# 1. Introduction

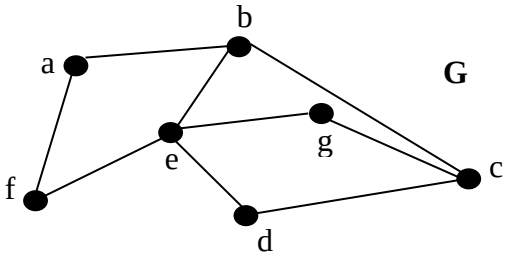
A **graph** consists of set of **nodes** (or **vertices**) and a set of **arcs** (or **edges**) that connect these vertices. A *graph G = (V, E)* consists of *V*, a nonempty set of *vertices* and *E*, a set of *edges*. Each edge has either one or two vertices associated with it, called its *endpoints*. An edge is said to *connect* its endpoints.

# 2. Graph Types

## 2.1. Simple Graph

A graph in which each edge connects two different vertices and where no two edges connect the same pair of vertices is called a simple graph.

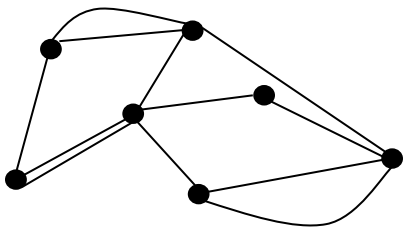
A simple graph G = (V, E) consists of V, a nonempty set of vertices, and E, a set of unordered pairs of distinct elements of V called edges. This kind of graph has undirected edges, no loops and no multiple edges. The figure below is an example of simple graph.




In the above graph G = (V, E), the set of vertices, V (G) or V = {a, b, c, d, e, f, g} and the set of edges E (G) or E = {{a, b}, {a, f}, {b, c}, {b, e}, {c, d}, {c, g}, {d, e}, {e, g},{e, f}}.

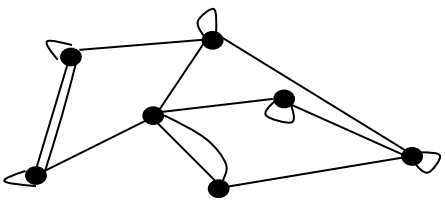
## 2.2. Multigraph

Graphs that may have **multiple edges** connecting the same vertices are called **multigraphs**. When there are *m* different edges associated to the same unordered pair of vertices {*u*, *v*}, we also say that {*u*, *v*} is an edge of multiplicity *m*. That is, we can think of this set of edges as *m* different copies of an edge {*u*, *v*}. This kind of graph has undirected edges, and no loops. The figure below is an example of a multigraph.



## 2.3. Pseudograph

Some graphs have edges that connect a vertex to itself. Such edges are called **loops**, and sometimes we may even have more than one loop at a vertex. Graphs that may include loops, and possibly multiple edges connecting the same pair of vertices or a vertex to itself, are sometimes called **pseudographs**.



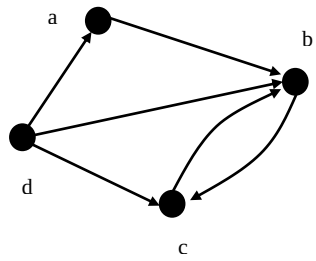
## 2.4. Directed Graph

The graphs have introduced so far are **undirected graphs**. Their edges are also said to be **undirected**. However, to construct a graph model, we may find it necessary to assign directions to the edges of a graph.

A *directed graph* (or *digraph*) *(V, E)* consists of a nonempty set of vertices *V* and a set of *directed edges* (or *arcs*) *E*. Each directed edge is associated with an ordered pair of vertices. The directed edge associated with the ordered pair (*u, v*) is said to *start* at *u* and *end* at *v*.

A directed graph may contain loops and it may contain multiple directed edges that start and end at the same vertices. A directed graph may also contain directed edges that connect vertices *u* and *v* in both directions; that is, when a digraph contains an edge from *u* to *v*, it may also contain one or more edges from *v* to *u*.

When a directed graph has no loops and has no multiple directed edges, it is called a **simple directed graph**. The figure below is an example of simple directed graph.




In the above graph G = (V, E), the set of vertices, V (G) or V = {a, b, c, d} and the set of edges E (G) or E = {(a, b), (b, c), (c, b), (d, a), (d, b), (d, c)}.

Directed graphs that may have **multiple directed edges** and **loops** are called **directed multigraphs**. When there are *m* directed edges, each associated to an ordered pair of vertices *(u, v),* we say that *(u, v)* is an edge of **multiplicity** *m*.

## 2.5. Mixed Graph

For some models we may need a graph where some edges are undirected, while others are directed. A graph with both directed and undirected edges is called a **mixed graph**.

| Type | Edge | Multiple Edges | Loops |
|---|---|---|---|
| Simple graph | Undirected | No | No |
| Multigraph | Undirected | Yes | No |
| Pseudograph | Undirected | Yes | Yes |
| Simple directed graph | Directed | No | No |
| Directed multigraph | Directed | Yes | Yes |
| Mixed graph | Directed and Undirected | Yes | Yes |

# 3. Graph Terminologies

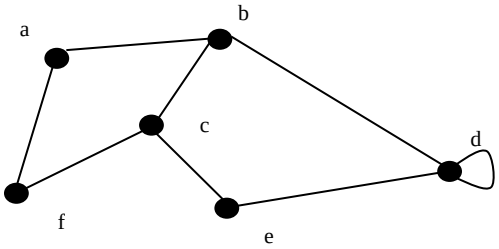
Two vertices u, v in an undirected graph G are called **adjacent** or **neighbors** in G if {u, v} is an edge. The edge **e** is called **incident with** the vertices u and v if e = {u, v}. This edge is also said to **connect** u and v, where u and v are end points of the edge.

