Homework 4

Zicong Mo

February 8, 2018

3. Assume, for the sake of contradiction, that the greedy algorithm in use does not produce the optimal number of trucks needed. Suppose there exists some optimal solution that differs from the schedule produced by the greedy algorithm.

   Let $0 < m_1 < ... < m_p$ be the last boxes to go on each truck determined by the greedy algorithm. Let $0 < n_1 < ... < n_q$ be the last boxes to go on each truck determined by the optimal solution.

   Because the optimal solution is different than the greedy algorithm, and the greedy algorithm always packs each truck as full as possible before it leaves, it follows that there must be at least one truck where the optimal algorithm sends off a truck before it is filled.

   However, because the greedy algorithm was able to place at least one more box on the truck, we know that there is room in the optimal solution for another box on the truck that the optimal algorithm did not put. Therefore, we can transform the optimal scheduling into the greedy scheduling just by placing another box on the truck. Note that this modification does not break the schedule produced by the optimal solution, as we simply removed a box from the next truck and placed it onto this one, which decreases the weight of the next truck. By doing this continuously until there exist no more differences between the two, we can transform the optimal solution to the solution produced by the greedy algorithm.

   But this contradicts the assumption that the greedy algorithm does not produce the minimal number of trucks, since we showed that the greedy algorithm and optimal solution produce the same number of trucks. Therefore, the greedy algorithm uses the minimum number of trucks.

7. Because each of the finishing portions of the jobs can be done in parallel, the completion time is limited by the time that the last one finishes. In particular, if all of the finishing jobs start simultaneously, then the completion time is simply the longest finishing time. Since the amount of time needed for the computer to preprocess is constant, we should start the job with the longest finishing time as soon as possible. Therefore, we claim that the optimal algorithm is to start the job with the longest finishing time first, then do jobs in decreasing order, ending with the job with the shortest finishing time.

   We prove our using induction on the number of jobs n.

   Base case: $n = 1$. Since there is only one job, our algorithm finds the correct solution.

   Assume starting the job with the longest finishing time finds the optimal schedule for n jobs. We show that given $n + 1$ jobs $j_1, ..., j_{n+1}$, the optimal schedule is to schedule the jobs in decreasing finishing time.

   Let S be any schedule that does not have the job with the longest finshing time j_f first. Suppose that $j_i \neq j_f$ is the first job in the schedule. Note that once the algorithm schedules j_i, there are n jobs remaining. Therefore, by the inductive hypothesis, the optimal way to schedule the next n jobs is by scheduling the longest finish time job j_f next. Because the two jobs j_i and j_f are consecutive, swapping their order does not affect any of the timings of the remaining jobs. If we swap them, then j_f can start earlier, and since the finishing time of j_i is less than that of j_f, swapping does not increase the completion time, and may potentially decrease the completion time. We can thus continuously swap j_i with the next job, until the finishing time of j_i is greater than the next job. But since we showed that any arbitrary schedule S without j_f as its first job cannot be better than a schedule with j_f as its first job, the schedule that our algorithm produces must be optimal.

   Therefore, the optimal algorithm is to run jobs in descending order of finish time.

12. (a) The claim is false.

For example, let $(b_1, t_1) = (100, 1)$, $(b_2, t_2) = 75$, $r = 75$.
Although $b_1 = 100 > 75 = t_1 r$, the schedule (b_2, b_1) is still valid.

(b) We show that sorting by increasing stream rate will produce a valid schedule if it exists.

Because we sort by increasing stream rate, after t seconds, the minimum possible number of bits has been sent through the stream. Therefore, if this number exceeds the parameter rt, then because we send the minimum number of bits, every other schedule also exceeds rt bits sent at this time, so there cannot be a valid schedule. If this number does not exceed rt for any t, then we simply have a valid schedule. Therefore sorting by increasing stream rate produces a valid schedule if it exists.

16.

```
prob16(x, t, e, n):
    sort x by occurence time
    for i from 1 to n:
        if t_j - e_j <= x_i <= t_j + e_j for some j:
            associate x_i with the earliest such j
        else:
            no association exists
```

The runtime of this algorithm is at most $O(n^2)$, since sorting is $O(n \log n)$, the outer for loop runs n times, and each of the n iterations scans through the entire list of n elements for possible j.

