Data Structure and Algorithms

Lab Report

Name:	Muhammad Arslan Ishaq
Registration #:	CSU-F16-112
Lab Report #:	09
Dated:	21-04-2018
Submitted To:	Mr. Usman Ahmed

The University of Lahore, Islamabad Campus
Department of Computer Science & Information Technology

Experiment # 9
Data Structure - Graph

Objective
The objectives of this lab session are to understand the Implementation of DFS on graph

1 Theory

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.

Graph
Graph is a non linear data structure, it contains a set of points known as nodes (or vertices) and set of linkes known as edges (or Arcs) which connets the vertices. A graph is defined as follows... Graph is a collection of vertices and arcs which connects vertices in the graph Graph is a collection of nodes and edges which connects nodes in the graph Generally, a graph G is represented as G = (V , E), where V is set of vertices and E is set of edges.

DFS ordering
It is also possible to use depth-first search to linearly order the vertices of a graph or tree. There are three common ways of doing this:

A preordering is a list of the vertices in the order that they were first visited by the depth-first search algorithm. This is a compact and natural way of describing the progress of the search, as was done earlier in this article. A preordering of an expression tree is the expression in Polish notation. A postordering is a list of the vertices in the order that they were last visited by the algorithm. A postordering of an expression tree is the expression in reverse Polish notation. A reverse postordering is the reverse of a postordering, i.e. a list of the vertices in the opposite order of their last visit. Reverse postordering is not the same as preordering.

Graph Terminology:-

We use the following terms in graph data structure...

Applications of DFS:-
1) For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.

2) Detecting cycle in a graph A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges. (See this for details)

3) Path Finding We can specialize the DFS algorithm to find a path between two given vertices u and z. i) Call DFS(G, u) with u as the start vertex. ii) Use a stack S to keep track of the path between the start vertex and the current vertex. iii) As soon as destination vertex z is encountered, return the path as the contents of the stack

See this for details.

4) Topological Sorting Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers [2].

5) To test if a graph is bipartite We can augment either BFS or DFS when we first discover a new vertex, color it opposited its parents, and for each other edge, check it doesnt link two vertices of the same color. The first vertex in any connected component can be red or black! See this for details.

6) Finding Strongly Connected Components of a graph A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. (See this for DFS based algo for finding Strongly Connected Components)

7) Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

Undirected Graph:-
A graph with only undirected edges is said to be undirected graph.

Directed Graph:-
A graph with only directed edges is said to be directed graph.

2 Lab Task

Write a C++ program to implement all the above described algorithms and display the following menu and ask the user for the desired operation.
The program should have the option for reusing it after you have completed the desired task.

2.1 Program

```
#include<iostream>
#include        <list>
using namespace std;

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();    // prints DFS traversal of the complete graph
};

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);
}

// The function to do DFS traversal. It uses recursive DFSUtil()
void Graph::DFS()
{
    // 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
    // starting from all vertices one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            DFSUtil(i, 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 Traversaln";
    g.DFS();

    return 0;
}
```

3 Conclusion

Depth first search is an interesting algorithm, and as you might suspect, it is particularly well suited for inspecting if a graph is connected; if the tree returned by depth first search contains all vertices in the graph, it is connected, otherwise, it is not.