Computer Science 180, Homework 4

Michael Wu
UID: 404751542

February 8th, 2018

Chapter 4, Problem 3

Assume that there is some optimal packing of packages S such that we can ship all the packages using n trucks. Then we can transform this packing into another optimal packing S' produced by our greedy packing algorithm. Let a truck that cannot hold the next package i in the mailing sequence, either because doing so would exceed the weight limit W or because no more packages remain, be called a full truck. Our greedy packing algorithm generates a sequence of full trucks. Now we will show that S can be transformed into a sequence of full trucks as well. If S does not assign the first truck t_1 enough packages such that it is a full truck, then we can remove the next package i from the following truck t_2 and place it in t_1. This is a new optimal packing, as t_2 now holds w_i less weight, and t_1 does not exceed the weight limit because it was not full prior to loading package i. We continue moving packages from t_2 to t_1 until t_1 is full. The same number of trucks n are used, but our packing now starts with a full truck. So any optimal packing S can be modified into another optimal packing that starts with a full truck. Looking at the remaining $n-1$ trucks from t_2 to t_n, we have another optimal packing of the remaining packages $S - \{\text{packages in } t_1\}$, since S is originally an optimal packing. We can make the first truck in this packing full as well. Repeating this procedure until we reach the last truck, we see that we have created a packing S' consisting entirely of full trucks that uses the same number n trucks as S. Thus our greedy algorithm, which generates S', generates an optimal solution.

Chapter 4, Problem 7

An algorithm that generates a schedule with the shortest completion time is scheduling the shortest finishing time last. This algorithm can be implemented in $O(n \log n)$ time using merge sort. The psuedocode is as follows

```
Merge Sort list of jobs L by the finish time
Initialize empty schedule S
for i from 0 to (length of L)-1
    S[(length of L)-1-i]=L[i]
return S
```

This is polynomial time, as the most expensive operation is the sorting. Otherwise, it only traverses through the list of jobs once.

Before we prove that this algorithm is correct, note that the shortest time on the supercomputer is $\sum p_i$, because all the jobs must be processed sequentially. Thus the completion time must be at least $\sum p_i$. We want show that any optimal solution S that has a minimum completion time t_a can contain the shortest finishing time job j_s last. If the shortest finishing time job j_s is not last, then swapping it with the current last job j_l cannot increase the completion time. Note that t_a must be at least as much as the preprocessing time plus the finishing time f_l of the last job, so $t_a \geq \sum p_i + f_l$. Let $\sum p_i + f_l$ be called the completion time t_l of the job j_l. If we swap j_s and j_l, the completion time of j_s will be $\sum p_i + f_s$. We have that $t_a \geq \sum p_i + f_l \geq \sum p_i + f_s$ because $f_l > f_s$, so the total completion time is not increased by positioning the shortest finishing time last. The total completion time is also not increased by moving the job j_l backwards, because it will complete in $t_l - \delta$ time, where δ is a positive amount of time it is moved backwards. Thus, $t_a \geq t_l > t_l - \delta$ and so any optimal solution S can have its shortest finishing time job j_s switched into the last position and remain an optimal solution.

Next we consider the set of all jobs minus the shortest completion time job. We order these jobs such that we obtain another optimal solution for this subset of jobs with a completion time t_b. Our original optimal completion time is t_a, so we know that we can complete this subset of jobs within time t_a, so $t_b \leq t_a$. For this subset we can also switch the shortest finishing time job into the last position without increasing the completion time t_b. Continuing to do this for every subset of decreasing size, we obtain a list of jobs ordered in decreasing order of finishing times f_i, such that the total completion time is less than or equal to t_a. Thus our optimal solution S has the same completion time t_a as a schedule generated by the shortest finishing time last algorithm, proving that our algorithm generates an optimal solution.

Chapter 4, Problem 12

a) This is false. Consider the streams

$$(b_1, t_1) = (10, 10), \qquad (b_2, t_2) = (50, 1)$$

with $r = 10$. Then $b_2 \nleq rt_2$, but at $t = 10$ we have sent over $10 < rt$ bits, and at $t = 11$ we have sent over $60 < rt$ bits. We have a valid schedule where $b_i \nleq rt_i$ is not true for all cases.

b) An algorithm to evaluate whether there exists a valid schedule is to check the expression

$$\sum_{i=1}^{n} b_i \leq r \sum_{i=1}^{n} t_i$$

If this is true, then a valid schedule exists. If it is false then a valid schedule does not exist. This algorithm runs in $O(n)$, as it simply requires a sum over the given streams.

If this expression is false, we prove that a valid schedule does not exist by definition. Our total bits sent is $\sum_{i=1}^{n} b_i$, and our finish time is $\sum_{i=1}^{n} t_i$. Thus if $\sum_{i=1}^{n} b_i \nleq r \sum_{i=1}^{n} t_i$, our constraint must be violated by definition at the end of our schedule because the total number of bits we sent over the time interval from 0 to $\sum_{i=1}^{n} t_i$ exceeds $r \sum_{i=1}^{n} t_i$.

