Computer Science C73 September 22, 2020
Scarborough Campus University of Toronto

Solutions for Homework Assignment #2

Answer to Question 1.

ALGORITHM. Intuitively, the algorithm works as follows: We let ℓ be the maximum time slot in which we can schedule an order; initially this is the slot $\max\{d_i : 1 \leq i \leq n\}$. Among all remaining (i.e., not previously scheduled) orders that can be scheduled in slot ℓ (i.e., orders whose deadline is ℓ or more), we choose the most profitable one, with ties broken arbitrarily. Now the slots left in which we can still schedule the remaining orders are the slots $1..\ell-1$. In fact, we can (and for reasons of running time efficiency we should) eliminate from consideration slots in the range $1..\ell-1$ that are greater than the maximum deadline of the remaining orders. We continue scheduling the remaining orders in these slots using the same logic, until we run out of slots or orders. This is expressed in pseudocode as follows:

```
  SCHEDULE($J$)
1 for $i := 1$ to $n$ do $S(i) := \infty$                        ▷ initialization
2 $\ell := \infty$
3 while $\ell > 1$ and $J \neq \varnothing$ do
4     $\ell := \min(\ell - 1, \max\{d_i : (i, d_i, p_i) \in J\})$   ▷ last slot to schedule orders left in $J$
5     $(j, d_j, p_j) :=$ most profitable order in $J$ with deadline $\geq \ell$   ▷ choose order to schedule in $\ell$
6     $S(j) := \ell$                                             ▷ schedule $j$ in slot $\ell$
7     $J := J - \{(j, d_j, p_j)\}$                               ▷ delete schedule $j$ in slot $\ell$
8 return $S$
```

CORRECTNESS. Intuitively, the algorithm is correct because it schedules as late as possible the most profitable order available for scheduling, as as to leave as much room as possible to schedule the remaining orders. We will use the "promising set" method to prove that it produces an optimal schedule, where the "set" in question now is the partial schedule S that the algorithm has produced so far. So we must define what it means for a schedule S^* to "extend" another schedule S. Intuitively it means that S^* schedules every order that S schedules in the same slot as S; it is possible that S^* schedules more orders than S. More precisely, S^* ***extends*** S if for every order i such that $S(i) \neq \infty$ (i.e., S schedules order i), $S^*(i) = S(i)$.

Let S_t be the schedule in variable S at the end of the t-th iteration of the loop in lines 3-7. It is obvious that S_t is indeed a schedule: it does not map distinct orders into the same slot, and it schedules every order by its deadline. We will prove:

Lemma 1. *For every iteration t of the loop, there is an optimal schedule S_t^* that extends S_t.*

Proof. By induction on the iteration number t. The basis is trivially true because $S_0(i) = \infty$ for every order i, so it is extended by any schedule.

For the induction step, suppose that there is an optimal schedule S_t^* that extends S_t. We will prove that, if iteration $t+1$ exists, then there is optimal schedule S_{t+1}^* that extends S_{t+1}. Let j be the order that is scheduled in iteration $t+1$, and ℓ be the slot in which order j is scheduled in that iteration. There are two exhaustive cases depending on whether j is scheduled in S_t^* (i.e., whether $S_t^* = \infty$).

CASE 1. $S_t^*(j) = \infty$ (i.e., j is not scheduled in S_t^*). Then some order j' is scheduled in slot ℓ in S_t^*, i.e., $S_t^*(j') = \ell$. (Otherwise we could schedule j in slot ℓ and improve the profit, contradicting that S_t^* is optimal). Let S_{t+1}^* be the function that is identical to S_t^* except that instead of j' we schedule j in slot ℓ: $S_{t+1}^*(j) = \ell = S_{t+1}(j)$ and $S_{t+1}^*(j') = \infty$. It is easy to check that this is a schedule. Note that when the greedy algorithm added j to the schedule in iteration $t+1$, j' was still in the set of unscheduled orders J. (Otherwise, we would have $S_t(j') = \ell'$ for some $\ell' > \ell$, contradicting that S_t^* extends S_t.) Also note that, since j' is scheduled in slot ℓ in S_t^*, $d_{j'} \geq \ell$. Since (a) $j' \in J$ during iteration $t+1$, (b) $d_{j'} \geq \ell$, and (c) the greedy algorithm added j, rather than j', to the schedule in iteration $t+1$, it follows that $p_j \geq p_{j'}$. Thus, S^*_{t+1} is at least as profitable as S^*_t, and since S^*_t is optimal so is S^*_{t+1}. By its definition and the induction hypothesis that S^*_t extends S_t, S^*_{t+1} extends S_{t+1}. So, S^*_{t+1} is an optimal schedule that extends S_{t+1}, as wanted.

