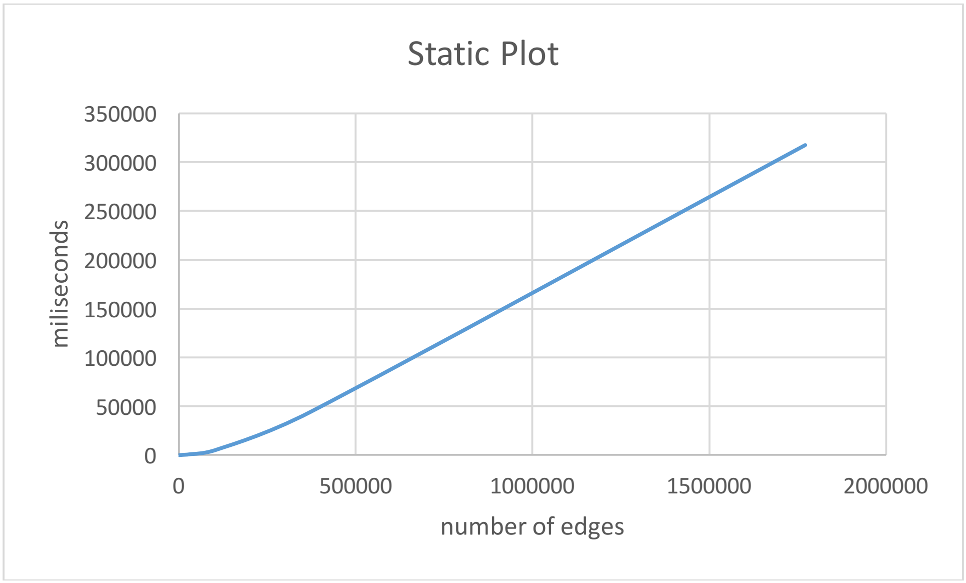
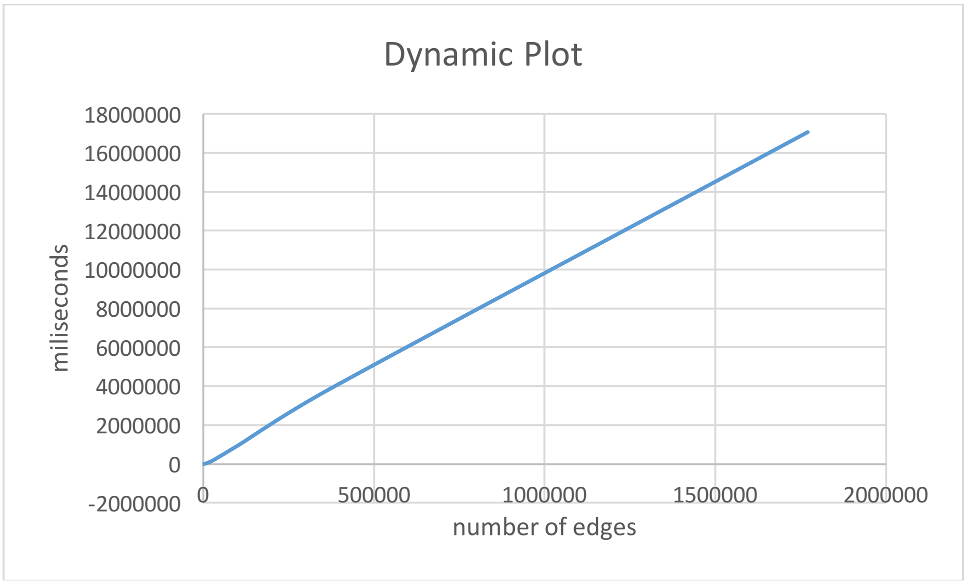
Report

a) I chose the Kruskal's algorithm. The Kruskal's algorithm is focusing on adding edge that both ends belong to different vertices set, which can effectively avoid cycle in the graph. The theoretical complexity for this algorithm should be |E|log|V|. In my computeMST function, I have a outer while loop and it reaches ending condition when the new MST graph size is equal to the graph size as indicated in the input graph file. So this could be O(|V|) in worst case. Then I have a checkSet function which is used to find which set the vertices belong to. I used a hashmap to store the sets. The key is set number and value is hashset. The worst case is that each hashset only contains a pair of nodes so that traversal could be 0.5O(|V|) but on average should be log|V| because the sets will join together once they have an edge connecting them. For the set union, on average, it should be log|V|. So in general, the total time complexity should be O|V| * log|V|. Here, the |E| is actually equal to |V|-1 in the MST so they can interchange each other.
As for my recomputeMST part, I added new edge to the current MST that is saved in previous step and apply the computeMST again on it. In the first run of computeMST, the time complexity could be much longer because the initial graph input is in very large scale. Later on, the input graph is the MST which is optimum so the running time should be about Vlog|V|.

Implementation explanation:
I build a Graph class and Edge class in java to help store the edges and graph. The data structure for graph is a hashmap containing hashmap. For outside hashmap, the key is smaller number of node and value is hashmap. For inner hashmap, the key is larger number node and value is weight. The inner hashmap "serves" like an adjacency list which can help me get any edge in the graph in O(1). When implementing the MST, I used priority queue imported from java library. The inner data structure is a hashset and inside of the hashset is the Edge class. The priority queue is implemented in ascending weight order. In the RunExperiments.java, I have several class variables such as Graph G and Graph mst PriorityQueue pq so that I can modify them without calling them into the argument.



Discussion:
The static plot behaves like a linear plot which is not corresponding to the time complexity analysis. There could be several reasons potentially affecting the result. First, there are 12 cases result. If we run more groups of edges, the result may be different. Second, the implementation could not be as efficient as theoretical Kruskal's Algorithm. In the parseEdges part, the complexity is O(n) where n is equal to total number of edges. While in computeMST, it actually reduced amount of edges used since there are many duplicates. This will slightly dominate a little bit of the linear behavior. Also, in my implementation of computeMST, for the stopping condition of the while loop, I stop the function when size of MST is equal to number of graph nodes. Randomly, it is highly possible the program will not run until the last element of priority queue. So the program will likely to terminate earlier than theory.



Discussion:
As I did recomputeMST based on the saved MST and new edge using computeMST, the recomputeMST should have same theoretical complexity of computeMST. This plot behaves also like a linear plot. First, it would be better if the running cases are more. Second, since this method is basically same with computeMST, it also behaves like a linear line. As mentioned in last discussion, the while loop in the computeMST could terminate earlier. Majorly, it just behaves like computeMST in my implementation.