Graph Data Stucture

A graph .

Let's try to understand this through an example. On facebook, everything is a node. That includes User, Photo, Album, Event, Group, Page, Comment, Story, Video, Link, Note...anything that has data is a node.

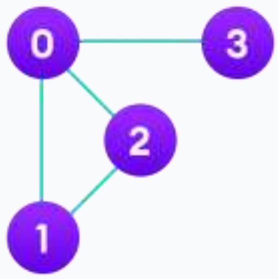
Every relationship is an edge from one node to another. Whether you post a photo, join a group, like a page, etc., a new edge is created for that relationship.

Example of graph data structure

All of facebook is then a collection of these nodes and edges. This is because facebook uses a graph data structure to store its data.

More precisely, a graph is a data structure (V, E) that consists of

- A collection of vertices V
- A collection of edges E, represented as ordered pairs of vertices (u,v)



Vertices and edges

In the graph,

```
V = {0, 1, 2, 3}

E = {(0,1), (0,2), (0,3), (1,2),(1,0),(2,0),(3,0),(2,1)}

G = {V, E}
```

Graph Terminology

- **Adjacency**: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them.
- **Path**: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.
- **Directed Graph**: A graph in which an edge (u,v) doesn't necessarily mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge.

Graph Representation

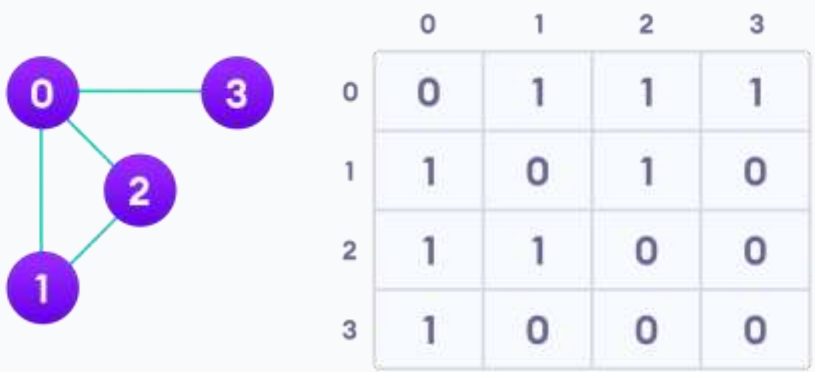
Graphs are commonly represented in two ways:

1. Adjacency Matrix

An adjacency matrix is a 2D array of V x V vertices. Each row and column represent a vertex.

If the value of any element `a[i][j]` is 1, it represents that there is an edge connecting vertex i and vertex j.

The adjacency matrix for the graph we created above is



	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

Graph adjacency matrix

Since it is an undirected graph, for edge (0,2), we also need to mark edge (2,0); making the adjacency matrix symmetric about the diagonal.

Edge lookup(checking if an edge exists between vertex A and vertex B) is extremely fast in adjacency matrix representation but we have to reserve space for every possible link between all vertices(V x V), so it requires more space.

2. Adjacency List

An adjacency list represents a graph as an array of linked lists.

The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

The adjacency list for the graph we made in the first example is as follows:

Adjacency list representation

An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a graph with millions of vertices, this can mean a lot of saved space.

Graph Operations

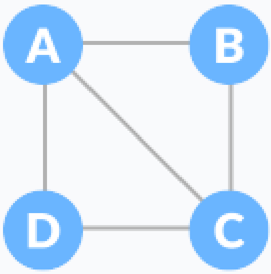
The most common graph operations are:

- Check if the element is present in the graph
- Graph Traversal
- Add elements(vertex, edges) to graph
- Finding the path from one vertex to another

Spanning Tree and Minimum Spanning Tree

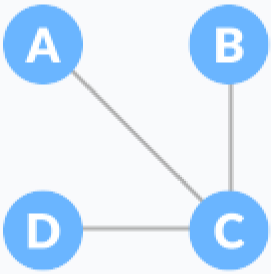
Before we learn about spanning trees, we need to understand two graphs: undirected graphs and connected graphs.

An **undirected graph** is a graph in which the edges do not point in any direction (ie. the edges are bidirectional).



Undirected Graph

A **connected graph** is a graph in which there is always a path from a vertex to any other vertex.



Connected Graph

Spanning tree

A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree.

The edges may or may not have weights assigned to them.

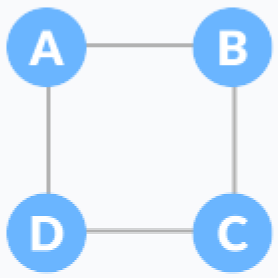
> The total number of spanning trees with n vertices that can be created from a complete graph is equal to $n^{(n-2)}$.

If we have `n = 4`, the maximum number of possible spanning trees is equal to $4^{4-2} = 16$. Thus, 16 spanning trees can be formed from a complete graph with 4 vertices.

Example of a Spanning Tree

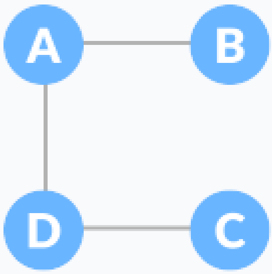
Let's understand the spanning tree with examples below:

Let the original graph be:

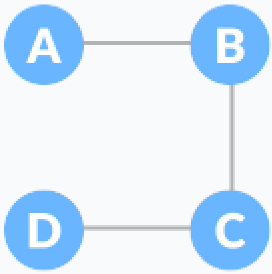


Normal graph

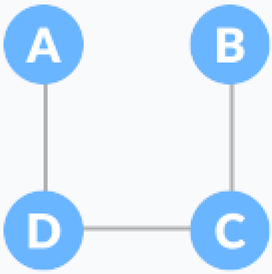
Some of the possible spanning trees that can be created from the above graph are:



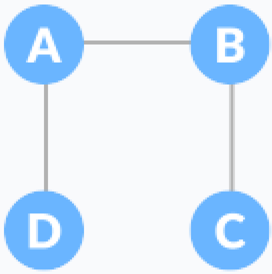
A spanning tree



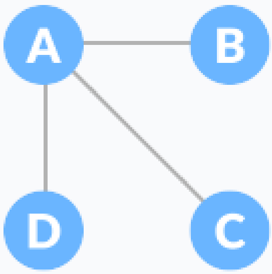
A spanning tree



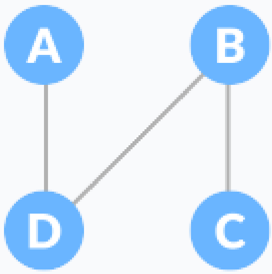
A spanning tree



A spanning tree



A spanning tree



A spanning tree

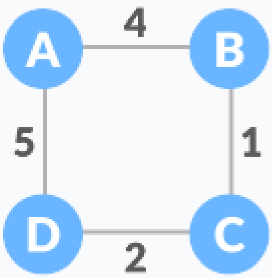
Minimum Spanning Tree

A minimum spanning tree is a spanning tree in which the sum of the weight of the edges is as minimum as possible.

Example of a Spanning Tree

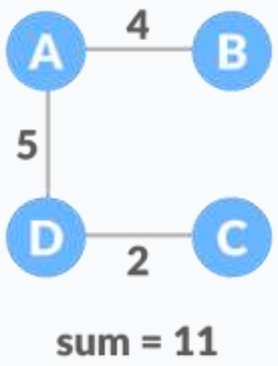
Let's understand the above definition with the help of the example below.

