Solution to Homework 04

November 7, 2014

Problem 1

In the problem of scheduling all activities discussed in class, someone claims that the algorithm still works correctly if we follow:

(1) Activities sorted by finishing time.

This algorithm is not correct. If we change the algorithm Greedy-Schedule-All-Intervals line 1 to be:

$$\text{sort the intervals according to increasing } f_p \text{ value} : f_1 \leq f_2 \leq \cdots \leq f_n$$

we end up with the following counter example giving a non-optimal output:

Let

$$I = \{[1,3), [2,5), [6,7), [8,9), [4,10)\}$$

We can see that the intervals in I are already in sorted order by finishing time. However, when we add these to our queues following the algorithm, we get the following output:

$$Q_1 = \{[1,3), [6,7), [8,9)\}$$
$$Q_2 = \{[2,5)\}$$
$$Q_3 = \{[4,10)\}$$

If we follow the original algorithm (sorting by start time), we can see that we get the following optimal output:

$$Q_1' = \{[1,3), [4,10)\}$$
$$Q_2' = \{[2,5), [6,7), [8,9)\}$$

Therefore, sorting by finishing time does not give an optimal solution.

Note: If you were able to change the "availability" to check based on start time of the last item in each queue and then sorted elements in decreasing finish times, then you would have a correct algorithm (a mirror of the original algorithm). However, this is not allowed since we cannot make changes aside from line 1.

(2) Activities left in arbitrary order:

This algorithm is also not correct. If we change the algorithm Greedy-Schedule-All-Intervals by removing line 1, then the same counter example I from part (1) will also give us a counter example for this algorithm being optimal. If we leave the input in arbitrary order then on input

$$I = \{[1,3), [2,5), [6,7), [8,9), [4,10)\}$$

we would end up with the same output

$$\begin{aligned} Q_1 &= \{[1,3),[6,7),[8,9)\} \\ Q_2 &= \{[2,5)\} \\ Q_3 &= \{[4,10)\} \end{aligned}$$

which is not the optimal output of

$$\begin{aligned} Q_1' &= \{[1,3),[4,10)\} \\ Q_2' &= \{[2,5),[6,7),[8,9)\} \end{aligned}$$

Problem 2

To solve the problem we can do the following: we will take every point on our outer polygon as a starting point. For each of these starting points we use our greedy algorithm to solve for the minimum number of edges.

Let us denote $P_1 = \{p_1, \dots p_n\}$ as the points making up the outer convex polygon and $P_2 = \{\hat{p}_1, \dots \hat{p}_m\}$. Let us initialize $R = \{r_1, \dots, r_n\}$ where each $r_i = 0$. Sort P_1 by x-value, then taking the leftmost point in P_1 as p_1, compute the angle between p_1 and p_i, for all $p_i \in P_1$. Then sort P_1 by increasing angles from p_1 relative to the vertical line through p_1 going clockwise. Then for each $p_i \in P_1$, we set $r_i = \text{MinEdges}(P_1, P_2, p_i)$. Finally we output the minimum value stored in R.

Algorithm 1 MinEdges(P_1, P_2, s)

```
1: $\Theta \leftarrow \{\Theta_1, \dots, \Theta_{m+1}\}$ and initialize $\Theta_i \leftarrow 2\pi$ for all $\Theta_i \in \Theta$.
2: for $\hat{p}_i \in P_2$ do
3:     $\Theta_i \leftarrow$ angle between $s$ and $\hat{p}_i$ (relative to vertical line through $s$, clockwise)
4: end for
5: $\sigma_{min} \leftarrow \min\{\Theta_i \mid \Theta_i \in \Theta\}$, $\sigma_{max} \leftarrow \max\{\Theta_i \mid \Theta_i \in \Theta\}$
6: $p \leftarrow$ ComputeIncidence($s, \sigma_{min}, P_1$), $q \leftarrow$ ComputeIncidence($s, \sigma_{max}, P_1$)
7: $edges \leftarrow 2$
8: while $\sigma_{min} \neq \Theta_{m+1}$ do
9:     for $\hat{p}_i \in P_2$ do
10:        $\Theta_i \leftarrow$ angle between $p$ and $\hat{p}_i$
11:    end for
12:    $\Theta_{m+1} \leftarrow$ angle between $p$ and $q$
13:    $\sigma_{min} \leftarrow \min\{\Theta_i \mid \Theta_i \in \Theta\}$
14:    $p \leftarrow$ ComputeIncidence($p, \sigma_{min}, P_1$)
15:    $edges \leftarrow edges + 1$
16: end while
17: return $edges$
```

