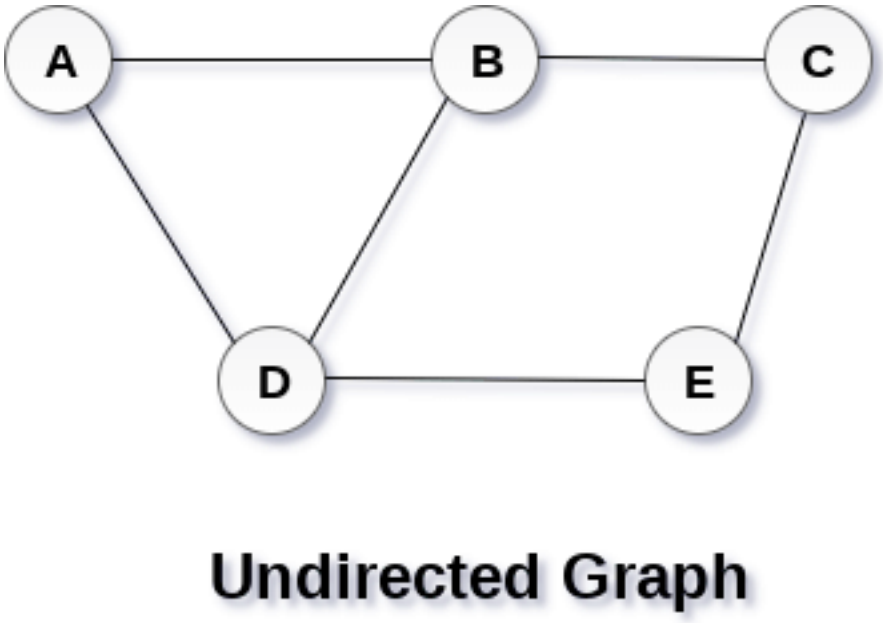
Graph

A graph can be defined as group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

Definition

A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.

A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.



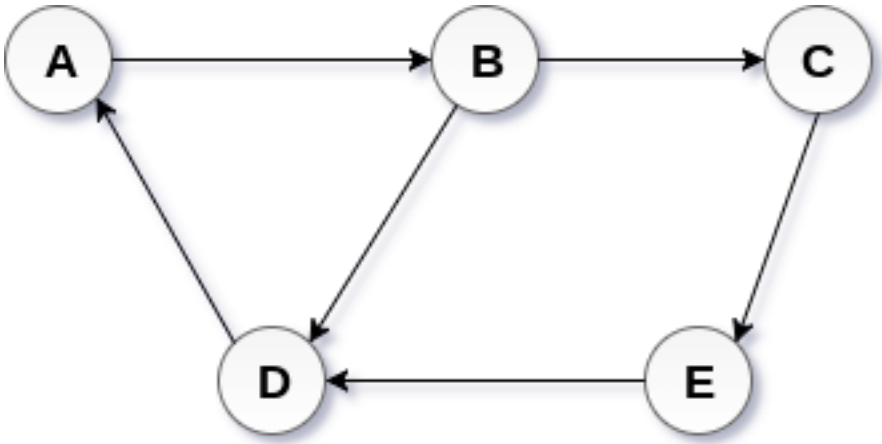
Undirected Graph

Directed and Undirected Graph

A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.

In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node.

A directed graph is shown in the following figure.



Directed Graph

Graph Terminology

Path

A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

Closed Path

A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if $V_0=V_N$.

Simple Path

If all the nodes of the graph are distinct with an exception $V_0=V_N$, then such path P is called as closed simple path.

Cycle

A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.

Connected Graph

A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.

Complete Graph

A complete graph is the one in which every node is connected with all other nodes. A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph.

Weighted Graph

In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.

Digraph

A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.

Loop

An edge that is associated with the similar end points can be called as Loop.

Adjacent Nodes

If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.

Degree of the Node

A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

Graph representation

In this article, we will discuss the ways to represent the graph. By Graph representation, we simply mean the technique to be used to store some graph into the computer's memory.

A graph is a data structure that consist a sets of vertices (called nodes) and edges. There are two ways to store Graphs into the computer's memory:

- **Sequential representation** (or, Adjacency matrix representation)
- **Linked list representation** (or, Adjacency list representation)

In sequential representation, an adjacency matrix is used to store the graph. Whereas in linked list representation, there is a use of an adjacency list to store the graph.

In this tutorial, we will discuss each one of them in detail.

Now, let's start discussing the ways of representing a graph in the data structure.

Sequential representation

In sequential representation, there is a use of an adjacency matrix to represent the mapping between vertices and edges of the graph. We can use an adjacency matrix to represent the undirected graph, directed graph, weighted directed graph, and weighted undirected graph.

If adj[i][j] = w, it means that there is an edge exists from vertex i to vertex j with weight w.

An entry A_{ij} in the adjacency matrix representation of an undirected graph G will be 1 if an edge exists between V_i and V_j. If an Undirected Graph G consists of n vertices, then the adjacency matrix for that graph is n x n, and the matrix A = [aij] can be defined as -

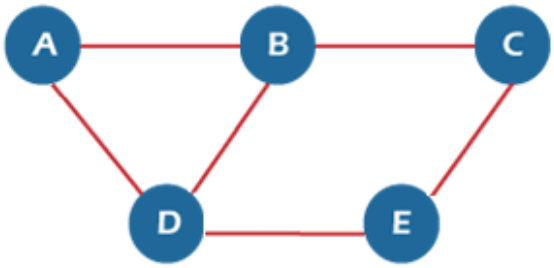
a_{ij} = 1 {if there is a path exists from V_i to V_j}

a_{ij} = 0 {Otherwise}

It means that, in an adjacency matrix, 0 represents that there is no association exists between the nodes, whereas 1 represents the existence of a path between two edges.

If there is no self-loop present in the graph, it means that the diagonal entries of the adjacency matrix will be 0.

Now, let's see the adjacency matrix representation of an undirected graph.



Undirected Graph

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

Adjacency Matrix

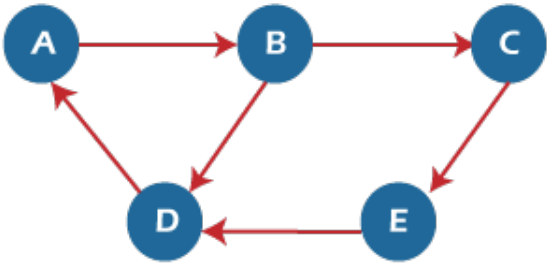
In the above figure, an image shows the mapping among the vertices (A, B, C, D, E), and this mapping is represented by using the adjacency matrix.

There exist different adjacency matrices for the directed and undirected graph. In a directed graph, an entry A_{ij} will be 1 only when there is an edge directed from V_i to V_j.

Adjacency matrix for a directed graph

In a directed graph, edges represent a specific path from one vertex to another vertex. Suppose a path exists from vertex A to another vertex B; it means that node A is the initial node, while node B is the terminal node.

Consider the below-directed graph and try to construct the adjacency matrix of it.



Directed Graph

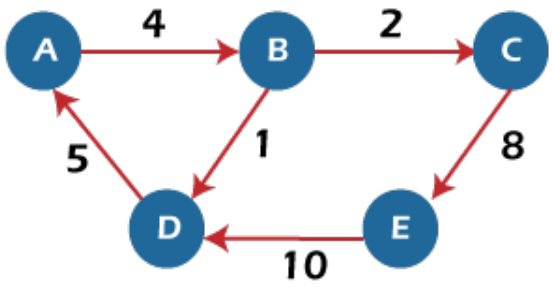
	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

Adjacency Matrix

In the above graph, we can see there is no self-loop, so the diagonal entries of the adjacent matrix are 0.

