Back To Course (/batchPage.php?batchId=154)

LIVE BATCHES

Learn

Classroom

Theory

Quiz

Overview Learn Problems Quiz Contest

Classroom Theory

Introduction to Graphs

A ***Graph*** is a data structure that consists of the following two components:

1. A finite set of vertices also called nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(digraph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

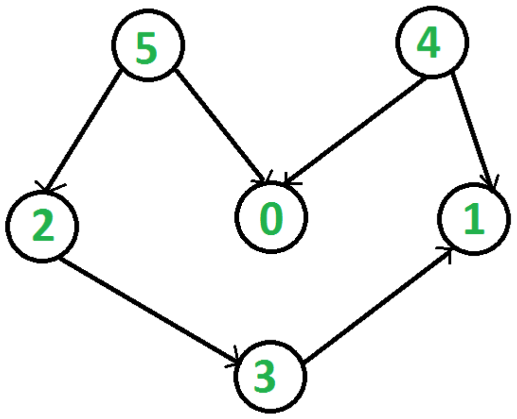
Graphs are used to represent many real-life applications:

- Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. For example Google GPS
- Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender and locale.

Directed and Undirected Graphs

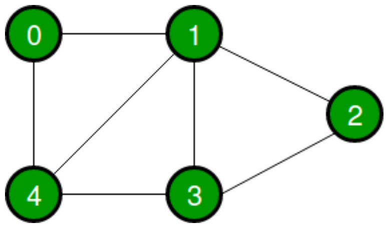
- **Directed Graphs**: The Directed graphs are such graphs in which edges are directed in a single direction.

  For Example, the below graph is a directed graph:



- **Undirected Graphs**: Undirected graphs are such graphs in which the edges are directionless or in other words bi-directional. That is, if there is an edge between vertices **u** and **v** then it means we can use the edge to go from both **u to v** and **v to u**.

  Following is an example of an undirected graph with 5 vertices:



Representing Graphs

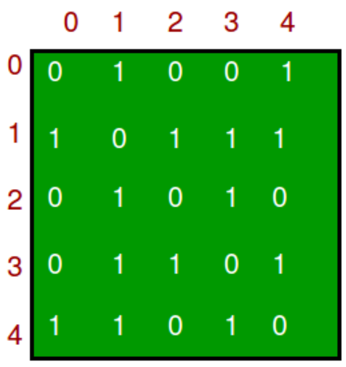
Following two are the most commonly used representations of a graph:

1. Adjacency Matrix.
2. Adjacency List.

Let us look at each one of the above two method in details:

- **Adjacency Matrix:** The Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

  The adjacency matrix for the above example undirected graph is:

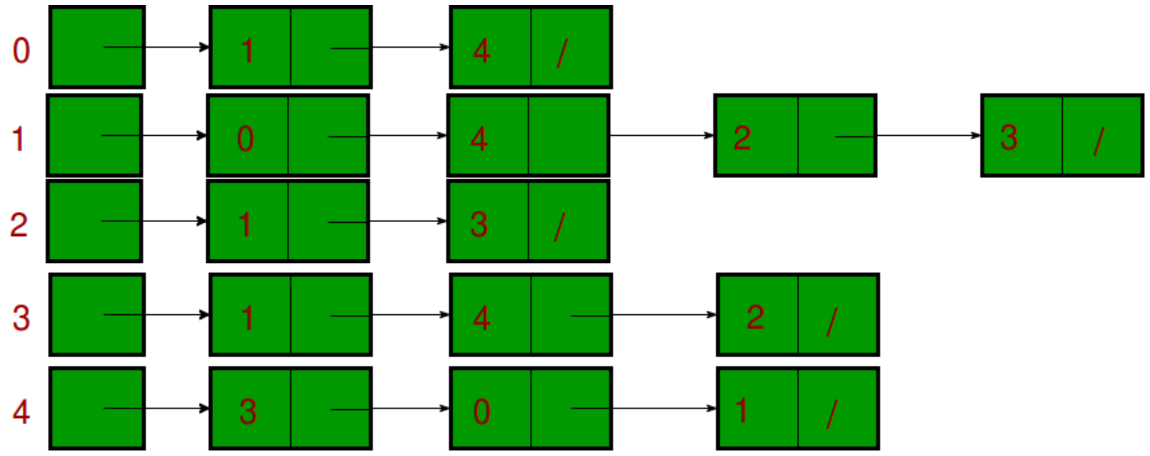
  

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

  Pros: Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done O(1).

  Cons: Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.
  Please see this (https://ide.geeksforgeeks.org/9je5j6jJ13) for a sample Python implementation of adjacency matrix.

- **Adjacency List:** Graph can also be implemented using an array of lists. That is every index of the array will contain a complete list. Size of the array is equal to the number of vertices and every index **i** in the array will store the list of vertices connected to the vertex numbered *i*. Let the array be array[]. An entry array[i] represents the list of vertices adjacent to the *i*th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above example undirected graph.



LIVE BATCHES

Below is the implementation of the adjacency list representation of Graphs:

Note: In below implementation, we use dynamic arrays (vector in C++/ArrayList in Java) to represent adjacency lists instead of a linked list. The vector implementation has advantages of cache friendliness.

