CS180, Winter 2018
Homework 4
Problems 4.3 4.7 4.12 4.16 4.30

Derek Xu

February 8, 2018

1 Problem 4.3

Notation:
Given a set of truck, where each truck can carry at most a weight of W,
$T = \{t_1, t_2, ..., t_n\}$
a set of packages that need to be delivered,
$P = \{i_1, i_2, ..., i_m\}$
a set of associated weights such that i_j corresponds to w_j,
$W = \{w_1, w_2, ..., w_m\}$
Let it be such that items can be mapped to a truck such that $\sum_{t_j} w_k < W$
Let it be such that the trucks are indexed by the order they come in.
Let it be such that the packages must leave in the order they are in.
Find a way to minimize $|T|$.

The current algorithm maps each of the trucks, in order, to the packages, in order, such that
$\sum_{t_j} w_j \leq W$ and
$i_j \epsilon t_u$ and $i_k \epsilon t_v$ and $u < v$ implies $j < k$

Proof that our "current" greedy algorithm gives optimal solution:
We shall prove this by considering the optimal solution to the problem and showing our "current" greedy algorithm provides the same solution.
Let the mapping of trucks to packages, $M^* = (T^*, P)$, be optimal.
Let the mapping of trucks to packages, $M = (T, P)$, be our solution.
Observe when we have a truck, in M^* that isn't filled to completion:
$M^* = ..., [(t_u), (i_v, i_{v+1}, ..., i_{v+n})], [(t_{u+1}), (i_{v+n+1}, i_{v+n+2}, ..., i_{v+n+m})], ...$
. . . where $w_v + w_{v+1} + ... + w_{v+n} < W - w_{v+1} - w_{v+2} - ... - w_{v+j}$
. . . (i.e. truck t_u is not filled to completion).
We can apply our "current" algorithm to this mapping and fill all instances of unfilled trucks to completion, such that we create a new mapping, $M = (T, P)$, that also returns the optimal solution:
$M = ..., [(t_u), (i_v, i_{v+1}, ..., i_{v+n+j})], [(t_{u+1}), (i_{v+n+j+1}, i_{v+n+2}, ..., i_{v+n+m})], ...$
. . . where $j \geq 1$ and $w_v + ... + w_{v+n+j} < W - w_{v+n+j+1} < W$

Moving packages $i_{v+n+1}, ..., i_{v+n+j}$ from truck t_{u+1} to t_u results in 2 cases:
Case 1:
The following trucks, t_k s.t. $k > u$, carry the same packages, which means $|T^*| = T$ where T is the set of trucks returned by our "current" algorithm.
Case 2:
The following trucks, t_k s.t. $k > u$, carry different packages, which can lead to either there still being the same number of trucks i.e. truck t_k s.t. $k > u$ can take truck t_{k+1}'s packages, and so on, until the final truck keeps packages equal to "original packages" - "the packages it gave to the previous truck" > 0 (in which case $|T^*| = T$), or a lesser number of trucks i.e. the final truck (or trucks) give all of its "original packages" to the previous truck. We note that the latter cannot be the case since we assumed that the trucks in the mapping $T*$ was optimal, in which the latter case would create a situation where $|T*| < |T|$, and thus cannot exist.

We are left with the conclusion that the "current algorithm" produces a mapping $M = (T, P)$ such that $|T| = |T^*|$
$\therefore$ "current algorithm" produces the optimal solution.

2 Problem 4.7

Notation:
Given a set of jobs, such that each job, $J_i = (p_i, f_i)$
$J = \{J_1, J_2, ..., J_n\}$
Let it be such that p_i can be done only one at a time.
Let it be such that f_i can be all done in parallel.
Let it be such that p_i must be finished before f_i.
Let it be such that $S = J_i, J_j, ..., J_k$ denote the schedule of jobs
. . . where job J_i is performed before job J_k.
Let t denote the time between the start of the J_i and the end of the J_k.
Find a way to minimize the total time spent, t.

Claim
We simply select the longest jobs (which can be done in parallel) to be done first.

Proof that "longest job algorithm" gives optimal solution:
We shall prove this by considering the optimal solution to the problem and showing our "longest job" algorithm provides the same solution.
Let S^* be the optimal way to schedule jobs.
Let S be the way to schedule jobs, according to the algorithm.
Observe when we do not schedule the longer job first:
$S^* = ..., J_i, ..., J_k, ...$
. . . where $i < k$ and $f_i < f_k$ We can apply our current algorithm, and see the following: $S = ..., J_{i+n}, J_k, J_{i+m}, ..., J_{k-1}, J_i, ...$
. . . where $n \epsilon [-1, k-i-1)$
. . . and $m \epsilon [1, k-i+1)$
. . . and $f_{i+n} \geq f_k \geq f_{i+m} \geq f_i$
. . . i.e. we flipped J_k with J_i. Notice, the time of completion for our algorithm is the same as that of the optimal algorithm. Consider what happens when we flip these jobs. 9 cases appear (numbered 1a-3c):
Case 1. $f_i >$time it takes to complete the rest of the jobs

a. $f_k >$time it takes to complete the rest of the jobs
b. $f_k =$time it takes to complete the rest of the jobs
c. $f_k <$time it takes to complete the rest of the jobs