Adjacency matrix for a weighted directed graph

It is similar to an adjacency matrix representation of a directed graph except that instead of using the '1' for the existence of a path, here we have to use the weight associated with the edge. The weights on the graph edges will be represented as the entries of the adjacency matrix. We can understand it with the help of an example. Consider the below graph and its adjacency matrix representation. In the representation, we can see that the weight associated with the edges is represented as the entries in the adjacency matrix.



weighted Directed Graph

	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

Adjacency Matrix

In the above image, we can see that the adjacency matrix representation of the weighted directed graph is different from other representations. It is because, in this representation, the non-zero values are replaced by the actual weight assigned to the edges.

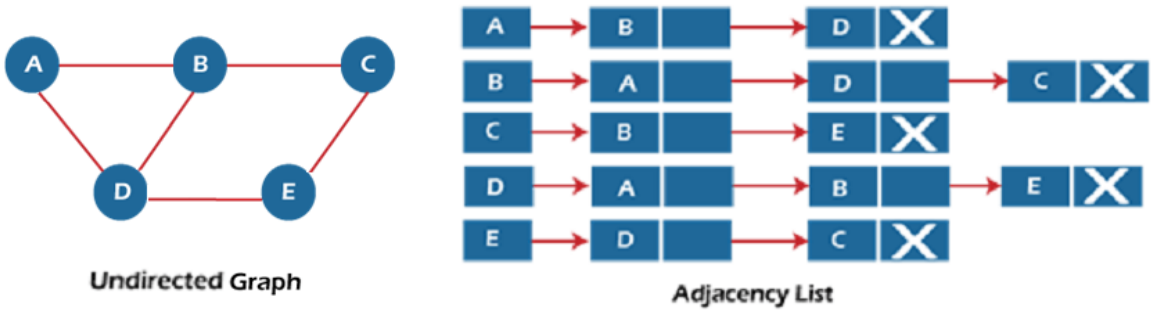
Adjacency matrix is easier to implement and follow. An adjacency matrix can be used when the graph is dense and a number of edges are large.

Though, it is advantageous to use an adjacency matrix, but it consumes more space. Even if the graph is sparse, the matrix still consumes the same space.

Linked list representation

An adjacency list is used in the linked representation to store the Graph in the computer's memory. It is efficient in terms of storage as we only have to store the values for edges.

Let's see the adjacency list representation of an undirected graph.

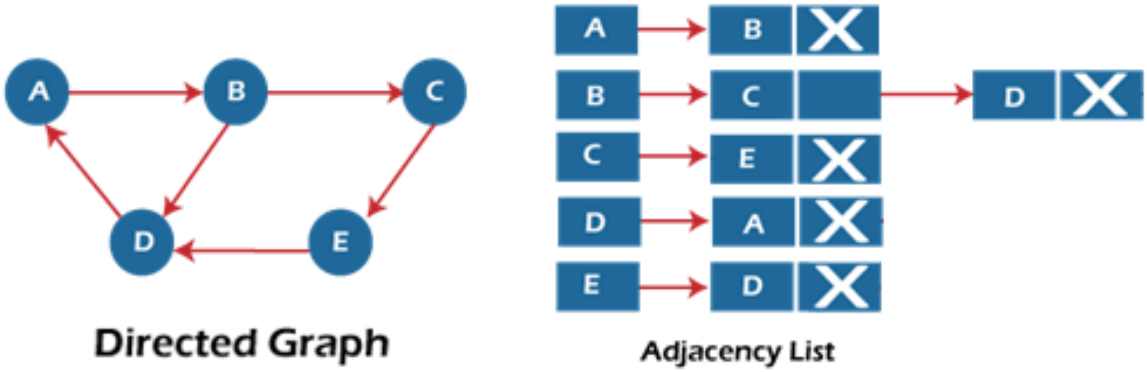


In the above figure, we can see that there is a linked list or adjacency list for every node of the graph. From vertex A, there are paths to vertex B and vertex D. These nodes are linked to nodes A in the given adjacency list.

An adjacency list is maintained for each node present in the graph, which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed, then store the NULL in the pointer field of the last node of the list.

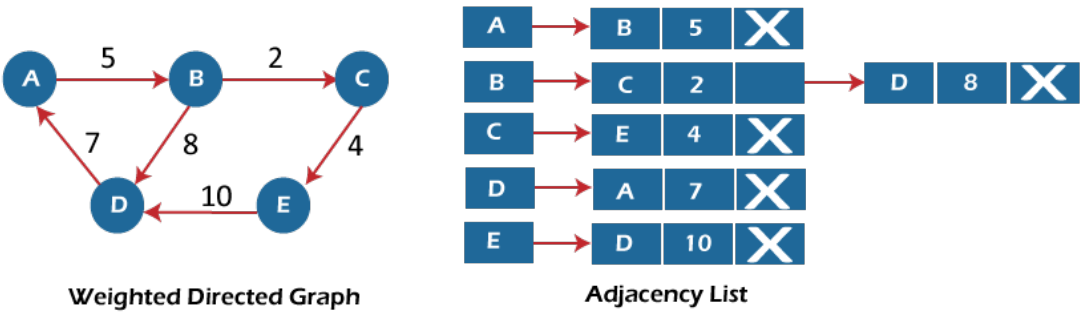
The sum of the lengths of adjacency lists is equal to twice the number of edges present in an undirected graph.

Now, consider the directed graph, and let's see the adjacency list representation of that graph.



For a directed graph, the sum of the lengths of adjacency lists is equal to the number of edges present in the graph.

Now, consider the weighted directed graph, and let's see the adjacency list representation of that graph.



Weighted Directed Graph

Adjacency List

In the case of a weighted directed graph, each node contains an extra field that is called the weight of the node.

In an adjacency list, it is easy to add a vertex. Because of using the linked list, it also saves space.

Implementation of adjacency matrix representation of Graph

Now, let's see the implementation of adjacency matrix representation of graph in C.

In this program, there is an adjacency matrix representation of an undirected graph. It means that if there is an edge exists from vertex A to vertex B, there will also an edge exists from vertex B to vertex A.

Here, there are four vertices and five edges in the graph that are non-directed.

```
/* Adjacency Matrix representation of an undirected graph in C */

#include <stdio.h>
#define V 4 /* number of vertices in the graph */

/* function to initialize the matrix to zero */
void init(int arr[][V]) {
  int i, j;
  for (i = 0; i < V; i++)
    for (j = 0; j < V; j++)
      arr[i][j] = 0;
}

/* function to add edges to the graph */
```

```c
void insertEdge(int arr[][V], int i, int j) {
  arr[i][j] = 1;
  arr[j][i] = 1;
}

/* function to print the matrix elements */
void printAdjMatrix(int arr[][V]) {
  int i, j;
  for (i = 0; i < V; i++) {
    printf("%d: ", i);
    for (j = 0; j < V; j++) {
      printf("%d ", arr[i][j]);
    }
    printf("\n");
  }
}

int main() {
  int adjMatrix[V][V];

  init(adjMatrix);
  insertEdge(adjMatrix, 0, 1);
  insertEdge(adjMatrix, 0, 2);
  insertEdge(adjMatrix, 1, 2);
  insertEdge(adjMatrix, 2, 0);
  insertEdge(adjMatrix, 2, 3);

  printAdjMatrix(adjMatrix);

  return 0;
}
```

BFS algorithm

In this article, we will discuss the BFS algorithm in the data structure. Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

There are many ways to traverse the graph, but among them, BFS is the most commonly used approach. It is a recursive algorithm to search all the vertices of a tree or graph data structure. BFS puts every vertex of the graph into two categories - visited and non-visited. It selects a single node in a graph and, after that, visits all the nodes adjacent to the selected node.