C++

```

// A simple representation of graph using STL
#include<bits/stdc++.h>
using namespace std;

// A utility function to add an edge in an
// undirected graph.
void addEdge(vector<int> adj[], int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}

// A utility function to print the adjacency list
// representation of graph
void printGraph(vector<int> adj[], int V)
{
    for (int v = 0; v < V; ++v)
    {
        cout << "\n Adjacency list of vertex "
             << v << "\n head ";
        for (auto x : adj[v])
           cout << "-> " << x;
        printf("\n");
    }
}

// Driver code
int main()
{
```

Run

Java

Output:

```
Adjacency list of vertex 0
head -> 1-> 4

Adjacency list of vertex 1
head -> 0-> 2-> 3-> 4

Adjacency list of vertex 2
head -> 1-> 3

Adjacency list of vertex 3
head -> 1-> 2-> 4

Adjacency list of vertex 4
head -> 0-> 1-> 3
```

Pros: Saves space O(|V|+|E|). In the worst case, there can be C(V, 2) number of edges in a graph thus consuming O(V^2) space. Adding a vertex is easier.

Cons: Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done O(V).

- Breadth First Traversal of a Graph

The ***Breadth First Traversal*** or ***BFS*** traversal of a graph is similar to that of the Level Order Traversal of Trees.

The BFS traversal of Graphs also traverses the graph in levels. It starts the traversal with a given vertex, visits all of the vertices adjacent to the initially given vertex and pushes them all to a queue in order of visiting. Then it pops an element from the front of the queue, visits all of its neighbours and pushes the neighbours which are not already visited into the queue and repeats the process until the queue is empty or all of the vertices are visited.

The BFS traversal uses an auxiliary boolean array say *visited[]* which keeps track of the visited vertices. That is if **visited[i] = true** then it means that the **i-th** vertex is already visited.

Complete Algorithm:

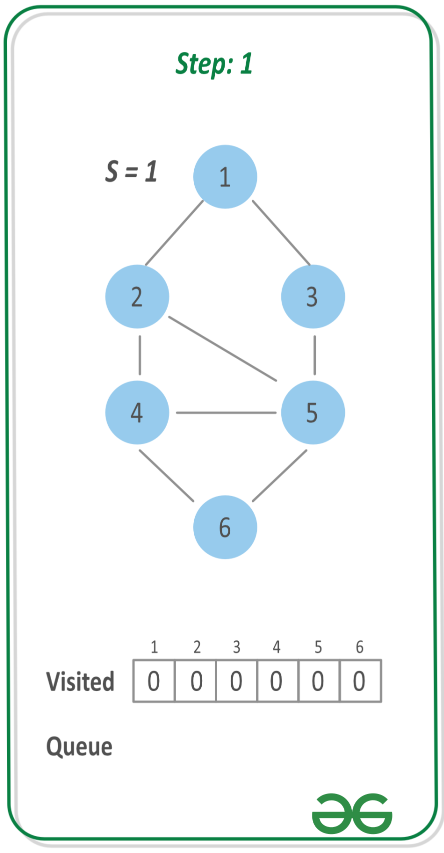
1. Create a boolean array say ***visited[]*** of size **V+1** where *V* is the number of vertices in the graph.
2. Create a Queue, mark the source vertex visited as **visited[s] = true** and push it into the queue.
3. Until the Queue is non-empty, repeat the below steps:
    - Pop an element from the queue and print the popped element.
    - Traverse all of the vertices adjacent to the vertex poped from the queue.
    - If any of the adjacent vertex is not already visited, mark it visited and push it to the queue.

Illustration:

Consider the graph shown in the below Image. The vertices marked **blue** are *not-visited* vertices and the vertices marked **yellow** are *visited*. The vertex numbered **1** is the source vertex, i.e. the BFS traversal will start from the vertex 1.

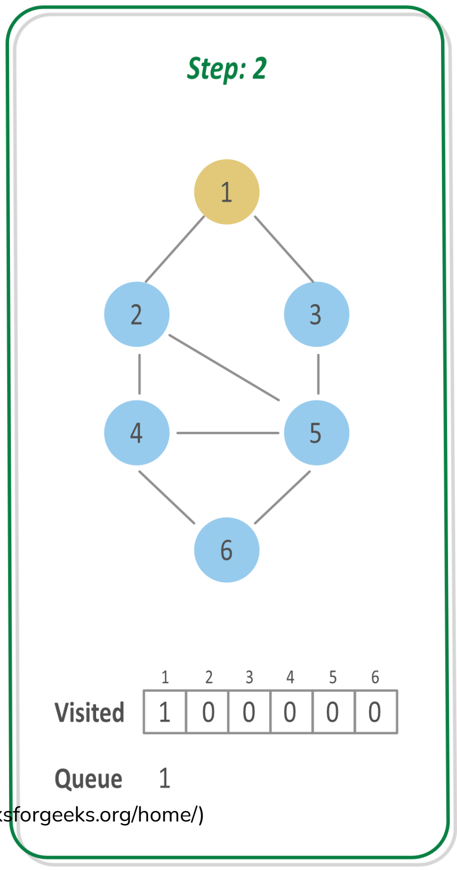
Following the BFS algorithm:

- Mark the vertex 1 visited in the visited[] array and push it to the queue.