The set of all neighbors of a vertex *v* of *G = (V, E)*, denoted by *N(v)*, is called the *neighbourhood* of *v*. If *A* is a subset of *V*, we denote by *N(A)* the set of all vertices in *G* that are adjacent to at least one vertex in *A*. So, $N(A) = \cup_{v \in A} N(v)$.

The **degree of a vertex** in an undirected graph is the number of edges incident with it, except a loop at a vertex. Loop in a vertex counts twice to the degree. Degree of a

vertex v is denoted by deg (v). A vertex of degree zero is called **isolated** vertex and a vertex with degree one is called **pendant** vertex. A pendant vertex is adjacent to exactly one other vertex.

**Example:** What are the degree of the vertices in the following graph? What is N(a) and N(a, b)?

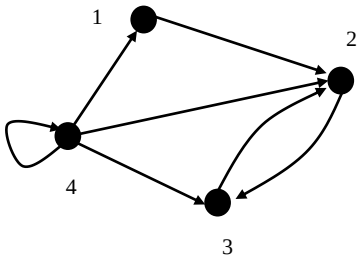



**Solution:** deg(a) = deg(f) = deg(e) = 2 ; deg(b) = deg(c) = 3; deg(d) = 4; N(a) = {b, f}; N(a, b) = {b, f, a, c, d}.

Let (u, v) be an edge representing edge of a directed graph G, u is called **adjacent to** v and v is called **adjacent from** u. The vertex u is called **initial vertex** and the vertex v is called **terminal or end vertex**. Loop has same initial and terminal vertex.

In directed graph the **in-degree** of a vertex v, denoted by $\deg^-(v)$, is the number of edges that have v as their terminal vertex. The **out-degree** of a vertex v, denoted by $\deg^+(v)$, is the number of edges that have v as their initial vertex. Loop at a vertex adds up both in-degree and out-degree to one more than calculated in-degree and out-degree.

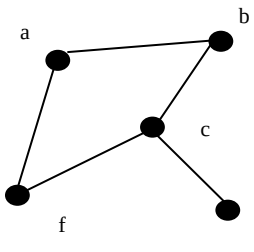
**Example:** Find the in-degree and out-degree of each vertex in the following graph.



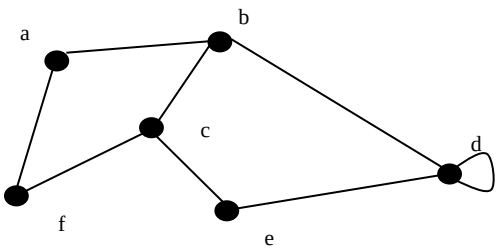

**Solution:** In-degrees of a graph are $\deg^-(1) = \deg^-(4) = 1$; $\deg^-(2) = 3$; $\deg^-(3) = 2$ and the out-degrees of a graph are $\deg^+(1) = \deg^+(2) = \deg^+(3) = 1$; $\deg^+(4) = 4$.

A **subgraph** of a graph G = (V, E) is a graph H = (W, F) where $W \subseteq V$ and $F \subseteq E$.

**Example:**




is subgraph of

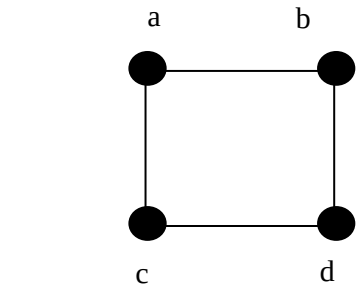



# 4. Graph Representation

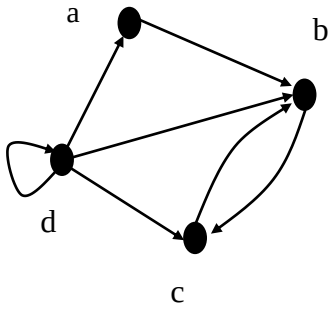
There are many useful ways to represent graphs. As explained earlier, one way to represent a graph without multiple edges is to list all the edges of this graph. Other techniques are **adjacency list**, **adjacency matrix**, and **incidence matrix**.

## 4.1. Adjacency List

This type of representation is suitable for the undirected graphs without multiple edges, and directed graphs. This representation looks as in the tables below.




| Edge List for Simple Graph | |
|---|---|
| Vertex | Adjacent Vertices |
| a | b, c |
| b | a, d |
| c | a, d |
| d | b, c |




| Edge List for Directed Graph | |
|---|---|
| Initial Vertex | End Vertices |
| a | b |
| b | c |
| c | b |
| d | a, b, c, d |

Carrying out graph algorithms using the representation of graphs by lists of edges, or by adjacency lists, can be cumbersome if there are many edges in the graph. To simplify computation, graphs can be represented using matrices. Two types of matrices commonly used to represent graphs will be presented here. One is based on the adjacency of vertices, and the other is based on incidence of vertices and edges.

