CSE 6140 Algorithms Assignment 2

Yashovardhan Jallan GTID: yjallan3

October 1st, 2018

Preamble

(a) List of People you worked with:
Somdut Roy

(b) Sources used:
http://www.cs.cmu.edu/afs/cs/academic/class/15210-s14/www/lectures/MST.pdf
https://www.ics.uci.edu/ eppstein/161/960206.html
https://courses.engr.illinois.edu/cs374/sp2017/labs/solutions/lab12-sol.pdf
http://www.cim.mcgill.ca/ langer/251/E8-MST.pdf
https://www.arl.wustl.edu/ jst/cse/542/exams/ex1sol.pdf

1 Question - 1 : Simple Complexity

(a) (a) $f = \Theta(g)$
(b) $f = \Theta(g)$
(c) $f = \Omega(g)$
(d) $f = O(g)$
(e) $f = \Omega(g)$
(f) $f = O(g)$

(b) Big-O Complexity of the Algorithm given:

```
Data: n
i = 1;
while i ≤ n do
    j = 0;
    k = i;
    while k > 0 do
        k = k/3 ;
        j + + ;
    end
    print i, j;
    i + +;
end
```

For each value of $i <= n$, k is initialized as i and it enters another while loop between lines 5-8 where the value of k is divided by 3 and its floor is assigned to k. j counts the number of steps before the value of k becomes zero and the while loop ends.

The output of the program is of the format :
$(i, \lfloor \log_3 i \rfloor + 1)$...$\forall$ i=1 *to* n.

For example, when $n = 100$, the output is
1 1
2 1
3 2
4 2
.
.
.
99 5
100 5

The number of computations in this entire loop is:
$\sum_{i=1}^{n} (2(\lfloor \log_3 k \rfloor + 1) + 3)$
= On solving, we get $O(log(n!))$

2 Question - 2 : Greedy 1

Let us visualize the leaks as points $l_1, l_2, l_3, ...l_n$ lying along a straight line, where l_n is the location of the n^{th} leak. Without loss of generality suppose the strips are given in sorted order; i.e., $\forall i < j, l_i < l_j$. Let the length of the strip be s. The greedy algorithm is as follows.

1. Start with $i = 1$
2. Lay the first strip at point l_i. This will cover all leaks $l_i...l_j$ in the interval $[l_i, l_i + s]$ where $l_j \leq l_i + s$.
3. Update $i = j + 1$
4. Repeat steps 2 and 3 until $i > n$ (all leaks have been covered)

The algorithm has a run-time complexity of $O(n)$. Note: if you assume that the strips are not sorted and your algorithm sorts them initially, the total complexity is then $O(nlog(n))$.

Let the solution obtained by the greedy algorithm be represented as $G = \{G_1, G_2..., G_t\}$, where G_i is a point on the line/strip representing the location of the starting point of the strip covering the interval on the pipe from $[G_i, G_i + s]$.

Without loss of generality, we say that for $\forall i < j$, we have $G_i \leq G_j$, i.e., that the solution is given by strips placed in order from left to right. Using a similar notation, the optimal solution will be represented as $O = \{O_1, O_2, ...O_r\}$.

Greedy Choice Property

Now O_1 lies from point $[O_1, O_1 + s]$ and must cover the first leak, l_1 (in order to be a feasible solution, it must cover all the leaks). By definition of the concept of a greedy choice, $G_1 = l_1$, since the greedy solution places the first strip as far right as possible while still covering l_1, it follows that $O_1 \leq G_1$.
Now let $O^{'}$ be the solution obtained by replacing O_1 with G_1 in O. Since l_1 is the first and leftmost leak, we know that there is no other leak lying in the space from $[O_1, G_1)$. Consequently, we see that strip at G_1 covers all the leaks that strip at O_1 covers. Hence, $O^{'} = \{G_1, O_2, ...O_r\}$ is also an optimal solution.

