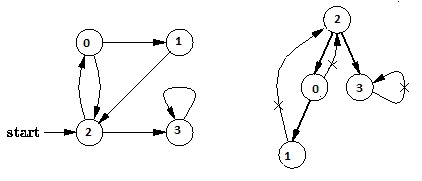
Depth First Traversal for a Graph

**LAB**

**Part 1: Run the program**

[Depth First Traversal (or Search)](http://en.wikipedia.org/wiki/Depth-first_search) for a graph is similar to [Depth First Traversal of a tree](http://www.geeksforgeeks.org/archives/618). 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.

[](http://d1gjlxt8vb0knt.cloudfront.net/wp-content/uploads/DFS.jpg)

Following are implementations of simple Depth First Traversal. The C++ implementation uses [adjacency list representation](http://en.wikipedia.org/wiki/Adjacency_list) of graphs. [STL](http://en.wikipedia.org/wiki/Standard_Template_Library)‘s [list container](http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html#LIST) is used to store lists of adjacent nodes.

// 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

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

void DFSUtil(int v, bool visited[]); // A function used by DFS

public:

Graph(int V); // Constructor

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

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

};

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.

}

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);

}

// DFS traversal of the vertices reachable from v. It uses recursive DFSUtil()

void Graph::DFS(int v)

{

// Mark all the vertices as not visited

bool \*visited = new bool[V];

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

visited[i] = false;

// Call the recursive helper function to print DFS traversal

DFSUtil(v, visited);

}

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 Depth First Traversal (starting from vertex 2) \n";

g.DFS(2);

return 0;

}

Output:

Following is Depth First Traversal (starting from vertex 2)

2 0 1 3

**Part 2: Run the program**

// DFS traversal of the vertices reachable from v to count the number of vertices in the graph.

int Graph::NumOfNodesDFS()

{

//add your code here

}

// new test case

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 << "The number of vertices \n";

g. NumOfNodesDFS();

return 0;

}

**Homework**

Implementing lab 1 using an iterative algorithm. Note that the program needs to return a list the vertices visited. The code may contain bugs. You need to fix them.

// DFS traversal of the vertices reachable from v. It uses an iterative algorithm DFSUtil()

list<int> Graph::DFS\_iterative(int v)

{

//add your code here

}

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 Depth First Traversal (starting from vertex 2) \n";

list<int> visitedList=g. DFS\_iterative (2);

list<int>::iterator i;

for(i= visitedList.begin(); i != visitedList.end(); ++i) {

cout << \*i << " ";

}

cout << endl;

return 0;

}