## 4.2. Adjacency Matrix

Given a simple graph G = (V, E) with |V| = n. Assume that the vertices of the graph are listed in some arbitrary order like $v_1, v_2, \ldots, v_n$. The adjacency matrix A of G ($A_G$), with respect to the order of the vertices is n-by-n zero-one matrix ($A = [a_{ij}]$) with the condition,

$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ is an edge of G,} \\ 0 & \text{otherwise.} \end{cases}$$

Since there are n vertices and we may order vertices in any order there are *n!* possible order of the vertices. The adjacency matrix depends on the order of the vertices, hence there are n! possible adjacency matrices for a graph with n vertices.

Adjacency matrix for undirected graph is *symmetric*, in case of the pseudograph or multigraph the representation is similar but the matrix here is not zero-one matrix rather the $(i, j)^{th}$ entry of the matrix contains the number of edges appearing between that pair of vertices.

In case of the directed graph we can extend the same concept as in undirected graph as dictated by the relation

$$a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ is an edge of G,} \\ 0 & \text{otherwise.} \end{cases}$$

The main difference is that the matrix may not be *symmetric*. Adjacency matrices can also be used to represent directed multigraphs. Again, such matrices are not zero–one matrices when there are multiple edges in the same direction connecting two vertices. In the adjacency matrix for a directed multigraph, *aij* equals the number of edges that are associated to *(vi , vj )*.

If the number of edges is few then the adjacency matrix becomes *sparse*. Sometimes it will be beneficial to represent graph with adjacency list in such a condition.

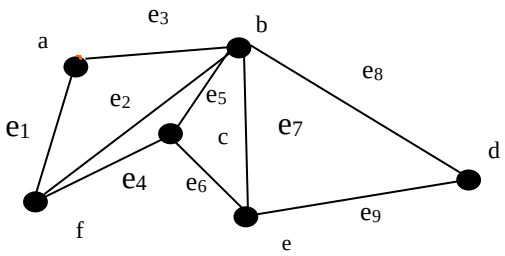
## 4.3. Incidence Matrix

This is another way of representing graph. Given an undirected graph G = (V, E). Assume that the vertices of the graph are $v_1, v_2, \ldots, v_n$ and the edges of the graph are $e_1, e_2, \ldots, e_m$. The incidence matrix of a graph with respect to the above ordering of V and E is n-by-m matrix M = $[m_{ij}]$, where

$$m_{ij} = \begin{cases} 1 & \text{when edge } e_j \text{ is incident with } v_i \\ 0 & \text{otherwise.} \end{cases}$$

When the graph is not simple then also the graph can be represented by using incidence matrix where multiple edges corresponds to two different columns with exactly same entries. Loops are represented with column with only one entry.

**Example 1:** Represent the following graph using adjacency matrix and incidence matrix.




**Solution:** Let the order of the vertices be a, b, c, d, e, f and edges order be $e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9$.

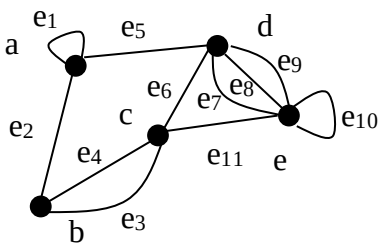
| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| a | 0 | 1 | 0 | 0 | 0 | 1 |
| b | 1 | 0 | 1 | 1 | 1 | 1 |
| c | 0 | 1 | 0 | 0 | 1 | 1 |
| d | 0 | 1 | 0 | 0 | 1 | 0 |
| e | 0 | 1 | 1 | 1 | 0 | 0 |
| f | 1 | 1 | 1 | 0 | 0 | 0 |

Adjacency Matrix

| | $e_1$ | $e_2$ | $e_3$ | $e_4$ | $e_5$ | $e_6$ | $e_7$ | $e_8$ | $e_9$ |
|---|---|---|---|---|---|---|---|---|---|
| a | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| b | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| c | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| d | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| e | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| f | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

Incidence Matrix

**Example 2:** Represent the following graph using adjacency matrix and incidence matrix.



**Solution:** Let the order of the vertices be a, b, c, d, e and edges order be $e_1$, $e_2$, $e_3$, $e_4$, $e_5$, $e_6$, $e_7$, $e_8$, $e_9$, $e_{10}$, $e_{11}$.

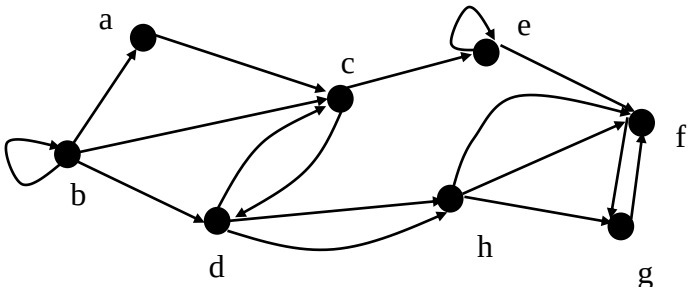
| | a | b | c | d | e |
|---|---|---|---|---|---|
| a | 1 | 1 | 0 | 1 | 0 |
| b | 1 | 0 | 2 | 0 | 0 |
| c | 0 | 2 | 0 | 1 | 1 |
| d | 1 | 0 | 1 | 0 | 3 |
| e | 0 | 0 | 1 | 3 | 1 |