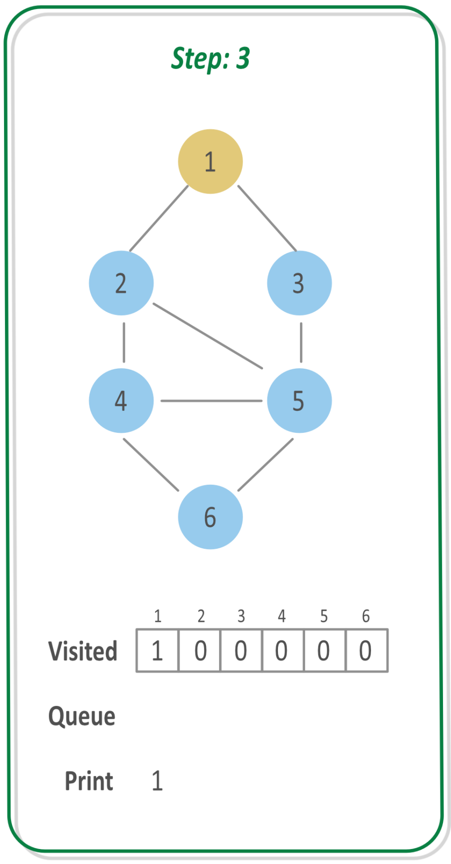
	1	2	3	4	5	6
Visited	0	0	0	0	0	0
Queue						

(https://practice.geeksforgeeks.org/home/)



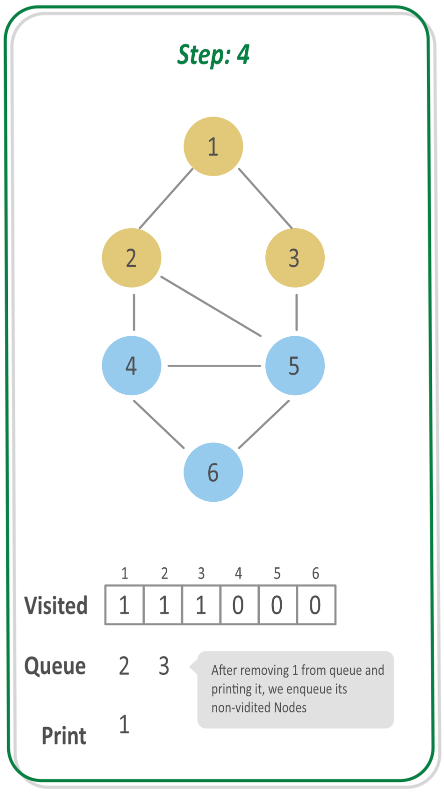
	1	2	3	4	5	6
Visited	1	0	0	0	0	0
Queue	1					

Step 3: POP the vertex at the front of queue that is 1, and print it.



	1	2	3	4	5	6
Visited	1	0	0	0	0	0
Queue						
Print	1					

Step 4: Check if adjacent vertices of the vertex 1 are not already visited. If not, mark them visited and push them back to the queue.


Step: 4
1
2
3
4
5
6
Visited
Queue
Print
After removing 1 from queue and printing it, we enqueue its non-vidited Nodes

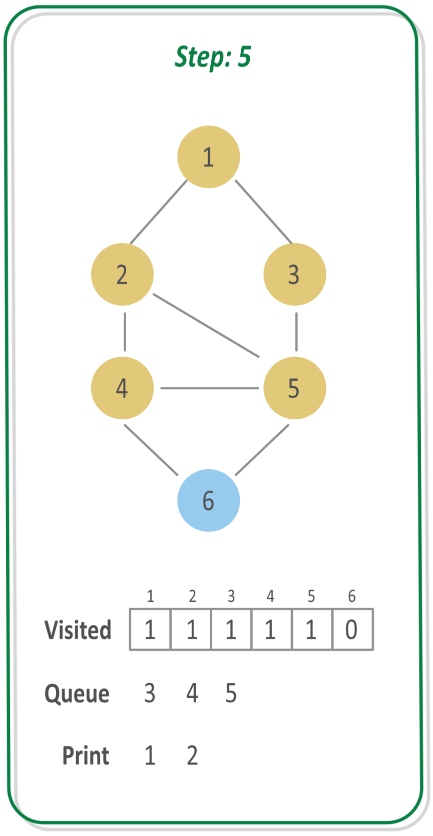
	1	2	3	4	5	6
Visited	1	1	1	0	0	0
Queue	2	3				
Print	1					

Step 5:

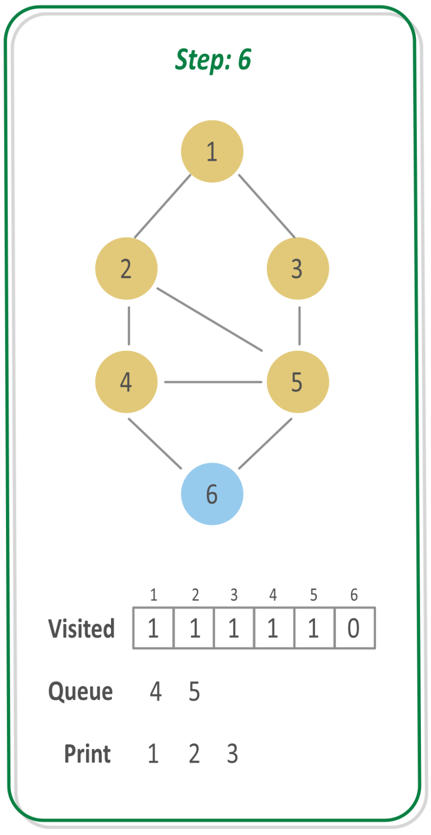
- POP the vertex at front that is 2 and print it.
- Check if the adjacent vertices of 2 are not already visited. If not, mark them visited and push them to queue. So, push 4 and 5.