For these cases, (i.e. 1a 1b 1c), f_i will finish after the last job is finished. This case only matters if the time it takes to finish f_i is longer than the time it takes any other job in J to finish. Observe the job f_k now. Because f_k takes longer time to complete than f_i, as defined earlier, the only way for f_k in sequence $S*$ to terminate faster than f_i in sequence S is if it started earlier than f_i. BUT, as defined, f_i starts in the new job at the same time as f_k i.e. the jobs run on the supercomputer done before each of these two jobs is the same. Thus, cases b and c cannot be true as f_k takes more time than f_i and in case a, as just aforementioned the time it takes for the jobs in sequence S^* to finish would be either greater or equal to the time it takes for S. Because S^* is the optimal solution, we can thus conclude the time is the same.

Case 2. f_i =time it takes to complete the rest of the jobs

a. f_k >time it takes to complete the rest of the jobs

b. f_k =time it takes to complete the rest of the jobs

c. f_k <time it takes to complete the rest of the jobs

For these cases, (i.e. 2a 2b 2c), f_i will finish when the last job is finished. This means that the optimal time provided by the schedule S is $\sum_{p_i \epsilon J} p_i$. Because for all the jobs to finish, all of the jobs must be run at least one at a time by the super computer, and we assumed that our S^* is the optimal solution, the following equality must hold: $\sum_{p_i \epsilon J} \leq$ run time of $S^* \leq p_i \sum_{p_i \epsilon J} p_i$ we now see that "time to finish jobs in S" = time to finish jobs in S^*"

Case 3. f_i <time it takes to complete the rest of the jobs

a. f_k >time it takes to complete the rest of the jobs

b. f_k =time it takes to complete the rest of the jobs

c. f_k <time it takes to complete the rest of the jobs

For these cases, (i.e. 3a 3b 3c), f_i will finish before the last job is finished. As we saw in case 2a, 2b, and 2c, the sequence produced by algorithm, S, matches that of the optimal solution S^*.

We are left with the conclusion that the "longest job algorithm" produces a sequence S finishes jobs just as quickly as the optimal solution S^*.
$\therefore$ "longest job algorithm" produces the optimal solution.

3 Problem 4.12

(a) Rigorous definition of problem:
Given a set of streams, such that each stream, $s_i = (b_i, t_i)$
$S = \{s_1, s_2, ..., s_n\}$
Let it be that $b_i > 0$ is the number of bits to be sent.
Let it be that $t_i > 0$ is the amount of time required to set b_i bits.
Let it be that only one stream can be sent at a time.
Let there be no gap time, i.e. streams must be sent constantly.
Let it be that in a given time from 0 to t, a maximum of rt bits can be sent.
Find if a schedule for sending streams exist.

The current algorithm simply checks the following inequality:
$b_i \leq rt_i$

This algorithm is wrong. Consider the following counterexample:
Let $S = \{s_1, s_2\}$ s.t. $s_1 = (1.01, 1), s_2 = (0.01, 1), r = 1$
Note, s_1 breaks the rule as $1.01 > 1 * 1$, but by sending s_2 then s_1, we are still upholding the condition that in time 0 to t less than rt bits are being sent.

(b) We notice that the ideal way to send streams is to send low bitrate streams first as they act as a kind of "buffer" for higher bitrate streams to be sent without violating the condition set by r.

Naturally, we claim:
Given S we resort it into a new set S' such that
$S' = \{s'_1, s'_2, ..., s'_n\}$ where $s'i = (b'_i, t'_i)$ and $b'_i/t'_i \leq b'_j/t'_j$ s.t. $i < j$
. . . then if $\sum_{i=1}^{n} b'_i/t'_i < r$, a schedule exists
Pseudocode:
```
1 if $\sum_S b_i/t_i < r$, a schedule exists
2 otherwise, a schedule does not exist.
```

NOTE: WE ARE ONLY DOING THE COMPARISON NOT THE SORTING AS THE QUESTION ONLY ASKS FOR IF THE SCHEDULE EXISTS NOT WHAT THE SCHEDULE IS.

Proof of correctness:
The key thing to note here is that 1. S' is the optimal sequence from streaming and 2. checking if the bitrate from time $t = 0$ to $t = n$ offers the bitrate for S' which is the highest. If these two observations are correct, our algorithm is correct, as it simply checks if the bitrate for the optimum solution ever goes over the limit.
Consider the optimum solution $S^* = \{..., s_i, ..., s_j, ...\}$ where s_i has a higher bit rate than s_j. If we flip these 2 sequences, as proposed by 1st note, we would have a lower high bit rate, as the bit rate from 0 up to and including s_i is greater than the bit rate from 0 up to and including s_j. This is simple mathematical fact stemming from the fact that s_i has higher bit rate than s_j and the streams leading up to s_i in our solution has either higher or equal bit rate as those leading up to s_j in our solution. Therefore, we can conclude that the 1st note is correct, as the sequence, S, offers a equal or lower max bit rate than the optimum solution S^*.
Consider that the max bit rate in S' was at a time, $t_o <$ "end of stream". We notice that there will always be a bitrate that is equal or higher than t_o from t_o to "end of stream". Adding this section of time into our calculations will thus either increase or equal the rate at time t_o. Therefore we can conclude our 2nd note is correct.

