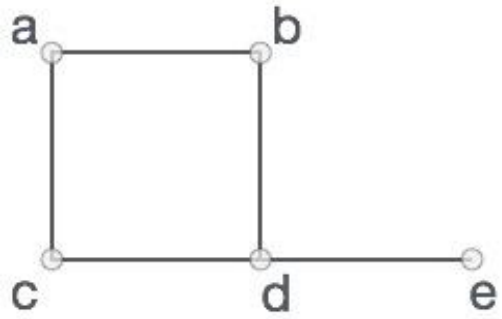
Graphs

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 –



In the above graph,

V = {a, b, c, d, e} E = {ab, ac, bd, cd, de}

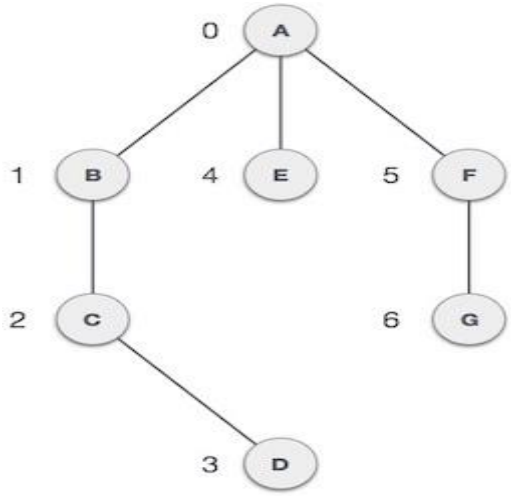
Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms −

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.

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.



Examples of Graph applications:

o Cities with distances between

o Roads with distances between intersection points

o Course prerequisites

o Network and shortest routes

o Social networks

o Electric circuits, projects planning and many more...

Graph Classifications

 There are several common kinds of graphs

o Weighted or unweighted

o Directed or undirected

o Cyclic or acyclic

o Multigraphs

Kinds of Graphs: Weighted and Unweighted

Graphs can be classified by whether or not their edges have weights

- *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:* course prerequisites

Kinds of Graphs: Directed and Undirected

Graphs can be classified by whether or their edges are have direction

- *Undirected Graphs:* each edge can be traversed in either direction
- *Directed Graphs:* each edge can be traversed only in a specified direction

Undirected Graphs

Undirected Graph: no implied direction on edge between nodes

✓ The example below is an undirected graph

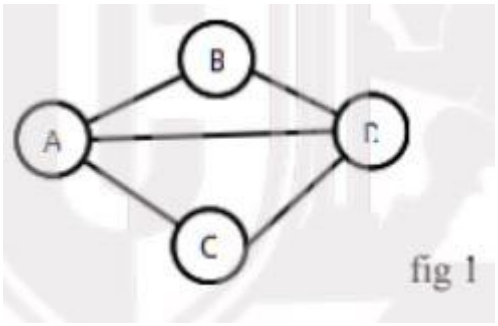


fig 1

✓ In diagrams, edges have no direction (ie there are no arrows)
✓ Can traverse edges in either directions
✓ In an undirected graph, an edge is an unordered pair
✓ Actually, an edge is a set of 2 nodes, but for simplicity we write it with parenthesis

 For example, we write (A, B) instead of {A, B}

 Thus, (A,B) = (B,A), etc

 If (A,B) ∈ E then (B,A) ∈ E

Directed Graphs

A graph whose edges are directed (ie have a direction)

- ✓ Edge drawn as arrow
- ✓ Edge can only be traversed in direction of arrow
- ✓ Example: E = {(A,B), (A,C), (A,D), (B,C), (D,C)}

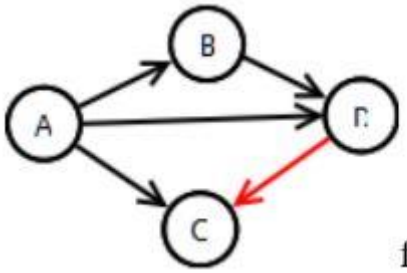


fig 2

- ✓ In a digraph, an edge is an ordered pair
- ✓ Thus: (u,v) and (v,u) are not the same edge
- ✓ In the example, (D,C) ∈ E, (C,D) ∉ E

