Graphs : Dijkstra's Single-Source Shortest Paths
Lab Assignment #13

by
Justin Anthony Timberlake

CS 303 Algorithms and Data Structures
November 29th, 2017

1. Problem Specification

The purpose of this lab was to generate the single source shortest path from a given Vertex, using an altered version of the DirectedGraph and Vertex class from recent assignments using Dijkstra's algorithm. Dijkstra's algorithm would change the Vertex parent values to inherit 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 set, containing the list of Vertex objects in ascending order of edge weights.

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.

f) initializeSingleSource(Vertex s) – sets all wKey values to maximum and parent values to null within every Vertex in vertList, also initializing the wKey value of the node s to 0.

g) relax(Vertex u, Vertex v, int I, PriorityQueue<Vertex> Q) – compares the wKey value at a Vertex with all the same value of another Vertex added to the current weight of the element i referenced within the adjacency list of u, updating depending on which total is shorter and then removing and re-adding the element to Q in order to update the ordering of which paths are shortest.

h) dijkstra(int s) – First initializes the graph by calling initializeSingleSource on the vertex at position s, initializes an arrayList S and a PriorityQueue<Vertex> Q, and adds all elements of vertList to Q. Then, similar to Prim's one Vertex u will be polled from the smallest position in Q and added to the set S. For every position of the adjacency list of u, relax is called with the current Vertex u, each Vertex at every position in its associated adjency list index, the specific number of the item in the adjacency list (for reference to this position being used again for the edgeWeight within the relax function), and the PriorityQueue itself, so that Q can only be updated if the condition is met. In retrospect, relax would have probably been better with my implementation if it were just a condition included within the dijkstra method, rather than a seperate method of its own, but trying to follow the pseudocode as much as possible yields to this structure.

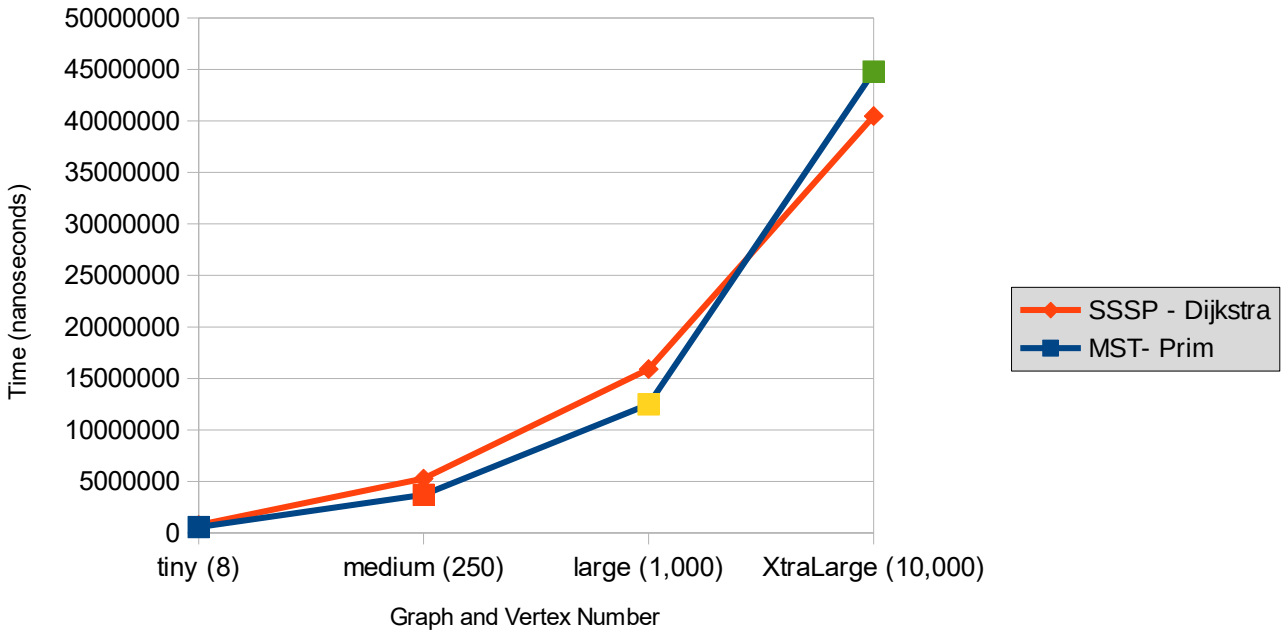
3. Testing Plan

The testing plan was largely the same for implementing DFS, BFS, and Prim as it was to implement Dijkstra's, however, while the paths could easily be printed as well, the only necessary print out was that of the given set S (in this case an arrayList) of the Vertex objects in the order of their associated weights traveling from the source Vertex (in this case 0). The adjacency list is printed, followed by the single-source shortest path weights from 0 to each of the Vertex objects. Dijkstra's 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 4 test cases, involving running the algorithm for single-source shortest paths from the Vertex 0, only varying the number V of Vertex objects as well as E of Edges. These differences were accounted for within the varying sizes of files provided.

5. Analysis and Conclusions



Naturally, it took longer to generate the MST and SSSP 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 for Prim's MST algorithm. For the tiny graph, the total amount of time it took to generate the single-source shortest paths from 0 as the source was 769,976 nanoseconds, 5,273,289 nanoseconds for the medium graph, 15,900,061 nanoseconds for the large graph, and finally 40,472,921 nanoseconds for the extra large graph, in the case of Dijkstra's algorithm. The running time for this algorithm is $O(V \lg V + E \lg V) = O(E \log V)$, where E is Edges and V is Vertices, which is the same approximate runtime of Prim's. Based on the graph, one can see their approximate similarity even though Dijkstra's seems to have a slightly shorter run time. That being said, it is somewhat irrelevant as Prim's and Dijkstra's algorithms have slightly different purposes as one generates a minimum spanning tree and the other generates shortest paths, which update according to newer shorter paths being found from one Vertex to another.

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.