Optimal Substructure Property

Let the optimal solution to the entire problem P be O. After placing the first strip at O_1, we are left with the task of solving the problem $P^{'}$, which is to cover all leaks to the right of the point O_1+s. G_1 is the first point by Greedy algorithm and it is no worse than O_1. Let the optimal solution to $P^{'}$ be $O^{'}$. By induction, we can prove that $G^{'}$ is no worse than $O^{'}$. Since, $cost(P) = cost(P^{'}) + 1(\text{strip})$, the globally optimum solution O, contains the solution $O^{'}$. Hence by optimal substructure property, we can say that the greedy algorithm is correct.

3 Question - 3 : Greedy 2

Solution

First, we need to sort the minerals based on their $\frac{Value}{Weight}$ ratio. Let the minerals be sorted such as $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq ....\frac{v_n}{w_n}$. The solution of this problem is to choose the maximum amount of mineral 1 (having the highest $\frac{Value}{Weight}$ ratio) and move onto mineral 2 and so on, until we reach the limit of the bag L. The time complexity of this approach is dominated by the sorting which is $O(nlog(n))$.

Exchange Argument

Let the solution obtained by the greedy algorithm be represented as $G = \{G_1, G_2..., G_n\}$, where G_i is the weight of the mineral i chosen as per the solution described above. Using a similar notation, the optimal solution will be represented as $O = \{O_1, O_2, ...O_n\}$.

The goal of the problem was to maximize the value of the bag. Let us consider the specific case of mineral 1. We can say that $G_1 * \frac{v_1}{w_1} \geq O_1 * \frac{v_1}{w_1}$. This is because the greedy choice was made to choose the maximum weight G_1

possible. To replace O_1 by G_1 in the optimal solution, we will need to make sure that the weight limit of the bag is not exceeded. If $G_1 > O_1$, the weight of the bag will increase by $G_1 - O_1$. So, we need to reduce the weight of the all the other minerals such that overall weight limit of L is maintained. Let the new weights be $\{O_2^{'}, O_3^{'}....O_n^{'}\}$. We can say for sure that the solution $O = \{G_1, O_2^{'}, ...O_n^{'}\}$ is still optimal as the overall value of the bag has not decreased for sure. This is because whatever value is lost by reducing the weights $\{O_2, O_3....O_n\}$ to $\{O_2^{'}, O_3^{'}....O_n^{'}\}$, is compensated by changing O_1 to G_1. Similarly, if we keep exchanging G_i and O_i pairs and compare them, we find our Greedy solution is always as good as the optimal solution. Hence, by exchange argument, we can say that Greedy solution is an optimal solution.

4 Question - 4 : Programming Assignment

For the programming assignment, I have used Python 3.6. I have chosen to implement the Kruskal's algorithm to find the minimum spanning tree. For each graph G, after computing the Minimum Spanning Tree T, the new MST by adding an edge $\{e\}$ is found using only $T \cup \{e\}$ edges. This is proven below:

Proof: Let there be a graph G which has a MST T using Kruskal's Algorithm. It can be shown that if an edge e is added to G, the new MST T' can be obtained using $T \cup \{e\}$ edges only.

First, we start with the original solution using the Kruskal's Algorithm, i.e MST T for the graph G. Let this be denoted by $\{e_1, e_2, ..., e_{n-1}\}$. Now when a new edge e is added, the new graph is $G' = G \cup \{e\}$.

Let us assume the edges are now sorted by their weight such i is the largest number such that weight(e_i)< weight(e). So, when we run Kruskal's algorithm on the new graph G', all the edges up until $\{e_1, e_2, ..., e_i\}$ will remain same as before. After this step, there are two possibilities,

1. Kruskal does not choose e as a component of the new MST T': In this case, the new tree T' will be same as T, and hence we could have started with G'=T in the algorithm. Hence, $T' \in T \cup \{e\}$.

2. Kruskal chooses e as a component of the new MST T': In this scenario, there can be two sub-cases.

   (a) In the set of edges, having followed the sequence of edges $< e_j >$ till $j = k$ like $\{e_1, e_2, ..., e_i, e, e_{i+1}, ..., e_k\}$, then instead of using e_{k+1}, we use another edge e' from outside of T. Now if that were to be used, it would have been part of the solution when T was computed. Therefore this scenario has a contradiction.