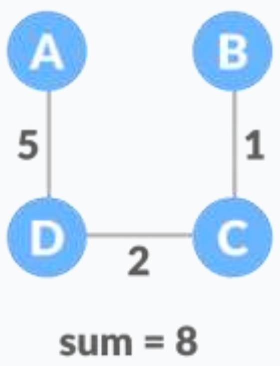
The initial graph is:



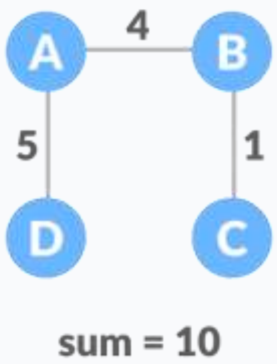
Weighted graph

The possible spanning trees from the above graph are:



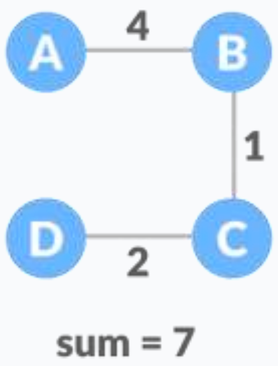


Minimum spanning tree - 1



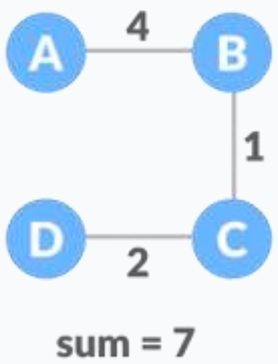
Minimum spanning tree - 2

Minimum spanning tree - 3



Minimum spanning tree - 4

The minimum spanning tree from the above spanning trees is:



Minimum spanning tree

The minimum spanning tree from a graph is found using the following algorithms:

1. Prim's Algorithm
2. Kruskal's Algorithm

Spanning Tree Applications

- Computer Network Routing Protocol
- Cluster Analysis
- Civil Network Planning

Minimum Spanning tree Applications

- To find paths in the map
- To design networks like telecommunication networks, water supply networks, and electrical grids.

Prim's Algorithm (example of dynamic programming)

Prim's Algorithm 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.

It’s a Dynamic programming algorithm

What is Dynamic Programming?

- Dynamic Programming (commonly referred to as DP) is an algorithmic technique for solving a problem by recursively breaking it down into simpler subproblems and using the fact that the optimal solution to the overall problem depends upon the optimal solution to it’s individual subproblems.
- The technique was developed by **Richard Bellman** in the 1950s.
- DP algorithm solves each subproblem just once and then remembers its answer, thereby avoiding re-computation of the answer for similar subproblem every time.
- It is the most powerful design technique for solving optimization related problems.
- It also gives us a life lesson - **Make life less complex.** There is no such thing as big problem in life. Even if it appears big, it can be solved by breaking into smaller problems and then solving each optimally.

Prim's algorithm starts with the single node and explore 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.

The algorithm is given as follows.

Algorithm

- **Step 1:** Select a starting vertex
- **Step 2:** Repeat Steps 3 and 4 until there are fringe vertices
- **Step 3:** Select an edge e connecting the tree vertex and fringe vertex that has minimum weight
- **Step 4:** Add the selected edge and the vertex to the minimum spanning tree T [END OF LOOP]
- **Step 5:** EXIT

Steps

How Prim's algorithm works

It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

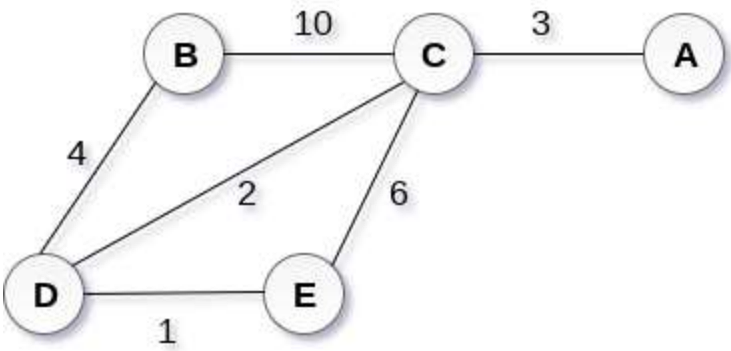
We start from one vertex and keep adding edges with the lowest weight until we reach our goal.

The steps for implementing Prim's algorithm are as follows:

1. Initialize the minimum spanning tree with a vertex chosen at random.
2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
3. Keep repeating step 2 until we get a minimum spanning tree

Example :

Construct a minimum spanning tree of the graph given in the following figure by using prim's algorithm.