Assume for the sake of contradiction that this algorithm does not find an association when one exists. Suppose P is some perfect matching between the intervals and x. Let x_i be the first difference between P and the matching that our algorithm finds. By definition of x_i, P has x_i matched to some interval $t_i \pm e_i$, while our algorithm has x_i matched to some interval $t'_i \pm e'_i$. Because our algorithm picks the earliest such interval, $x_i \leq t'_i + e'_i \leq t_i + e_i$.

However, because P is a perfect matching, P must have paired the interval $t'_i \pm e'_i$ with some other x_j. Since this pairing must be valid, we have $t'_i - e'_i \leq x_j \leq t'_i + e'_i$. Finally, because $t_i \pm e_i$ is the earliest interval fitting x, we have $t_i - e_i \leq t'_i - e'_i$. Putting all of these inequalities together, we have

$$t_i - e_i \leq t'_i - e'_i \leq (x_i, x_j) \leq t'_i + e'_i \leq t_i + e_i$$

for some ordering of x_i and x_j. But this means that the perfect matching could have swapped the intervals mapped to x_i and x_j and gotten the exact same matching as our algorithm. Since the choice of x_i is completely arbitrary, there cannot be an association found by another algorithm that is not found by our greedy algorithm.

Therefore, our algorithm is able to find an association if it exists in $O(n^2)$.

30. We can categorize the nodes of the Steiner tree into two categories: the terminal nodes X of size k, and the added nodes $A = Z - X$ of size a.

We claim that any added node in the Steiner tree must have degree at least 3.
To show this, assume for the sake of contradiction that there exists an added node a in the optimal Steiner tree that has degree 2. Let the two nodes that a is connected to be u and v, that is, the Steiner tree contains the edges $(a, u), (a, v)$. If there is an edge (u, v) in the Steiner tree, then there is a cycle, so this optimal Steiner tree cannot contain (u, v). Because G is by definition a fully connected graph of n nodes, there exists an edge between (u, v) in G. Since the optimal Steiner tree does not contain (u, v), then the weight $w_{au} + w_{av}$ must be less than or equal to w_{uv}. If they are equal, then we can simply remove the added node and replace with (u, v). If the sum is less than the individual weight, the triangle inequality is violated. Therefore, the only added nodes in the Steiner tree with degree 2 must be those directly between two other nodes, and can therefore be removed completely and replaced with the edge connecting the two nodes.
It is fairly easy to see that there cannot be any added node of degree 1, as the node can simply be removed without breaking the connection between the terminal nodes, which decreases the weight of the Steiner tree, contradicting the optimality of the tree.

Because the Steiner tree is a spanning subtree on $X + A$, the Steiner tree is a spanning subtree on $k + a$ nodes, and therefore has $k + a - 1$ edges. By the handshake lemma, the sum of the degrees of all nodes is $2k + 2a - 2$. Thus, for each added node with degree at least 3, there must be some node with degree 1. Since added nodes have degree at least 3, this node of degree 1 must be one of the k terminal nodes. Therefore, the number of added nodes is limited by the number of terminal nodes: $0 \leq a \leq k$. The size of $Z = A + X$ is therefore given by $k \leq a + k \leq 2k$. There are $\binom{n}{k}$ ways to select the number of nodes for a tree of size k, and the runtime to find the minimal spanning tree for this tree is $O(k^2)$. Since the minimum-weight Steiner tree must be a minimal spanning tree, checking all minimal spanning trees of all possible sizes will allow us to find the desired Steiner tree.

The brute force check of all possible Steiner trees for the optimal tree is a function of the number of trees:

$$f(n) = O(k^2)\binom{n}{k} + O(k^2)\binom{n}{k+1} + \cdots + O(k^2)\binom{n}{2k}$$

Because $\binom{n}{k} = \frac{n(n-1)\ldots(n-k+1)}{k!} = O(n^k)$ and $k \leq n$,

$$O(f(n)) = O(kk^2n^{2k}) = O(n^{2k+3}) = O(n^{O(k)})$$

Therefore finding a minimum-weight Steiner tree can be solved in time $O(n^{O(k)})$.