Applications of BFS algorithm

The applications of breadth-first-algorithm are given as follows -

- BFS can be used to find the neighboring locations from a given source location.
- In a peer-to-peer network, BFS algorithm can be used as a traversal method to find all the neighboring nodes. Most torrent clients, such as BitTorrent, uTorrent, etc. employ this process to find "seeds" and "peers" in the network.
- BFS can be used in web crawlers to create web page indexes. It is one of the main algorithms that can be used to index web pages. It starts traversing from the source page and follows the links associated with the page. Here, every web page is considered as a node in the graph.
- BFS is used to determine the shortest path and minimum spanning tree.
- BFS is also used in Cheney's technique to duplicate the garbage collection.
- It can be used in ford-Fulkerson method to compute the maximum flow in a flow network.

Algorithm

The steps involved in the BFS algorithm to explore a graph are given as follows -

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set

their STATUS = 2

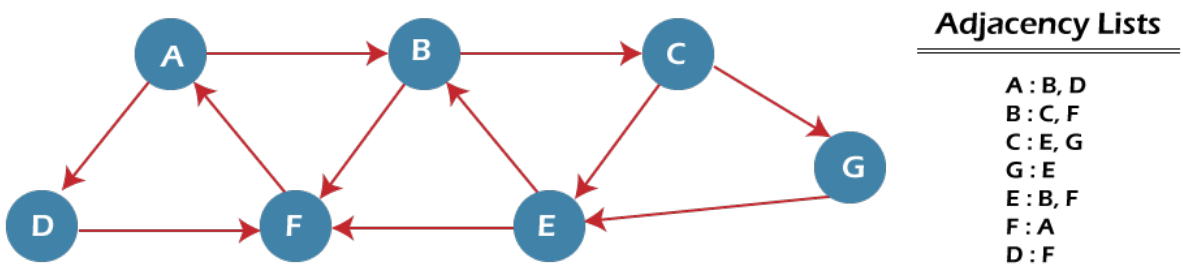
(waiting state)

[END OF LOOP]

Step 6: EXIT

Example of BFS algorithm

Now, let's understand the working of BFS algorithm by using an example. In the example given below, there is a directed graph having 7 vertices.



In the above graph, minimum path 'P' can be found by using the BFS that will start from Node A and end at Node E. The algorithm uses two queues, namely QUEUE1 and QUEUE2. QUEUE1 holds all the nodes that are to be processed, while QUEUE2 holds all the nodes that are processed and deleted from QUEUE1.

Now, let's start examining the graph starting from Node A.

Step 1 - First, add A to queue1 and NULL to queue2.

1. QUEUE1 = {A}
2. QUEUE2 = {NULL}

Step 2 - Now, delete node A from queue1 and add it into queue2. Insert all neighbors of node A to queue1.

1. QUEUE1 = {B, D}
2. QUEUE2 = {A}

Step 3 - Now, delete node B from queue1 and add it into queue2. Insert all neighbors of node B to queue1.

1. QUEUE1 = {D, C, F}
2. QUEUE2 = {A, B}

Step 4 - Now, delete node D from queue1 and add it into queue2. Insert all neighbors of node D to queue1. The only neighbor of Node D is F since it is already inserted, so it will not be inserted again.

1. QUEUE1 = {C, F}
2. QUEUE2 = {A, B, D}

Step 5 - Delete node C from queue1 and add it into queue2. Insert all neighbors of node C to queue1.

1. QUEUE1 = {F, E, G}
2. QUEUE2 = {A, B, D, C}

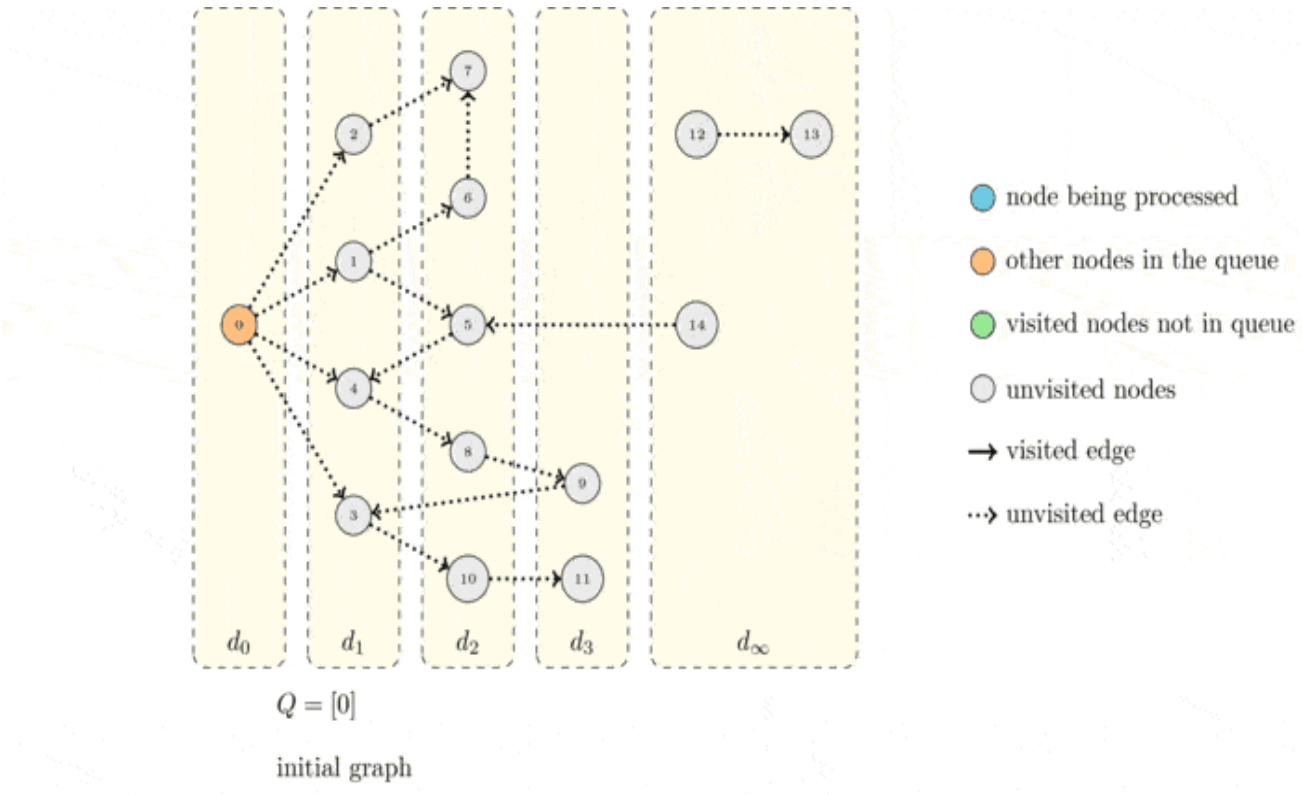
Step 5 - Delete node F from queue1 and add it into queue2. Insert all neighbors of node F to queue1. Since all the neighbors of node F are already present, we will not insert them again.

1. QUEUE1 = {E, G}
2. QUEUE2 = {A, B, D, C, F}

Step 6 - Delete node E from queue1. Since all of its neighbors have already been added, so we will not insert them again. Now, all the nodes are visited, and the target node E is encountered into queue2.

1. QUEUE1 = {G}
2. QUEUE2 = {A, B, D, C, F, E}

++



DFS (Depth First Search) algorithm

In this article, we will discuss the DFS algorithm in the data structure. It is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

Because of the recursive nature, stack data structure can be used to implement the DFS algorithm. The process of implementing the DFS is similar to the BFS algorithm.

