Adrian Gellert

Professor Maximillian Bender

CS231 Lecture A, Lab D

18 December 2022

Adrian's Project 9: Quickest Path Around The United States

ABSTRACT

This project implemented many methods and data structures from previous projects. That is, in this project we used recursion, priority queues, hashmaps, lists, array lists, and stacks. The purpose of this project was to implement these methods and data structures within a graph data structure. A graph data structure includes vertices and edges that go between the vertices. In the case of this project, the vertices contained information regarding the state capital and its location. The edges, in contrast, contained information regarding the cost of going between the vertices. The ending goal of the project was to attempt to find a minimum Hamiltonian path around the United States via a brute force algorithm. Seeing that such a method takes way too long to run, we chose different algorithms to approximate the solution. I implemented Prim's algorithm in my project, where you choose a starting vertex in the map and go from vertex to vertex across minimal cost edges until all vertices are visited.

RESULTS

Minimum Spanning Tree approximation found:

Total cost of traveling this path: 833863

[{<VA>, <NC>: 9154.0}, {<VA>, <MD>: 8328.0}, {<MD>, <DE>: 4684.0}, {<NJ>, <DE>: 6515.0}, {<NJ>, <CT>: 11068.0}, {<NY>, <CT>: 6390.0}, {<RI>, <NY>: 9418.0}, {<RI>, <MA>: 3520.0}, {<NH>, <MA>: 3930.0}, {<NH>, <ME 04330>: 8936.0}, {<VT 05602>, <ME 04330>: 13682.0}, {<VT 05602>, <PA>: 27344.0}, {<SC>, <PA>: 31061.0}, {<SC>, <GA>: 11348.0}, {<GA>, <AL>: 8647.0}, {<MS>, <AL>: 14452.0}, {<MS>, <AR>: 15219.0}, {<LA>, <AR>: 21641.0}, {<LA>, <FL>: 22942.0}, {<TN>, <FL>: 27829.0}, {<TN>, <KY 40601>: 11819.0}, {<KY 40601>, <IN>: 9446.0}, {<OH>, <IN>: 9696.0}, {<WV>, <OH>: 9525.0}, {<WV>, <MI>: 23390.0}, {<MI>, <IL>: 21407.0}, {<MO>, <IL>: 11936.0}, {<MO>, <KS>: 11895.0}, {<OK>, <KS>: 15489.0}, {<TX>, <OK>: 20643.0}, {<TX>, <NE>: 43168.0}, {<NE>, <IA>: 10203.0}, {<MN>, <IA>: 13090.0}, {<WI>, <MN>: 14185.0}, {<WI>, <ND>: 36836.0}, {<SD 57501>, <ND>: 12280.0}, {<WY>, <SD 57501>: 23663.0}, {<WY>, <UT>: 23419.0}, {<UT>, <MT>: 24882.0}, {<MT>, <ID>: 27004.0}, {<OR>, <ID>: 26183.0}, {<WA>, <OR>: 9145.0}, {<WA>, <NV>: 39444.0}, {<NV>, <CA>: 9312.0}, {<CO>, <CA>: 61969.0}, {<NM>, <CO>: 20795.0}, {<NM>, <AZ>: 26931.0}]

Size = 47

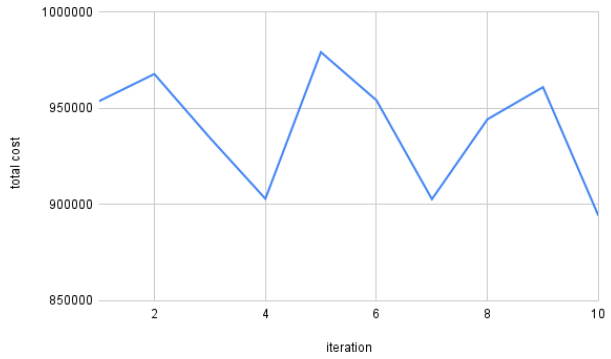

Figure 1: Graph showing periodicity of total path cost from calling tspApprox() multiple times (exploration)

RESULTS DISCUSSION

Figure 1 shows that there is a periodicity to the performance of the depth first search when randomness is added to the method. Specifically, we can see that randomness of movement through the search causes the total path cost to cycle between 900,000 and 100,000. When we include starting from a random point, the difference between the minimum and maximum of this cycle increases.

REFLECTION

minTSP runtime:

To analyze the runtime of this method, we first need to analyze the runtime of the allHamCycles method. On surface value, the runtime of allHamCycles include a loop to add all edges of a path to an edge sequence. This loop depends on the length of the path. Thus, the time complexity of this loop is O(n). There is also the condition that we must make this edge sequence for every path available in the graph. Thus, including both of these, the total runtime would be O(n*n), which is O(n^2). However, the method also includes recursion, which does the above on smaller and smaller portions of the graph. Therefore, the total runtime of the method O(n!).

Disregarding allHamCycles, the worst-case scenario runtime of minTSP would be O(n) because the longest portion of the method is where it loops through each edge in a path. This portion depends on the length of the path. However, to obtain the path itself, the method relies on allHamCycles, which has a time complexity longer than any part of minTSP. Therefore, the time complexity of this method is equivalent to that of allHamCycles, O(n!).

mst runtime:

First, this method chooses a vertex, finds all the edges that go out from that vertex and adds those edges to a queue. The method to add items to the queue has an O(log n) runtime, and the number of additions depends on the number of edges that the vertex is connected to. Therefore, in totality this portion of the method has an O(n log(n)) runtime. In the next portion of the method, an edge is polled from the queue, and for every vertex in the map, the edge is added to the mst and the

edges spanning from the vertex are added to the queue. This portion runs until all vertices have been visited. Since adding and removing from the queue takes O(log n) time and the method does this for n vertices, this portion of the method takes O(n log(n)) runtime as well. In totality then, the mst method has a runtime of O(n log(n)).

tspApprox runtime:

This method first stores an mst of the graph. Since the mst method takes O(n log(n)) runtime this step will take that long. The next portion of the method uses Prim's algorithm to search through the tree to find an approximation to the traveling salesman problem. Specifically it takes each edge from the tree and adds it to the approximation to be returned if it does not contain a vertex in the visited array. Because the method depends on lengths/sizes, that of the tree and that of the visited array, this portion takes O(n) time to run. Finally, there is an ending loop to determine the total cost of going through the path, which depends on the length of the path. Since the longest portion of the method is the part to obtain the mst, this method takes O(n log(n)) runtime. A thing to note about this method is that I added randomness to both where the starting vertex is and how vertices are added to the stack of vertices to still go through.

Other methods in this class are readData and shortestPaths. readData takes a file and makes a graph containing vertices and edges based on the contents of the city-state pairs within it. shortestPaths makes a HashMap containing distances from each vertex to another vertex in the map. Then, it offers these distances to a priority queue which can then be used to recreate the hashmap containing only those vertex to vertex paths that are of minimal length.

COLLABORATION

Jessie, Camila, Jaime, and I were all staying up to finish the coding portion of this assignment. We helped each other in terms of walking through how to implement Prim's algorithm. We also received coding help from the TA who was there.