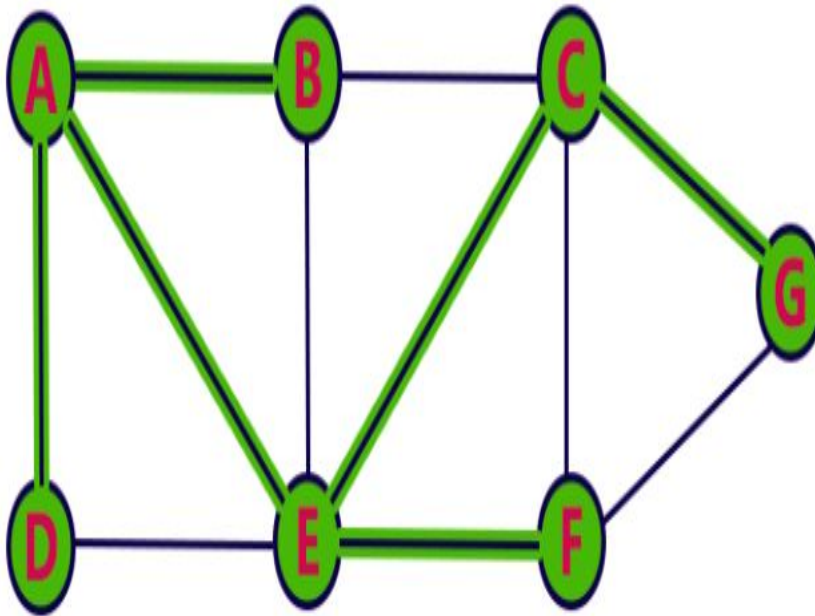


Queue

| | | | | | | |
|--|--|--|--|---|---|--|
| | | | | C | F | |
|--|--|--|--|---|---|--|

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

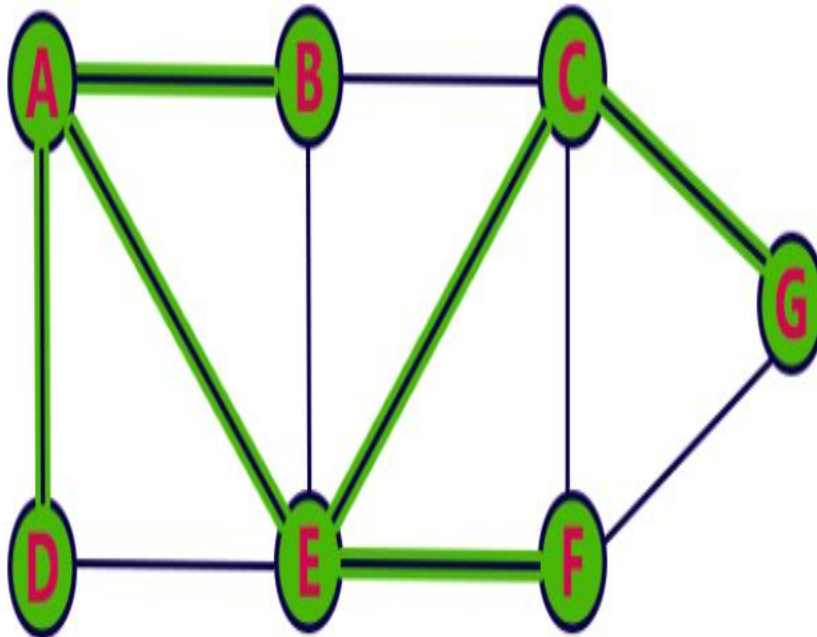


Queue

| | | | | | | |
|--|--|--|--|--|---|---|
| | | | | | F | G |
|--|--|--|--|--|---|---|

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

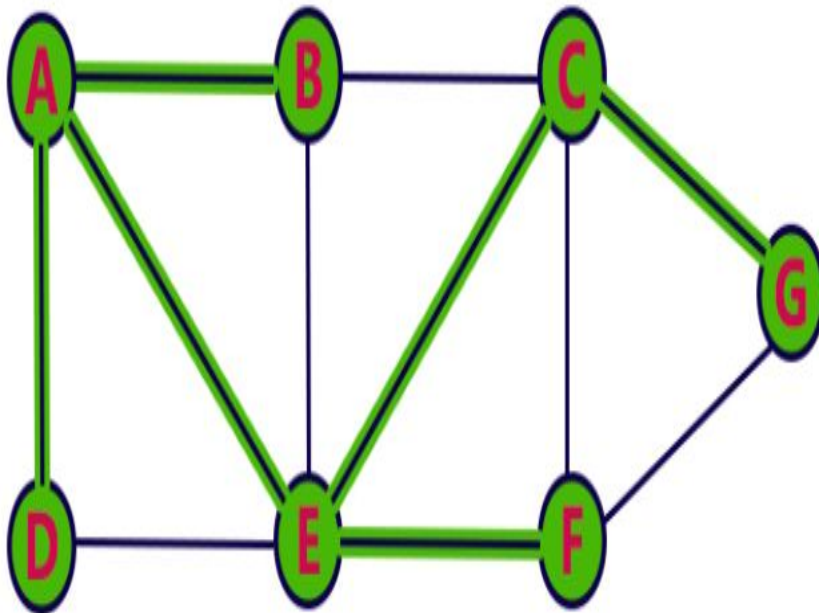


Queue



Step 8:

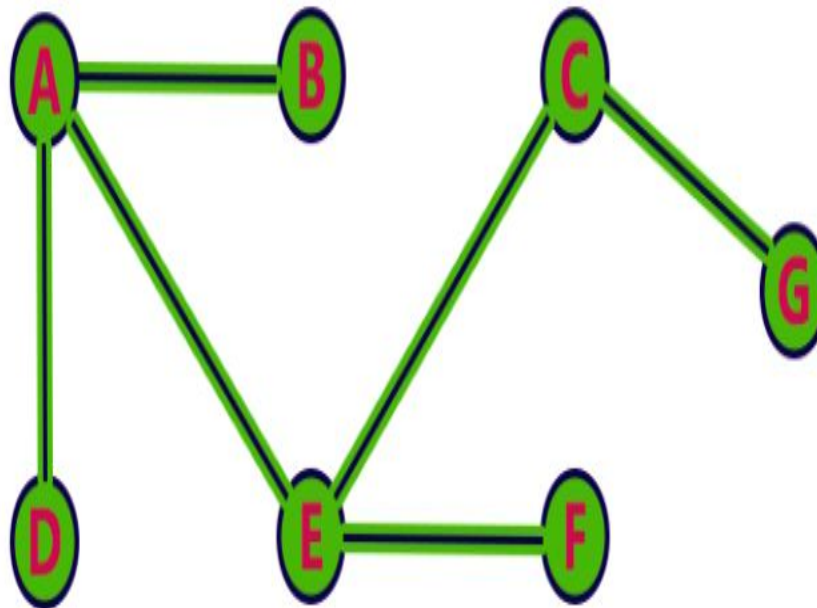
- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue



- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Complexity Analysis

Assume that graph is connected. Depth-first search visits every vertex in the graph and checks every edge its edge. Therefore, DFS complexity is $O(V + E)$. As it was mentioned before, if an adjacency matrix is used for a graph representation, then all edges, adjacent to a vertex can't be found efficiently, that results in $O(V^2)$ complexity.





Breadth First Search(BFS) Complexity

1. Assume a particular node has been designated as starting point
2. Let A be the last node Visitor and A has A neighbor N1, N2 until Nk.
3. A Breadth First Search will

- Visit N1 then N2 and so forth through Nk.
- Proceed to traverse all the unvisited neighbours of N1 then
- Traverse the immediate neighbor of N2.....Nk in a similar fashion.

Breadth First Search algorithm used in

- Prim's MST algorithm.
- Dijkstra's single source shortest path algorithm.

Like depth first search, BFS traverse a connected component of a given graph and defines a spanning tree.





Pre Requisites

As with the depth first search (DFS), the discovery edges form a spanning tree, which in

this case we call the BSF-tree.

BSF used to solve following problem

- Testing whether graph is connected.
- Computing a spanning forest of graph.
- Computing, for every vertex in graph, a path with the minimum number of edges between
- Start vertex and current vertex or reporting that no such path exists.
- Computing a cycle in graph or reporting that no such cycle exists.

Analysis

Total running time of BFS is $O(V + E)$.






Depth First Search(DFS) Complexity

As with the depth first search (DFS), the discovery edges form a spanning tree, which in this case we call the BSF-tree.

BSF used to solve following problem

- Testing whether graph is connected.
- Computing a spanning forest of graph.
- Computing, for every vertex in graph, a path with the minimum number of edges between start vertex and current vertex or reporting that no such path exists.
-  Computing a cycle in graph or reporting that no such cycle exists.

Analysis

Total running time of BFS is $O(V + E)$.





Complexity

Graph Algorithms

| Algorithm | Time Complexity | | Space Complexity |
|--------------------------|--------------------|--------------------|------------------|
| | Average | Worst | Worst |
| Dijkstra's algorithm | $O(E \log V)$ | $O(V ^2)$ | $O(V + E)$ |
| A* search algorithm | $O(E)$ | $O(b^d)$ | $O(b^d)$ |
| Prim's algorithm | $O(E \log V)$ | $O(V ^2)$ | $O(V + E)$ |
| Bellman–Ford algorithm | $O(E \cdot V)$ | $O(E \cdot V)$ | $O(V)$ |
| Floyd-Warshall algorithm | $O(V ^3)$ | $O(V ^3)$ | $O(V ^2)$ |
| Topological sort | $O(V + E)$ | $O(V + E)$ | $O(V + E)$ |



Applications of Graphs

- Finding Minimum Spanning Trees (MST) – Prim's and Kruskal's Algorithm
- Shortest path - Dijkstra's Algorithm
- Transitive closure – Warshall's Algorithm
- Topological sort.



× ○ DIGITAL LEARNING CONTENT



Parul[®] University



www.paruluniversity.ac.in