Algorithm 2 ComputeIncidence(p, σ, P)

```
1: $A \leftarrow \{a_i \mid a_i$ is the angle between $p$ and $p_i \in P\}$
2: $k \leftarrow$ index of $\min\{a_i \mid a_i \in A\}$
3: **while** $\sigma < a_k$ **do**
4:     $k \leftarrow (k = n)\ ?\ 1 : k + 1$ {if $k$ is $n$ then set $k = 1$, otherwise increment $k$}
5: **end while**
6: {We know the line from $p$ at angle $\sigma$ will pass between points $p_k$ and $p_{k+1}$}
7: $\ell \leftarrow p_k$, $r \leftarrow p_{k+1}$ {again, we can assume if $k$ is $n$ take $k+1$ to be 1}
8: {Now we will compute where the line from $p$ at angle $\sigma$ intersects $P$}
9: $m \leftarrow \frac{\ell_y - r_y}{\ell_x - r_x}$, $b \leftarrow \ell_y - m\ell_x$ {line between $\ell$ and $r$: $y = mx + b$}
10: $m' \leftarrow \tan(\frac{\pi}{2} - \sigma)$, $b' \leftarrow p_y - m'p_x$ {line through $p$ at angle $\sigma$: $y = m'x + b'$}
11: $u \leftarrow \frac{b' - b}{m - m'}$
12: $v \leftarrow mu + b$
13: **return** $(u, v)$
```

The way our algorithm works is given a point we find the edge that has the largest angle between the point of incidence and the outer polygon, but creates a line that doesn't go through the inner polygon. Then we find where this line crosses the outer polygon to find a new point of incidence and repeat until we hit the starting point.

Proof. Greedy Choice:
Given our point of incidence we take the edge with the largest angle closest to the inner polygon. Suppose there was an optimal solution that had an smaller angle to the outer polygon. Then change the angle more towards the inner polygon without causing an issue (since P_1 is convex, we won't run into the issue of needing more edges to go around part of P_1). So we can make this choice without issue. ☐

Proof. Optimal Substructure:
Given we have partially bounded the inner polygon. Given an optimal solution for the remainder of the bounding box, assume we could have fewer edges. Then that means one of the points further in the optimal solution would have a larger angle with the outer polygon and the point of incidence. This is a contradiction since we chose the point with the largest angle possible. ☐

Runtime:
Our preprocessing to sort P_1, compute angles, and sort again takes $O(n \log n)$. Given a point s, MinEdges loop on line 2 is $O(m)$, line 5 is $O(m)$, line 6 is $O(n)$, loop on line 8 is run $O(n)$ times, and the inner loop on line 9 and line 13 are $O(m)$ and ComputeIncidence is $O(n)$. Finding the minimum in R is $O(n)$. Putting this all together we get $O(n^2(n + m))$.

Problem 3

A minimum spanning tree is a connected graph, so for any cut of the original graph (X, Y) where $X \cup Y = V$,we have one edge connecting from X to Y in the MST. And as it is a minimum spanning tree, we can only choose the edge with minimum weight from the edges connecting from X to Y. And as the weights are distinct, we have a unique choice for each cut. So enumerate all cuts for the graph, each time we get a unique edge for the MST. So the MST is unique.

Problem 4

Order the n files sorted by increasing length.

Proof. Greedy Choice:
Our algorithm takes the shortest file available first. This is optimal because any other order will increase average time. Suppose instead you have an optimal order where for some $i_j > 1$, L_{i_j} is the smallest file (the

smallest file is not first). Then we have $L_{i_j} < L_{i_j-1}$. Let $\delta = L_{i_j-1} - L_{i_j}$. Consider the sum T_{i_j-1} for the sequence $L_{i_1}, \ldots, L_{i_j-1}, L_{i_j}, \ldots, L_{i_n}$. Consider also the sum $T'_{i'_j-1}$ for the sequence $L_{i'_1}, \ldots, L_{i'_j-1}, L_{i'_j}, \ldots, L_{i'_n}$, where $i'_j = i_j - 1$ and $i'_j - 1 = i_j$. We can see that this inversion lends to a larger sum in the sequence without this inversion, i.e. $T_{i_j-1} = T_{i'_j-1} + \delta$. Since all other values of T will be the same, we get that $\frac{1}{n}\sum T_{i_j} > \frac{1}{n}\sum T_{i'_j}$. This happens as long as the solution contains an inversion with the smallest file. Therefore the smallest file must come first so that it is not contained in an inversion. Thus our greedy choice is correct. □