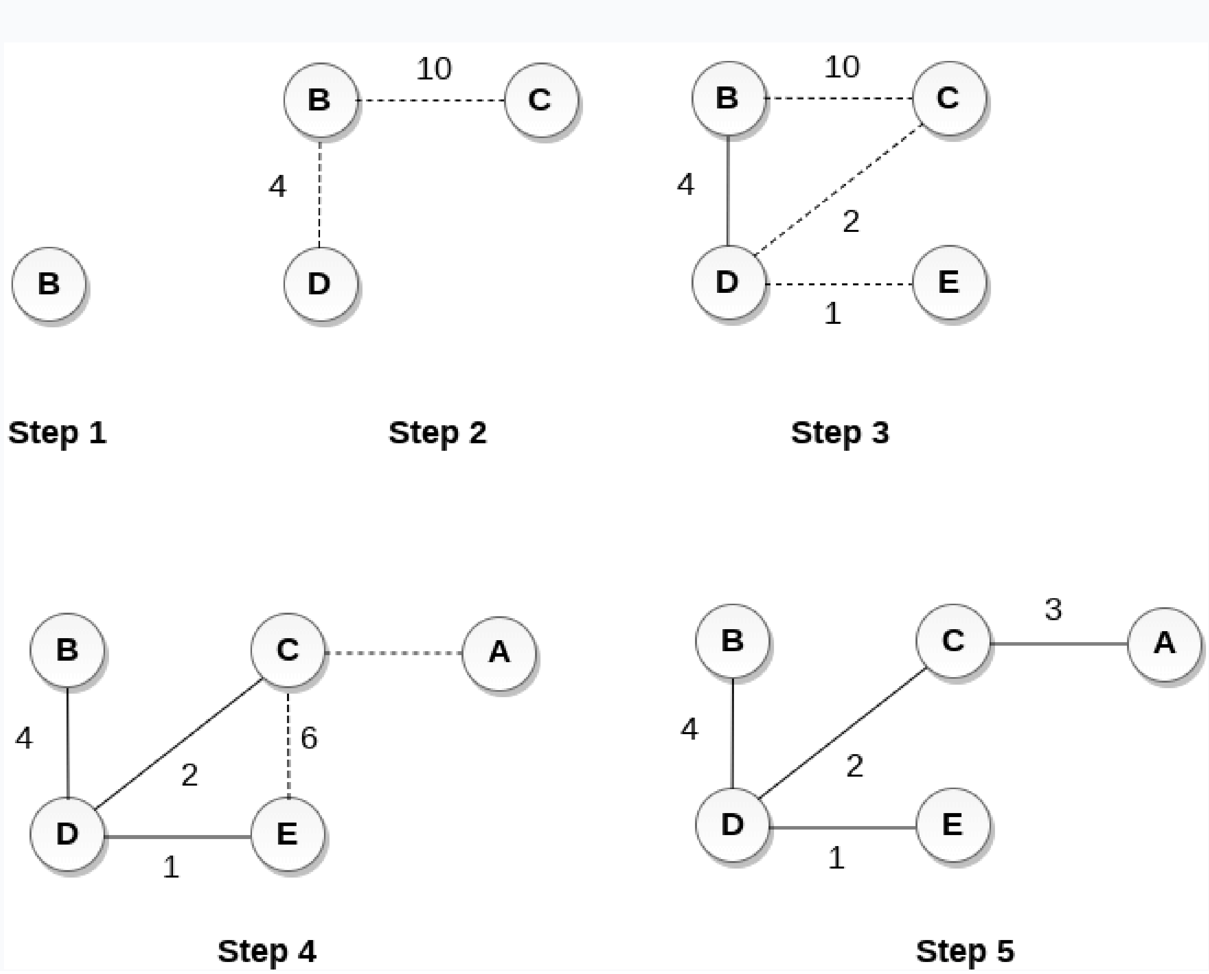
Solution

- **Step 1 :** Choose a starting vertex B.
- **Step 2:** Add the vertices that are adjacent to B. the edges that connecting the vertices are shown by dotted lines.
- **Step 3:** Choose the edge with the minimum weight among all. i.e. BD and add it to MST. Add the adjacent vertices of D i.e. C and E.
- **Step 3:** Choose the edge with the minimum weight among all. 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.
- **Step 4:** Choose the edge with the minimum weight i.e. CA. We can't choose CE as it would cause cycle in the graph.

The graph produces in the step 4 is the minimum spanning tree of the graph shown in the above figure.

The cost of MST will be calculated as;

cost(MST) = 4 + 2 + 1 + 3 = 10 units.



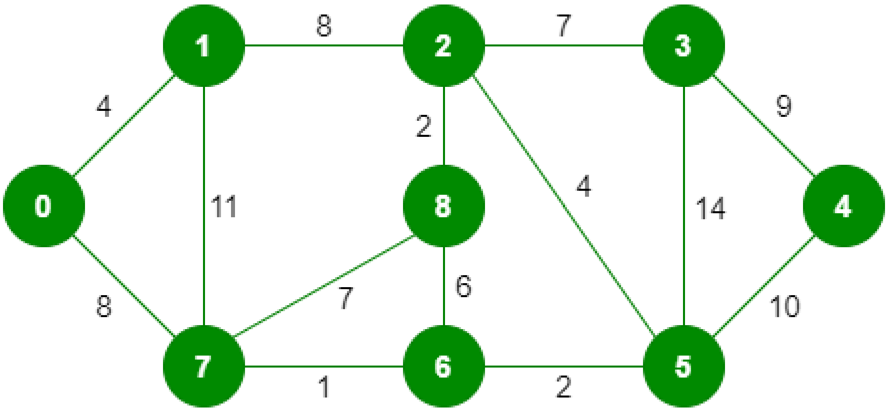
Kruscals algorithm

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

1. *Sort all the edges in ascending 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.*

Step #2 uses the Union-Find algorithm to detect cycles. So we recommend reading the following post as a prerequisite.

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.

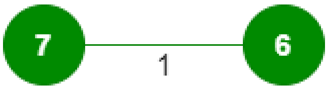
```
After sorting:
```

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

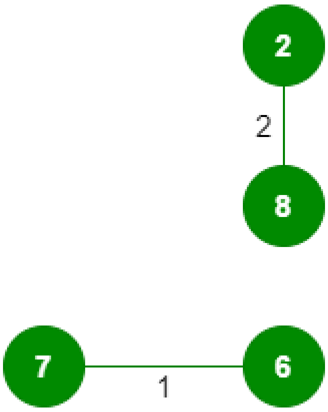
14 3 5

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

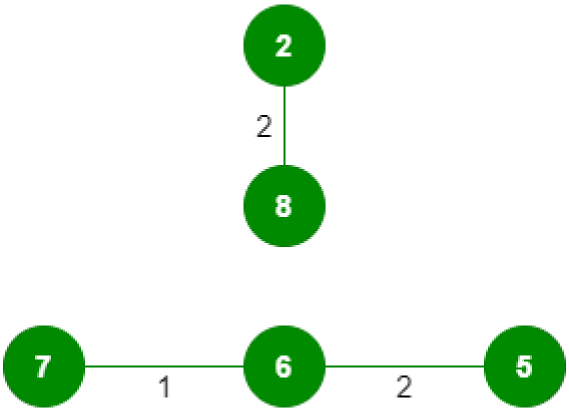
1. *Pick edge 7-6:* No cycle is formed, include it.



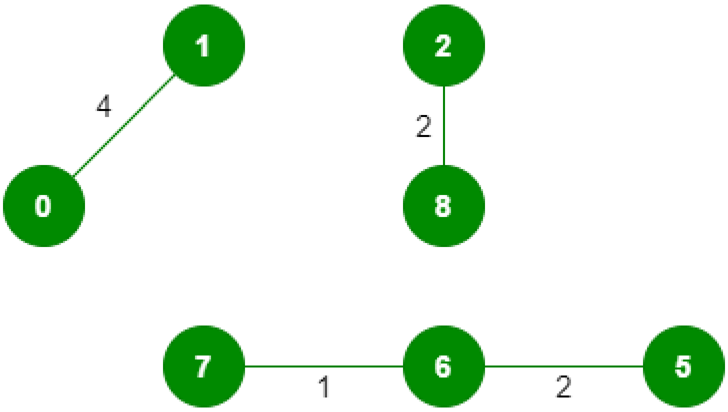
2. *Pick edge 8-2:* No cycle is formed, include it.



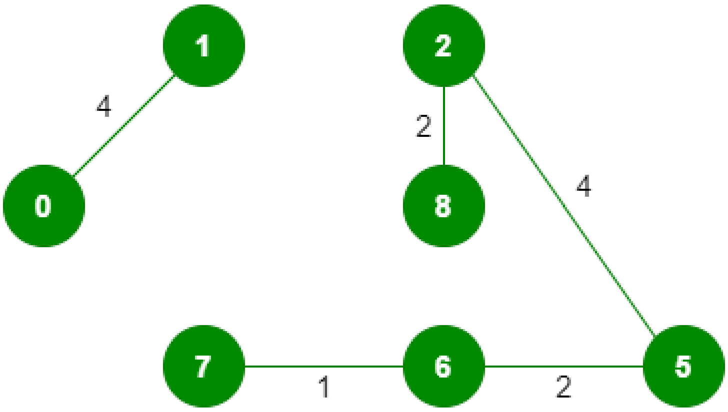
3. *Pick edge 6-5:* No cycle is formed, include it.