The step by step process to implement the DFS traversal is given as follows -

1. First, create a stack with the total number of vertices in the graph.
2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
5. If no vertex is left, go back and pop a vertex from the stack.
6. Repeat steps 2, 3, and 4 until the stack is empty.

Applications of DFS algorithm

The applications of using the DFS algorithm are given as follows -

- DFS algorithm can be used to implement the topological sorting.
- It can be used to find the paths between two vertices.
- It can also be used to detect cycles in the graph.
- DFS algorithm is also used for one solution puzzles.
- DFS is used to determine if a graph is bipartite or not.

Algorithm

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

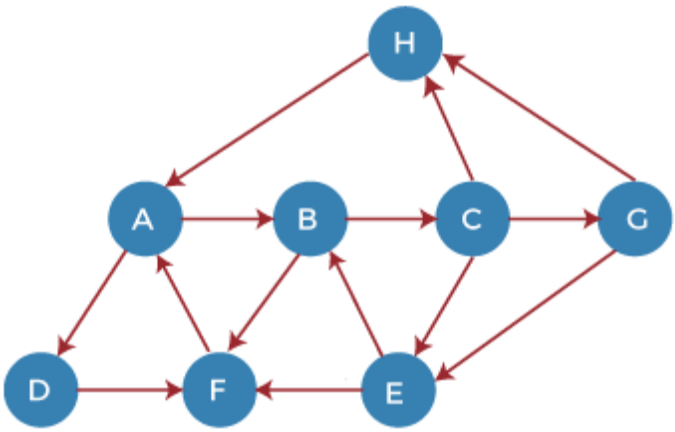
Step 6: EXIT

Pseudocode

```
1.  DFS(G,v)   ( v is the vertex where the search starts )
2.        Stack S := {};   ( start with an empty stack )
3.        for each vertex u, set visited[u] := false;
4.        push S, v;
5.        while (S is not empty) do
6.          u := pop S;
7.          if (not visited[u]) then
8.            visited[u] := true;
9.            for each unvisited neighbour w of uu
10.             push S, w;
11.         end if
12.       end while
13.    END DFS()
```

Example of DFS algorithm

Now, let's understand the working of the DFS algorithm by using an example. In the example given below, there is a directed graph having 7 vertices.



Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

Now, let's start examining the graph starting from Node H.

Step 1 - First, push H onto the stack.

1. STACK: H

Step 2 - POP the top element from the stack, i.e., H, and print it. Now, PUSH all the neighbors of H onto the stack that are in ready state.

1. Print: H
2. STACK: A

Step 3 - POP the top element from the stack, i.e., A, and print it. Now, PUSH all the neighbors of A onto the stack that are in ready state.

1. Print: A
2. STACK: B, D

Step 4 - POP the top element from the stack, i.e., D, and print it. Now, PUSH all the neighbors of D onto the stack that are in ready state.

1. Print: D
2. STACK: B, F

Step 5 - POP the top element from the stack, i.e., F, and print it. Now, PUSH all the neighbors of F onto the stack that are in ready state.

1. Print: F
2. STACK: B

Step 6 - POP the top element from the stack, i.e., B, and print it. Now, PUSH all the neighbors of B onto the stack that are in ready state.

1. Print: B
2. STACK: C

Step 7 - POP the top element from the stack, i.e., C, and print it. Now, PUSH all the neighbors of C onto the stack that are in ready state.

1. Print: C
2. STACK: E, G

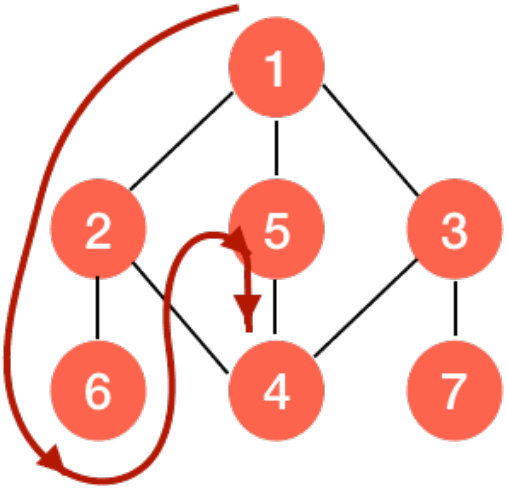
Step 8 - POP the top element from the stack, i.e., G and PUSH all the neighbors of G onto the stack that are in ready state.

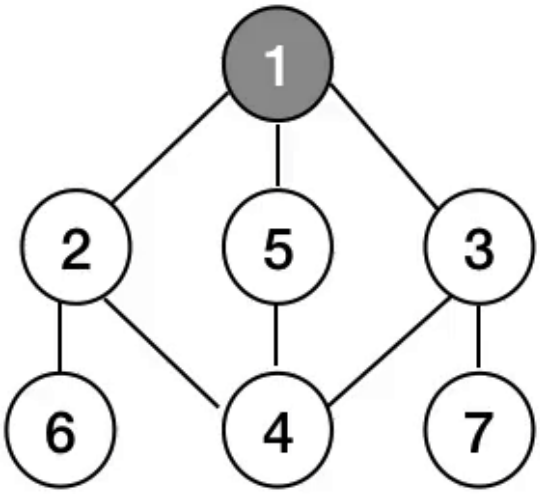
1. Print: G
2. STACK: E

Step 9 - POP the top element from the stack, i.e., E and PUSH all the neighbors of E onto the stack that are in ready state.

1. Print: E
2. STACK:

++++++++++++++++++++++++++++++++

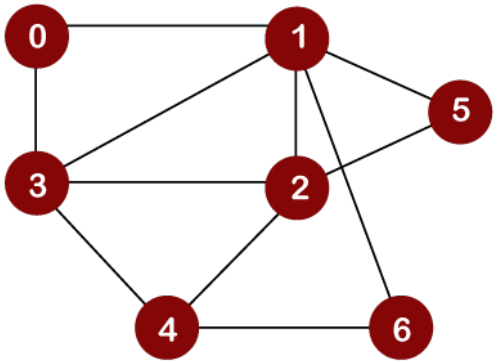




What is BFS?

BFS stands for ***Breadth First Search***. It is also known as **level order traversal**. The Queue data structure is used for the Breadth First Search traversal. When we use the BFS algorithm for the traversal in a graph, we can consider any node as a root node.

Let's consider the below graph for the breadth first search traversal.



Suppose we consider node 0 as a root node. Therefore, the traversing would be started from node 0.

Once node 0 is removed from the Queue, it gets printed and marked as a ***visited node.***

Once node 0 gets removed from the Queue, then the adjacent nodes of node 0 would be inserted in a Queue as shown below:

Result : 0

Now the node 1 will be removed from the Queue; it gets printed and marked as a visited node

Once node 1 gets removed from the Queue, then all the adjacent nodes of a node 1 will be added in a Queue. The adjacent nodes of node 1 are 0, 3, 2, 6, and 5. But we have to insert only unvisited nodes in a Queue. Since nodes 3, 2, 6, and 5 are unvisited; therefore, these nodes will be added in a Queue as shown below:

Result : 0 , 1

The next node is 3 in a Queue. So, node 3 will be removed from the Queue, it gets printed and marked as visited as shown below:

Result : 0, 1, 3

Once node 3 gets removed from the Queue, then all the adjacent nodes of node 3 except the visited nodes will be added in a Queue. The adjacent nodes of node 3 are 0, 1, 2, and 4. Since nodes 0, 1 are already visited, and node 2 is present in a Queue; therefore, we need to insert only node 4 in a Queue.