Proof of Time Complexity:
The sum comparison simply does a linear scan over S and sums up b_i/t_i for each element s_i. This can be done in O(n) time.

4 Problem 4.16

Notation:
Given a set of times,
$T = \{t_1, t_2, ..., t_n\}$
a set of errors associated to said times,
$E = \{e_1, e_2, ..., e_n\}$
a set of neighborhoods associated to said times,
$N = (T \pm E) = \{n_1, n_2, ..., n_n\}$ s.t. $n_i = (t_i - e_i, t_i + e_i)$
a set of actions
$X = \{x_1, x_2, ..., x_n\}$
Find if for each $x_i \epsilon X$, $\exists n_i \epsilon N$ — $x_i \epsilon n_i$

Claim
We can find the optimal solution by attempting to match each element in X to an element in N, with a for loop.
Pseudo Code:

```
1 Initialize $N$
2 Sort $N$ in order of end time and sort $X$
3 Initialize $N'$ to keep track which neighborhoods have already been mapped
4 for $x_i$ in $X$:
5       for $n_i$ in $N$:
6             if $x_i$ exists in $n_i$:
7                   remove $n_i$ from $N$
8                   break (move to next $x_i$ and start on line 5)
9             if $n_i$ is last element in $N$:
10             mapping does not exist
11 if we mapped all $x_i$ in $X$:
12       mapping exists
```

Proof of correctness:
Assume that the algorithm gives a wrong output. This falls under 2 cases:
Case 1: Algorithm says mapping exists but mapping does not exist
This is false as we map each x_i to a neighborhood n_i in lines 4 through 10 before we can return that mapping exists. Thus we found the mapping in order to return mapping exists to return such.
Case 2: Algorithm says mapping does not exist but mapping exists
This is false because we only return false when we cannot find a mapping. i.e.

$\exists x_i$ s.t. $\exists$ no n_i that includes x_i and has not already been mapped. Because we sorted X and N beforehand, this means that there is no mapping which can include both x_i and x_{i+1} or x_{i-1} or there just does not exist of mapping for x_i itself. In either case, we only return no mapping when no mapping exists.
Because the algorithm cannot give the wrong output, our proof by contradiction is complete, and our algorithm works.

Proof of Time Complexity
assume X is of size $\approx n$ and T is of size n. initializing the set N will be a linear scan through E and T, taking O(n) time. Sorting N and X with merge wort will take O(nlogn) time. the 2 nested for loops scans through N and X resulting in O(n^2) time. The if condition at the end on line 11 and 12 takes O(1) time. In total, the algorithm takes O(n^2) time.

5 Problem 4.30

Notation:
I will be following the same convention as the textbook. The problem is to find a minimum weight Steiner tree in a graph $G = (V, E)$, given subgraph, $X \subseteq V$ in time $O(n^{O(k)})$.

The Steiner tree nodes, Z, must follow: $X \subseteq Z \subseteq V$.
We rewrite this subset as $Z = X \cup Y$. We notice that Y can only exist if it is exploited to connect all the nodes in X with less cost than without Y. We notice that such a case cannot exist if Y has 2 or less nodes connected to it. This is because adding any extra nodes to only 2 extra nodes would involve adding branches (and in effect cost). The following is a proof of this principle: Given any set of nodes, N, adding an extra node, v, that connects less than 2 nodes, will either result in an extra edge, or the removal of some edges and the addition of some edges. In the former case, adding any edge will always increase the total cost of the Steiner tree because edges cannot be of negative value. In the latter case, by the triangle equality, the removal of the edges and addition for edges will increase cost. Observe, in this case, an the extra node will at one or more points create 2 edges a, b, that was before connected by c, where $w_c < w_a + w_b$ (specified in problem). Thus, Y can be thought of as a set of nodes that have degree strictly greater than 2. we can see an example of when this could help in a triangle. Given a equilateral triangle with 3 nodes as vertexes, cost as edge length, and 3 edges as edges, we can add another node at the center of the 3 vertexes and connect the 3 vertex nodes to that node. In this case, adding a node in the middle would reduce the total cost of the Steiner tree. From discrete math, when we define that tree as having k nodes, we notice that because there must be at least 1 leaf node (and 1 root node) the number of nodes having degree 3, or $|Y|$, must be $\leq k-1$. from here we see that the total number of nodes $|X \cup Y| < 2k$, in which case we can simply test all $\binom{n}{2k} = O(n^{O(k)})$ possibilities. This proves both correctness and run time.