Degree of a Node

The degree of a node is the number of edges incident on it. In the example above: (fig 1)

- ✓ Degree 2: B and C
- ✓ Degree 3: A and D
- ✓ A and D have odd degree, and B and C have even degree

Can also define in-degree and out-degree

o In-degree(incoming): Number of edges pointing to a node

o Out-degree(outgoing): Number of edges pointing from a node

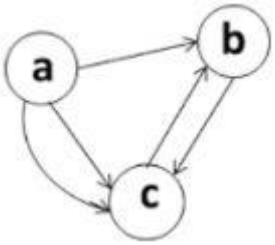
Terminology Involving Paths

- ✓ ***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 o 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

Cyclic and Acyclic Graphs

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

Multigraph: A graph with self loops and parallel edges is called a multigraph.



Connected and Unconnected Graphs and Connected Components

- An ***undirected*** graph is connected if every pair of vertices has a path between it o Otherwise it is unconnected
- A ***directed graph*** is strongly connected if every pair of vertices has a path between them, in both directions

Data Structures for Representing Graphs

Two common data structures for representing graphs:

- ✓ Adjacency lists
- ✓ Adjacency matrix

Adjacency List Representation

An adjacency list is a way in which graphs can be represented in the computer's memory. This structure consists of a list of all nodes in G. Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it. The key advantages of using an adjacency list are:

- It is easy to follow and clearly shows the adjacent nodes of a particular node.

- It is often used for storing graphs that have a small-to-moderate number of edges.

That is, an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.

- Adding new nodes in G is easy and straightforward when G is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered. Each node has a list of adjacent nodes

Example (undirected graph): (fig 1)

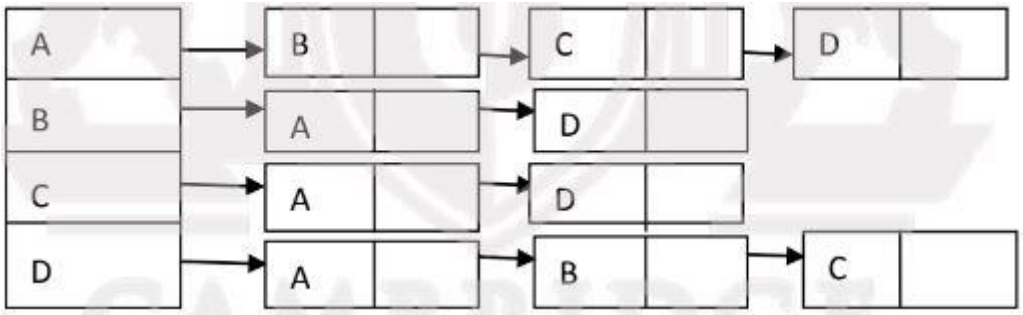


Fig (1) adjacency lsit for the graph of fig3

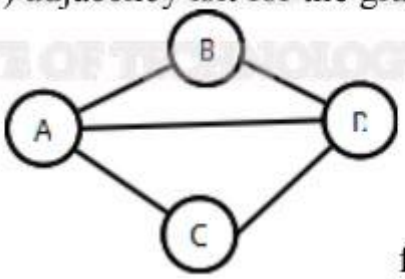


fig 3

- Example (directed graph):
- A: B, C, D
- B: D
- C: Nil
- D: C

Adjacency Matrix Representation

An adjacency matrix is used to represent which nodes are adjacent to one another. By definition, two nodes are said to be adjacent if there is an edge connecting

them. In a directed graph G, if node v is adjacent to node u, then there is definitely an edge from u to v. That is, if v is adjacent to u, we can get from u to v by traversing one edge. For any graph G having n nodes, the adjacency matrix will have the dimension of n * n. In an adjacency matrix, the rows and columns are labelled by graph vertices. An entry aij in the adjacency matrix will contain 1, if vertices vi and vj are adjacent to each other. However, if the nodes are not adjacent, aij will be set to zero. It. Since an adjacency matrix contains only 0s and 1s, it is called a bit matrix or a Boolean matrix. The entries in the matrix depend on the ordering of the nodes in G. Therefore, a change in the order of nodes will result in a different adjacency matrix.