Result : 0, 1, 3

Now, the next node in the Queue is 2. So, 2 would be deleted from the Queue. It gets printed and marked as visited as shown below:

Result : 0, 1, 3, 2,

Once node 2 gets removed from the Queue, then all the adjacent nodes of node 2 except the visited nodes will be added in a Queue. The adjacent nodes of node 2 are 1, 3, 5, 6, and 4. Since the nodes 1 and 3 have already been visited, and 4, 5, 6 are already added in the Queue; therefore, we do not need to insert any node in the Queue.

The next element is 5. So, 5 would be deleted from the Queue. It gets printed and marked as visited as shown below:

Result : 0, 1, 3, 2, 5

Once node 5 gets removed from the Queue, then all the adjacent nodes of node 5 except the visited nodes will be added in the Queue. The adjacent nodes of node 5 are 1 and 2. Since both the nodes have already been visited; therefore, there is no vertex to be inserted in a Queue.

The next node is 6. So, 6 would be deleted from the Queue. It gets printed and marked as visited as shown below:

Result : 0, 1, 3, 2, 5, 6

Once the node 6 gets removed from the Queue, then all the adjacent nodes of node 6 except the visited nodes will be added in the Queue. The adjacent nodes of node 6 are 1 and 4. Since the node 1 has already been visited and node 4 is already added in the Queue; therefore, there is not vertex to be inserted in the Queue.

The next element in the Queue is 4. So, 4 would be deleted from the Queue. It gets printed and marked as visited.

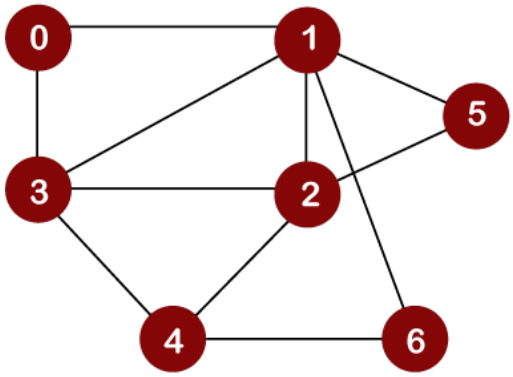
Once the node 4 gets removed from the Queue, then all the adjacent nodes of node 4 except the visited nodes will be added in the Queue. The adjacent nodes of node 4 are 3, 2, and 6. Since all the adjacent nodes have already been visited; so, there is no vertex to be inserted in the Queue.

What is DFS?

DFS stands for Depth First Search. In DFS traversal, the stack data structure is used, which works on the LIFO (Last In First Out) principle. In DFS, traversing can be started from any node, or we can say that any node can be considered as a root node until the root node is not mentioned in the problem.

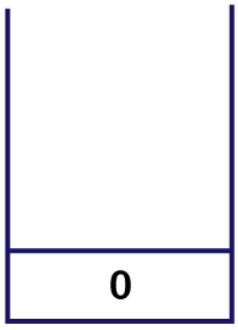
In the case of BFS, the element which is deleted from the Queue, the adjacent nodes of the deleted node are added to the Queue. In contrast, in DFS, the element which is removed from the stack, then only one adjacent node of a deleted node is added in the stack.

Let's consider the below graph for the Depth First Search traversal.

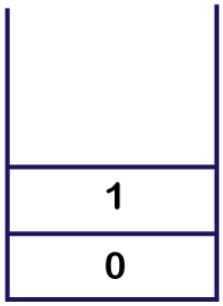


Consider node 0 as a root node.

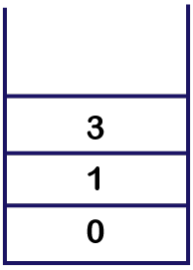
First, we insert the element 0 in the stack as shown below:



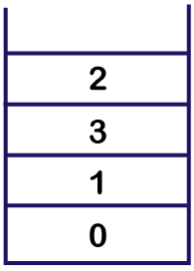
The node 0 has two adjacent nodes, i.e., 1 and 3. Now we can take only one adjacent node, either 1 or 3, for traversing. Suppose we consider node 1; therefore, 1 is inserted in a stack and gets printed as shown below:



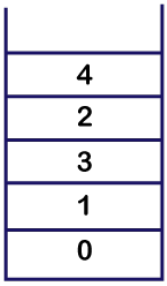
Now we will look at the adjacent vertices of node 1. The unvisited adjacent vertices of node 1 are 3, 2, 5 and 6. We can consider any of these four vertices. Suppose we take node 3 and insert it in the stack as shown below:



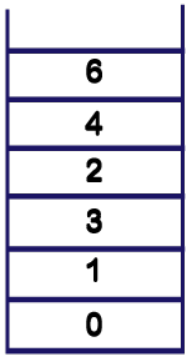
Consider the unvisited adjacent vertices of node 3. The unvisited adjacent vertices of node 3 are 2 and 4. We can take either of the vertices, i.e., 2 or 4. Suppose we take vertex 2 and insert it in the stack as shown below:



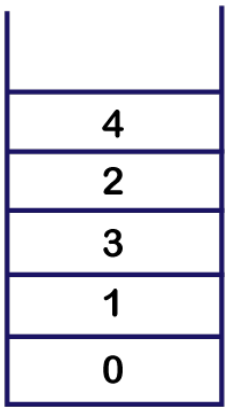
The unvisited adjacent vertices of node 2 are 5 and 4. We can choose either of the vertices, i.e., 5 or 4. Suppose we take vertex 4 and insert in the stack as shown below:



Now we will consider the unvisited adjacent vertices of node 4. The unvisited adjacent vertex of node 4 is node 6. Therefore, element 6 is inserted into the stack as shown below:

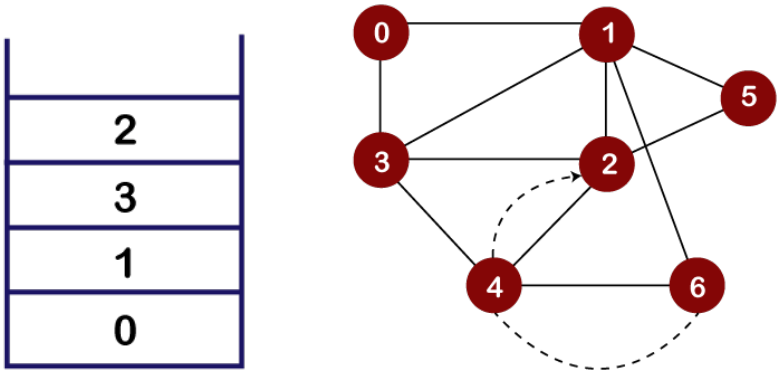


After inserting element 6 in the stack, we will look at the unvisited adjacent vertices of node 6. As there is no unvisited adjacent vertices of node 6, so we cannot move beyond node 6. In this case, we will perform **backtracking**. The topmost element, i.e., 6 would be popped out from the stack as shown below:

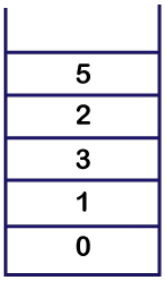


x

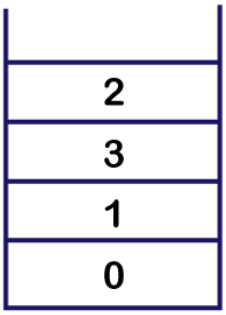
The topmost element in the stack is 4. Since there are no unvisited adjacent vertices left of node 4; therefore, node 4 is popped out from the stack as shown below:



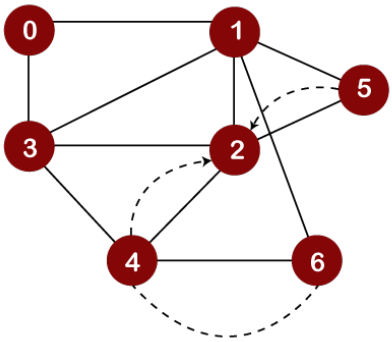
The next topmost element in the stack is 2. Now, we will look at the unvisited adjacent vertices of node 2. Since only one unvisited node, i.e., 5 is left, so node 5 would be pushed into the stack above 2 and gets printed as shown below:



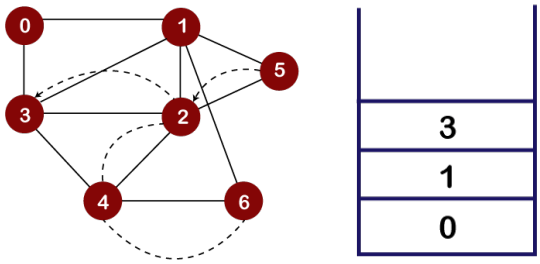
Now we will check the adjacent vertices of node 5, which are still unvisited. Since there is no vertex left to be visited, so we pop the element 5 from the stack as shown below:



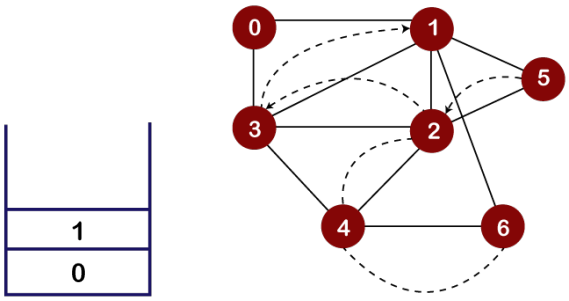
We cannot move further 5, so we need to perform backtracking. In backtracking, the topmost element would be popped out from the stack. The topmost element is 5 that would be popped out from the stack, and we move back to node 2 as shown below:



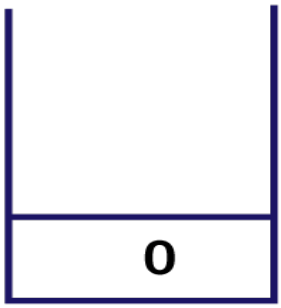
Now we will check the unvisited adjacent vertices of node 2. As there is no adjacent vertex left to be visited, so we perform backtracking. In backtracking, the topmost element, i.e., 2 would be popped out from the stack, and we move back to the node 3 as shown below:

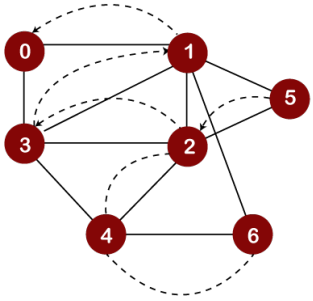


Now we will check the unvisited adjacent vertices of node 3. As there is no adjacent vertex left to be visited, so we perform backtracking. In backtracking, the topmost element, i.e., 3 would be popped out from the stack and we move back to node 1 as shown below:



After popping out element 3, we will check the unvisited adjacent vertices of node 1. Since there is no vertex left to be visited; therefore, the backtracking will be performed. In backtracking, the topmost element, i.e., 1 would be popped out from the stack, and we move back to node 0 as shown below:





We will check the adjacent vertices of node 0, which are still unvisited. As there is no adjacent vertex left to be visited, so we perform backtracking. In this, only one element, i.e., 0 left in the stack, would be popped out from the stack as shown below:

As we can observe in the above figure that the stack is empty. So, we have to stop the DFS traversal here, and the elements which are printed is the result of the DFS traversal.

Differences between BFS and DFS

The following are the differences between the BFS and DFS:

	BFS	DFS
Full form	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
Technique	It a vertex-based technique to find the shortest path in a graph.	It is an edge-based technique because the vertices along the edge are explored first from the starting to the end node.
Definition	BFS is a traversal technique in which all the nodes of the same level are explored first, and then we move to the next level.	DFS is also a traversal technique in which traversal is started from the root node and explore the nodes as far as possible until we reach the node that has no unvisited adjacent nodes.
Data Structure	Queue data structure is used for the BFS traversal.	Stack data structure is used for the BFS traversal.
Backtracking	BFS does not use the backtracking concept.	DFS uses backtracking to traverse all the unvisited nodes.
Number of edges	BFS finds the shortest path having a minimum number of edges to traverse from the source to the destination vertex.	In DFS, a greater number of edges are required to traverse from the source vertex to the destination vertex.
Optimality	BFS traversal is optimal for those vertices which are to be searched closer to the source vertex.	DFS traversal is optimal for those graphs in which solutions are away from the source vertex.
Speed	BFS is slower than DFS.	DFS is faster than BFS.
Suitability for decision tree	It is not suitable for the decision tree because it requires exploring all the neighboring nodes first.	It is suitable for the decision tree. Based on the decision, it explores all the paths. When the goal is found, it stops its traversal.
Memory efficient	It is not memory efficient as it requires more memory than DFS.	It is memory efficient as it requires less memory than BFS.

What is a spanning tree?

A spanning tree can be defined as the subgraph of an undirected connected graph. It includes all the vertices along with the least possible number of edges. If any vertex is missed, it is not a spanning tree. A spanning tree is a subset of the graph that does not have cycles, and it also cannot be disconnected.

A spanning tree consists of (n-1) edges, where 'n' is the number of vertices (or nodes). Edges of the spanning tree may or may not have weights assigned to them. All the possible spanning trees created from the given graph G would have the same number of vertices, but the number of edges in the spanning tree would be equal to the number of vertices in the given graph minus 1.

A complete undirected graph can have $\mathbf{n^{n-2}}$ number of spanning trees where **n** is the number of vertices

Applications of the spanning tree

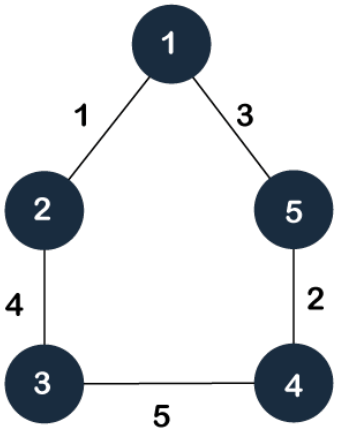
Basically, a spanning tree is used to find a minimum path to connect all nodes of the graph. Some of the common applications of the spanning tree are listed as follows -

- Cluster Analysis
- Civil network planning
- Computer network routing protocol

Now, let's understand the spanning tree with the help of an example.

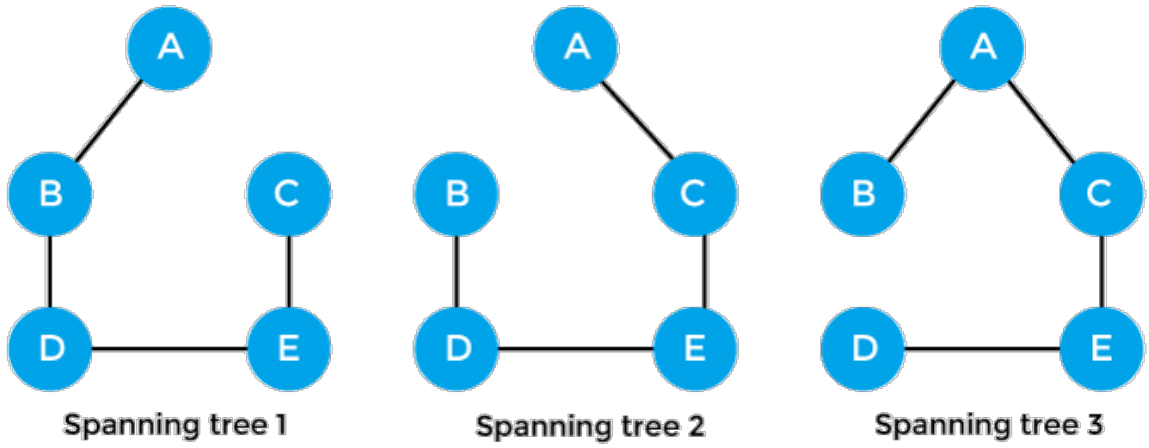
Example of Spanning tree

Suppose the graph be -



As discussed above, a spanning tree contains the same number of vertices as the graph, the number of vertices in the above graph is 5; therefore, the spanning tree will contain 5 vertices. The edges in the spanning tree will be equal to the number of vertices in the graph minus 1. So, there will be 4 edges in the spanning tree.

