Graphs : (Un)directed Depth First Searching
Lab Assignment #11

by
Justin Anthony Timberlake

CS 303 Algorithms and Data Structures
November 8th, 2017

1. Problem Specification

The purpose of this lab was to implement the DFS algorithm to work with DirectedGraph and UndirectedGraph, which were implemented in the previous lab. Following the execution of a Depth First Search on a DirectedGraph (shown in class with “mediumG.txt”), the topological sort of the vertices was also to be generated based on the vertices in the “tinyDG.txt”.

2. Program Design

The steps taken to develop the program were:

a) Ignoring the sample code provided, developed DepthFirstPaths.java, which implements DFS driver class as well as adding DFS to both Undirected and Directed Graphs.

b) Implement the DFSVisit method.

c) Make slight alterations to include adding a defined list of Vertex objects, making it easier to iterate though in the driver class to suit the printout format as defined in the lab report.

d) Change the return type of DFS to LinkedList<Vertex> and use this return type in topologicalSort when returning the resultant LinkedList of Vertex objects pushed into the front position upon their individual completion. Also adjust DFSVisit to set the time of each step at the beginning so that the minimal steps displayed is corrected from being off by to and that when, for example, a graph is traversed from one Vertex to itself, 0 steps are taken, instead of 2. Also, added the push statement at the end of the method so that no additional calculations had to be done in topologicalSort.

e) Show the same printouts paths from 0 to any other node in the graph, showing when each cannot be reached at any given step as well, followed by the topological sorted order of the Vertex objects themselves by key value.

No additional methods were used in the primary (driver) class.

The following methods were used in secondary class, Vertex.

a) Vertex(int k) - constructor for objects of type Vertex, setting the key equal to k

b) getKey() - returns the integer key value of a given Vertex object.

c) getColor() - returns the String color value of a given Vertex object.

d) getParent() - returns the Vertex parent object of a given Vertex object.

e) setColor(String col) – sets the color value equal to the String col.

f) setDistance(int d) – sets the distance value equal to the integer d.

g) setParent(Vertex p) – sets the parent object equal to the Vertex p.

h) getFTime() - returns the final time of steps taken by traversal of a given Vertex object.

i) setFTime(int f) - – sets the integer value fTime equal to the integer f.

The following methods were used in secondary class, Graph.

a) Graph() – constructor for objects of type Graph, setting V = E = 0, for vertices and edges.

b) Graph(BufferedReader reader) – takes a BufferedReader object to read in the files, parse each line, and add a vertex for each entry in the files with an adjacency list.

c) addEdge(int v, int w) – method to be overloaded for adding an edge to a given Graph type.

d) tostring() - generates a string printout of an adjacency list from a particular graph and returns this string to be printed out.

The following methods were used in secondary class, UndirectedGraph.

a) UndirectedGraph(BufferedReader reader) – exactly the same as in Graph

b) addEdge(int v, int w) – Given two integers, adds an edge going in each direction from each of the values' corresponding vertices.

c) tostring() - exactly the same as in Graph

d) BFS (Vertex s) – launches a breadth first search from the given Vertex to all possible other values in the UndirectedGraph.

e) printPath (Vertex s, Vertex v) – Given a source vertex and an endpoint, will print out a post-BFS traversal of all paths from the Vertex v to s recursively.

f) DFS() - initializes the total time of steps taken for a given Vertex at 0, sending this time value to all values within the array of Vertex objects by calling DFSVisit on each.

g) DFSVisit(Vertex u, int time) – Changes the color of the current Vertex in a graph to gray for being processed, and then calls itself recursively on all elements within the adjacency list of Vertex objects before setting the node to black to show it has been processed, setting the final time of the current Vertex object, and finally updating the value within the list of Vertex objects to match the calculations done on the Vertex within this method, u.

The following methods were used in secondary class, DirectedGraph.

a) DirectedGraph(BufferedReader reader) – exactly the same as in Graph

b) addEdge(int v, int w) – Given two integers, adds an edge going in one direction from v's corresponding vertex to w's corresponding vertex.

c) tostring() - exactly the same as in Graph

d) DFS() - exactly the same as in UndirectedGraph

e) DFSVisit(Vertex u, int time) – exactly the same as in UndirectedGraph

3. Testing Plan

The testing plan was largely the same for implementing DFS as it was to implement BFS, except the printout was neater and more clean cut, showing the depth first paths from 0 to any other given vertex including the ones which did not exist (which were denoted as such) and the number of steps each Vertex required to reach that result. Following this, the topological sort for the given graph (tinyDG.txt) was also generated.

4. Test Cases

There were no test cases in this lab, but for comparison, calling DFS on a directed graph took 3,690,669 nanoseconds for these processes (since one accomplishes the other, they cannot be separated in the given algorithm, using tinyDG.txt. This result can be compared with the results from the last lab below.

For generating a graph with an adjacency matrix and then using a BFS for the graph on one particular value, the runtime was 2,619,263 nanoseconds with the tiny graph file as input, 12,452,756 nanosecond with the medium graph file as input, and 21,042,695,952 nanoseconds with the large graph file as input.

For the first two inputs, using an adjacency matrix helped the runtime significantly with 1,025,977 nanoseconds for the tiny graph input file and 5,150,821 nanoseconds for the medium graph input file. However, for the large graph input file, the program could not construct the MatrixGraph object as Java ran out of heap space, perhaps as a result of a restriction upon my computer or other limitations within Java itself.

5. Analysis and Conclusions

Using DFS on the tiny graph yields a faster runtime than did BFS. That is the only comparison which can be made from this lab, given the very limited new information. Addtionally, we can see the topological sorting of the Vertex objects has also been listed. In this instance, we can see that the first Vertex to be displayed is 7 because 7 has no other paths which will lead to it from any other Vertex. It is the only Vertex, which has paths leading from it others, but absolutely none leading to it.

6. References

All code in this assignment was either presented in prior labs and assignments or provided through means of pseudocode within the in-class portion of the lab.