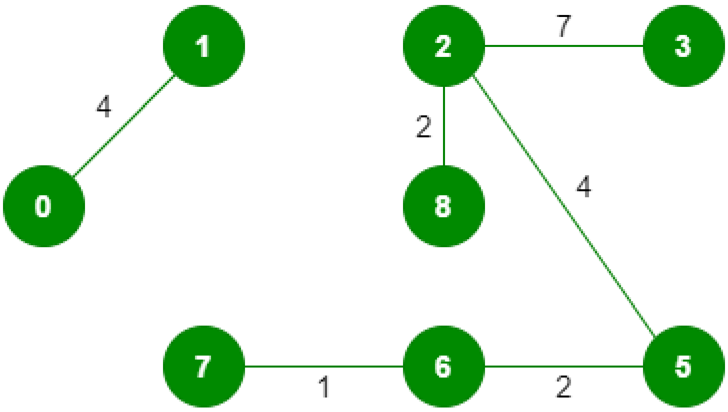
4. *Pick edge 0-1:* No cycle is formed, include it.



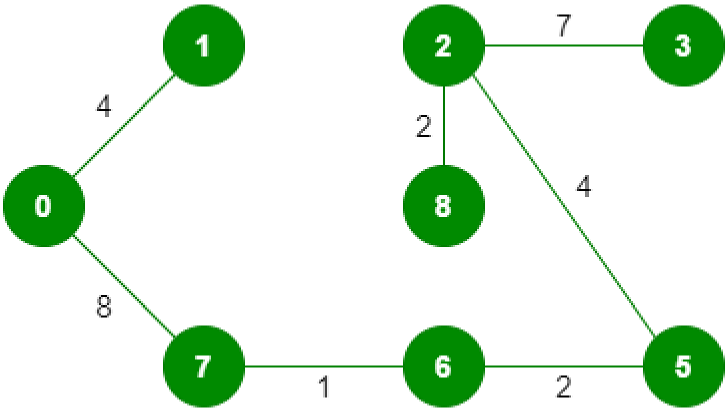
5. *Pick edge 2-5:* No cycle is formed, include it.



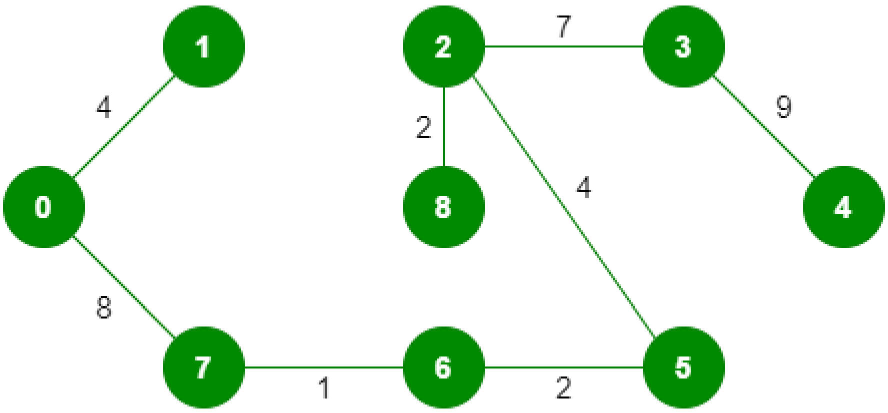
6. *Pick edge 8-6:* Since including this edge results in the cycle, discard it.
7. *Pick edge 2-3:* No cycle is formed, include it.



8. *Pick edge 7-8:* Since including this edge results in the 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 the 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.

Both **Prim's and Kruskal's algorithm** finds the Minimum Spanning Tree and follow the Greedy approach of problem-solving, but there are few **major differences between them**.

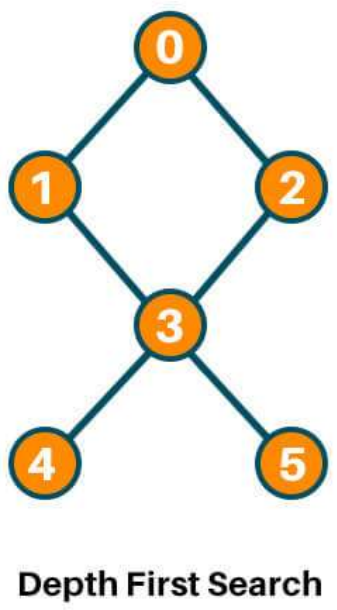
Prim's Algorithm	Kruskal's Algorithm
It starts to build the Minimum Spanning Tree from any vertex in the graph.	It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph.
It traverses one node more than one time to get the minimum distance.	It traverses one node only once.
Prim's algorithm has a time complexity of O(V^2), V being the number of	Kruskal's algorithm's time complexity is O(E log V), V being the number of vertices.

Prim’s Algorithm	Kruskal’s Algorithm
vertices and can be improved up to O(E log V) using Fibonacci heaps.	
Prim’s algorithm gives connected component as well as it works only on connected graph.	Kruskal’s algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components
Prim’s algorithm runs faster in dense graphs.	Kruskal’s algorithm runs faster in sparse graphs.
Prim’s algorithm uses List Data Structure.	Kruskal’s algorithm uses Heap Data Structure.

Depth First Search

Implementing DFS in Java | Depth First Search Algorithm

-



Graph traversals are some of the more subtle topics to learn before one can take a deep dive into the more complex algorithmic problems related to graphs. In this article, we will be having an in-depth look at DFS Algorithm and how to implement Depth-First Search in Java.

What is Depth First Search?

Graph traversal is the process by which one can travel from one node (called the source) to all other nodes of the graph. The order of nodes traced out during the process of traversal depends on the algorithm used. Graph traversal is of two main types: [Breadth-first Search] and depth-first Search.

Depth First Search (DFS) is an algorithm of graph traversal that starts exploring from a source node (generally the root node) and then explores as many nodes as possible before backtracking. Unlike breadth-first search, the exploration of nodes is very non-uniform by nature.

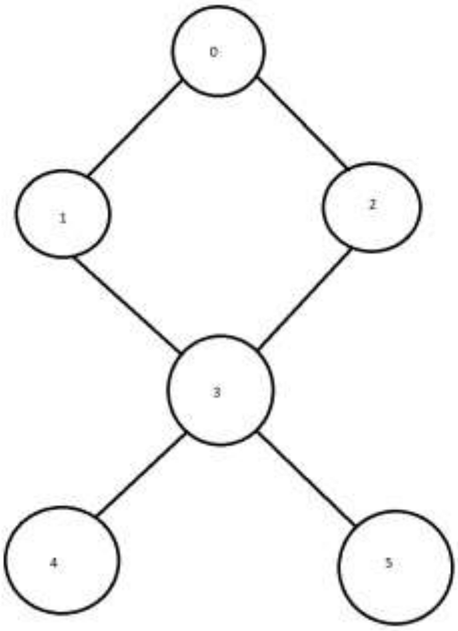
DFS Algorithm

The general process of exploring a graph using depth-first search includes the following steps:

- Take the input for the adjacency matrix or adjacency list for the graph.
- Initialize a stack.
- Push the root node (in other words, put the root node at the beginning of the stack).
- If the root node has no neighbors, stop here. Else push the leftmost neighboring node which hasn't already been explored into the stack. Continue this process till a node is encountered that has no neighbors (or whose neighbors have all been added to the stack already) – stop the process, pop the head, and then continue the process for the node that is popped.
- Keep repeating this process till the stack becomes empty.

DFS Algorithm Example

Let's work with a small example to get started. We are using the graph drawn below, starting with 0 as the root node.



o

Iteration 1: Push(0).