Some of the possible spanning trees that will be created from the above graph are given as follows -



Properties of spanning-tree

Some of the properties of the spanning tree are given as follows -

- There can be more than one spanning tree of a connected graph G.
- A spanning tree does not have any cycles or loop.
- A spanning tree is **minimally connected,** so removing one edge from the tree will make the graph disconnected.
- A spanning tree is **maximally acyclic,** so adding one edge to the tree will create a loop.
- There can be a maximum $\mathbf{n^{n-2}}$ number of spanning trees that can be created from a complete graph.
- A spanning tree has **n-1** edges, where 'n' is the number of nodes.
- If the graph is a complete graph, then the spanning tree can be constructed by removing maximum (e-n+1) edges, where 'e' is the number of edges and 'n' is the number of vertices.

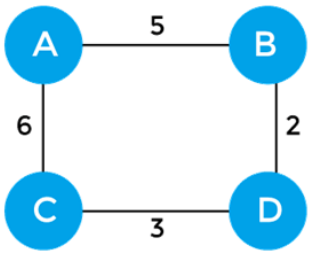
So, a spanning tree is a subset of connected graph G, and there is no spanning tree of a disconnected graph.

Minimum Spanning tree

A minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree. In the real world, this weight can be considered as the distance, traffic load, congestion, or any random value.

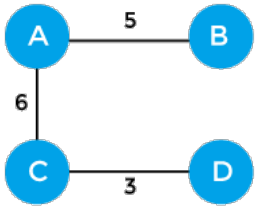
Example of minimum spanning tree

Let's understand the minimum spanning tree with the help of an example.

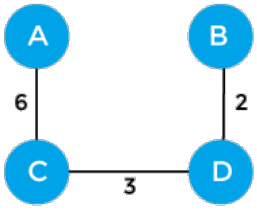


Weighted graph

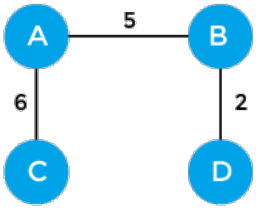
The sum of the edges of the above graph is 16. Now, some of the possible spanning trees created from the above graph are -



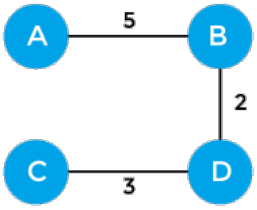
Sum = 14
Minimum spanning tree - 1



Sum = 11
Minimum spanning tree - 2

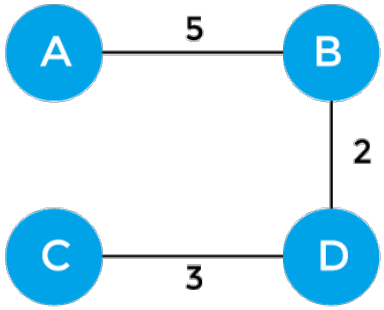


Sum = 13
Minimum spanning tree - 3



Sum = 10
Minimum spanning tree - 4

So, the minimum spanning tree that is selected from the above spanning trees for the given weighted graph is -



Sum = 10

Applications of minimum spanning tree

The applications of the minimum spanning tree are given as follows -

- Minimum spanning tree can be used to design water-supply networks, telecommunication networks, and electrical grids.
- It can be used to find paths in the map.

Algorithms for Minimum spanning tree

A minimum spanning tree can be found from a weighted graph by using the algorithms given below -

- Prim's Algorithm
- Kruskal's Algorithm

Prim's Algorithm

Before starting the main topic, we should discuss the basic and important terms such as spanning tree and minimum spanning tree.

Spanning tree - A spanning tree is the subgraph of an undirected connected graph.

Minimum Spanning tree - Minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.

Now, let's start the main topic.

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 minim

Prim's algorithm starts with the single node 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.

How does the prim's algorithm work?

Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows -

- First, we have to initialize an MST with the randomly chosen vertex.
- Now, we have to 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.
- Repeat step 2 until the minimum spanning tree is formed.

The applications of prim's algorithm are -

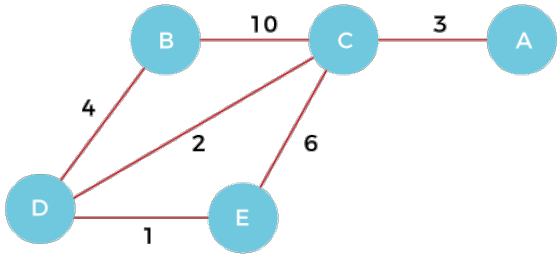
- Prim's algorithm can be used in network designing.
- It can be used to make network cycles.

- It can also be used to lay down electrical wiring cables.

Example of prim's algorithm

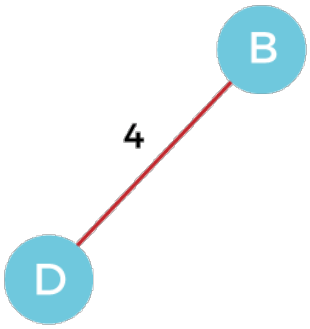
Now, let's see the working of prim's algorithm using an example. It will be easier to understand the prim's algorithm using an example.

Suppose, a weighted graph is -

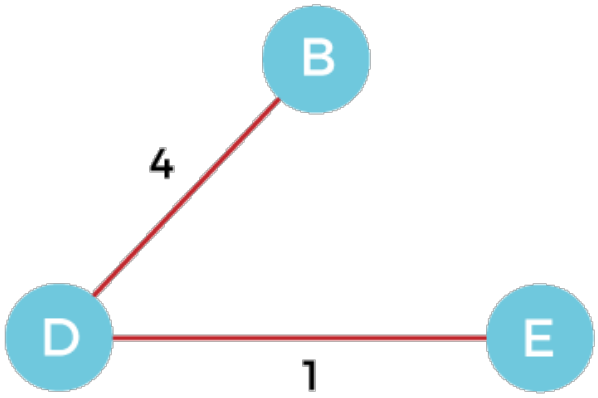


Step 1 - First, we have to choose a vertex from the above graph. Let's choose B.

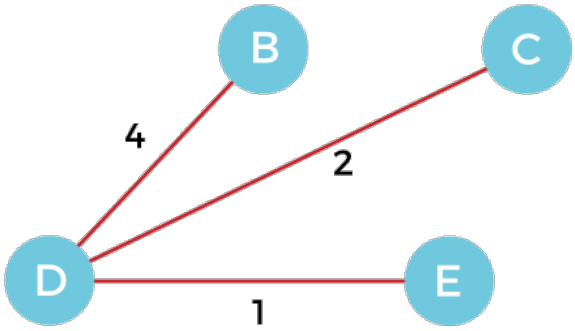
Step 2 - Now, we have to choose and add the shortest edge from vertex B. There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.