Step 6:

- POP the vertex at front that is 3 and print it.
- Check if the adjacent vertices of 3 are not already visited. If not, mark them visited and push them to queue. So, donot push

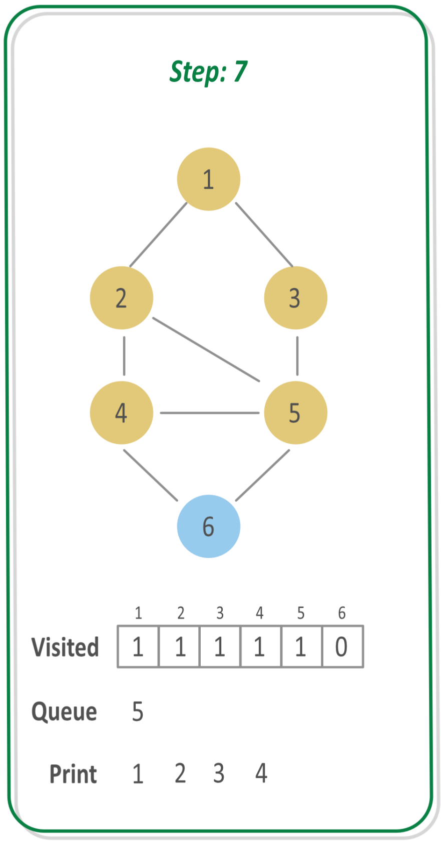


anything.



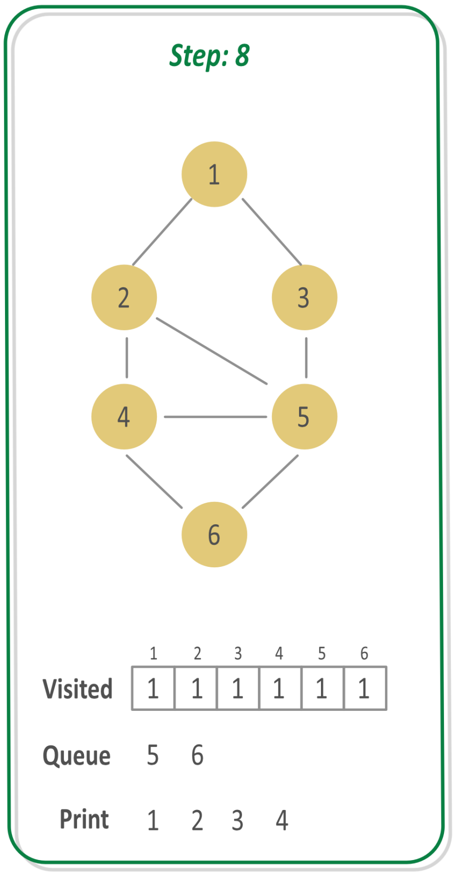
Step 7:

- POP the vertex at front that is 4 and print it.



Step 8:

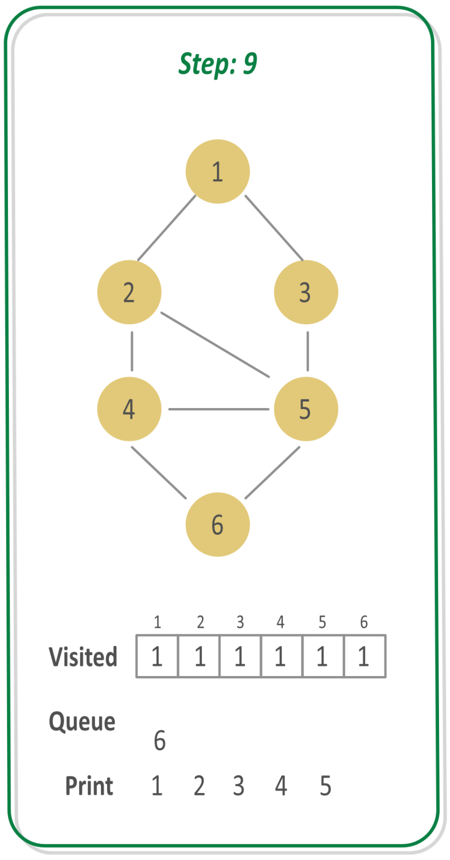
- Check if the adjacent vertices of 4 are not already visited. If not, mark them visited and push them to queue. So, push 6 to the queue.



Since the Queue is empty now, it means that the complete graph is traversed.

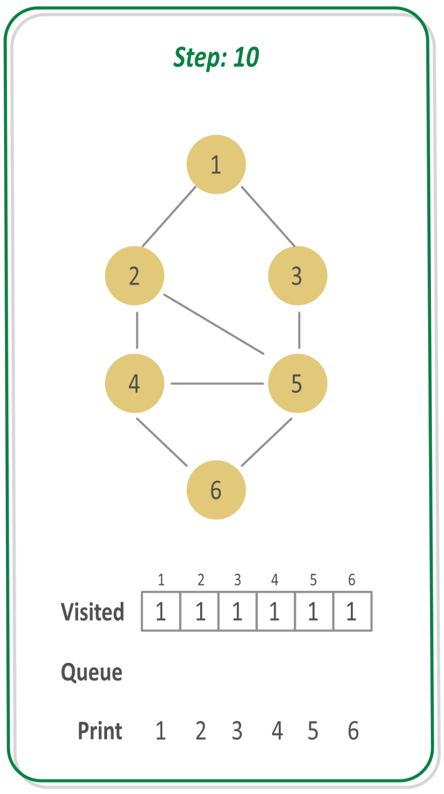
Step 9:

- POP the vertex at front, that is 5 and print it.
- Since, all of its adjacent vertices are already visited, donot push anything.