Adjacency Matrix

| | $e_1$ | $e_2$ | $e_3$ | $e_4$ | $e_5$ | $e_6$ | $e_7$ | $e_8$ | $e_9$ | $e_{10}$ | $e_{11}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *a* | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| *b* | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| *c* | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| *d* | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| *e* | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |

Incidence Matrix

**Example 3:** Represent the following directed graph using adjacency matrix.




**Solution:** Let the order of the vertices be a, b, c, d, e, f, g and h. The adjacency matrix is

| | a | b | c | d | e | f | g | h |
|---|---|---|---|---|---|---|---|---|
| a | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| b | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| c | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| d | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 2 |
| e | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| f | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| g | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| h | 0 | 0 | 0 | 0 | 0 | 2 | 1 | 0 |

## 5. Shortest Path Algorithm

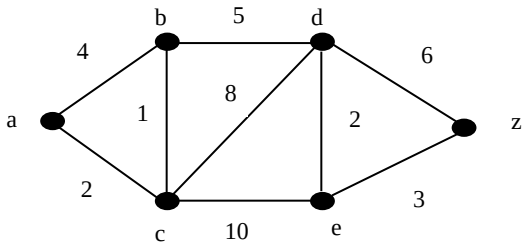
Many problems can be modeled using graphs with weights assigned to their edges. Graphs that have a number assigned to each edge are called **weighted graphs**. Several types of problems involving weighted graphs arise frequently. Determining a path of least length (**i.e. shortest path**) between two vertices in a graph is one such problem.

There are several different shortest path algorithms to find shortest path between two vertices in a graph. Here, we will present a **greedy algorithm** discovered by the Dutch mathematician Edsger Dijkstra in 1959. The version we will describe solves this problem in undirected weighted graphs where all the weights are positive. It is easy to adapt it to solve shortest-path problems in directed graphs.

**Dijkstra's Algorithm**

**procedure** *Dijkstra*(*G*: weighted connected simple graph, with all weights positive)
{*G* has vertices $a = v_0, v_1, \ldots, v_n = z$ and lengths $w(v_i, v_j)$ where $w(v_i, v_j) = \infty$ if $\{v_i, v_j\}$ is not an edge in *G*}
**for** *i* := 1 **to** *n*
    $L(v_i) := \infty$
$L(a) := 0$
$S := \emptyset$
{the labels are now initialized so that the label of *a* is 0 and all other labels are ∞, and *S* is the empty set}
**while** $z \notin S$
    *u* := a vertex not in *S* with *L(u)* minimal
    $S := S \cup \{u\}$
    **for** all vertices *v* not in *S*
        **if** $L(u) + w(u, v) < L(v)$ **then** $L(v) := L(u) + w(u, v)$
        {this adds a vertex to *S* with minimal label and updates the labels of vertices not in *S*}
**return** *L(z)* {*L(z)* = length of a shortest path from *a* to *z*}

**Example:** Use Dijkstra's algorithm to find the length of shortest path between the vertices ***a*** and ***z*** in the weighted graph given below.




*Solution:*

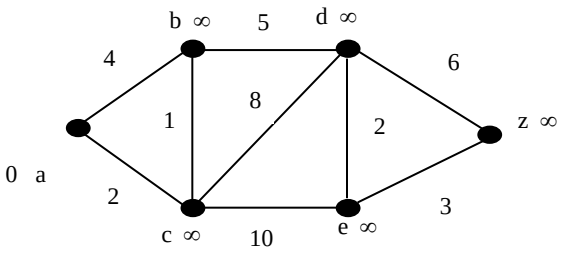



Fig (a)

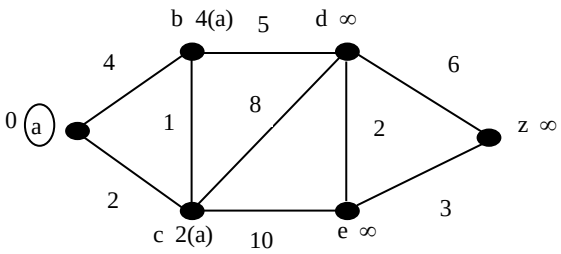



Fig (b)

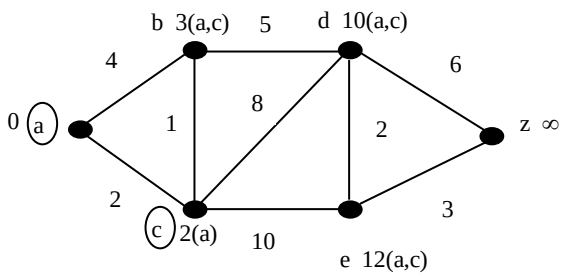



Fig (c)

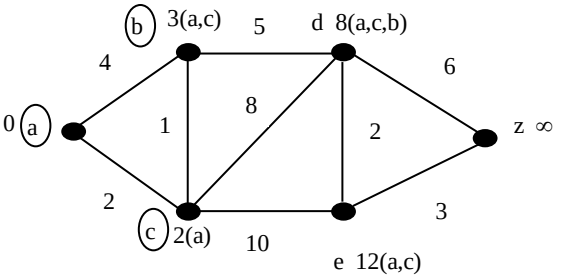



Fig (d)

