Hongsup OH
GTID:903232384

4.Programming assignment

a) Data Structure

Here is a list of data structures that I have used for my implementation:

Union-Find - Best time complexity for Union-Find can be reduced to $\log N$ by using weighted quick-union with path compression and when done for m nodes, I get time complexity as O(m log(N)).
Dictionary - I have used this for avoiding any computation for those edges which have been added to graph but exist multiple times with different weights. But as I have already sorted the edges in the order of increasing weight, I can be assured that if an edge with same source and destination is observed again, then it's weight will be more than the edge that was observed for the first time and hence I can just ignore it. Dictionary helps me detect collisions in O(1) time.

b) Time Complexity

Before I start calculating time complexities for both the methods, let's define some variables.

E - total number of edges

V - total number vertices

I use Kruskal's algorithm to compute MST. And expected running time is O(E log(E)) or O(E log(V)). The reason why I choose Kruskal's algorithm is that I think it has simpler data structure (sparse graphs), and Kruskal's algorithm works well in sparse graph (Prim's algorithm works well in dense graph). Then, I explain time complexity of computeMST(), and recomputeMST().

computeMST( ) - First step is sorting the edges as per their weights, time complexity for this step is $Big(O) = E\log(E), or\ Big(O) = E\log(V)$. In this problem, the number of edge (E) is much larger than the number of vertices (V). So I think time complexity is O(E log(E)), since it is worse than E log(V); (log(E) > log(V)). Now, I just need to find time complexity for union find. Starting from an empty data structure, any sequence of E unions (total number of edges) and find operations on V objects (total number of nodes) takes $O(E\log(V))$, which is smaller than O(E log(E)). Thus, total time complexity for computeMST( ) is O(E log(E))

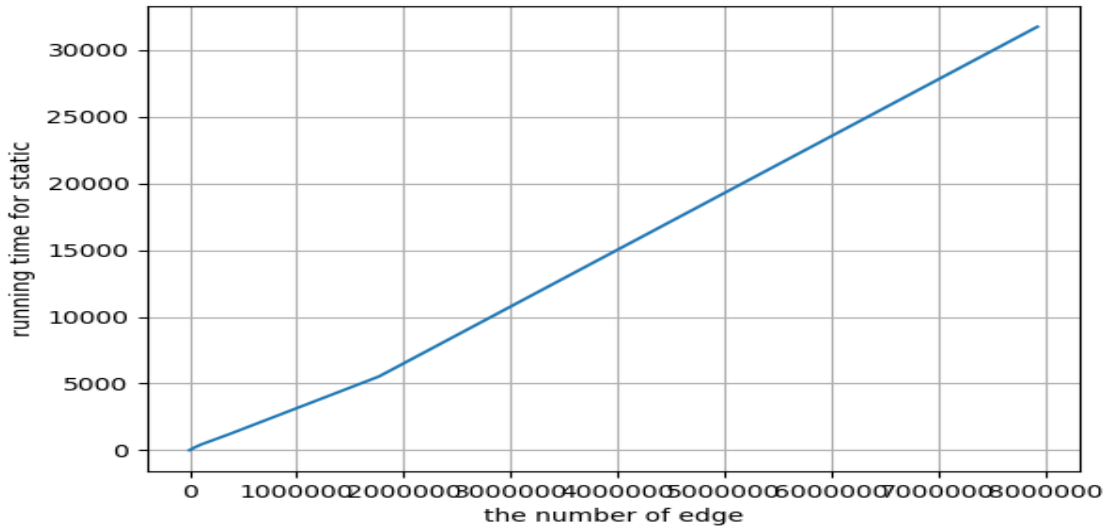
recomputeMST( ) - Let's list the algorithm for recomputing MST:
1.(u, v) is the new edge e*, find shortest path, p, between u and v in T, MST.
2.Find the edge in p that has the maximum weight $w*$.
3.If $w < w*$, then $T* = T - e* + e$ the MST for $G + e$, with $cost(T*) = cost(T) - w* + w$
4.If not, then $T' = T$ is the MST of $G + e$ (graph + edge)

Only step that can take some significant time in this algorithm is find shortest path, which takes linear time i.e. $O(V)$ for a minimum spanning tree.

c) Static Computation Plot

My Big O for static computation is $E\log(E)$, and the graph (figure1) shows exactly same behavior of E log(E). Therefore, I think theoretical time complexity (E log(E)) shows similar