Step 10:

- POP the vertex at front, that is 6 and print it.
- Since, all of its adjacent vertices are already visited, donot push anything.



Implementation:

C++

```

// C++ program to implement BFS traversal
// of a Graph

#include <bits/stdc++.h>
using namespace std;

// A utility function to add an edge in an
// undirected graph.
void addEdge(vector<int> adj[], int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}

// Function to perform BFS traversal of the given Graph
void BFS(vector<int> adj[], int V)
{
    // Initialize a boolean array
    // to keep track of visited vertices
    bool visited[V + 1];

    // Mark all vertices not-visited initially
    for (int i = 1; i <= V; i++)
        visited[i] = false;

    // Create a Queue to perform BFS
    queue<int> q;

    // Our source vertex is vertex
```

Run

Java

Output:

```
1 2 3 4 5 6
```

LIVE BATCHES

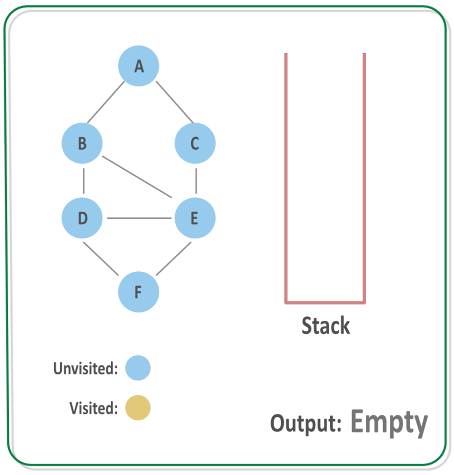
- Depth First Traversal of a Graph

The Depth-First Traversal or the DFS traversal of a Graph is used to traverse a graph depth wise. That is, it in this traversal method, we start traversing the graph from a node and keep on going in the same direction as far as possible. When no nodes are left to be traversed along the current path, backtrack to find a new possible path and repeat this process until all of the nodes are visited.

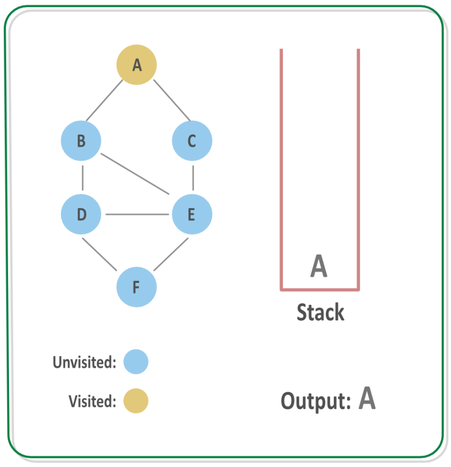
We can implement the DFS traversal algorithm using a recursive approach. While performing the DFS traversal the graph may contain a cycle and the same node can be visited again, so in order to avoid this we can keep track of visited array using an auxiliary array. On each step of the recursion mark, the current vertex visited and call the recursive function again for all the adjacent vertices.

Illustration:

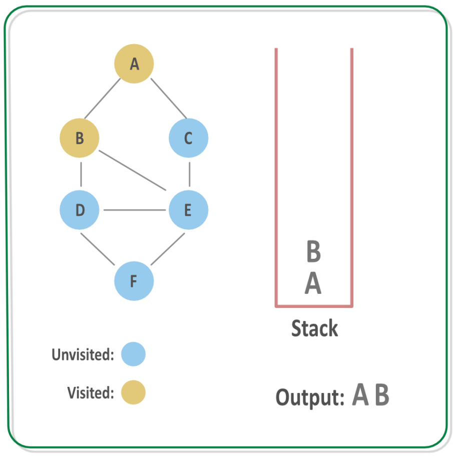
Step 1: Consider the below graph and apply the DFS algorithm recursively for every node using an auxiliary stack for recursive calls and an array to keep track of visited vertices.



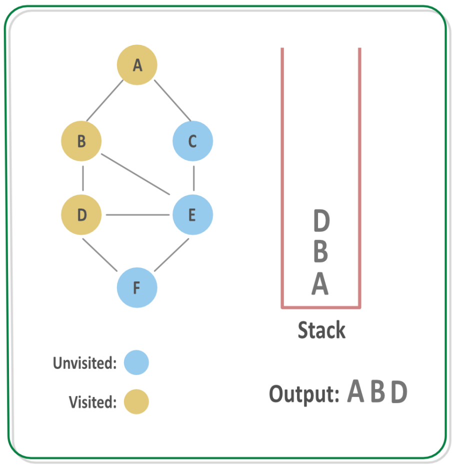
Step 2: Process the vertex A, mark it visited and call DFS for its adjacent vertices. Suppose the first adjacent vertex processed is B. The vertex A is put on the auxilary stack for now.



Step 3: Process the vertex B, mark it visited and call DFS for its adjacent vertices. Suppose the first adjacent vertex processed is D. The vertex B is put on the auxilary stack for now.

