Graphs : Minimum Spanning Trees of Directed Graphs with Prim’s Algorithm Lab Assignment #12

by
Justin Anthony Timberlake

CS 303 Algorithms and Data Structures
November 15th, 2017

1. Problem Specification

The purpose of this lab was to generate a minimum spanning tree, using an altered version of the DirectedGraph and Vertex class from recent assignments using Prim's algorithm. Prim's algorithm would change the Vertex parent values to inhert in a tree that traverses each Vertex exactly once, so that the appropriate shortest path through the tree from the root Vertex to any other Vertex can be displayed in a simple, easy to read format.

2. Program Design

The steps taken to develop the program were:

a) First implemented the edgeWeight merely as a property of an individual Vertex object and put new Vertex objects into my adjacency list instead of referencing the ones in vertList as before, just to complete the in-class portion of the assignment and printout the visualized modified adjacency list with the associated weights for each edge of the tiny and medium graphs.

b) Added the vertList reliant printPaths method to the code just as in the DirectedGraph class from before, shifting back to the previous implementation of the graph which relied on the adjacency list as well as vertList. This required the Edge class to be made instead of just multiple Vertex objects with overriding references everytime an edge contained the same Vertex as a previously processed edge did.

c) Implemented Prim's algorithm in the WeightedGraph class using a PriorityQueue Q and the edges within each position of the adjacency list for every Vertex as it is polled from Q as "u" as shown in the algorithm. Then, based on the neighbors, the most eligible (satisfies all necessary conditions) Vertex is chosen from the graph to become the next position in the minimum spanning tree. One noticeable caveat and hang up for this program ending in never-ending stack overflows was that the order of Q had to be recalculated after modifications to Vertex objects were made, so every time that a wKey was set to a smaller value than infinity, it needed to be removed from Q and then returned, but in its new, higher priority position. This was the primary problem for edges which went both ways as in an UndirectedGraph.

d) Generate benchmarks and text printouts for later comparison.

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

The following methods were used in secondary class, Vertex, not all of which were used in this lab :

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.

j) getWkey() - returns the wKey double of this Vertex (key from the pseudocode)

k) setWKey(double w) – sets the double value wKey equal to w

l) compareTo(Object v) – overloading the comparison method from the implemented interface, Comparable<Object>

The following methods were used in secondary class, Edge.

a) Edge() - constructor for objects of type Edge

b) Edge(Vertex v, double ew) - constructor for objects of type Edge, setting the Vertex v1 = v and double edgeWeight = ew.

c) Vertex getVertex() - returns the Vertex v1 associated with this Edge.

d) setVertex(Vertex v) - sets the value of v1 to the Vertex object v.

d) getEdgeWeight() - returns the double edgeWeight associated with this Edge

d) setEdgeWeight(double ew) - sets the value of edgeWeight to the double ew.

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, WeightedGraph.

a) WeightedGraph(BufferedReader reader) – exactly the same as in Graph, except that it is modified to accept 3 inputs per entry, with the third evaluating to a double edgeWeight for each Edge object

b) addEdge(int v, int w, double x) – Given two integers and a double, adds an edge going in one direction from v's corresponding vertex to w's corresponding vertex from the list of Vertex objects (vertList) along with the associatd edgeWeight x.

c) tostring() - exactly the same as in Graph, except it includes the edgeWeights of each Edge

d) printPath(int s, int v) – prints out the shortest path from the source Vertex at vertList[s] to the destination Vertex at vertList[v], used to determine resultant paths from each graph algorithm implemented so far.

e) MSTPrim(Vertex r) – Generates the minimum spanning tree from a given source Vertex r, implemented with a PriorityQueue for processing each neighboring Vertex object in the graph at a time, altered to keep track of which Vertex objects have been processed by reusing the old color properties from former assignments. This particular MST will be generated using Prim's Algorithm.

3. Testing Plan

The testing plan was largely the same for implementing DFS and BFS as it was to implement Prim's, showing the various paths along the MST from 0 to any other given Vertex, all of which necessarily existed even though this was a directed graph. Initially these paths were printed out for testing purposes, however, in the end, Prim's MST algorithm was called on each of the 4 sizes of the directed and weighted graphs, recording the time for each of the corresponding runtimes to be printed out upon execution of the program.

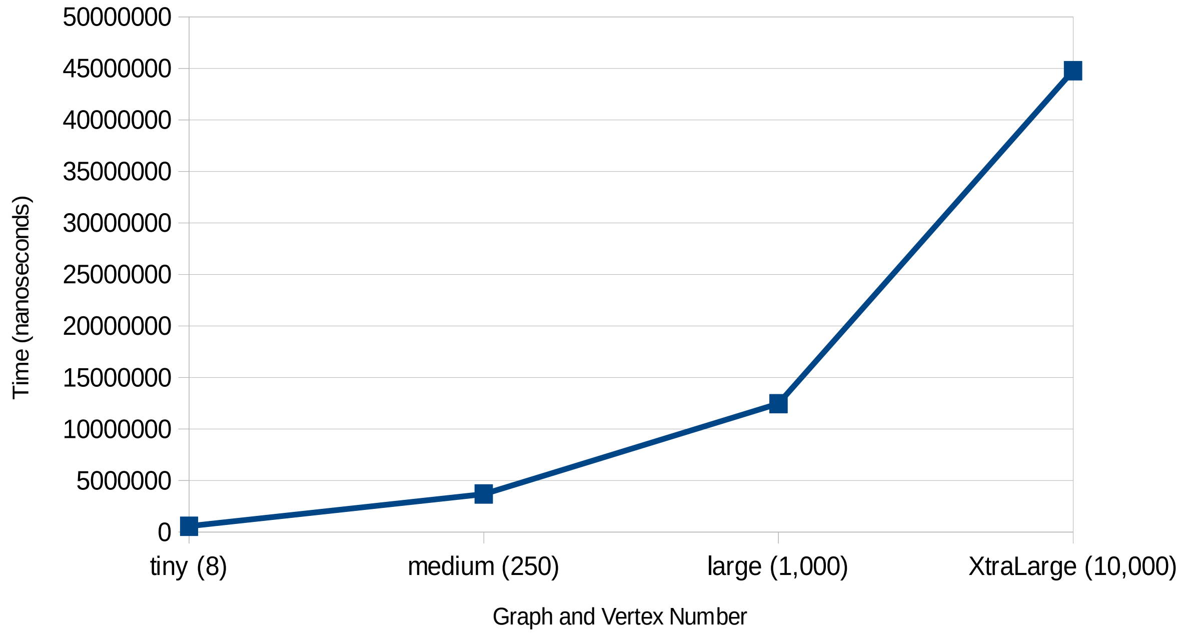
4. Test Cases

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



Naturally, it took longer to generate the MST for larger graphs.

The total time elapsed for tiny graph, medium graph, large graph, and extra large graph were 576,791 nanoseconds, 3,694,225 nanoseconds, 12,467,368 nanoseconds, and 44,776,726 nanoseconds respectively. The running time for this algorithm is O(VlogV + ElogV) = O(ElogV), where E is Edges and V is Vertices. Though the algorithm contains a nested for loop within a while loop, leading one to expect n^2 time, V is essentially n in this case, being the number of vertices and also the size that Q will be after an n time for loop to input all vertices into Q as a PriorityQueue. The inner for loop will always be significantly less than V however as it only corresponds to the number of edges each Vertex will have, and on those, only the elements which will be selected for the MST will perform the majority of the calculations within this for loop (though every element within the corresponding LinkedList in that position of the adjacency list will necessarily have to be visited and determined as meeting two separate conditions regardless).

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.