Stack after iteration 1 :

0

Iteration 2: Push(1).

Stack after iteration 2 :

1	0

Iteration 3: Push(3).

Stack after iteration 3 :

3	1	0

Iteration 4: Push(4).

Stack after iteration 4 :

4	3	1	0

Iteration 5: Pop(4).

Stack after iteration 5 :

3	1	0

Iteration 6: Push(5).

Stack after iteration 6 :

5	3	1	0

Iteration 7: Pop(5).

Stack after iteration 7 :

3	1	0

Iteration 8: Push(2).

Stack after iteration 8 :

2	3	1	0

Iteration 9: Pop(2).

Stack after iteration 9 :

3	1	0

Iteration 10: Pop(3).

Stack after iteration 10 :

1	0

Iteration 11: Pop(1).

Stack after iteration 11 :

0

Iteration 12: Pop(0).

Stack after iteration 11 :

One thing that should be pretty noticeable is that DFS runs comparatively longer than that of an equivalent BFS structure for the same graph. Since the stack is again empty at this point, we will stop the process.

```
iterative_depth_first_search(node)
  stack = []
  stack.append(node)
  while len(stack) != 0:
    current = stack.pop()
    if not current.visited:
      visit(current)
      current.visited = True
      for neighbor in current.neighbors:
        if not neighbor.visited:
          stack.append(neighbor)
```

Edge Cases

Now, there’s always the risk that the graph being explored has one or more cycles. This means that there’s a chance of getting back to a node that we have already explored.

How do we determine if a node has been explored or not? It’s simple – we simply maintain an array for all the nodes. The array at the beginning of the process will have all of its elements initialized to 0 (or false).

Once a node is explored, the corresponding element in the array will be set to 1 (or true). We simply push nodes to the stack if the value of their corresponding element in the array is 0 (or false). There’s still another problem to solve.

What happens if the graph given is a disconnected graph (meaning that it has multiple connected components instead of a single component)? This would mean that the results obtained would be skewed because all nodes would never be explored. The solution is to iterate through the unexplored nodes and manually use the DFS algorithm to explore each component individually.

Of course, this means that one would need to take the help of an array to mark the nodes that have already been explored up to a certain point.

Applications of Depth-First Search Algorithm

Depth First Search has a lot of utility in the real world because of the flexibility of the algorithm. These include:

- All traversal methods can be used for the detection of cycles in graphs. Cycle detection is done using DFS by checking for back edges.
- Both DFS and BFS can be used for producing the minimum spanning tree and for finding the shortest paths between all pairs of nodes (or vertices) of the graph.
- DFS can be used for topological sorting of a graph. In topological sorting, the nodes of the graph are arranged in the order in which they appear on the edges of the graph.
- DFS can be used to check if a graph is bipartite or not. A bipartite graph is such that all nodes in the graph can be divided into two sets such that the edges of the graph connect one vertex from each set.
- DFS can be used for finding the strongly connected components of a graph. Strongly connected components are such that all of the nodes in the component are connected to one another.

Implementing DFS in Java

There are multiple ways to implement DFS in Java. We will be using an adjacency list for the representation of the graph and will be covering both recursive as well as and iterative approaches for the implementation of the algorithm. The graph used for the demonstration of the code will be the same as the one used for the above example.

Recursive implementation

The recursive implementation for the DFS algorithm in Java is as follows:

```
import java.io.*;

import java.util.*;



class Graph {
```

```java
    private int V;                              //number of nodes

    private LinkedList<Integer> adj[];              //adjacency list

    public Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i = 0; i < v; ++i)
{
          adj[i] = new LinkedList();
         }

    void addEdge(int v, int w)
    {
        adj[v].add(w);                              //adding an edge to the adjacency
list (edges are bidirectional in this example)
    }



    void DFSUtil(int vertex, boolean nodes[])
    {

        nodes[vertex] = true;                          //mark the node as explored
        System.out.print(vertex + " ");
        int a = 0;

        for (int i = 0; i < adj[vertex].size(); i++)  //iterate through the linked list
and then propagate to the next few nodes
            {
                a = adj[vertex].get(i);
```

```
            if (!nodes[a])                    //only propagate to next nodes which
haven't been explored
            {
                DFSUtil(a, nodes);
            }
        }
    }

    void DFS(int v)
    {
        boolean already[] = new boolean[V];              //initialize a new
boolean array to store the details of explored nodes
        DFSUtil(v, already);
    }

    public static void main(String args[])
    {
        Graph g = new Graph(6);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 0);
        g.addEdge(1, 3);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 4);
        g.addEdge(3, 5);
        g.addEdge(4, 3);
        g.addEdge(5, 3);

        System.out.println(
```

```
            "Following is Depth First Traversal: ");

        g.DFS(0);
    }
}
```

Iterative implementation

The iterative implementation of DFS in Java follows:

```
import java.util.*;


 class Graph
    {
        int V;                              //number of nodes

        LinkedList<Integer>[] adj;                  //adjacency list

        Graph(int V)
        {
            this.V = V;
            adj = new LinkedList[V];

            for (int i = 0; i < adj.length; i++)
                adj[i] = new LinkedList<Integer>();

        }

        void addEdge(int v, int w)
        {
```

```
        adj[v].add(w);                          //adding an edge to the adjacency
list (edges are bidirectional in this example)
    }

    void DFS(int n)
    {
        boolean nodes[] = new boolean[V];

        Stack<Integer> stack = new Stack<>();

        stack.push(n);                              //push root node to the stack
        int a = 0;

        while(!stack.empty())
        {
            n = stack.peek();                     //extract the top element of the
stack
            stack.pop();                          //remove the top element from the
stack

            if(nodes[n] == false)
            {
                System.out.print(n + " ");
                nodes[n] = true;
            }

            for (int i = 0; i < adj[n].size(); i++)  //iterate through the linked
list and then propagate to the next few nodes
        {
            a = adj[n].get(i);
            if (!nodes[a])                  //only push those nodes to the stack
which aren't in it already
```

```
                {
                    stack.push(a);                    //push the top element to the
stack
                }
            }

            }
        }

    public static void main(String[] args)
    {
        Graph g = new Graph(6);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 0);
        g.addEdge(1, 3);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 4);
        g.addEdge(3, 5);
        g.addEdge(4, 3);
        g.addEdge(5, 3);

        System.out.println("Following is the Depth First Traversal");
        g.DFS(0);
    }
}
```

Output:

Following **is** Depth First Traversal:

0 1 3 4 5 2

Time and Space Complexity

The running time complexity of the DFS algorithm in Java is O(V+E) where V is the number of nodes in the graph, and E is the number of edges.