   (b) e_{k+1} is not used after we include e in T'. This is a possibility. It will still have the same list of sorted edges as T after appending the edge

$e_{k+1} to T'$. Therefore, T' will look like $\{e_1, e_2, ..., e_i, e, e_{i+1}, ..., e_k, e_{k+1}, ..., e_n\}$, such that the new T' is improved by weight(removed edge) - weight(e). This case also agrees with $T' \in T \cup \{e\}$.

So, in all cases, we can say that the new MST using recomputeMST can be found using new edge appended to the previous MST. **End Proof.**

Reason to choose Kruskal, Space/Time Complexity and Data Structures used

Now that we have proved the above, all the subsequent re-computations of the Minimum Spanning Tree have close to V edges, where V is the number of vertices. Hence, the new graphs are very sparse as $|V| << |V|^2$. The choice was made to use Kruskal's algorithm because for a graph with V vertices and E edges, it runs in O(E log V) time. In our case, this is similar to O(V log V) time for all the re-computations which is better than Prim. Prim's algorithm performs better in a dense graph which has a lots of edges. It can run in O(E + V log V) time, if coded efficiently.

The time complexity of my code is dominated by the sort operation in computeMST function which is $O(Elog(V))$. The way I have coded it, the recomputeMST function simply append the new edge e to the previous MST and then calls the computeMST function, and hence just by itself is of $O(1)$, and including computeMST is of $O(Elog(V))$.

The data structures used for the assignment are pretty simple. I initially experimented with NetworkX but realized that I was creating an inefficient code by first reading everything into NetworkX graph class format and extracting all information back into python list to sort. So, I ditched networkX and simply used a list of tuples of the form $[(w_1, u_1, v_1), ((w_2, u_2, v_2), ..., (w_e, u_e, v_e)]$ to store my graph. The space complexity of this is $O(E)$ where E is the total number of edges. Additionally, I am using Python dictionary data-structure to implement the Union-find algorithm. This has a space complexity of $O(V)$ where V is the total number of vertices. Overall, the space complexity of my algorithm is $O(E)$.

Plots for Static Computation

My algorithm seems to be very efficient from a run-time perspective. For the largest graph rmat1618, the static computation takes 10 seconds in python. By comparison, inbuilt Minimum Spanning Tree algorithm of NetworkX takes 29 seconds (I checked this to compare). For the subsequent dynamic computations, the run times are much less than 1 second for all the graphs. To run all the graphs, with all the dynamic re-computations, it shouldn't take more than 5-10 mins.

Please see the plot of run-time versus number of edges here. A lot of the time

values that I obtained for smaller graphs are zero because they are smaller than machine precision and are reported as zero. Only for the largest 4-5 graphs, I received some non-zero values.

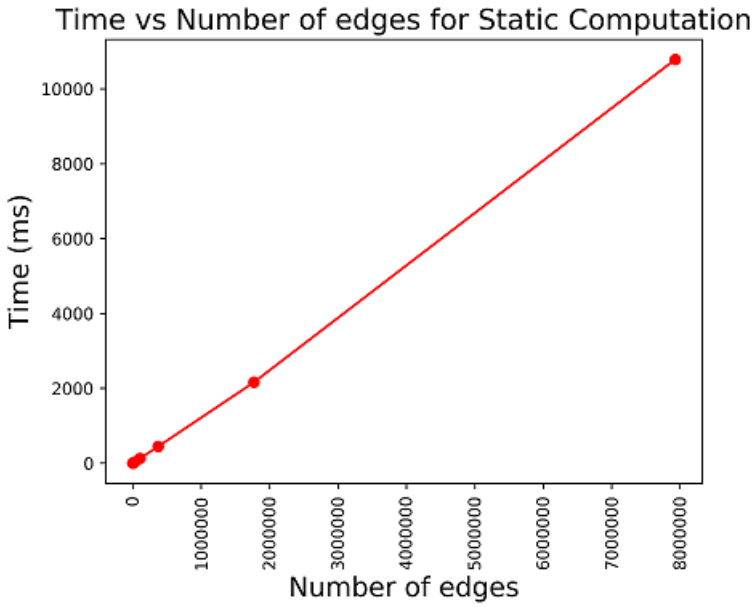

Figure 1: Plot of Time vs Number of Edges (normal scaled)

