CS325-001-F17
Group Assignment 3 Write up
November 14, 2017

Daniel Schroeder
W Keith Striby

We worked with Evan Milton and Alea Weeks to compose a solution.

First Minimum Spanning Tree Pseudo Code:

```
//The first row of numbers from input.txt have already been placed into heap.
//An array, label, equal to the amount of rows/columns marks whether or not a vertex has
//been used from it.
//myEdges is used to add an edges information once it's determined to part of the MST.

primMST(heap, e, last_edge)
        (w, (v1, u1)) = e                   //bad edges not to be used in 2nd and 3rd MST
        (w2, (v2, u2)) = last_edge
        //used to ignore the last edge in the 2nd MST to create the 3rd MST
        MSTtotal = 0                        //The total edge amount in MST

        while the heap is not empty:
                (weight, (v, u)) = pop the minimum weight (root) from the heap, & vertex info
                if (v1 != v or v2 != u) and (v2 != v or u2 != u):
                //used to compare "bad edge" when finding 2nd and 3rd MST

                        if label[u] != label[v] and label[v] == 1:
                        //look at both vertices of the edge just popped off the heap
                        //if the vertex 'v' is already in the MST

                                label[u] = 1
                                myEdges.append((weight, (v, u)))
                                //place edge & vertex info into myEdges[]
                                heapInsert(u, readRow(u+1), myHeap)
                                //add all edges of the vertex that was just appended into the heap
                                MSTtotal += weight          //add the weight to the MST total

                        else if label[u] != label[v] and label[u] == 1:
                        //look at both vertices of the edge just popped off the heap
                        //if the vertex 'u' is already in the MST

                                label[v] = 1
                                myEdges.append((weight, (v, u)))
                                //place edge & vertex info into myEdges[]
                                heapInsert(v, readRow(v+1), myHeap)
                                //add all edges of the vertex that was just appended into the heap
                                MSTtotal += weight          //add the weight to the MST total

        return MSTtotal, myEdges
```

Second Minimum Spanning Tree Pseudo Code:

```
//first_mst recreated the 1st MST and holds the MST value and each edge

for bad_edge in first_mst[1]:                //first_mst[1] cycles through each edge in 1st MST

    myEdges = []                             //re-initialize myEdges
    myHeap = []                              //re-initialize myHeap
    label = [0 for x in range(n)]            //zero out label
    heapq.heapify(myHeap)                    //convert myHeap into a heap data struct

    heapInsert(0, readRow(1), myHeap)
    //insert the first vertex/row into the heap with all of its edges

    label[0] = 1                             //update label array to reflect first row added
    second_MST = primMST(myHeap, bad_edge, (-1, (-1, -1)))
    //Call primMST with the heap, iterating through each edge in the first MST, but each
    //iteration provides another edge in the first MST to be ignored to create a different
    //tree from the first MST and is held in second_MST.

    if min == -1 or second_MST[0] < min:
    //min is initially set to -1, then is set to the value primMST sets second_MST[0] and is
    //the sum for the tree just found. We want second_MST[0] to be less than min to find
    //the second MST.

        min = second_MST[0]
        //store the value of the 2nd MST
        MSTedgesMin = second_MST[1]
        //store the edges of the 2nd MST to find the disjoint set
```