Since the algorithm requires a stack for storing the nodes that need to be traversed at any point in time, the space complexity is the maximum size of the stack at any point of time. Since this can extend to V slots for a linear graph, the maximum space complexity is O(V).

```
// Java program to print DFS
//mtraversal from a given given
// graph
import java.io.*;
import java.util.*;

// This class represents a
// directed graph using adjacency
// list representation
class Graph {
    private int V; // No. of vertices

    // Array  of lists for
    // Adjacency List Representation
    private LinkedList<Integer> adj[];

    // Constructor
    @SuppressWarnings("unchecked") Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i = 0; i < v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w); // Add w to v's list.
```

```
    }

    // A function used by DFS
    void DFSUtil(int v, boolean visited[])
    {
        // Mark the current node as visited and
print it
        visited[v] = true;
        System.out.print(v + " ");

        // Recur for all the vertices adjacent to
this
        // vertex
        Iterator<Integer> i =
adj[v].listIterator();
        while (i.hasNext())
        {
            int n = i.next();
            if (!visited[n])
                DFSUtil(n, visited);
        }
    }

    // The function to do DFS traversal.
    // It uses recursive
    // DFSUtil()
    void DFS(int v)
    {
        // Mark all the vertices as
        // not visited(set as
        // false by default in java)
        boolean visited[] = new boolean[V];

        // Call the recursive helper
        // function to print DFS
        // traversal
        DFSUtil(v, visited);
    }

    // Driver Code
    public static void main(String args[])
    {
        Graph g = new Graph(4);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
```

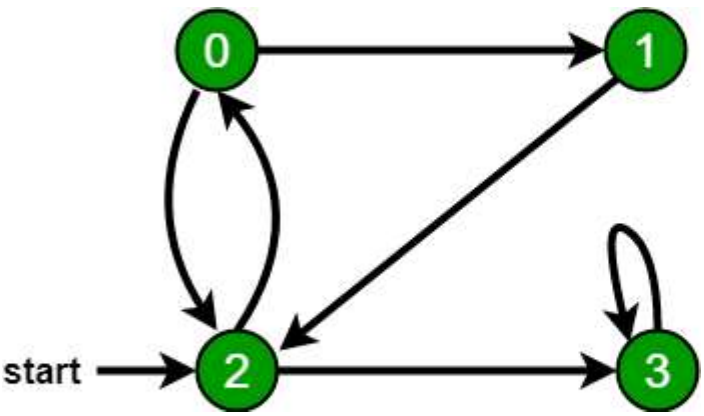
```
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        System.out.println(
            "Following is Depth First Traversal "
            + "(starting from vertex 2)");

        g.DFS(2);
    }
}
```

Breadth First Search or BFS for a Graph

Breadth-First Traversal (or Search) for a graph is similar to Breadth-First Traversal of a tree The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.
For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth-First Traversal of the following graph is 2, 0, 3, 1.



- Following is the implementations of simple Breadth-First Traversal from a given source.
  The implementation uses an adjacency list representation of graphs.

```
// Java program to print BFS traversal from a
given source vertex.
```

```java
// BFS(int s) traverses vertices reachable from
s.
import java.io.*;
import java.util.*;

// This class represents a directed graph using
adjacency list
// representation
class Graph
{
    private int V;   // No. of vertices
    private LinkedList<Integer> adj[];
//Adjacency Lists

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v,int w)
    {
        adj[v].add(w);
    }

    // prints BFS traversal from a given source s
    void BFS(int s)
    {
        // Mark all the vertices as not
visited(By default
        // set as false)
        boolean visited[] = new boolean[V];

        // Create a queue for BFS
        LinkedList<Integer> queue = new
LinkedList<Integer>();

        // Mark the current node as visited and
enqueue it
        visited[s]=true;
        queue.add(s);

        while (queue.size() != 0)
```

```
        {
            // Dequeue a vertex from queue and
print it
            s = queue.poll();
            System.out.print(s+" ");

            // Get all adjacent vertices of the
dequeued vertex s
            // If a adjacent has not been
visited, then mark it
            // visited and enqueue it
            Iterator<Integer> i =
adj[s].listIterator();
            while (i.hasNext())
            {
                int n = i.next();
                if (!visited[n])
                {
                    visited[n] = true;
                    queue.add(n);
                }
            }
        }
    }

    // Driver method to
    public static void main(String args[])
    {
        Graph g = new Graph(4);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        System.out.println("Following is Breadth
First Traversal "+
                           "(starting from vertex
2)");

        g.BFS(2);
    }
}
```

Shortest Path o Level Setting : Dijkstra's algorithm o Level Correcting: All-pairs shortest path, Floyd-Warshall algorithm

Dijkstra’s shortest path algorithm | Greedy Algo

Given a graph and a source vertex in the graph, find the shortest paths from the source to all vertices in the given graph.

Dijkstra’s algorithm is very similar to Prim’s algorithm for minimum spanning tree. Like Prim’s MST, we generate a SP*T (shortest path tree)* with a given source as a root. We maintain two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, we find a vertex that is in the other set (set of not yet included) and has a minimum distance from the source.

Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Algorithm

1) Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

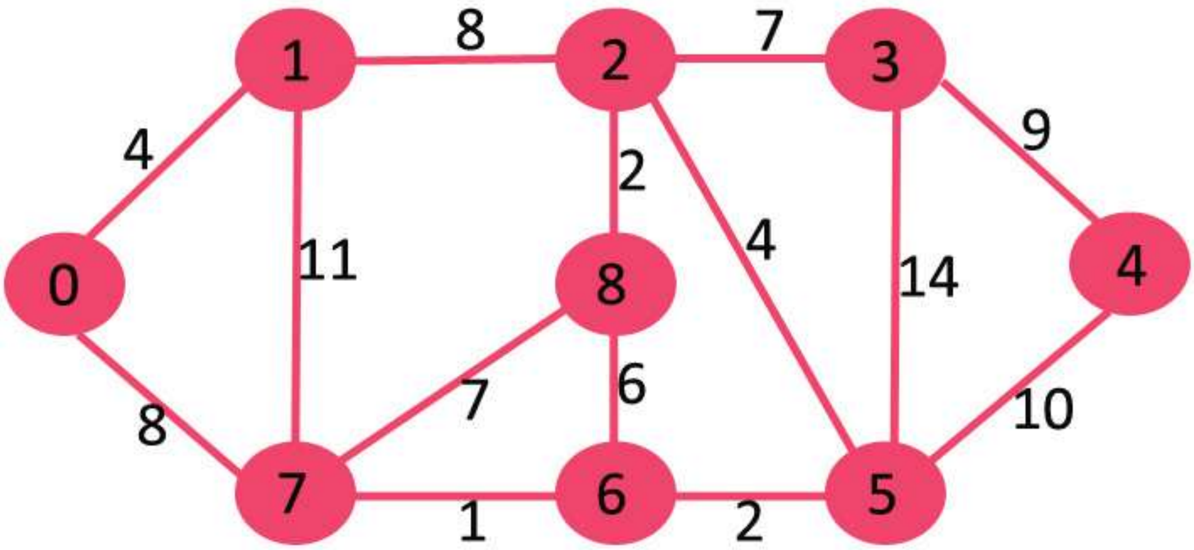
3) While *sptSet* doesn’t include all vertices

….**a)** Pick a vertex u which is not there in *sptSet* and has a minimum distance value.

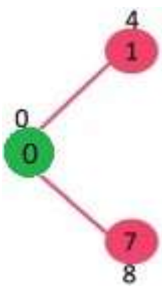
….**b)** Include u to *sptSet*.