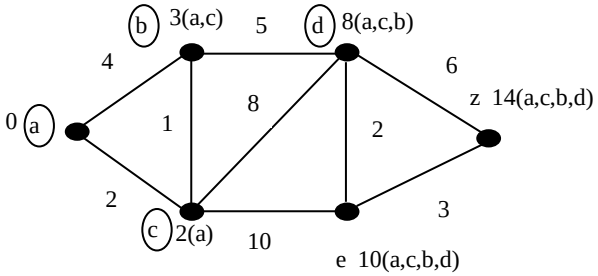
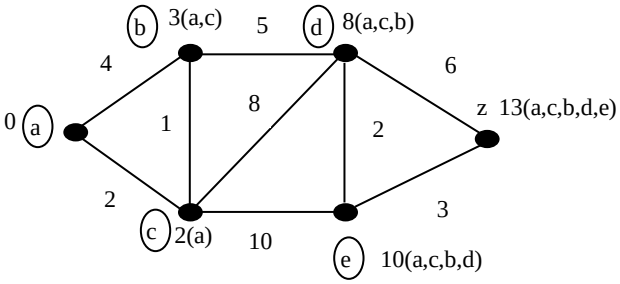



Fig (e)




Fig(f)

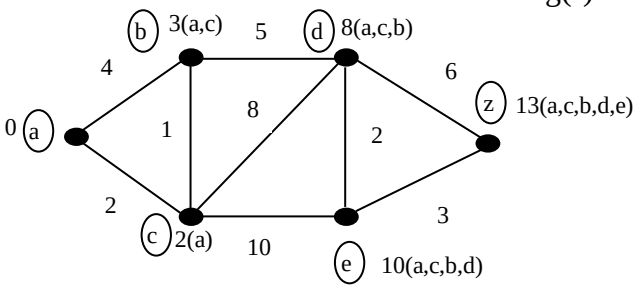
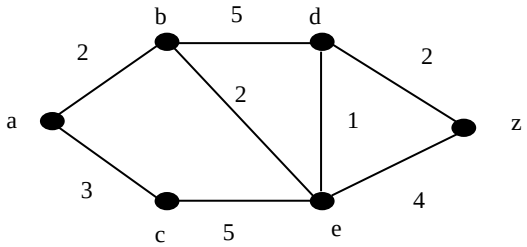



Fig (g)

Hence, the length of the shortest path is 13 and the shortest path is a, c, b, d, e, z.

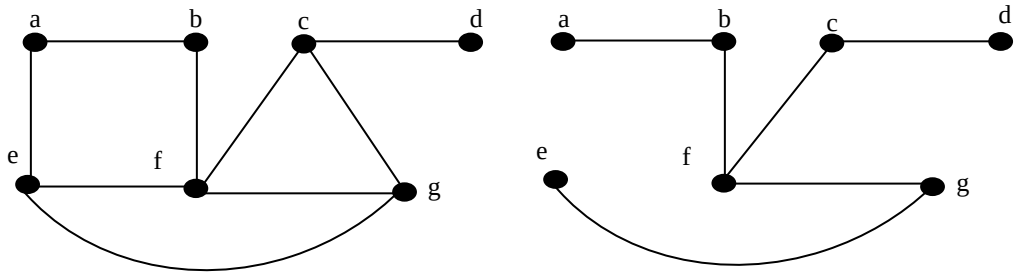
**Exercise:** Find the length of the shortest path between a and z in the given graph.




# 6. Spanning Trees and Minimum Spanning Trees

Let G be a simple connected graph. A spanning tree of G is a subgraph of G that is a tree containing every vertex of G.

**Example:** Find a spanning tree of the simple graph G on the left side below.




*Solution:* The left graph is connected but is not tree because it contains simple circuits. By removing the edges {a, e}, {e, f} and {c, g} from the left graph, we obtain a simple graph with no simple circuits on the right side. This subgraph is a spanning tree.

This tree is not the only spanning tree of the graph. The graph can also have other spanning trees.

A **minimum spanning tree** in a connected weighted graph is a spanning tree that has the smallest possible sum of weights of its edges.

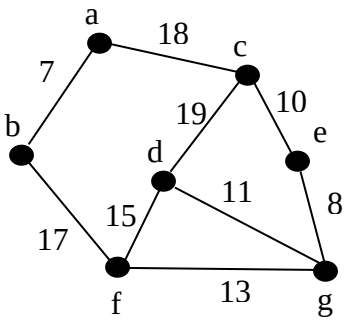
## 6.1. Prim's Algorithm

**Prim's Algorithm** is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized. Prim's algorithm starts with a randomly chosen the single vertex and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

**Algorithm:**

1. Initialize minimum spanning tree with the randomly chosen vertex.
2. Find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.
3. Repeat step 2 until minimum spanning tree is formed.

**Example:** Find the minimum spanning tree of the following graph using Prim's algorithm.




*Solution:* First, we have to choose a vertex randomly from the graph. Let's choose vertex b.