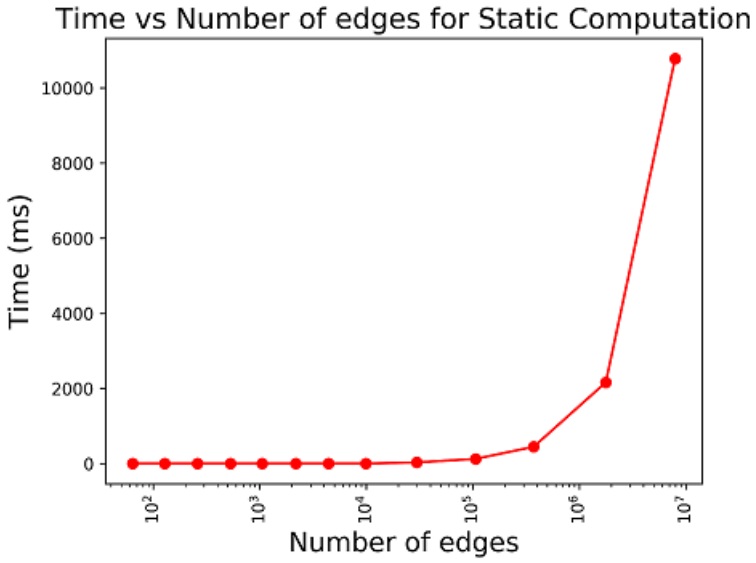

Figure 2: Plot of Time vs Number of Edges (log-scaled)

Plots for Dynamic Re-Computation

Please find the plots for Dynamic re-computation for some of the graphs here. As you can see from Figure 3., for rmat1618 that the time for the static computation is close to 10,000 millisecond or 10 seconds. But, the dynamic computation is only a very small fraction of it. Similar characteristic is observed for other graphs shown in Figure 4-6. I have only included plots for the largest four graphs because all the other graphs have their times less than my machine precision and it shows zero.