Aij = 1 if there is an edge from Vi to Vj

0 otherwise

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 ∞ 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):

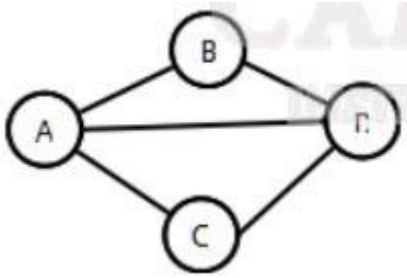
A B C D

A 0 1 1 1

B 1 0 ∞ 1

C 1 ∞ 0 1

D 1 1 1 0



 Example directed graph (assume self-edges allowed):

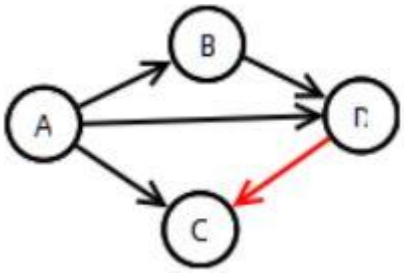
A B C D

A ∞ 1 1 1

B ∞ ∞ ∞ 1

C ∞ ∞ ∞ ∞

D ∞ ∞ 1 ∞



Disadvantage:

Adjacency matrix representation is easy to represent and feasible as long as the graph is small and connected. For a large graph ,whose matrix is sparse, adjacency matrix representation wastes a lot of memory. Hence list representation is preferred over matrix representation.

Graph traversal algorithms

Traversing a graph, is the method of examining the nodes and edges of the graph. There are two standard methods of graph traversal. These two methods are:

- ✓ Breadth-first search
- ✓ Depth-first search

While breadth-first search uses a queue as an auxiliary data structure to store nodes for further processing, the depth-first search scheme uses a stack.

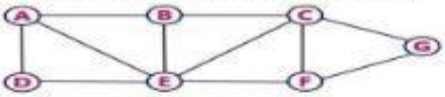
Breadth-first search algorithm

- ✓ Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes.
- ✓ Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes, and so on, until it finds the goal.
- ✓ That is, we start examining the node A and then all the neighbours of A are examined. In the next step, we examine the neighbours of neighbours of A, so on and so forth.
- ✓ This means that we need to track the neighbours of the node and guarantee that every node in the graph is processed and no node is processed more than once.
- ✓ This is accomplished by using a queue that will hold the nodes that are waiting for further processing.

Algorithm for BFS traversal

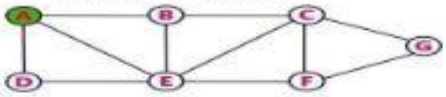
- ✓ Step 1: Define a Queue of size total number of vertices in the graph.
- ✓ Step 2: Select any vertex as starting point for traversal. Visit that vertex and insert it
- ✓ into the Queue.
- ✓ Step 3: Visit all the adjacent vertices of the verex which is at front of the Queue
- ✓ which is not visited 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 the enqueue or dequeue order gives the BFS traversal order.

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.

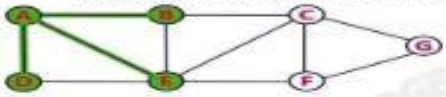


Queue

A						

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..

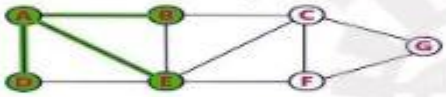


Queue

	D	E	B			

Step 3:

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

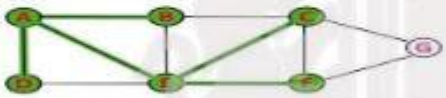


Queue

		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.

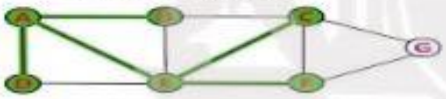


Queue

			B	C	F	

Step 5:

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

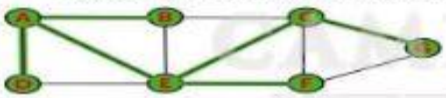


Queue

				C	F	

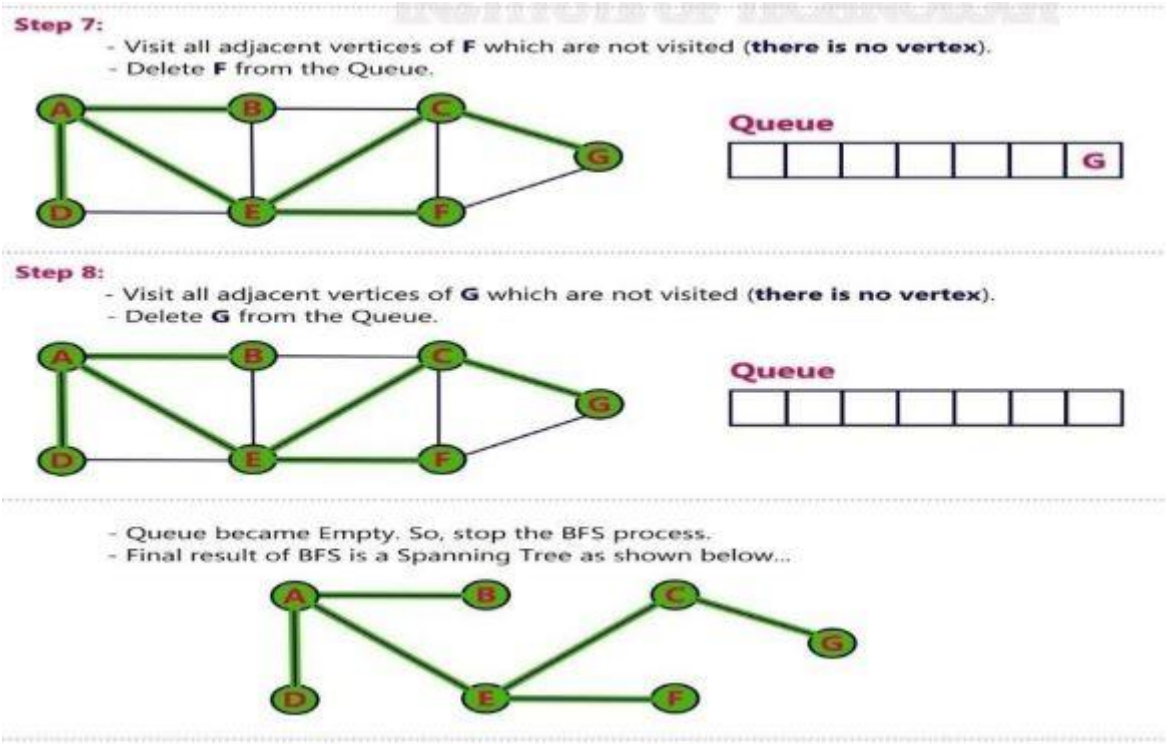
Step 6:

- 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



Depth-first Search Algorithm

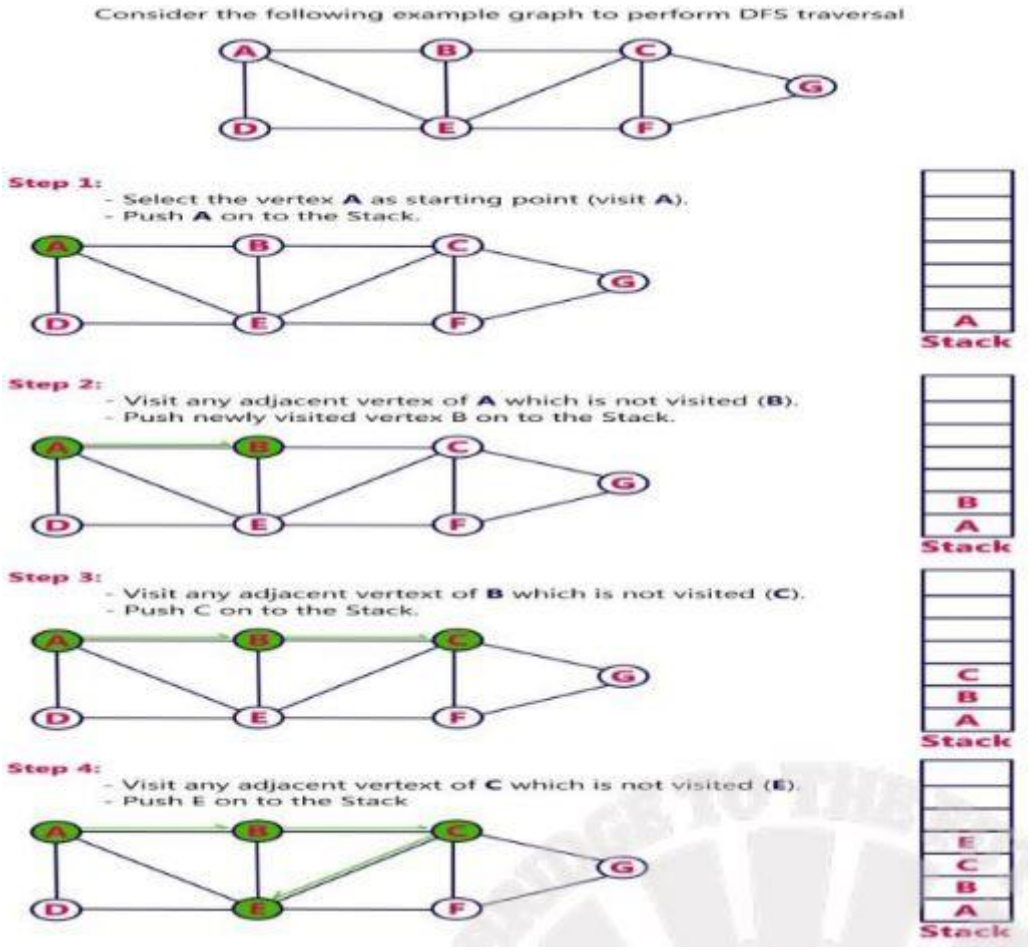
- Depth-first search begins at a starting node A which becomes the current node.
- Then, it examines each node N along a path P which begins at A. That is, we process a neighbour of A, then a neighbour of neighbour of A, and so on.
- During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.
- The algorithm proceeds like this until we reach a dead-end (end of path P). On reaching the deadend, we backtrack to find another path P.
- The algorithm terminates when backtracking leads back to the starting node A.

In this algorithm, edges that lead to a new vertex are called discovery edges and edges that lead to an already visited vertex are called back edges. Observe that this algorithm is similar to the in-order traversal of a binary tree. Its

implementation is similar to that of the breadth- first search algorithm but here we use a stack instead of a queue.

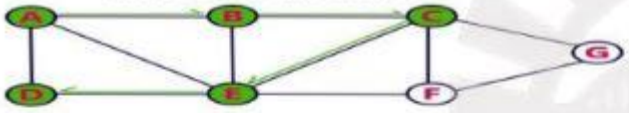
We use the following steps to implement DFS traversal...

- ✓ 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



Step 5:

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



Step 6:

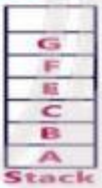
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

Step 7:

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

Step 8:

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



Step 9:

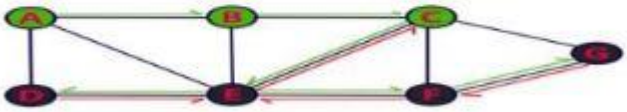
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

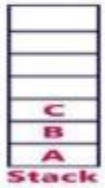
Step 10:

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

Step 11:

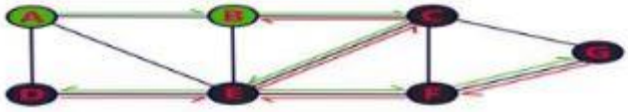
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