….**c)** Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

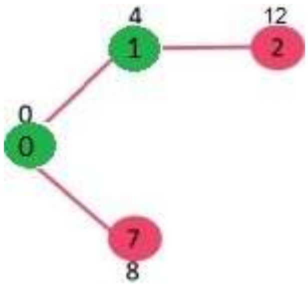
Let us understand with the following example:



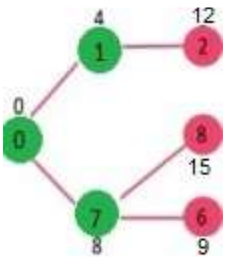
The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with a minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update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



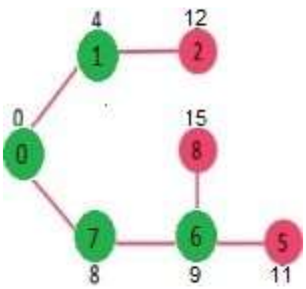
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



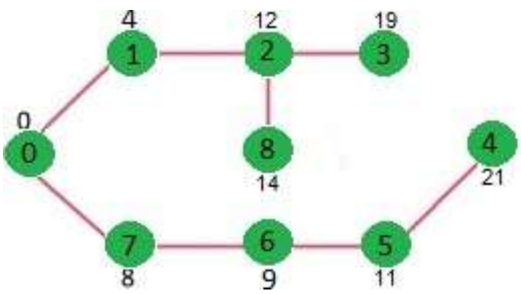
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



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



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



How to implement the above algorithm?

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

-

```
// A Java program for Dijkstra's single source shortest path algorithm.
// The program is for adjacency matrix representation of the graph
```

```java
import java.util.*;
import java.lang.*;
import java.io.*;

class ShortestPath {
    // A utility function to find the vertex with minimum distance
value,
    // from the set of vertices not yet included in shortest path tree
    static final int V = 9;
    int minDistance(int dist[], Boolean sptSet[])
    {
        // Initialize min value
        int min = Integer.MAX_VALUE, min_index = -1;

        for (int v = 0; v < V; v++)
            if (sptSet[v] == false && dist[v] <= min) {
                min = dist[v];
                min_index = v;
            }

        return min_index;
    }

    // A utility function to print the constructed distance array
    void printSolution(int dist[])
    {
        System.out.println("Vertex \t\t Distance from Source");
        for (int i = 0; i < V; i++)
            System.out.println(i + " \t\t " + dist[i]);
    }

    // Function that implements Dijkstra's single source shortest path
    // algorithm for a graph represented using adjacency matrix
    // representation
    void dijkstra(int graph[][], int src)
    {
        int dist[] = new int[V]; // The output array. dist[i] will hold
        // the shortest distance from src to i

        // sptSet[i] will true if vertex i is included in shortest
        // path tree or shortest distance from src to i is finalized
        Boolean sptSet[] = new Boolean[V];

        // Initialize all distances as INFINITE and stpSet[] as false
        for (int i = 0; i < V; i++) {
            dist[i] = Integer.MAX_VALUE;
            sptSet[i] = false;
```

```
        }

        // Distance of source vertex from itself is always 0
        dist[src] = 0;

        // Find shortest path for all vertices
        for (int count = 0; count < V - 1; count++) {
            // Pick the minimum distance vertex from the set of vertices
            // not yet processed. u is always equal to src in first
            // iteration.
            int u = minDistance(dist, sptSet);

            // Mark the picked vertex as processed
            sptSet[u] = true;

            // Update dist value of the adjacent vertices of the
            // picked vertex.
            for (int v = 0; v < V; v++)

                // Update dist[v] only if is not in sptSet, there is an
                // edge from u to v, and total weight of path from src
to
                // v through u is smaller than current value of dist[v]
                if (!sptSet[v] && graph[u][v] != 0 && dist[u] !=
Integer.MAX_VALUE && dist[u] + graph[u][v] < dist[v])
                    dist[v] = dist[u] + graph[u][v];
        }

        // print the constructed distance array
        printSolution(dist);
    }

    // Driver method
    public static void main(String[] args)
    {
        /* Let us create the example graph discussed above */
        int graph[][] = new int[][] { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                                      { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                                      { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                                      { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                                      { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                                      { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                                      { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                                      { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                                      { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };
        ShortestPath t = new ShortestPath();
        t.dijkstra(graph, 0);
```

```
    }
}
```

Floyd Warshall Algorithm | DP-16

The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

Example:

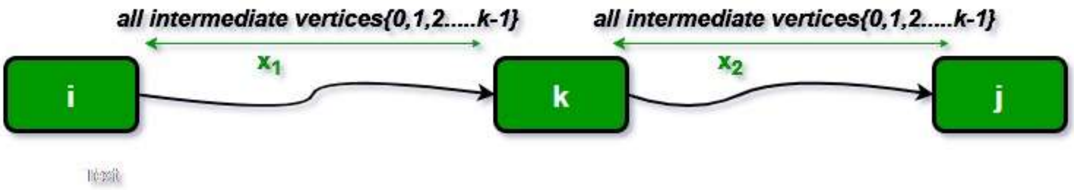
```
Input:
       graph[][] = { {0,   5,  INF, 10},
                    {INF,  0,   3,  INF},
                    {INF, INF, 0,   1},
                    {INF, INF, INF, 0} }
which represents the following graph
             10
       (1)------->(4)
        |         /|\
      5 |          |
        |          | 1
       \|/         |
       (2)------->(3)
            3
Note that the value of graph[i][j] is 0 if i is equal to j
And graph[i][j] is INF (infinite) if there is no edge from vertex
i to j.

Output:
Shortest distance matrix
      0      5      8      9
    INF      0      3      4
    INF    INF      0      1
    INF    INF    INF      0
```

Floyd Warshall Algorithm

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.

1) k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.
2) k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] > dist[i][k] + dist[k][j]
The following figure shows the above optimal substructure property in the all-pairs shortest path problem.



•

```
// A Java program for Floyd Warshall All Pairs Shortest
// Path algorithm.
import java.util.*;
import java.lang.*;
import java.io.*;


class AllPairShortestPath
{
    final static int INF = 99999, V = 4;

    void floydWarshall(int graph[][])
    {
        int dist[][] = new int[V][V];
        int i, j, k;

        /* Initialize the solution matrix
           same as input graph matrix.
           Or we can say the initial values
           of shortest distances
           are based on shortest paths
           considering no intermediate
           vertex. */
        for (i = 0; i < V; i++)
            for (j = 0; j < V; j++)
                dist[i][j] = graph[i][j];
```

```
        /* Add all vertices one by one
           to the set of intermediate
           vertices.
          ---> Before start of an iteration,
               we have shortest
               distances between all pairs
               of vertices such that
               the shortest distances consider
               only the vertices in
               set {0, 1, 2, .. k-1} as
               intermediate vertices.
          ----> After the end of an iteration,
                vertex no. k is added
                to the set of intermediate
                vertices and the set
                becomes {0, 1, 2, .. k} */
        for (k = 0; k < V; k++)
        {
            // Pick all vertices as source one by one
            for (i = 0; i < V; i++)
            {
                // Pick all vertices as destination for
the
                // above picked source
                for (j = 0; j < V; j++)
                {
                    // If vertex k is on the shortest
path from
                    // i to j, then update the value of
dist[i][j]
                    if (dist[i][k] + dist[k][j] <
dist[i][j])
                        dist[i][j] = dist[i][k] +
dist[k][j];
                }
            }
        }

        // Print the shortest distance matrix
        printSolution(dist);
    }

    void printSolution(int dist[][])
    {
        System.out.println("The following matrix shows
the shortest "+
```

```
                              "distances between every pair
of vertices");
        for (int i=0; i<V; ++i)
        {
            for (int j=0; j<V; ++j)
            {
                if (dist[i][j]==INF)
                    System.out.print("INF ");
                else
                    System.out.print(dist[i][j]+"   ");
            }
            System.out.println();
        }
    }

    // Driver program to test above function
    public static void main (String[] args)
    {
        /* Let us create the following weighted graph
           10
        (0)------->(3)
        |         /|\
        5 |          |
        |          | 1
        \|/         |
        (1)------->(2)
           3            */
        int graph[][] = { {0,   5,  INF, 10},
                          {INF, 0,   3, INF},
                          {INF, INF, 0,   1},
                          {INF, INF, INF, 0}
                        };
        AllPairShortestPath a = new
AllPairShortestPath();

        // Print the solution
        a.floydWarshall(graph);
    }
}
```