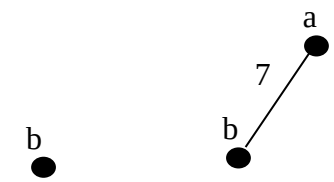



Fig (a)




Fig (b)

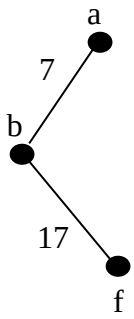



Fig (c)

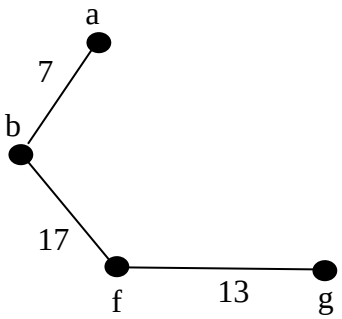



Fig (d)

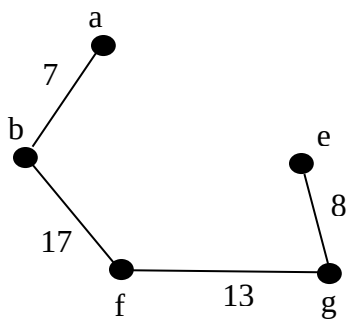



Fig (e)

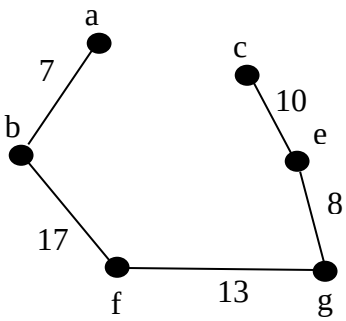



Fig (f)

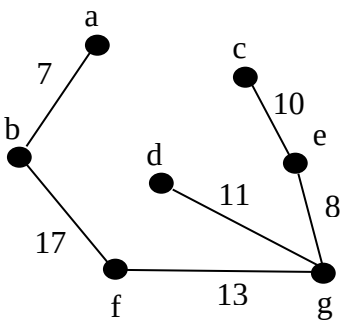



Fig (g)

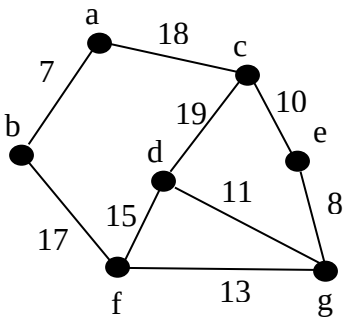
## 6.2. Kruskal's Algorithm

Kruskal's algorithm is also used to construct minimum spanning trees. To carry out Kruskal's algorithm, choose an edge in the graph with minimum weight. Successively add edges with minimum weight that do not form a simple circuit with those edges already chosen. Stop after n-1 edges have been selected.

**Algorithm:**

1. Take the edge with the lowest weight and add it to the spanning tree. If the edge to be added creates a cycle, then reject the edge.
2. Continue to add the edges until we reach all vertices, and a minimum spanning tree is created.

**Example:** Find the minimum spanning tree of the following graph using Kruskal's algorithm.




*Solution:*

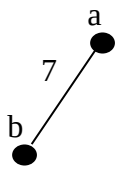



Fig (a)

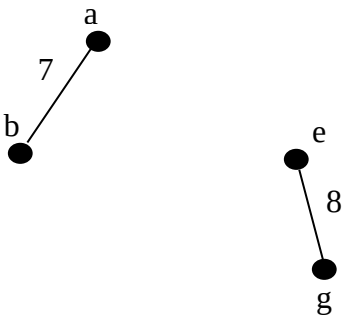



Fig (b)

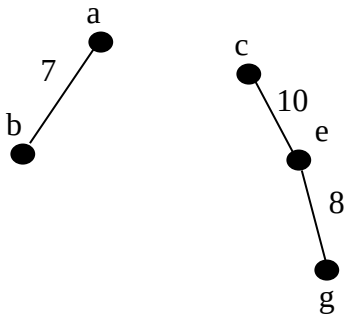



Fig (c)

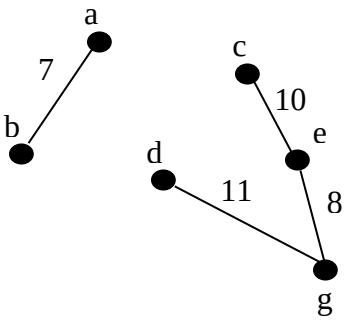



Fig (d)

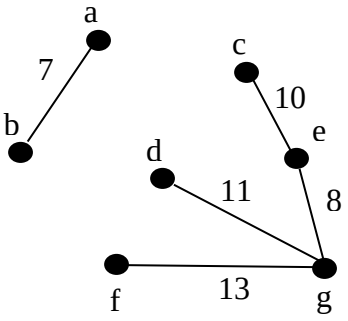



Fig (e)

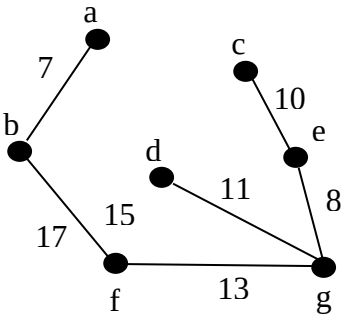



Fig (f)

# 7. Graph Traversal

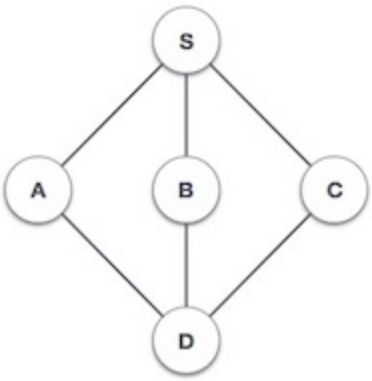
Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices visited in the search process. There are two graph traversal techniques: DFS (Depth First Search) and BFS (Breadth First Search).