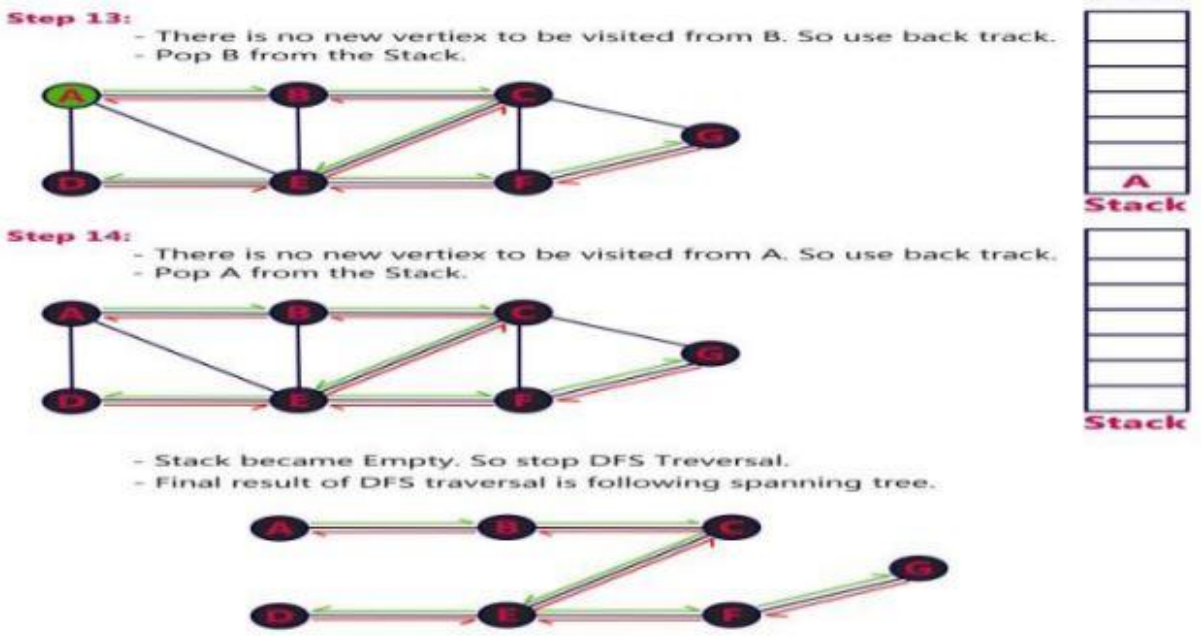




Step 12:

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



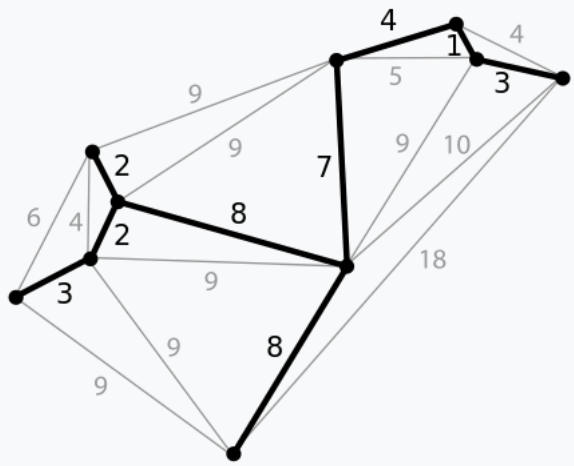


Applications OF graphs

Graphs are constructed for various types of applications such as:

- ✓ In circuit networks where points of connection are drawn as vertices and component wires become the edges of the graph.
- ✓ In transport networks where stations are drawn as vertices and routes become the edges of the graph.
- ✓ In maps that draw cities/states/regions as vertices and adjacency relations as edges.
- ✓ In program flow analysis where procedures or modules are treated as vertices and calls to these procedures are drawn as edges of the graph.
- ✓ Once we have a graph of a particular concept, they can be easily used for finding shortest paths, project planning, etc.
- ✓ In flowcharts or control-flow graphs, the statements and conditions in a program are represented as nodes and the flow of control is represented by the edges.
- ✓ In state transition diagrams, the nodes are used to represent states and the edges represent legal moves from one state to the other.
- ✓ Graphs are also used to draw activity network diagrams. These diagrams are extensively used as a project management tool to represent the interdependent relationships between groups, steps, and tasks that have a significant impact on the project.

Minimum Spanning Trees



A planar graph and its minimum spanning tree. Each edge is labeled with its weight, which here is roughly proportional to its length.