Proof. Optimal Substructure:
Consider we have chosen file L_{i_j} with shortest length. Then consider the remaining files $L_1, \ldots, L_{i_j-1}, L_{i_j+1}, \ldots L_n$. Choosing the next file does not impact the sum T_{i_j} so our future choices cannot impact our current solution. So if we get an optimal solution for the remaining files, we also have an optimal solution for L_{i_j} and the remaining files. □

You can consider the following as a simple example of the idea above: instead of $\{1, 2, 3, 4, 5\}$ suppose you chose the order $\{1, 2, 4, 3, 5\}$. In both sequences reading the files 1 and 2 takes the same amount of time. But reading the third file in the first sequence takes 1 less time unit than in the second sequence. To summarize let me denote T_3 for sequence 1 as T_3^1 and for sequence 2 as T_3^2. Similarly we have T_4^1 and T_4^2. Here we get $T_3^1 = 6$ and $T_3^2 = 7$. We will then have $T_4^1 = 10$ and $T_4^2 = 10$,and so on. We see that if the inversion were further apart then there would be a larger impact, but here all the other T values are the same. So we would end up with a larger average for the second sequence since the summation contains a larger value.

Problem 5

Note that we want to design an $O(n)$ algorithm, so sorting is not allowed. The algorithm is as follows:

Denote the value of items as $P = \{p_1, \ldots, p_n\}$. Denote weights as $W = \{w_1, \ldots, w_n\}$. Denote fractions $X = \{x_1, \ldots, x_n\}$. Denote knapsack capacity as K. The following algorithm will leave X with the appropriate weights given an input of P, W, K.

First compute $R = \{\frac{v_1}{w_1}, \ldots, \frac{v_n}{w_n}\}$. Initialize all values $x_i = 0$ for $i = 1 \ldots n$. Call Knap(P, W, R, X, K).

Algorithm 3 Knap(P, W, R, X, K)

```
1: Use SELECT algorithm to find the median of $R$.
   Denote $m$ as median (note that $m = \frac{v_j}{w_j}$ for some $j$).
2: Divide items into 3 sets: $G = \{i \mid \frac{v_i}{w_i} > m\}$, $E = \{i \mid \frac{v_i}{w_i} = m\}$ and $L = \{i \mid \frac{v_i}{w_i} < m\}$. Compute $W_G = \sum_{i \in G} w_i$ and $W_E = \sum_{i \in E} w_i$.
3: if $W_G > K$ then
4:     We do not take any item from $G$. Call Knap$(P', W', R', X, K)$,
       where $P' = \{p_i \mid i \in G\}$, $W' = \{w_i \mid i \in G\}$, $R' = \{\frac{p_i}{w_i} \mid i \in G\}$
5: end if
6: if $W_G + W_E \geq K$ then
7:     We are finished.
8:     Take all items from $G$ first, by setting $x_i = 1$ for all $i \in G$ and updating $K = K - w_i$ after each iteration (this can be done since $W_G \leq K$).
9:     Take as many items as possible from $W_E$ by setting $x_i = 1$ for $i \in E$ until $K < w_i$, updating $K = K - w_i$ after each iteration.
10:    If $K = 0$ then we are done. Otherwise $E$ has at least one item remaining so take $i \in E$. Since $0 < K < w_i$, let $x_i = \frac{K}{w_i}$.
11:    Return;
12: end if
13: Otherwise we have $W_G + W_E < K$
14: Set $x_i = 1$ for all $i \in G \cup E$.
15: Set $K = K - W_G - W_E$.
16: Call Knap$(P', W', R', X, K)$, where $P' = \{p_i \mid p_i \in P, i \notin G \cup E\}$, $W' = \{w_i \mid w_i \in W, i \notin G \cup E\}$, $R' = \{\frac{p_i}{w_i} \mid \frac{p_i}{w_i} \in R, i \notin G \cup E\}$
```

Runtime:
Notice that each case removes at least half of the elements that we need to consider (we either remove the elements larger than the median or smaller). Each call runs SELECT, which is $O(n)$ time, partitioning the elements in line 2 is $O(n)$ time, and making the updates to K and the sets $P', W',$ and R' all take $O(n)$ time (we only have each element in one of these sets). the recursion formula for the algorithm is $T(n) = T(\frac{n}{2}) + O(n)$, because each time we can handle at least half of the items. So by Master Theorem, $T(n) = O(n)$.

