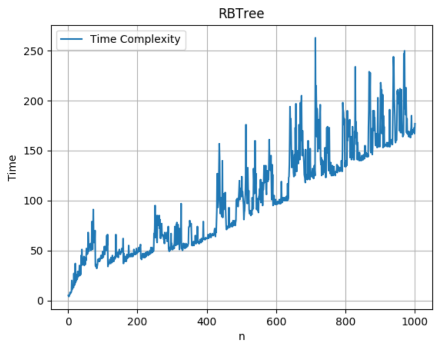
HW4

Miguel A. Montoya Z. (A01226045)

October 28, 2017

1. Implement RB-Trees insert

   Insertions in a RB Tree takes $O(lg(n))$.
   The following graph show the insertion of n elements to a RB Tree. The curve is similar to the theoretical complexity.

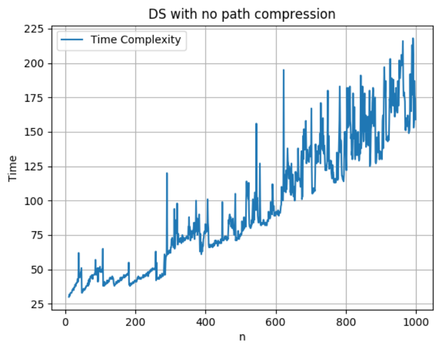
   

   The code used is annexed in **RBTree.cpp**

2. Implement Disjoint Set Representation

   (a) Weighted Union by Rank

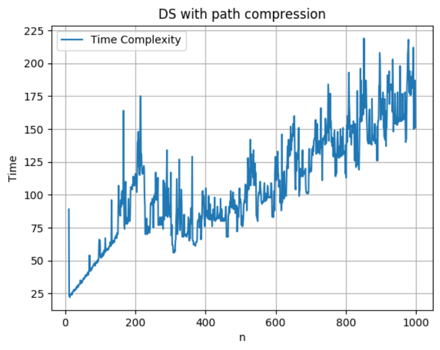
Weighted Union by Rank mixed with regular Find-Set operations takes $O(m + n\,lg(n))$. Where m are the number of operations and n the number of singletons. The following graph represents the time taken to execute this mix of operations. It resembles roughly to the complexity specified above.



The code used is annexed in **DisjointSet.cpp**

(b) Union by Rank + Path Compression

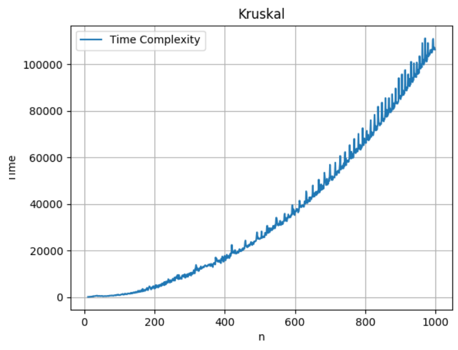
Union by Rank mixed with Find-Set with path compression operations takes $O(m + n\,lg(n))$. Where m are the number of operations and n the number of singletons.
The following graph represents the time taken to execute this mix of operations. This implementation gives a smaller execution time at end values and resembles even better the theoretical time complexity. The only downside is a highest complexity with small n's.



The code used is annexed in **DisjointSet.cpp**

3. Implement Kruskal

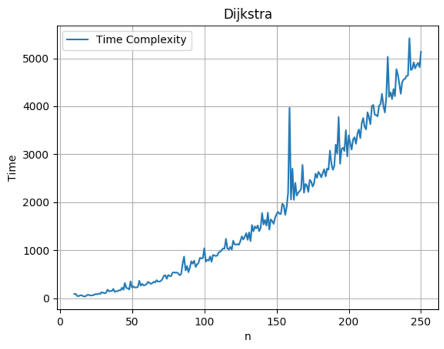
Kruskal algorithm takes $O(E\,log(E))$.
The following graph represents the time taken to execute this algorithm in a randomly generated complete graph with E edges. The implementation proves fully the theoretical time complexity of the Kruskal algorithm.



The code used is annexed in **Kruskal.cpp**

4. Implement Dijkstra

   Dijkstra algorithm takes $O(E + V\,lg(V))$.
   The following graph represents the time taken to execute this algorithm in a randomly generated complete graph with E edges, only if the given alpha is bigger than generated alpha. The implementation resembles the theoretical time complexity of the Dijkstra algorithm.



   The code used is annexed in **Dijkstra.cpp**

5. From Cormen

   (a) 21-3 (Tarjan's off-line least-common-ancestors algorithm)

      i. Argue that line 10 executes exactly once for each pair.

         Because when running on v, u isn't BLACK, so it won't execute the print. When we now execute on u, v was set BLACK previously, and the print will be executed.

ii. Argue that at the time of the call LCA(u), the number of sets in the disjoint-set data structure equal the depth of u in T.

At the start of LCA, u has a depth d, and there are d items. At the start of LCA, it increases to $d + 1$ disjoints sets. When calling again LCA, $d + 1$ is the depth of v (Inside making another set, and a union, leaving the number of sets as $d + 1$). The depth calculation holds for any run of LCA.

iii. Prove that LCA correctly prints the least common ancestor of u and v for each pair.

Because when running on v, all of it's ancestors will point to the LCA. u now will start the same process, but when it finds a BLACK node, it means we must return that node's ancestor (w).

iv. Analyze the running time of LCA, assuming that we use the implementation of the disjoint-set data structure.

Most lines takes constant time.
The first for loop takes $\mathcal{O}(|T| \, \alpha \, |T|)$, and the second takes $\mathcal{O}(|P|)$. Making the total $\mathcal{O}(|T| \, \alpha \, |T| + |P|)$.

(b) 23-4 (Alternative minimum-spanning-tree algorithms)

i. MAYBE-MST-A(G, w)

The algorithm returns a MST, because it never removes an edge that must be in the MST. An edge e it's only removed if it takes weight off and serves only as a bridge.
First, the edges are sorted in $\mathcal{O}(E \; lg(E))$. To check if e is a bridge, a DFS is executed in $\mathcal{O}(V + E)$. Giving a general total time of $\mathcal{O}(E^2)$

ii. MAYBE-MST-B(G, w)

The algorithm doesn't return a MST, because it can take the edges in such an order that all the lightest edges between two vertex serve only as a bridge (Making a cycle), so those wouldn't be taken into account.
Using disjoint sets to store the connected nodes. If an union within the same subset is made, a cycle would be created. This need V calls to Make-set and E call to both find and union, giving $\mathcal{O}(V \alpha E)$.

iii. MAYBE-MST-C(G, w)

The algorithm returns a MST, because if adding an edge that results in a cycle, a deletion of the heaviest edge in the cycle is made, to keep the characteristics of the MST.
Taking the previous answer, we just need to add a method to find the maximum weight edge on a cycle. Using a DFS to find the cycle and max edge could be implemented. Giving a running time of $\mathcal{O}(VE)$.

(c) 24-3 (Arbitrage)

i. Give an algorithm to determine if a sequence exists.

By negation the logarithm of each entry in the conversion table, and applying Bellman Ford to look for negative cycles. if the algorithm return true, this means there exists a sequence.

ii. Give an algorithm to print such sequence if exists.

Executing relaxation $V - 1$ times like Bellman Ford and storing the d values of each node. Executing the relaxation V more times and looking for the nodes that decreased in value. After that, using Disjoint Sets we can determine the exacts steps of the cycle.