- ✓ A **minimum spanning tree** (**MST**) or **minimum weight spanning tree** is a subset of the edges of a connected, edge-weighted (un)directed graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.
- ✓ That is, it is a spanning tree whose sum of edge weights is as small as possible. More generally, any edge-weighted undirected graph (not necessarily connected) has a **minimum spanning forest**, which is a union of the minimum spanning trees for its connected components.
- ✓ There are quite a few use cases for minimum spanning trees. One example would be a telecommunications company trying to lay cable in a new neighborhood.
- ✓ \If it is constrained to bury the cable only along certain paths (e.g. roads), then there would be a graph containing the points (e.g. houses) connected by those paths.
- ✓ Some of the paths might be more expensive, because they are longer, or require the cable to be buried deeper; these paths would be represented by edges with larger weights.

- ✓ Currency is an acceptable unit for edge weight – there is no requirement for edge lengths to obey normal rules of geometry such as the triangle inequality.
- ✓ A *spanning tree* for that graph would be a subset of those paths that has no cycles but still connects every house; there might be several spanning trees possible.
- ✓ A *minimum spanning tree* would be one with the lowest total cost, representing the least expensive path for laying the cable.

Kruskal's Minimum Spanning Tree Algorithm

What is Minimum Spanning Tree?

- ✓ Given a connected and undirected graph, a *spanning tree* of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees.
- ✓ A *minimum spanning tree (MST)* or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

How many edges does a minimum spanning tree has?

- ✓ A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

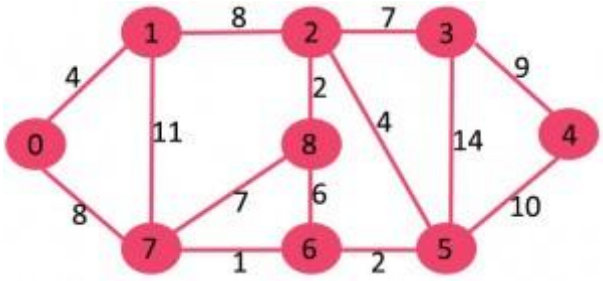
Below are the steps for finding MST using Kruskal's algorithm

1. *Sort all the edges in non-decreasing order of their weight.*

2. *Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.*

3. *Repeat step#2 until there are (V-1) edges in the spanning tree.*

The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example: Consider the below input graph.



The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

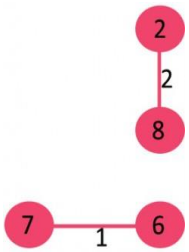
After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

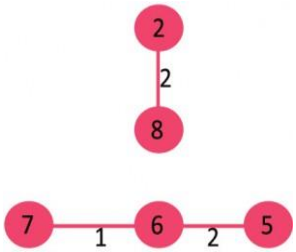
Now pick all edges one by one from sorted list of edges

1. *Pick edge 7-6:* No cycle is formed, include it.

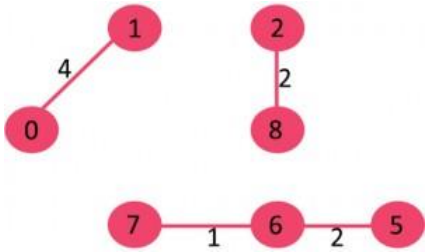
2. *Pick edge 8-2:* No cycle is formed, include it.



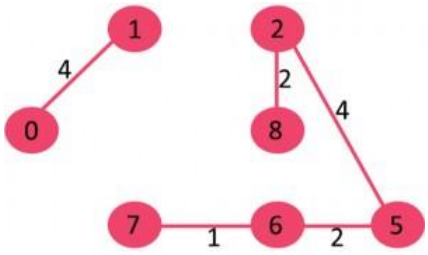
3. *Pick edge 6-5:* No cycle is formed, include it.



4. *Pick edge 0-1:* No cycle is formed, include it.

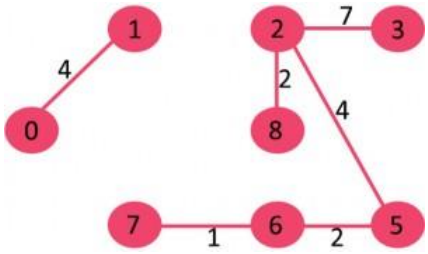


5. *Pick edge 2-5:* No cycle is formed, include it.



