CSE 6140 - Homework 1

Alexander Winkles

For assignment 1, I worked with Willian Kong on several problems. I utilized the class textbooks (Kleinberg, CLRS, and Benoit) as well as the website "GeeksForGeeks" for different perspectives on designing the algorithms. Overall, I enjoyed this assignment. However, I still feel uncomfortable proving optimality of algorithms, so more practice and examples with that would be appreciated! Additionally, I think the size of the graph files for the final problem were a bit too large. Once the first few graphs were solved, I think it became unnecessary to keep analyzing more, as we knew our algorithm worked.

Problem 1

(a) Let $k \geq \lceil \log n \rceil$. We wish to design an algorithm of complexity $O(\log n)$. This suggests the idea of an algorithm similar to the methodology behind the Bisection Method.

Algorithm 1 Bisection Search for Box Dropping

```
1: procedure BisectionSearch(l, n)
2:     j ← n                                   ▷ Highest Floor
3:     i ← l                                   ▷ Lowest Floor
4:     m ← (i+j)/2
5:     if m = break then                       ▷ Checks if box breaks on floor m
6:         BisectionSearch(1, m − 1)
7:     else
8:         BisectionSearch(m, n)
       return m
```

The nature of the algorithm guarentees that this algorithm will take at most $\log n$ steps, so we will not run out of boxes.

(b) Let $k < \lceil \log n \rceil$. We wish to design an algoirthm of complexity $O(k + \frac{n}{2^{k-1}})$. To accomplish this, we will use the bisection search from above, but only using $k-1$ boxes rather than k. The last box will then search the remaining interval by going up each floor and throwing the final box off. Because of this methodology, there will be at most $n/2^{k-1}$ floors in the interval, leading to the desired complexity of $O(k + n/2^{k-1})$.

(c) Let $k = 2$. We wish to design an algorithm of complexity $O(\sqrt{n})$.

Algorithm 2 Box Dropping for $k = 2$

```
1: procedure FloorSearch(n)
2:     m ← 0                                   ▷ Initializes maximum floor
3:     Nstep ← √n                              ▷ Number for floors to iterate over
4:     for i = 1 : n : Nstep do
5:
6:         if i ≠ break then
7:             m ← i
8:             break
9:     for i = m + 1 : m + √n do
10:
11:        if i ≠ break then
12:            m ← i
13:            break
       return m
```

Problem 2

Algorithm 3 Greedy 1

```
1: procedure MinimizeStrips(C, n)
2:     m ← 1                                              ▷ number of strips needed
3:     Sort nodes c_i from leftmost to rightmost so that c_1 < c_2 < ... < c_n
4:     minNode ← c_1
5:     for i = 1 : n do
6:         if c_i − minNode > 9 then
7:             m ← m + 1
8:             minNode ← c_i
9:     return m
```

Essentially, this algorithm works by sorting all the nodes ("holes") from left to right. From here, it starts at the leftmost strip then adds other nodes that will fit within the 9 inches of strip. Once all the nodes for a strip are filled, a new strip is added to the next node and the process is repeated. Given that this has a single for loop, its time complexity is $T(n) \in O(n)$.

Consider the subproblem of several nodes clustered together in such a way that there is no more than 9 inches between the leftmost and rightmost nodes. With our greedy choice, this subproblem will be covered completely by a single strip, which is trivially optimal. Thus, as the algorithm will treat all such subproblems this way, we can inductively see that the algorithm is optimal.

Problem 3

Algorithm 4 Greedy 2

```
1: procedure MaximizeValue(L, M)
2:     weightUsed ← 0
3:     B ← ∅
4:     sort minerals m_i by v_i/w_i from highest value to lowest
5:     for m_i ∈ M do
6:         if weightUsed + w_i ≤ L then
7:             B = B ∪ {m_i}
8:             weightUsed + w_i
9:             M = M − {m_i}
10:            break
11:    if weightUsed < L then
12:        pick the mineral next in line in M and call it m, with weight of w and value of v
13:        B = B ∪ {m * w/(L − weightUsed)}
14:    return B
```

In order for this algorithm to be optimal, the bag must be completely filled in order to maximize profits.