## 7.1. Depth Frist Search (DFS)

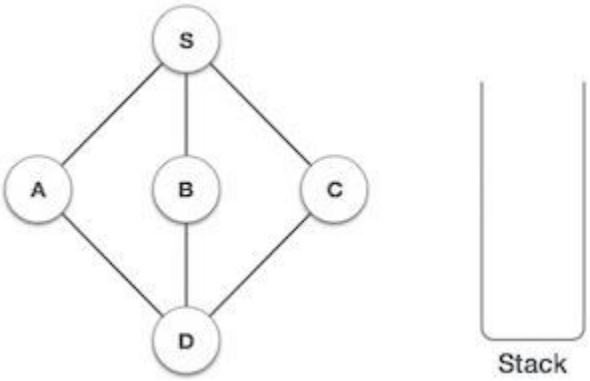
Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration. We use the following steps to implement DFS traversal.

- 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.

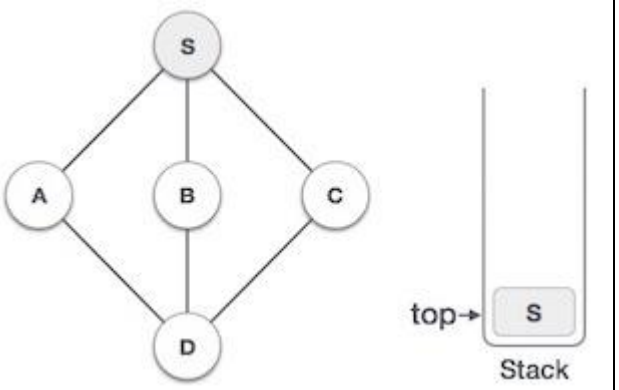
**Example**: Traverse the graph below using depth first search.

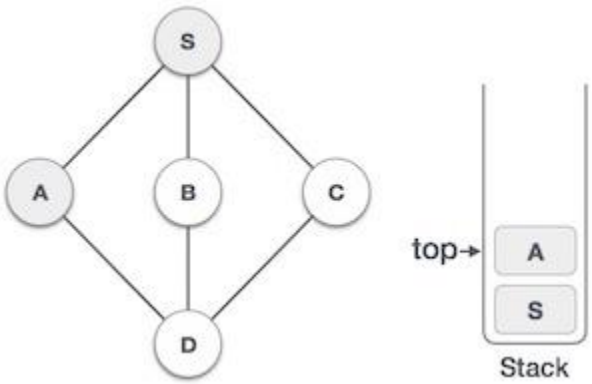
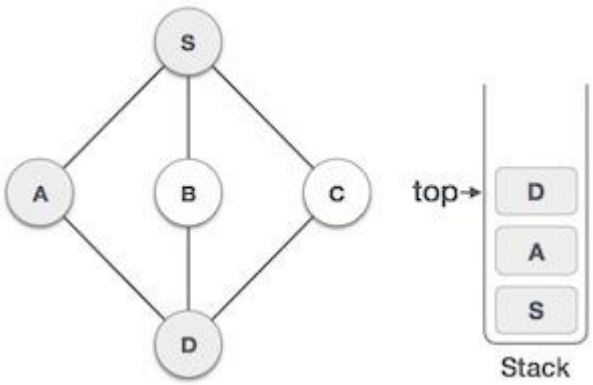
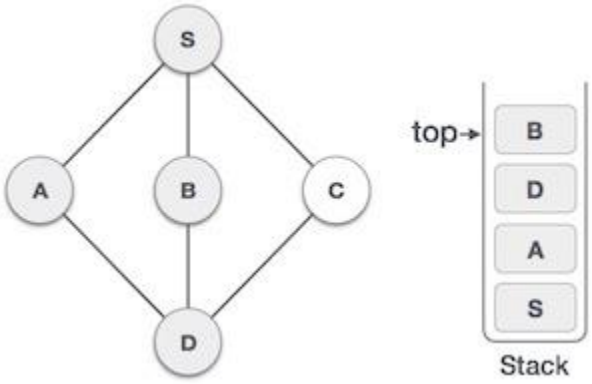
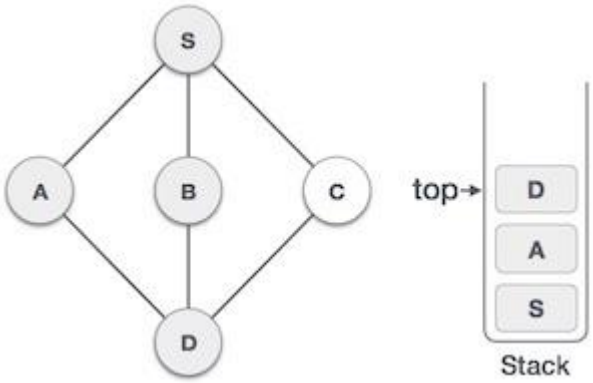
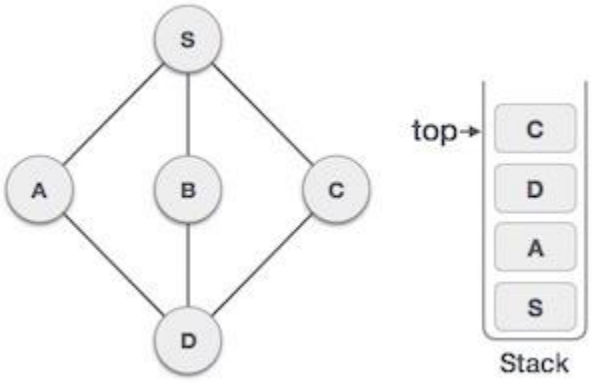