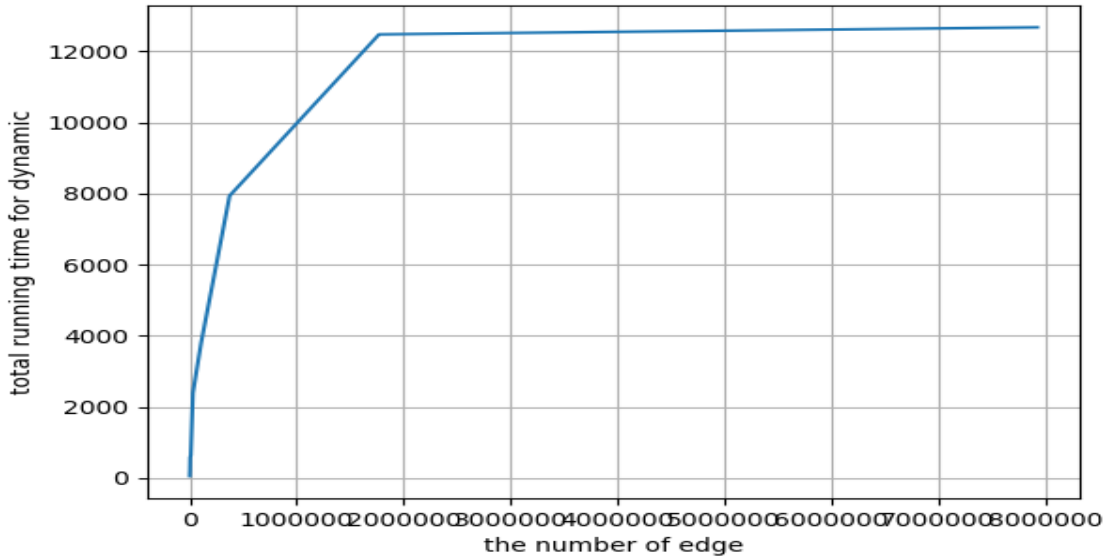
behavior of real situation.



<Figure1. Static Computation Plot>

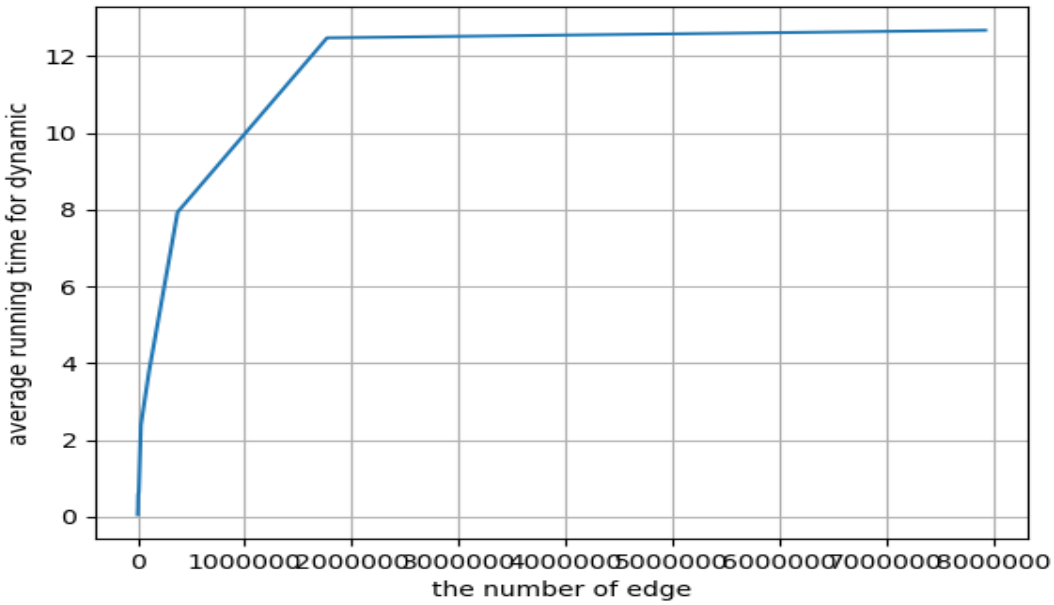
d) Dynamic Computation Plot

My Big O for dynamic computation is V. Thus, I expect linear graph. But the result is different with my expectation. Locally, the graph (figure2) shows linearity, however slope of the graph becomes decreased, as the number of edges increase. Even, last section of graph (from rmat1517 to rmat1618) shows almost same running time. My conclusion is that as the number of edges increases, time complexity for dynamic programming slowly increase (based on figure2, running time would be converged to about 12000). And theoretical time complexity is not exactly matched with real situation.



<Figure2. Dynamic Total Running Time Plot>

Now, I show average recompute time's graph to check it would match theoretical time complexity (O(V)). I calculate average time by total recompute time/1000.



<Figure3. Dynamic Average Running Time Plot>

In any case (13 data), number of change is 1000, so average running time is just calculated by total running time / 1000. Thus, average running time graph (figure3) shows same trend with total running time graph (figure2).