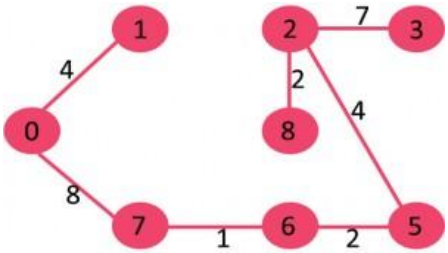
6. *Pick edge 8-6:* Since including this edge results in cycle, discard it.

7. *Pick edge 2-3:* No cycle is formed, include it.



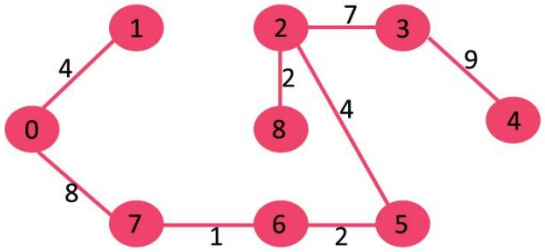
8. *Pick edge 7-8:* Since including this edge results in cycle, discard it.

9. *Pick edge 0-7:* No cycle is formed, include it.



10. *Pick edge 1-2:* Since including this edge results in cycle, discard it.

11. *Pick edge 3-4:* No cycle is formed, include it.



Since the number of edges included equals (V – 1), the algorithm stops here.

Prim's Minimum Spanning Tree (MST)

- Like Kruskal's algorithm, Prim's algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included.
- At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.
- A group of edges that connects two set of vertices in a graph is called cut in graph theory. *So, at every step of Prim's algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the verices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).*

How does Prim's Algorithm Work?

- The idea behind Prim's algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a *Spanning* Tree.
- And they must be connected with the minimum weight edge to make it a *Minimum* Spanning Tree.

Algorithm

1) Create a set *mstSet* that keeps track of vertices already included in MST.
2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
3) While mstSet doesn't include all vertices

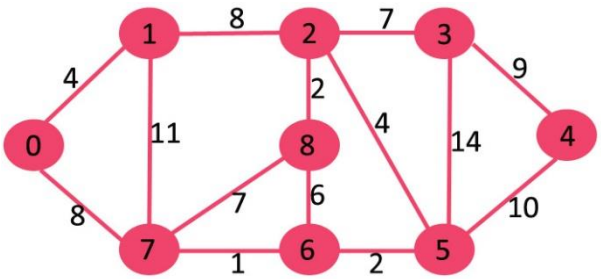
a) Pick a vertex u which is not there in mstSet and has minimum key value.

b) Include u to mstSet.

c) Update key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v

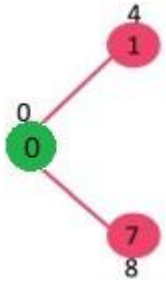
The idea of using key values is to pick the minimum weight edge from cut. The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

Let us understand with the following example:

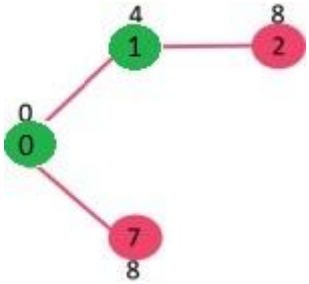


The set *mstSet* is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with

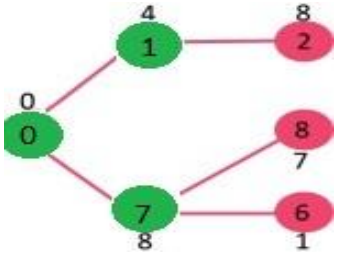
minimum key value. The vertex 0 is picked, include it in *mstSet*. So *mstSet* becomes {0}. After including to *mstSet*, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.



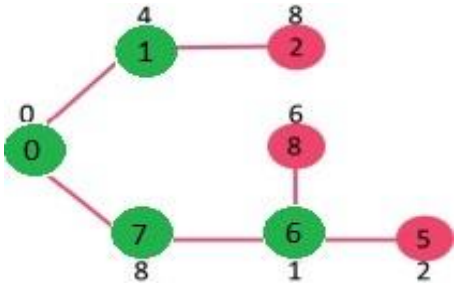
Pick the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.



Pick the vertex with minimum key value and not already included in MST (not in mstSET). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So mstSet now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (1 and 7 respectively).



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



We repeat the above steps until *mstSet* includes all vertices of given graph. Finally, we get the following graph.