**Breadth First Traversal Of A Graph:**

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree (See method 2 of this post). 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.



**(Source vertex is to be given by the user)**

**Breadth First Traversal Of A Tree:** level order traversal

**// Program to print BFS traversal from a given**

**// source vertex. BFS(int s) traverses vertices**

**// reachable from s.**

**#include<iostream>**

**#include <list>**

**using namespace std;**

**// This 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;**

**public:**

**Graph(int V); // Constructor**

**// function to add an edge to graph**

**void addEdge(int v, int w);**

**// prints BFS traversal from a given source s**

**void BFS(int s);**

**};**

**Graph::Graph(int V)**

**{**

**this->V = V;**

**adj = new list<int>[V];**

**}**

**void Graph::addEdge(int v, int w)**

**{**

**//a directed graph**

**adj[v].push\_back(w); // Add w to v’s list.**

**}**

**void Graph::BFS(int s)**

**{**

**//source node: the node from which we want to start, is user choice**

**// Mark all the vertices as not visited**

**bool \*visited = new bool[V];**

**for(int i = 0; i < V; i++)**

**visited[i] = false;**

**// Create a queue for BFS**

**list<int> queue;**

**// Mark the current node as visited and enqueue it**

**visited[s] = true;**

**queue.push\_back(s);**

**// 'i' will be used to get all adjacent**

**// vertices of a vertex**

**list<int>::iterator i;**

**while(!queue.empty())**

**{**

**// Dequeue a vertex from queue and print it**

**s = queue.front();**

**cout << s << " ";**

**queue.pop\_front();**

**// Get all adjacent vertices of the dequeued**

**// vertex s. If a adjacent has not been visited,**

**// then mark it visited and enqueue it**

**for (i = adj[s].begin(); i != adj[s].end(); ++i)**

**{**

**if (!visited[\*i])**

**{**

**visited[\*i] = true;**

**queue.push\_back(\*i);**

**}**

**}**

**}**

**}**

**// Driver program to test methods of graph class**

**int main()**

**{**

**// Create a graph given in the above diagram**

**Graph g(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);**

**cout << "Following is Breadth First Traversal "**

**<< "(starting from vertex 2) \n";**

**g.BFS(2);**

**return 0;**

**}**

Now, this is BFS for directed graph. Now, BFS of undirected graph follows the same approach. However, due to different construction of the graph, the traversal will be different.

**Depth First Traversal Of A Graph:**

Depth First Traversal (or Search) for a graph is similar to Depth 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 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 Depth First Traversal of the following graph is 2, 0, 1, 3.

**Recursive DFS:**

We can do it using stack.

**// 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);**

**};**

**Graph::Graph(int V)**

**{**

**this->V = V;**

**adj = new list<int>[V];**

**//because, before initialization, adj was pointer to an array of list<int>**

**}**

**void Graph::addEdge(int v, int w)**

**{**

**adj[v].push\_back(w); // Add w to v’s list.**

**}**

**void Graph::DFSUtil(int v, bool visited[])**

**{**

**// Mark the current node as visited and**

**// print it**

**visited[v] = true;**

**cout << v << " ";**

**// Recur for all the vertices adjacent**

**// to this vertex**

**list<int>::iterator i;**

**for (i = adj[v].begin(); i != adj[v].end(); ++i)**

**if (!visited[\*i])**

**DFSUtil(\*i, visited);**

**}**

**Iterative Depth First Search:**

**// An Iterative C++ program to do DFS traversal from**

**// a given source vertex. DFS(int s) traverses vertices**

**// reachable from s.**

**#include<bits/stdc++.h>**

**using namespace std;**

**// This class represents a directed graph using adjacency**

**// list representation**

**class Graph**

**{**

**int V; // No. of vertices**

**list<int> \*adj; // adjacency lists**

**public:**

**Graph(int V); // Constructor**

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

**void DFS(int s); // prints all vertices in DFS manner**

**// from a given source.**

**};**

**Graph::Graph(int V)**

**{**

**this->V = V;**

**adj = new list<int>[V];**

**}**

**void Graph::addEdge(int v, int w)**

**{**

**adj[v].push\_back(w); // Add w to v’s list.**

**}**

**// prints all not yet visited vertices reachable from s**

**void Graph::DFS(int s)**

**{**

**// Initially mark all vertices as not visited**

**vector<bool> visited(V, false);**

**// Create a stack for DFS**

**stack<int> stack;**

**// Push the current source node.**

**stack.push(s);**

**while (!stack.empty())**

**{**

**// Pop a vertex from stack and print it**

**s = stack.top();**

**stack.pop();**

**// Stack may contain same vertex twice. So**

**// we need to print the popped item only**

**// if it is not visited.**

**if (!visited[s])**

**{**

**cout << s << " ";**

**visited[s] = true;**

**}**

**// Get all adjacent vertices of the popped vertex s**

**// If a adjacent has not been visited, then push it**

**// to the stack.**

**for (auto i = adj[s].begin(); i != adj[s].end(); ++i)**

**if (!visited[\*i])**

**stack.push(\*i);**

**}**

**}**

**// Driver program to test methods of graph class**

**int main()**

**{**

**Graph g(5); // Total 5 vertices in graph**

**g.addEdge(1, 0);**

**g.addEdge(0, 2);**

**g.addEdge(2, 1);**

**g.addEdge(0, 3);**

**g.addEdge(1, 4);**

**cout << "Following is Depth First Traversal\n";**

**g.DFS(0);**

**return 0;**

**}**

Similarly, DFS for directed graph as well as undirected graph follows the same approach.