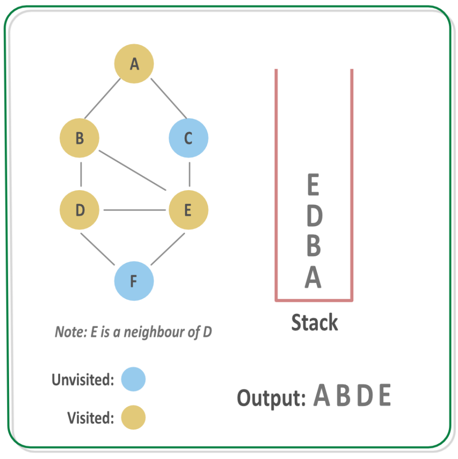


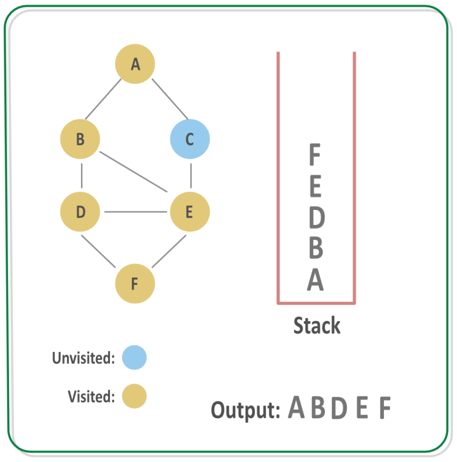
Step 4: Process the vertex D, mark it visited and call DFS for its adjacent vertices. Suppose the first adjacent vertex processed is E. The vertex D is put on the auxilary stack for now.



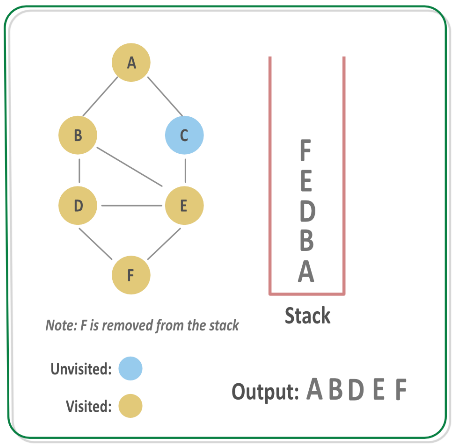
Step 5: Process the vertex E, mark it visited and call DFS for its adjacent vertices. Suppose the first adjacent vertex processed is F. The vertex E is put on the auxilary stack
for now.

Step 6: Process the vertex F, mark it visited and call DFS for its adjacent vertices. There are no adjacent vertex of the vertex F, so backtrack to find a new path. The vertex F is put on the auxilary stack for now.

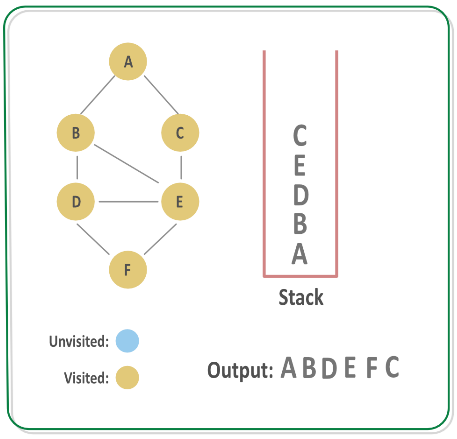




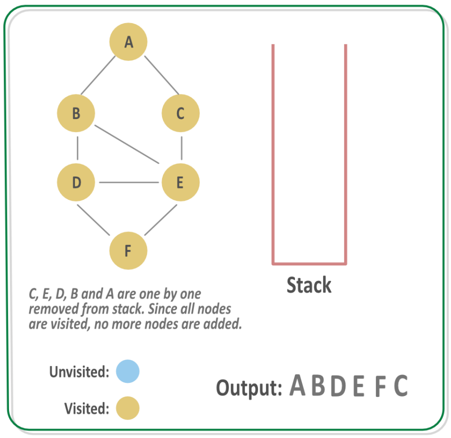
Step 7: Since the vertex F has no adjacent vertices left unvisited, backtrack and go to previous call, that is process any other adjacent vertex of node E, that is C.



Step 8: Process the vertex C, mark it visited and call DFS for its adjacent vertices. The vertex C is put on the auxilary stack for now.



Step 9: Since there are no adjacent vertex of C, backtrack to find a new path and keep removing nodes from stack until a new path is found. Since all of the nodes are processed so the stack will get empty.


A
B
C
D
E
F
Stack
C, E, D, B and A are one by one removed from stack. Since all nodes are visited, no more nodes are added.
Unvisited:
Visited:
Output: A B D E F C

Implementation:

C++

```

// C++ program to print DFS traversal from
// a given vertex in a  given graph
#include<iostream>
#include<list>
using namespace std;

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

    // Pointer to an array containing
    // adjacency lists
    list<int> *adj;

    // A recursive function used by DFS
    void DFSUtil(int v, bool visited[]);
public:
    Graph(int V);   // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // DFS traversal of the vertices
    // reachable from v
    void DFS(int v);
};

```

Run

Java

Output:

```
Following is Depth First Traversal (starting from vertex 2)
2 0 1 3
```

+ Detecting Cycle in a Graph

- Dijkstra's Algorithm for Shortest Path in a Weighted Graph

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

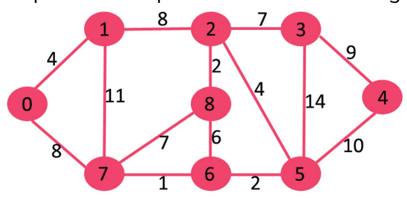
Dijkstra's algorithm is a variation of the BFS algorithm. In Dijkstra's Algorithm, a SPT *(shortest path tree)* is generated with given source as root. Each node at this SPT stores the value of the shortest path from the source vertex to the current vertex. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

Below is 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 weighted graph.