Theorem 1. *The bag has been completely filled by the algorithm.*

Proof. This result is easy to see. At the end of the for loop, there are two possibilities:

(a) weightUsed $= L$

(b) weightUsed $< L$

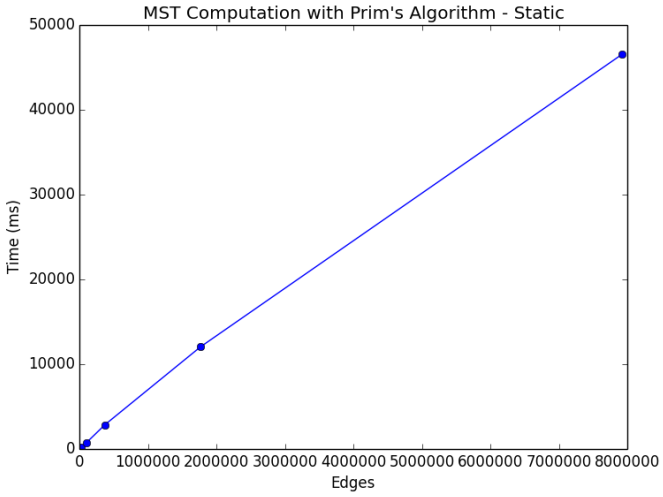
If it is the first case, we are done. However, if it is the second case, then we can fit $L -$ weightUsed amount into the bag. Our algorithm provides this by adding a portion of the next mineral scaled with weight $L -$ weightUsed, so after this our bag is again full. This completes our proof. □

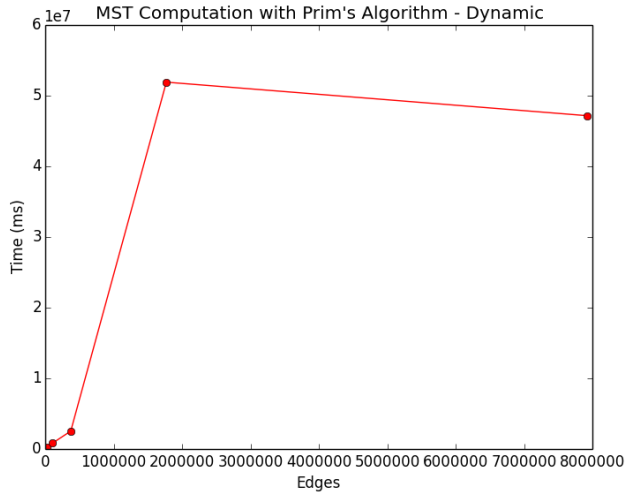
Theorem 2. *The greedy algorithm is optimal.*

Proof. Consider an optimal algorithm $\mathcal{O}$. Our goal is to use an exchange argument to show that the greedy solution A is optimal. Arrange the minerals in both as such in the algorithm, so m_i is first if and only if v_i/w_i is largest. Consider every mineral o_i in $\mathcal{O}$ that differs from a_i in A. We can exchange this mineral with the corresponding mineral in A. If they are of different weights, we can instead exchange it for the appropriate portion of a_i. By definition of the greedy algorithm, $v_{a_i} \geq v_{o_i}$, where the equality holds otherwise $\mathcal{O}$ is not optimal. By continuously exchanging minerals using the above procedure, we arrive at an optimal solution no worse that the one we started with that is identical to A, so thus A must be optimal. □

Problem 4

For this problem, I chose to implement Prim's algorithm in Python. I chose this problem because it was the less intuitive of the two for me, so I thought that writing code for it would help me gain a better understanding. Rather than designing a second algorithm for recomputeMST, I chose to instead append the new edge to my graph then run computeMST again. Because I implemented a priority queue utilizing Python's heapq package, my complexity should theoretically be $O(m \log n)$, where n is the number of nodes and m is the number of edges.





From the graphs above, the static computation appears to behave linearly with respect to the edges. The dynamic, on the other hand, appears more akin to $m \log n$, which is expected given the theoretical complexity of Prim's algorithm. I assume the difference in the graphs can be justified by the addition of edges during the dynamic process. Since occasionally new edges are added to the graph, this increases the time it would take to calculate a single instance of Prim's, whihc leads to an overall increase in the time required to run the computation.