Third Minimum Spanning Tree Pseudo Code:

```
//first_mst recreated the 1st MST previously found and holds the MST value and each edge

for bad_edge in first_mst[1]:                // first_mst[1] cycles through each edge in MST

    myEdges = []                             //re-initialize myEdges
    myHeap = []                              //re-initialize myHeap
    label = [0 for x in range(n)             //zero out label
    heapq.heapify(myHeap)                    //convert myHeap into a heap data struct

    heapInsert(0, readRow(1), myHeap)
    //insert the first vertex/row into the heap with all of its edges

    label[0] = 1                             //update label array to reflect first row added
    third_MST = primMST(myHeap, bad_edge, new_edge[0])
    //new_edge[0] is the edge in the 2nd MST that was not in the 1st MST.
    //Call primMST with the heap, iterating through each edge in the first MST, but each
    //iteration provides another edge in the first MST to be ignored and new_edge is also
    //ignored. This creates a different tree from the 2nd MST and is held in third_MST.

    if (min == -1 or third_MST[0] < min) and third_MST[0] > secondMSTtotal:
    //min is initially set to -1, then is set to the value primMST sets third_MST[0] and is the
    //sum for the tree just found. We want third_MST[0] to be smaller than min but greater
    //than the total for the second MST, which will eventually create the third MST.

        min = third_MST[0]
        //hold the new value of the third MST
        MSTedgesMintwo = second_MST[1]
        //hold the edges of the third MST (not necessary)
```

Proof for the first MST:
To find the Minimum Spanning Tree of a graph G, we used Prims algorithm. Prims algorithm effectively adds a preliminary vertex to the MST and then adds all its edges into a priority queue heap. The algorithm then enters a while loop that continues until the heap is empty, popping minimum edges off the heap.

It will check if the vertices of that edge are already in the MST, if not it will add that vertex, insert all its edges into the heap, and continue the loop until all vertices have been added to the minimum spanning tree connect by all their lightest edges.

This works for the first MST because every edge that is popped off the edge is either:

1. A safe edge from a vertex in the tree to a vertex that has not been added.
2. Or an edge between two vertexes already in the MST which is ignored.

Proof for the second MST:
For every edge we added to the first MST we run Prim's algorithm again, but ignore that edge to create a different minimum spanning tree.
For every edge e $\exists$ MST_1 find a new MST that replaces e with a new edge e'

This means that MST_2 differs from MST_1 by one edge.

We take the minimum of all MST_2's that we compute by disregarding a single edge of MST_1.
We know that our new edge e' is the next smallest edge in graph G. Since e' was not in MST_1 we know that e' must be larger than or equal to all edges in MST_1 which makes the weight of MST_2 larger than or equal to MST_1 confirming it is the second minimum spanning tree.

Proof for the third MST:
When we calculate the third MST, we perform the same actions as MST_2 except we remember what our added edge was (e') from MST_2 and make sure that we do not include it in our analysis of MST_3.

This works because for every edge in the MST_1, we run prims algorithm ignoring that edge (essentially removing it from the MST) and try to create a new MST with a new edge e'. But, we know what the last e' was for MST_2 and we ignore that as well, to ensure that we do not re-create the same MST as MST_2.

This is an edge case that need to be considered because if MST_1 differs from MST_2 by the edge (u, v), and, in turn, gains the new edge (u, x).

It is possible for the MST_3 to be formed from removing the same edge from MST_1 (u, v) but gaining a different edge (u, y).

For this reason, we need to remember the edge that was added to MST_2, so that when our algorithm chooses to ignore the MST_1 edge(u, x) it will ignore it initial intuition to choose the minimum edge(u, x) and move on to the second minimum edge(u, y) to receive MST_3.

Runtime Analysis:
We heap insert the first vertex and neighbors which results in:

$$O(\log E).$$

Then we enter a while loop that runs at most:

$$O(2E).$$

From that while-loop, we call heap insert at most 2E times, giving us:

$$\begin{aligned} O(2E \log E) &= O(E \log V^2) \\ &= O(2E \log V) \\ &= O(E \log V). \end{aligned}$$

We have a double for-loop that gets the difference of edges in MST_1 and MST_2 to store what edge we added for MST_2 so that we can ignore it in the analysis of MST_3.

This has a RT of: $O(V^2)$

For the second and third minimum spanning tree we run primMST() in a for each loop for every edge in MST_1 resulting in:

$$O(VE \log E) \text{ or } O(VE \log V).$$

Therefore, we get an overall runtime of:

$$2O(VE \log E) + O(E \log E) + O(V^2)$$

Since: $O(VE \log E) > O(E \log E) > O(V^2)$

We get an overall RT of:
$O(VE \text{ Log } E)$