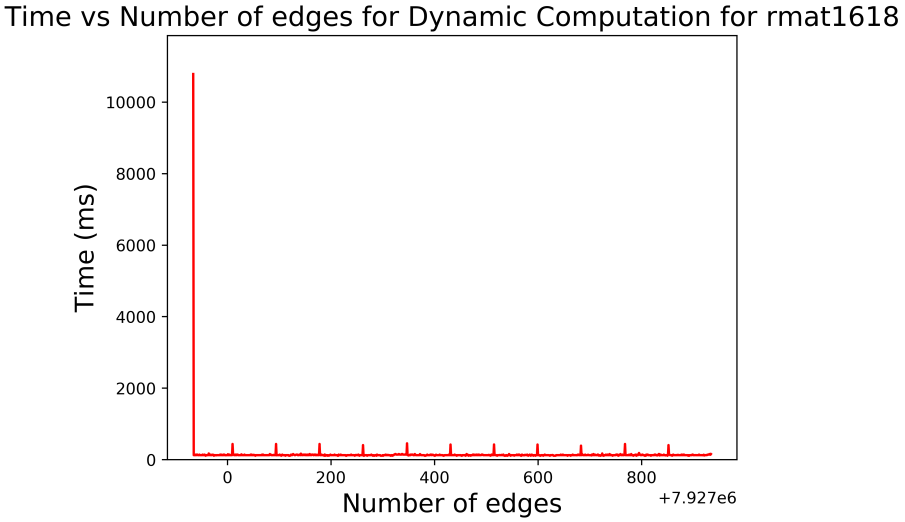

Figure 3: Plot of Time vs Additional Edges Added

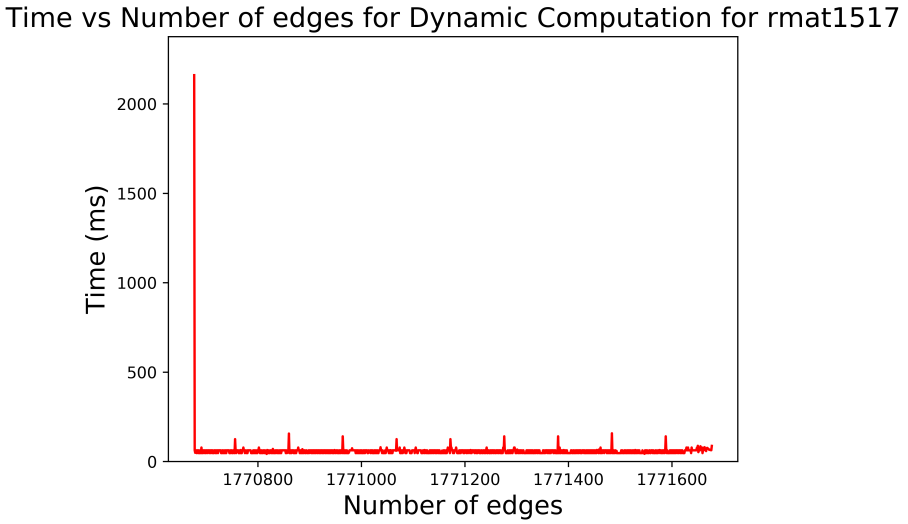

Figure 4: Plot of Time vs Additional Edges Added

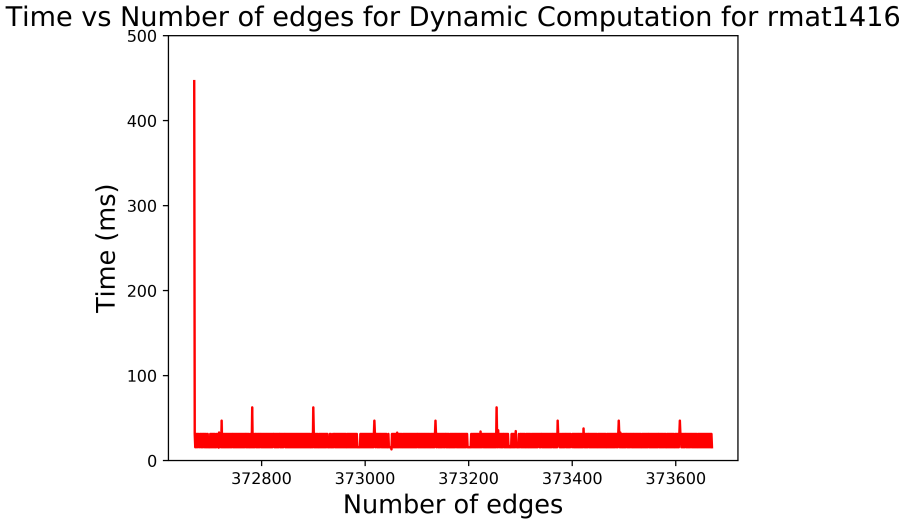

Figure 5: Plot of Time vs Additional Edges Added

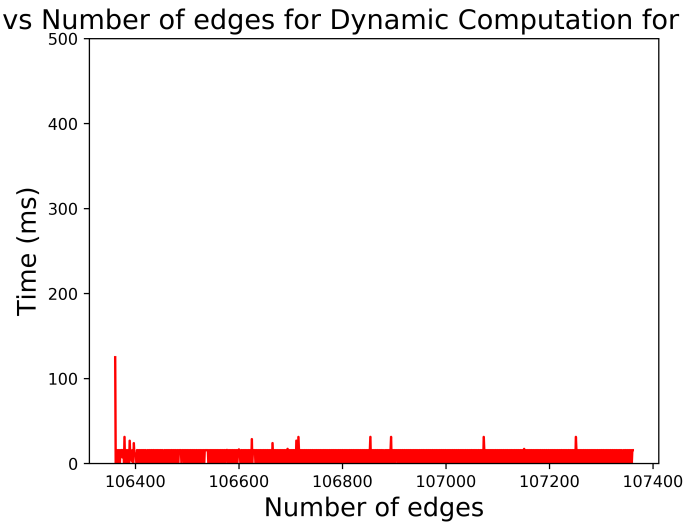

Figure 6: Plot of Time vs Additional Edges Added