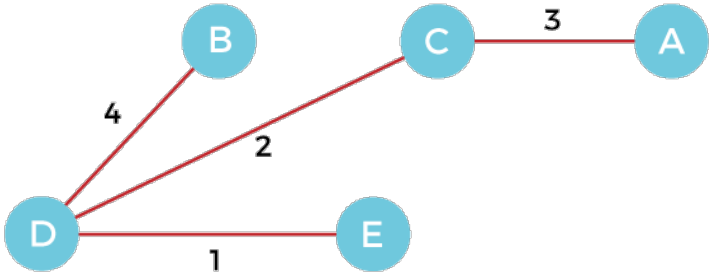
Step 3 - Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.



Step 4 - Now, select the edge CD, and add it to the MST.



Step 5 - Now, choose the edge CA. Here, we cannot select the edge CE as it would create a cycle to the graph. So, choose the edge CA and add it to the MST.



So, the graph produced in step 5 is the minimum spanning tree of the given graph. The cost of the MST is given below -

Cost of MST = 4 + 2 + 1 + 3 = 10 units.

Algorithm

1. Step 1: Select a starting vertex
2. Step 2: Repeat Steps 3 and 4 until there are fringe vertices

3. Step 3: Select an edge 'e' connecting the tree vertex and fringe vertex that has minimum weight
4. Step 4: Add the selected edge and the vertex to the minimum spanning tree T

5. [END OF LOOP]
6. Step 5: EXIT

Kruskal's Algorithm

In this article, we will discuss Kruskal's algorithm. Here, we will also see the complexity, working, example, and implementation of the Kruskal's algorithm.

But before moving directly towards the algorithm, we should first understand the basic terms such as spanning tree and minimum spanning tree.

Spanning tree - A spanning tree is the subgraph of an undirected connected graph.

Minimum Spanning tree - Minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.

Now, let's start with the main topic.

Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph. It follows the greedy approach that finds an optimum solution at every stage instead of focusing on a global optimum.

How does Kruskal's algorithm work?

In Kruskal's algorithm, we start from edges with the lowest weight and keep adding the edges until the goal is reached. The steps to implement Kruskal's algorithm are listed as follows -

- First, sort all the edges from low weight to high.
- Now, 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.

- Continue to add the edges until we reach all vertices, and a minimum spanning tree is created.

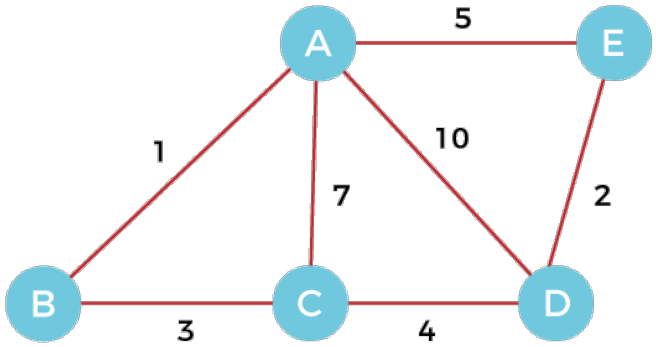
The applications of Kruskal's algorithm are -

- Kruskal's algorithm can be used to layout electrical wiring among cities.
- It can be used to lay down LAN connections.

Example of Kruskal's algorithm

Now, let's see the working of Kruskal's algorithm using an example. It will be easier to understand Kruskal's algorithm using an example.

Suppose a weighted graph is -



The weight of the edges of the above graph is given in the below table -

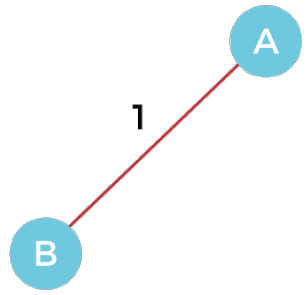
Edge	AB	AC	AD	AE	BC	CD	DE
Weight	1	7	10	5	3	4	2

Now, sort the edges given above in the ascending order of their weights.

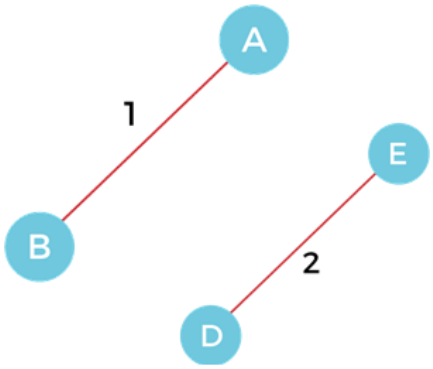
Edge	AB	DE	BC	CD	AE	AC	AD
Weight	1	2	3	4	5	7	10

Now, let's start constructing the minimum spanning tree.

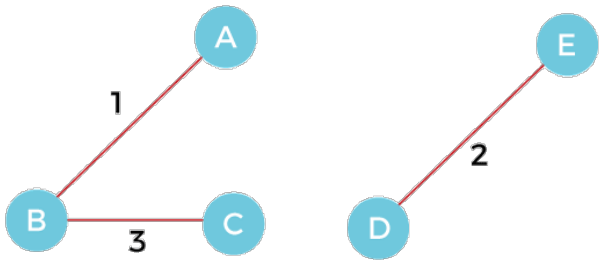
Step 1 - First, add the edge **AB** with weight **1** to the MST.



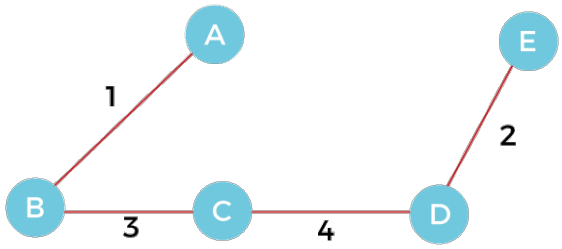
Step 2 - Add the edge **DE** with weight **2** to the MST as it is not creating the cycle.



Step 3 - Add the edge **BC** with weight **3** to the MST, as it is not creating any cycle or loop.



Step 4 - Now, pick the edge **CD** with weight **4** to the MST, as it is not forming the cycle.

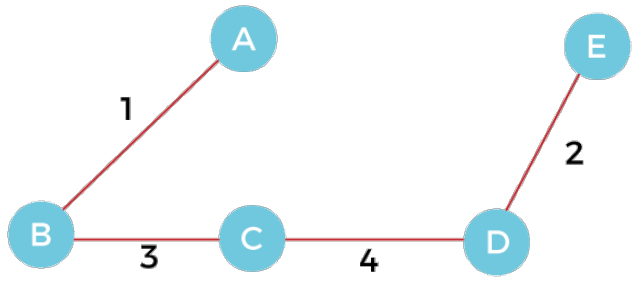


Step 5 - After that, pick the edge **AE** with weight **5.** Including this edge will create the cycle, so discard it.

Step 6 - Pick the edge **AC** with weight **7.** Including cycle, so discard it.

Step 7 - Pick the edge **AD** with weight **10.** Including this edge will also create the cycle, so discard it.

So, the final minimum spanning tree obtained from the given weighted graph by using Kruskal's algorithm is -



The cost of the MST is = AB + DE + BC + CD = 1 + 2 + 3 + 4 = 10.

Now, the number of edges in the above tree equals the number of vertices minus 1. So, the algorithm stops here.

Algorithm

1. Step 1: Create a forest F in such a way that every vertex of the graph is a separ ate tree.
2. Step 2: Create a set E that contains all the edges of the graph.
3. Step 3: Repeat Steps 4 and 5 **while** E is NOT EMPTY and F is not spanning
4. Step 4: Remove an edge from E with minimum weight
5. Step 5: IF the edge obtained in Step 4 connects two different trees, then add it to the forest F
6. (**for** combining two trees into one tree).
7. ELSE
8. Discard the edge
9. Step 6: END

   ○