Andrew Knox

CSC 221 DS&A

5/1/2019

Writeup for Program 6

**Overview of Program:**

This program involves finding the minimum spanning tree on an undirected and weighted graph for multiple graphs. For this project, vertexes and edges will be called Huts and Roads, and graphs will be called Villages. The program gives 1 to 100 data sets that need to be parsed by the user, where each data set is a Village. All information about the Village is included in the data set (connections, # of roads, # of huts). Each road has a weight and the user has to find the minimum cost of all the roads needed such that all huts can be traveled to from any hut in the village. Additionally, the user must write an algorithm such that, given the maximum amount of input, all Villages’ minimum costs can be found in less than a minute.

**Pseudocode for Algorithm:**

I used Prim’s algorithm to solve the problem. My method creates a PriorityQueue of Roads that contains any Road that connects a known Hut to other Huts. To start the algorithm, one Hut is marked as known, and the PriorityQueue builds from there. Pseudocode:

ArrayList H of all Hut objects;

Boolean array B indicating which Huts are considered known;

PriorityQueue PQ of possible Roads to be examined

While (PQ is not empty OR not all Huts are marked as known) {

Remove head of PriorityQueue (least costly road)

If the Road points to an unknown Hut, then add the weight of Road to total cost and mark previously unknown Hut as known

}

I calculated my time complexity for this method. The variables I used were:

E = edges = Roads, 0 < E <= 150

V = vertexes = Huts, 1 < V <= 26

N = # of Roads for a given Hut, 0 < N <= 15

The most time-consuming command is:

unKnown.addAll(hut.closeHuts);

* This step within the while loop takes the most time, O(N\*Log(E)), because adding to a priority queue costs O(Log(X)) where X = the number of entries in the Priority Queue. The priority queue will not have more than E elements and will add N roads to the Priority Queue. Thus, this step is O(N\*Log(E)). The while statement loops O(E) times, so the time complexity of the method overall is O(E\*N\*Log(E)).

**Choice of Data Structures:**

Hut:

A Hut object H needs a dynamic list that contains all Road objects R such that R connects H to a different Hut. I used a LinkedList instead of an ArrayList because LinkedLists are better for modifying elements in a dynamic list whereas ArrayLists are better for accessing and storing elements in a dynamic list. Each time a Road is removed from the PriorityQueue and determined to be in the minimum spanning tree, my algorithm removes the complementary road: for example, if A to B is determined to be in the MST, the Road object B to A in hut B’s LinkedList is removed. Since there will be many modifications, LinkedList is better than ArrayList.

Main:

Main tracks Hut objects in an ArrayList because the list of total Hut objects will not change. I prefer an ArrayList to an array because I personally feel it’s more readable and I’m more comfortable with using one, however, for maximum efficiency an array should be used. This is because the list of Huts will not change, and does not need the dynamic allocation of an ArrayList or LinkedList.

PrimsAlg:

Method PrimsAlg uses two data structures: an array to indicate which Huts are known vs unknown, and a PriorityQueue of Roads to be examined. The array does not need dynamic allocation, but it does need to track the index of each Hut, and therefore an array is the most efficient data structure to use. The PriorityQueue is also the most efficient algorithm for the purpose of this method because the method continually loops and retrieves the least costly road. Thus, the methods we need to optimize are findMin and deleteMin. findMin is constant time whereas deleteMin is linear time for a PriorityQueue, so it’s a good implementation. It’s also the easiest data structure to use in this algorithm because a PriorityQueue is primarily used to retrieve the least costly element in the heap. It’s common sense for our purposes.

**Implementation:**

I wrote my own classes Hut and Road, which resembled the general structure of classes Vertex and Edge. For all the data structures, I used Java classes.

**Challenges:**

Initially I wanted to examine each road from highest to low and check if deleting that road would still allow travel between all Huts. However, after looking back in the book at Prim’s algorithm, I based my algorithm off that because it seemed simpler than what I was thinking.

**Results:**

216

30

36

These results include the two given Village data sets, and a data set I added where there are 9 Huts and many roads. I envisioned the Huts in a circle; one Hut in the middle has roads to all other 8 Huts, whereas the other 8 Huts have 8 roads leading to all the others except for the one directly across from it in the circle. Each road has a cost of 1 + the last road’s weight, starting with weight 1.