If $\sum_{i=1}^{n} b_i \leq r \sum_{i=1}^{n} t_i$, we can always generate a valid schedule. Let the bitrate of a stream (b_i, t_i) be $h = \frac{b_i}{t_i}$. Sending our streams in order of increasing bitrate ensures that we have a valid schedule. We prove this by first defining our average bitrate after m streams as

$$h_m = \frac{\sum_{i=1}^{m} b_i}{\sum_{i=1}^{m} t_i}$$

We wish to show that $h_m \leq h_{m+1} \leq h_n$ when we order our bitrates in increasing order, because this means that we never exceed our constraint and thus we will have a valid schedule since $h_n \leq r$. We have that

$$h_{m+1} = \frac{\sum_{i=1}^{m+1} b_i}{\sum_{i=1}^{m+1} t_i} = \frac{\sum_{i=1}^{m} b_i + b_{m+1}}{\sum_{i=1}^{m} t_i + t_{m+1}}$$

Since we order by increasing bitrate, the stream $m + 1$ has a bitrate $\frac{b_{m+1}}{t_{m+1}}$ which is greater than any previous stream's bitrate. Thus, $\frac{b_{m+1}}{t_{m+1}} \geq h_m$ because

h_m is an average of the previous stream's bitrates. Thus our numerator in h_{m+1} increases by a magnitude more than $h_m t_{m+1}$, so $h_{m+1} \geq h_m$. Thus we never exceed our constraint and we can generate a valid schedule.

Chapter 4, Problem 16

An efficient algorithm to decide whether the association exists is as follows

```
I = sorted list of intervals by end time t_i+e_i
X = sorted list of event times x_i
for i from 0 to n-1
    for j from 0 to n-1
        if X[i] in interval I[j] and I[j] is not assigned
            assign X[i] to I[j]
            break
    if X[i] is not assigned
        return no
return yes
```

Essentially we sort the approximate time stamps by the value $t_i + e_i$, which is the end time of the margin of error interval. We then sort the event times, and choose the earliest event time x_1. We traverse the list of sorted time stamps and assign the event to the approximate time stamp that contains x_1 with the earliest end time. We continue to do this for the remaining events x_i, each time assigning the earliest event to the time stamp that contains x_i with the earliest end time. If we cannot match an event, we return no. Otherwise we have matched all the events to time stamps and we return yes. The runtime of this algorithm is $O(n^2)$, as we can sort in $O(n \log n)$ time and we have two nested loops over n elements.

We prove that this algorithm is correct by noting that if it returns yes, then it must assign an event to every time stamp, so an association must exist.

In the opposite direction we must show that if given a set of time stamps and events that can be associated, our algorithm must return yes. Let there be a valid pairing S of time stamps and events. We will show that for any valid pairing, we can always assign the earliest event with time x_1 to the time stamp (t_a, e_a) that contains x_1 with the earliest end time $t_a + e_a$. If x_1 is not assigned to the time stamp (t_a, e_a) in S, let it be assigned to the timestamp (t_b, e_b) and let x_a be assigned to (t_a, e_a). Then we try to swap x_1 and x_a so that they are assigned to (t_a, e_a) and (t_b, e_b), respectively. We know that x_1 can be assigned to (t_a, e_a), since it contains x_1 by definition. Additionally, we know that $x_1 \leq x_a$, because x_1 is the earliest event. But also $t_b - e_b \leq x_1 \leq x_a$, since (t_b, e_b) contains x_1. Finally, we also have

$x_a \leq t_a + e_a \leq t_b + e_b$, since (t_a, e_a) has an earlier end time than (t_b, e_b). Thus we have

$$t_b - e_b \leq x_a \leq t_b + e_b$$

so x_a can be validly assigned to (t_b, e_b), and we can swap x_1 and x_a. This results in another valid pairing. We then mark x_1 and (t_a, e_b) as permanently assigned, and only consider the remaining events. This forms a subset of our original problem. We continue to assign each leading event x_i to the time stamp that contains x_i with the earliest end time, resulting in a valid pairing S' that matches the pairing generated by our algorithm. So if there exists a valid pairing S, our algorithm will match all the events and time stamps and return yes. Therefore our algorithm is correct.

Chapter 4, Problem 30

Let $Z = (X \cup Y)$, so that the non terminal nodes in our minimum-weight Steiner tree T are contained in the set Y. Call Y the set of Steiner nodes. Observe that any Steiner node a in T must have at least 3 edges connected to it. If a has one edge, we can simply remove a from Y to produce another valid Steiner tree with a lower weight. If a has two edges, we can remove a from Y and draw an edge between the two neighbors of a. This produces another valid Steiner tree with a weight less than or equal to our original Steiner tree, because our graph G obeys the triangle inequality. Thus all $a \in Y$ have a degree of at least 3. T is also connected, so we know that each terminal node in X must have at least degree 1. So the sum of degrees d over T is $d \geq 3|Y| + |X|$. Because our tree T must have $|Z| - 1$ edges, the sum of its degrees is $d = 2|Z| - 2$. Then

$$\begin{aligned}
2|Z| - 2 &\geq 3|Y| + |X| \\
2|X| + 2|Y| - 2 &\geq 3|Y| + |X| \\
|X| - |Y| - 2 &\geq 0 \\
|Y| &\leq |X| - 2
\end{aligned}$$

So the number of Steiner nodes $|Y|$ must be at least two less than the number $k = |X|$ of terminal nodes. Then we can find the minimum weight Steiner tree on X by iterating over every set Y of Steiner nodes in G with the property $|Y| \leq k - 2$ and finding the smallest minimum spanning tree for $Z = (X \cup Y)$. There are

$$\sum_{i=0}^{k-2} \binom{|G - X|}{i} = \sum_{i=0}^{k-2} \binom{n-k}{i}$$

such sets. This sum is bounded by

$$\sum_{i=0}^{k-2} \binom{n-k}{i} \leq \sum_{i=0}^{k-2} n^i = 1 + n + n^2 + \ldots + n^{k-2}$$

Thus the number of sets scales in $O(n^{O(k)})$ time. The minimum spanning tree algorithm can run in $O(n^2)$ time, so multiplying these time complexities together yields a time complexity of $O(n^{O(k)})$.