**Step 1:** Initialize stack




**Step 2:** Select a vertex S as the starting point (visit S). Push S on to Stack.



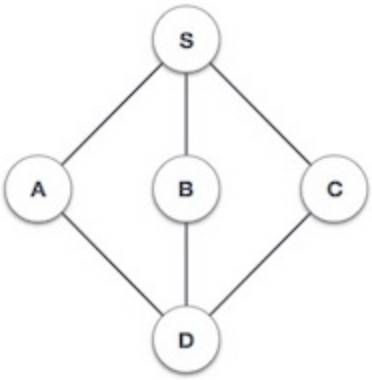
| | |
|---|---|
| **Step 3:** Visit any adjacent vertex of S which is not visited (A). Push newly visited vertex A on to the Stack.<br>  | **Step 4:** Visit any adjacent vertex of A which is not visited (D). Push newly visited vertex D on to the Stack.<br>  |
| **Step 5:** Visit any adjacent vertex of D which is not visited (B). Push newly visited vertex B on to the Stack.<br>  | **Step 6:** There is no new vertex to be visited from B. Backtrack from B and pop B from the Stack.<br>  |
| **Step 7:** Visit any adjacent vertex of D which is not visited (C). Push newly visited vertex C on to the Stack<br>  | As C does not have any unvisited adjacent vertex so we keep popping the stack until we find a vertex that has an unvisited adjacent vertex. In this case, there's none and we keep popping until the stack is empty.<br>Hence, depth first search is **S, A, D B, C**. |

## 7.2. Breadth Frist Search (BFS)

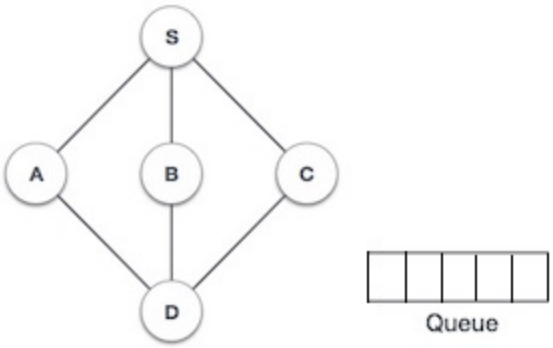
Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration. We use the following steps to implement BFS traversal.

- 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.

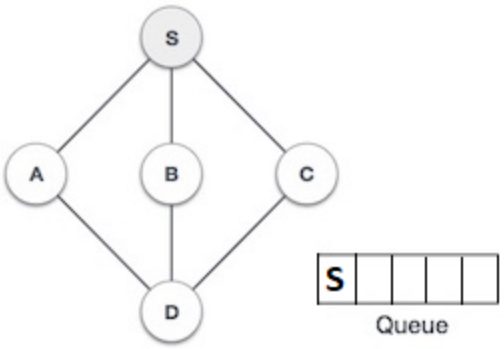
**Exercise**: Traverse the graph below using breadth first search.



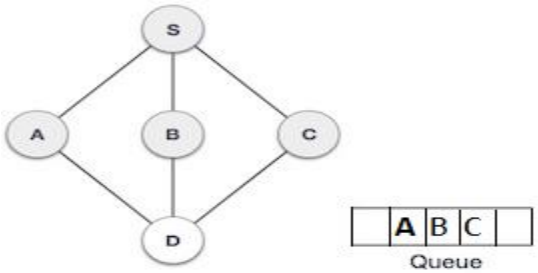

**Step 1:** Initialize queue.



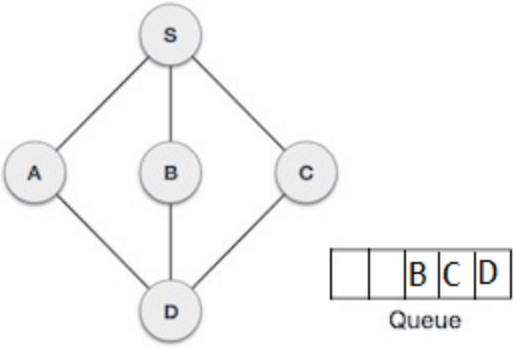

Step 2: Select the vertex S as starting point (visit S). Insert S into the Queue.




**Step 3:** Visit all adjacent vertices of S which are not visited (A, B, C). Insert newly visited vertices into the Queue and delete S from the Queue.




**Step 4:** Visit all adjacent vertices of A which are not visited (D). Insert newly visited vertex into the Queue and delete A from the Queue.



| | |
|---|---|
| At this stage, we are left with no unvisited vertices. But as per the algorithm we keep on removing in order to get all unvisited vertices. When the queue gets emptied, the program is over.<br><br>Hence, depth first search is **S, A, B, C, D**. | |