Proof. Correctness:
Suppose our algorithm outputs $x_i < opt_i$ where opt_i is the optimal value for x_i and w.l.o.g. all objects have different unit costs (this can be done since you could shift the weights slightly to get the same result). Then at some step in our recursion $i \in L$ or $i \in E$ and $W_G + W_E \geq K$.

- Case 1: $i \in E$.
  If $W_G > K$ then there was at least one element with a better unit cost that would have been in the optimal solution and leaving $x_i = 0$, contradicting our assumption.
  The case where $W_G + W_E \geq K$ and $W_G < K$ cannot happen since we assumed there are no multiple elements that would have the same unit cost and SELECT always gives us exactly one element.

- Case 2: $i \in L$.
  If $W_G > K$ then there was at least one element with a better unit cost that would have been in the optimal solution and leaving $x_i = 0$, contradicting our assumption.
  If $W_G + W_E \geq K$ and $W_G < K$, then again we know that there was at least one element with a better unit cost that would have been taken in the optimal solution and still leaving $x_i = 0$. This again contradicts our assumption.

Thus, no matter what i was chosen, our algorithm would have set $x_i = opt_i$. □

Problem 6

Given the input of n 1-d points as $P = \{p_1, \ldots, p_n\}$, we can use the following greedy algorithm to cover the points with intervals.

Algorithm 4 OneDIntervals(P)

```
1: Sort $P$ in increasing order.
2: Let $I = \{(p_1, p_1 + 1)\}$
3: $last \leftarrow p_1 + 1$
4: for $i = 2$ to $n$ do
5:     if $p_i > last$ then
6:         Insert the interval $(p_i, p_i + 1)$ into $I$.
7:         $last \leftarrow p_i + 1$
8:     end if
9: end for
```

Here, we take the greedy choice of covering the leftmost point that is uncovered with a unit interval.

Correctness:

Proof. Greedy choice:
Our algorithm takes the smallest point in sorted order and assigns that the first interval. Since any optimal solution must cover this point and there are no points with smaller x-value, taking this interval starting with the smallest point will cover any points the interval from the optimal solution would have covered. (i.e., if I is an optimal solution and interval $i = [x_1, x_2]$ covers p_1, then since no points are smaller than p_1 the interval $[p_1, p_1 + 1]$ will not exclude any points from i). □

Proof. Optimal substructure:
Given points $P = \{p_1, \ldots, p_n\}$ in sorted increasing order, consider the subproblem $P' = \{p_i \in P \mid p_i > p_1 + 1\}$ (all points outside the first interval). If P' is empty, then we have an optimal solution covering all points in P with one unit interval. If P' is not empty, then we would require at least one more interval to cover some or all of the points in P'. If this interval covering the points in P' could have covered all the points in $P \setminus P'$ as well, then this interval would have to start either at p_1 or to the left of p_1, contradicting that P' is not empty (since if this were the case, then there would have to be points in P' that have a distance at most 1 from p_1, meaning that they wouldn't belong to P' to begin with). □

Let us consider this problem in 2-D. Suppose, w.l.o.g., we take the unit square going up and right from a given point. So if a point $p = (1, 1)$ is taken as a basis to axis-aligned unit square, we add the axis-aligned unit square $(1, 1), (1, 2), (2, 1), (2, 2)$ (up and right) to our solution. Suppose we take the point-set $P = \{(1, 2), (1.5, 1.5), (2, 2)\}$. Since our algorithm would first sort by x-value, we would add the axis-aligned unit square starting at $(1, 2)$. Then since $(1.5, 1.5)$ lies below that, we would add the axis-aligned unit square starting at $(1.5, 1.5)$. We know that $(2, 2)$ is covered by both unit squares we have added so we are done. However, the axis-aligned unit square starting at $(1, 1)$ covers all 3 points, so the optimal solution has only one unit square, not two. Therefore our greedy algorithm does not work in the 2-D case.