DFS---- To check whether given vertex is reachable from the source

Application

1. Computer network if you want to check whether the message is reachable from one machine to another
2. whether 2 cities are connected by road
3. Whether water is reachable from source to some destination

```
DFS(source):
 s <- new stack
 visited <- {} // empty set
 s.push(source)
 while (s is not empty):
  current <- s.pop()
  if (current is in visited):
    continue
  visited.add(current)
  // do something with current
  for each node v such that (current,v) is an edge:
    s.push(v)
```

Example

```
    1
   / | \
  4  |  2
    3 /
```

```
Actions         Stack          Visited
=======         =====          =======
push 1          [1]            {}
pop and visit 1 []             {1}
 push 2, 4, 3   [2, 4, 3]      {1}
pop and visit 3 [2, 4]         {1, 3}
 push 1, 2      [2, 4, 1, 2]   {1, 3}
pop and visit 2 [2, 4, 1]      {1, 3, 2}
 push 1, 3      [2, 4, 1, 1, 3] {1, 3, 2}
pop 3 (visited) [2, 4, 1, 1]   {1, 3, 2}
pop 1 (visited) [2, 4, 1]      {1, 3, 2}
```

pop 1 (visited)	[2, 4]	{1, 3, 2}
pop and visit 4	[2]	{1, 3, 2, 4}
push 1	[2, 1]	{1, 3, 2, 4}
pop 1 (visited)	[2]	{1, 3, 2, 4}
pop 2 (visited)	[]	{1, 3, 2, 4}

Applications

Depth First Search has a lot of utility in the real world because of the flexibility of the algorithm. These include:-

- All traversal methods can be used for the detection of cycles in graphs. Cycle detection is done using DFS by checking for back edges.
- Both DFS and BFS can be used for producing the minimum spanning tree and for finding the shortest paths between all pairs of nodes (or vertices) of the graph.
- DFS can be used for topological sorting of a graph. In topological sorting, the nodes of the graph are arranged in the order in which they appear on the edges of the graph.
- DFS can be used to check if a graph is bipartite or not. A bipartite graph is such that all nodes in the graph can be divided into two sets such that the edges of the graph connect one vertex from each set.
- DFS can be used for finding the strongly connected components of a graph. Strongly connected components are such that all of the nodes in the component are connected to one another.

BFS algorithm

BFS Algorithm

The general process of exploring a graph using breadth-first search includes the following steps:-

- Take the input for the adjacency matrix or adjacency list for the graph.
- Initialize a queue.

- Enqueue the root node (in other words, put the root node into the beginning of the queue).
- Dequeue the head (or first element) of the queue, then enqueue all of its neighboring nodes, starting from left to right. If a node has no neighboring nodes which need to be explored, simply dequeue the head and continue the process. (Note: If a neighbor which is already explored or in the queue appears, don't enqueue it – simply skip it.)
- Keep repeating this process till the queue is empty.

Applications

Breadth-First Search has a lot of utility in the real world because of the flexibility of the algorithm. These include:-

- Discovery of peer nodes in a peer-to-peer network. Most torrent clients like BitTorrent, uTorrent, qBittorent use this mechanism for discovering something called "seeds" and "peers" in the network.
- Web crawling uses graph traversal techniques for building the index. The process considers the source page as the root node and starts traversing from there to all secondary pages linked with the source page (and this process continues). Breadth-First Search has an innate advantage here because of the reduced depth of the recursion tree.
- GPS navigation systems use Breadth-First Search for finding neighboring locations using the GPS.
- Garbage collection is done with Cheney's algorithm which utilizes the concept of breadth-first search.

S.NO	BFS	DFS
1.	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
2.	BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
3.	BFS can be used to find single source shortest path in an unweighted graph, because in BFS, we reach a vertex with	In DFS, we might traverse through more edges to reach a destination vertex from a source.

	minimum number of edges from a source vertex.	
3.	BFS is more suitable for searching vertices which are closer to the given source.	DFS is more suitable when there are solutions away from source.
4.	BFS considers all neighbors first and therefore not suitable for decision making trees used in games or puzzles.	DFS is more suitable for game or puzzle problems. We make a decision, then explore all paths through this decision. And if this decision leads to win situation, we stop.
5.	The Time complexity of BFS is O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.	The Time complexity of DFS is also O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.
6.	Here, siblings are visited before the children	Here, children are visited before the siblings

Types
ofalgorithm

Divide and Conquer Algorithm

example merge sort, quick sort

o Greedy algorithm

Will always select the best possible solution to lead towards the answer

example Kruskal’s algorithm, Dijekstras algorithm

o Dynamic Programming algorithm

At every stage we need to take the decision to get the best solution

prim’s algorithm, Floyed warshall

o Brute force algorithm

Brute Force algorithm is a **typical problem-solving technique where the possible solution for a problem is uncovered by checking each answer one by one**, by determining whether the result satisfies the statement of a problem or not

o Backtracking algorithms

DFS, 8 Queen problem

o Branch-and-bound algorithms

Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. The Branch and Bound Algorithm technique solves these problems relatively quickly. Let us consider the **0/1 Knapsack problem** to understand Branch and Bound.
used to find all possible solutions

o Stochastic algorithm

A stochastic program is **an optimization problem in which some or all problem parameters are uncertain, but follow known probability distributions**.

image detection Algorithm

Time complexity

Time complexity is **the amount of time taken by an algorithm to run**, as a function of the length of the input. It measures the time taken to execute each statement of code in an algorithm.

Space Complexity:
The term Space Complexity is misused for Auxiliary Space at many places. Following are the correct definitions of Auxiliary Space and Space Complexity.
Auxiliary Space is the extra space or temporary space used by an algorithm.

Space Complexity of an algorithm is the total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

Sorting Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	**$\Omega(N)$**	**$\Theta(N^2)$**	**$O(N^2)$**	**$O(1)$**
Selection Sort	**$\Omega(N^2)$**	**$\Theta(N^2)$**	**$O(N^2)$**	**$O(1)$**
Insertion Sort	**$\Omega(N)$**	**$\Theta(N^2)$**	**$O(N^2)$**	**$O(1)$**

Sorting Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Merge Sort	**Ω(N log N)**	**Θ(N log N)**	**O(N log N)**	**O(N)**
Heap Sort	**Ω(N log N)**	**Θ(N log N)**	**O(N log N)**	**O(1)**
Quick Sort	**Ω(N log N)**	**Θ(N log N)**	**$O(N^2)$**	**O(log N)**