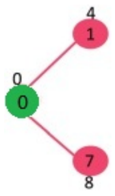
Algorithm:

1. Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from 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:
    - Pick a vertex u which is not there in *sptSet* and has minimum distance value.
    - Include u to *sptSet.*
    - 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 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.

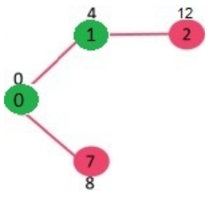
Let us understand the above algorithm with the help of an example. Consider the below given graph:



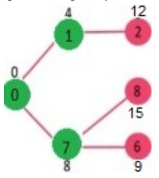
The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update 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. 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.



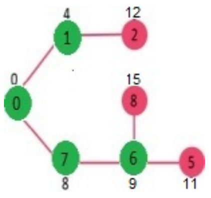
(https://media.geeksforgeeks.org/wp-content/cdn-uploads/MST1.jpg)
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



(https://media.geeksforgeeks.org/wp-content/cdn-uploads/DIJ2.jpg)
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).



(https://media.geeksforgeeks.org/wp-content/cdn-uploads/DIJ3.jpg)
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.



(https://media.geeksforgeeks.org/wp-content/cdn-uploads/DIJ4.jpg)
We repeat the above steps until *sptSet* doesn't include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).

Implementation:
Since at every step we need to find the vertex with minimum distance from the source vertex from the set of vertices currently not added to the SPT, so we can use a min heap for easier and efficient implementation. Below is the complete algorithm using priority_queue(min heap) to implement Dijkstra's Algorithm:

```
1) Initialize distances of all vertices as infinite.

2) Create an empty priority_queue pq. Every item
   of pq is a pair (weight, vertex). Weight (or
   distance) is used as the first item of pair
   as the first item is by default used to compare
   two pairs

3) Insert source vertex into pq and make its
   distance as 0.

4) While either pq doesn't become empty
    a) Extract minimum distance vertex from pq.
       Let the extracted vertex be u.
    b) Loop through all adjacent of u and do
       following for every vertex v.

           // If there is a shorter path to v
           // through u.
           If dist[v] > dist[u] + weight(u, v)

               (i) Update distance of v, i.e., do
                     dist[v] = dist[u] + weight(u, v)
               (ii) Insert v into the pq (Even if v is
                    already there)

5) Print distance array dist[] to print all shortest
   paths.
```

C++

```cpp
        // vertex label is stored in second of pair (it
        // has to be done this way to keep the vertices
        // sorted distance (distance must be first item
        // in pair)
        int u = pq.top().second;
        pq.pop();

        // Get all adjacent of u.
        for (auto x : adj[u])
        {
            // Get vertex label and weight of current adjacent
            // of u.
            int v = x.first;
            int weight = x.second;

            // If there is shorted path to v through u.
            if (dist[v] > dist[u] + weight)
            {
                // Updating distance of v
                dist[v] = dist[u] + weight;
                pq.push(make_pair(dist[v], v));
            }
        }
    }

    // Print shortest distances stored in dist[]
    printf("Vertex Distance from Source\n");
    for (int i = 0; i < V; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}
```

Run

Java

Output:

```
Vertex   Distance from Source

0              0

1              4

2              12

3              19

4              21

5              11

6              9

7              8

8              14
```

Time Complexity: The time complexity of the Dijkstra's Algorithm when implemented using a min heap is O(E * logV), where E is the number of Edges and V is the number of vertices.

Note: The Dijkstra's Algorithm **doesn't work** in the case when the Graph has negative edge weight.

+ Bellman-Ford Algorithm for Shortest Path

+ Number of Strongly Connected Components in an Undirected Graph

- Prim's Minimum Spanning Tree Algorithm

What is Minimum Spanning Tree?

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

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

Prim's Algorithm

Prim's algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

A group of edges that connects two set of vertices in a graph is called cut in graph theory (http://en.wikipedia.org/wiki/Cut_%28graph_theory%29). *So, at every step of Prim's algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the vertices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).*
How does Prim's Algorithm Work? The idea behind Prim's algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a *Spanning* Tree. And they must be connected with the minimum weight edge to make it a *Minimum* Spanning Tree.

Algorithm:

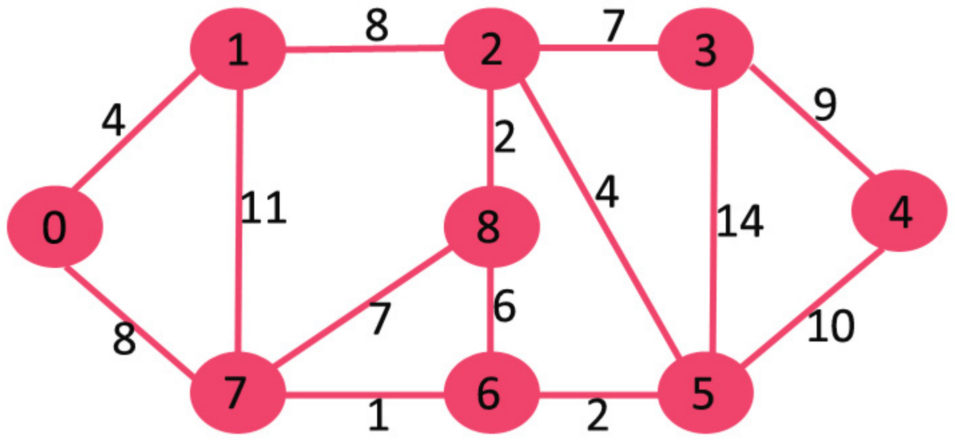
1. Create a set *mstSet* that keeps track of vertices already included in MST.
2. Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.

