Nicholas Strohmeyer
EE 360C SU2022
Nur Touba, Branch Archer

Programming Assignment Report # 2

(a) Minimum Student Cost

For this problem, I basically implemented Dijkstra's algorithm using a minimum-heap. The main difference is that I am only trying to return the actual value of the minimum cost and not the exact path. For this reason and also because the Student class did not have a "parent" attribute, my algorithm does not store the shortest paths tree that Dijkstra's algorithm creates, rather just calculates the minimum cost value. A second difference is that because we know our destination node, we need not continue until all nodes are explored. Once our destination node is added to the explored set, it means that the shortest path has been found and we can stop the algorithm.

One other note for both algorithms is that we never have to set nodes to "inf" because the student constructor will automatically set minCost to the maximum int value. However, we do need to make sure to set source nodes to 0.

Pseudocode:

```
public int findMinimumStudentCost(Student Start, Student dest) {
        1. Start.setMinCost = 0  // set start min cost to 0, top of the heap
        2. Initialize a minimum Heap "Q" using buildHeap method & list of students
        3. Initialize S = {} (empty set using an ArrayList) && currLength = 0
                - "currLength" will store the minimum path to the node we are currently
                exploring
        4.  while (Q  is not empty)  {
        5.      Explore Node =  Extract Min from Q
        6.      currLength = Node.getMinCost()  // when it is extracted the minCost =
                                                shortest path
        7.      if  (Node.name = dest):  BREAK ;   // stop here if it is our destination node
        8.      Get Node's Neighbors & Prices
        9.      For ( each Neighbor Node called "check" ) :{ // relaxing (unexplored) edges
        10.             If ( check is already included in S ) : continue;  // (already explored)
                        else :   get Price ( node, check) & continue
        11.             if ( currLength + Price  <  check.getMinCost() ):
        12.                     update check's key in Q ( new minCost)
                }
        13.     Add node --> S    // for loop finishes meaning node is fully explored
              }
        14.  return currLength ;   // while loop broke meaning we extracted the destination
                                        node. Return it's minCost value
}
```

Runtime Analysis:

Heap Operations:

Line 2 : Build Heap I believe should run in O(n) time the way I designed it. I will put the argument for this at the end of the report

Line 5: Extract Min should be O(Log V) where V = # of students

Line 12: ChangeKey is also O(Log V)

Overall :

Lines 4 - 12 : We will at most traverse through V nodes & possibly less

Line 10-12 : By going through each node & all of its neighbors, we are essentially traversing each edge twice (in the worst case when we need to go through all nodes)
Therefore overall traversal can be said to be O(2E) = O(E)

Within Lines 4-12:

- We do heap operations in line 5 = O(VLogV)
- And in line 12 : O(2ELogV)

Lines 1-4 : Either constant time or build heap, which is bounded by O(V)

Overall : O(VlogV) + O(2ELogV) + O(V) = **O(ELogV)**

(b) Minimum Total Class Cost

This algorithm essntially needs to build a minimum-spanning-tree out of the provided network. My algorithm is basically an implementation of Prim's algorithm using a minimum heap to remember "lightest" edges to a particular node. This algorithm is actually pretty similar to Dijkstra's, excpet instead of a key in Q representing the currently known shortest distance to get to that node from a specified start, it represents the currently known "lightest" edge connecting an unexplored node to the set of explored nodes S.

Again, we will not be remembering the actual edges/final tree structure. One way we could do this would be to impelement a parent attribute in the Student class. Since this algorithm just needs to return the minimum total cost, we will just be accumulating the cost of cut edges as we go without remembering the edge itself.

PseudoCode:

public int **findMinimumClassCost()** {

1. Initialize an arbitrary root to begin our search
   - root = students.get(0) // we just choose the Oth element from the given students array
   - root.setMinCost() = 0
   - also intialize int **totalCost = 0**;
2. Build Heap "Q"
3. While (Q is not empty){
4. Node = Q.extractMin() // the node is selected based on "cut edge" the base case is the root which will be explroed first
5. **totalCost** += Node.getMinCost() //final answer accumulates all cut edges
5. if (Q is empty now): BREAK ; // we are done if no nodes left on the Q
6. Get Node's edges & costs ; //constant time , just pointing to stored arrays

```
7.      for ( all of Node's edges ) {
8.          get neighbor "Check" & weight of the edge "cost"
9.          if ( Check is still in Q )  &&  ( cost < Check.getMinCost()  ) :
10.             Change Key for Check  (also sets the minCost attribute)
            // NOTE:  both conditions must be satisfied. If Check is not in
                      Q anymore, it means it has been fully explored and the proper
                      edges already selected. If (cost < curr minCost) it means we
                      have found a lighter edge connecting Check to S (the set of
                      explored nodes)
            }
        }
11.     return totalCost;
}
```

Runtime Analysis

As before, lines 1-4 are bounded by O(V) which is the cost of building the heap.
For analyzing the loop:

* We know we will have to extract every node off Q before the algorithm finishes.
* We also know that for each node, we will look at all of its edges
* Therefore, again the total # of traversals will be bounded by 2E

Line 4: Will execute V times and so it is O(VlogV)
Line 10: While not every traversal will hit line 10, we can say that it is bounded by the worst case = 2ELogV oeprations but in practice it will be better than this because if we are checking an edge a second time it means one of those nodes has been explored before, so we will not execute line 10. Therefore Line 10 is = O(ElogV)
Overall : O(V) + O(VLogV) + O(ELogV) = **O(ELogV)**

(c) Final Notes:

Build Heap Runtime Analysis:

My buildHeap() function starts with the first non-leaf and calls heapifyDown() on all non-leaf nodes. This means any node at level 1 (level 0 is the leaf level) will be swapped at most once (down). Likewise, any node at level 2 will be swapped at most twice; any node at level 3, at most 3 swaps etc. This will produce the expansion that Prof. Touba demonstrated in lecture 10 which he proved will lead to **O(V).**

A small note on why this algortihm will work : after calling heapify down on some node, the only places our heap may not be valid is above/before the current node. However, because we go in reverse order, the algorithm works "bottom-to-top" so these nodes will be heapified correctly later on.