3. While mstSet doesn't include all vertices:
    - Pick a vertex *u* which is not there in *mstSet* and has minimum key value.
    - Include *u* to mstSet.

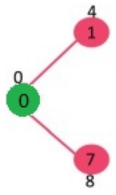
- Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u-v* is less than the previous key value of *v*, update the key value as weight of *u-v*.

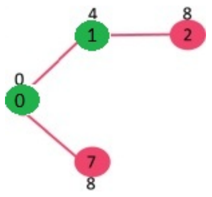
The idea of using key values is to pick the minimum weight edge from cut (http://en.wikipedia.org/wiki/Cut_(graph_theory)). The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

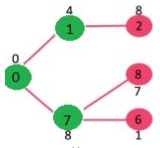
Let us understand this with the help of following example:

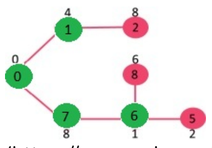


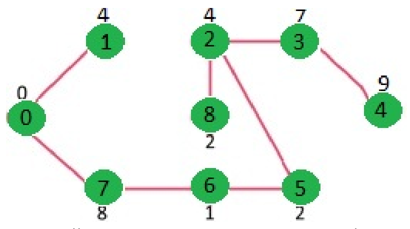
(https://www.cdn.geeksforgeeks.org/wp-content/uploads/Fig-11.jpg)
The set *mstSet* is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with the minimum key value. The vertex 0 is picked, include it in *mstSet*. So *mstSet* becomes {0}. After including to *mstSet*, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.


(https://www.cdn.geeksforgeeks.org/wp-content/uploads/MST1.jpg)
Pick the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.


(https://www.cdn.geeksforgeeks.org/wp-content/uploads/MST2.jpg)
Pick the vertex with minimum key value and not already included in MST (not in mstSET). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So mstSet now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (1 and 7 respectively).


(https://www.cdn.geeksforgeeks.org/wp-content/uploads/MST3.jpg)
Pick the vertex with minimum key value and not already included in MST (not in mstSET). Vertex 6 is picked. So mstSet now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.


(https://www.cdn.geeksforgeeks.org/wp-content/uploads/MST4.jpg)
We repeat the above steps until *mstSet* includes all vertices of given graph. Finally, we get the following graph.



(https://www.cdn.geeksforgeeks.org/wp-content/uploads/MST5.jpg)

How to implement the above algorithm?

We use a boolean array mstSet[] to represent the set of vertices included in MST. If a value mstSet[v] is true, then vertex v is included in MST, otherwise not. Array key[] is used to store key values of all vertices. Another array parent[] to store indexes of parent nodes in MST. The parent array is the output array which is used to show the constructed MST.

C++

```

// A C++ program for Prim's Minimum
// Spanning Tree (MST) algorithm. The program is
// for adjacency matrix representation of the graph
#include <bits/stdc++.h>
using namespace std;

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]
int printMST(int parent[], int graph[V][V])
{
    cout<<"Edge \tWeight\n";
```

Run

Java

Output:

```
Edge   Weight
0 - 1    2
1 - 2    3
0 - 3    6
1 - 4    5
```

Time Complexity of the above program is O(V^2). If the input graph is represented using adjacency list (https://www.cdn.geeksforgeeks.org/archives/27134), then the time complexity of Prim's algorithm can be reduced to O(E log V) with the help of binary heap.

+ Articulation Points (or Cut Vertices) in a Graph

+ Kosaraju's Algorithm | Strongly Connected Components

+ Bridges in a Graph

LIVE BATCHES

+ Sample Problems on Graphs

Report An Issue

If you are facing any issue on this page. Please let us know.

LIVE BATCHES

GeeksforGeeks

(https://www.geeksforgeeks.org/)

5th Floor, A-118,
Sector-136, Noida, Uttar Pradesh - 201305

feedback@geeksforgeeks.org (mailto:feedback@geeksforgeeks.org)

(https://www.facebook.com/geeksforgeeks.org/)(https://www.instagram.com/geeks_for_geeks/)(https://in.linkedin.com/company/geeksforgeek

Company

About Us (https://www.geeksforgeeks.org/about/)

Careers (https://www.geeksforgeeks.org/careers/)

Privacy Policy (https://www.geeksforgeeks.org/privacy-policy/)

Contact Us (https://www.geeksforgeeks.org/about/contact-us/)

Terms of Service (https://practice.geeksforgeeks.org/terms-of-service/)

Learn

Algorithms (https://www.geeksforgeeks.org/fundamentals-of-algorithms/)

Data Structures (https://www.geeksforgeeks.org/data-structures/)

Languages (https://www.geeksforgeeks.org/category/program-output/)

CS Subjects (https://www.geeksforgeeks.org/articles-on-computer-science-subjects-gq/)

Video Tutorials (https://www.youtube.com/geeksforgeeksvideos/)

Practice

Courses (https://practice.geeksforgeeks.org/courses/)

Company-wise (https://practice.geeksforgeeks.org/company-tags/)

Topic-wise (https://practice.geeksforgeeks.org/topic-tags/)

How to begin? (https://practice.geeksforgeeks.org/faq.php)

Contribute

Write an Article (https://www.geeksforgeeks.org/contribute/)

Write Interview Experience (https://www.geeksforgeeks.org/write-interview-experience/)

Internships (https://www.geeksforgeeks.org/internship/)

Videos (https://www.geeksforgeeks.org/how-to-contribute-videos-to-geeksforgeeks/)