Case 2. $S^*_t(j) = \ell'$ for some $\ell' \neq \infty$ (i.e., j is scheduled in some slot ℓ' in S^*_t). If $\ell' = \ell$ then we take $S^*_{t+1} = S^*_t$, and we are done: We have an optimal schedule that extends S_{t+1}.

So, in the rest of the proof assume $\ell' \neq \ell$. There are two exhaustive subcases, depending on whether in S^*_t some order is scheduled in slot ℓ

- Subcase 2a. No order is scheduled in slot ℓ in S^*_t (i.e., there is no j' such that $S^*_t(j') = \ell$). Then let S^*_{t+1} be identical to S^*_t, except that S^*_{t+1} schedules j in slot ℓ instead of slot ℓ'; i.e., $S^*_{t+1}(j) = \ell = S_{t+1}(j)$. It is clear that S^*_{t+1} is a schedule and has the same profit as S^*_t, so it is optimal. Furthermore, it extends S_{t+1} (by its definition and the induction hypothesis that S^*_t extends S_t). Thus, S^*_{t+1} is an optimal schedule that extends S_{t+1}, as wanted.
- Subcase 2b. Some order j' is scheduled in slot ℓ in S^*_t (i.e., for some j', $S^*_t(j') = \ell$). We will prove that

$$\ell > \ell' \tag{*}$$

  Suppose, for contradiction, that $\ell' > \ell$ (recall that $\ell' \neq \ell$). Since the algorithm schedules order j in iteration $t+1$, $j \in J$ before that iteration. And since j is scheduled in slot ℓ' in S^*_t, $d_j \geq \ell'$. Therefore, the algorithm considers slot ℓ' before iteration $t+1$ (i.e., it does not skip it in line 4). When the algorithm considered slot ℓ', j was still in J at that time, and $d_j \geq \ell'$ (as we just argued); so, the algorithm schedules some order j'' in slot ℓ'. Since S^*_t extends S_t, S^*_t also schedules j'' in slot ℓ'. So, by the hypothesis of Case 2, $j'' = j$. Thus, the algorithm schedules order j in slot ℓ'. But this contradicts that the algorithm schedules j in slot $\ell \neq \ell$. This proves $(*)$.

  So S^*_t schedules order j' in slot ℓ and order j in slot $\ell' < \ell$. Now let S^*_{t+1} be identical to S^*_t except that the slots of j and j' are swapped: $S^*_{t+1}(j) = \ell$ and $S^*_{t+1}(j') = \ell'$. S^*_{t+1} is a schedule because (a) $d_j \geq \ell$ (since S_{t+1} schedules j in slot ℓ), and (b) $d'_j \geq \ell'$ (since S^*_t schedules j' in $\ell > \ell'$). S^*_{t+1} has the same profit as S^*_t and so it is also optimal. Finally, by its definition and the induction hypothesis that S^*_t extends S_t, S^*_{t+1} extends S_{t+1}. So, S^*_{t+1} is an optimal schedule that extends S_{t+1}, as wanted. □

First we note that the loop terminates because in each iteration an order is removed from J, so the loop cannot be executed more than n times. Let S be the schedule returned by the algorithm. By the lemma, there is an optimal schedule S^* that extends S. We claim that $S^* = S$. Suppose, for contradiction, that $S^* \neq S$. Then (since S^* extends S) there is some order j such that $S(j) = \infty$ and $S^*(j) = \ell \neq \infty$. Since $S(j) = \infty$, j is in J at the end of the algorithm and therefore it is in J in every iteration. Since $S^*(j) = \ell$, $d_j \geq \ell$ and so the algorithm considered slot ℓ (it did not skip it), yet it did not schedule any order in that slot. Since j is always in J, the only reason why no order was scheduled in slot ℓ by the algorithm is that $d_j < \ell$. But this contradicts that S^* schedules order j in slot ℓ. Thus, $S^* = S$ and the algorithm returns an optimal schedule.

Running time. Line 1 takes $O(n)$ time and line 2 takes $O(1)$ time. As noted above, the loop in line 3 is executed at most n times. Lines 4 and 5 can be done in $O(n)$ time each (by scanning the set of remaining orders J). Lines 6 and 7 can be done in $O(1)$ time each. So, each iteration of the loop takes $O(n)$ time and the entire algorithm runs in $O(n^2)$ time.

The algorithm can be implemented in $O(n \log n)$ time by considering the orders in decreasing deadline, starting with the maximum time slot that can be used to schedule an order, and keeping all the orders with deadline greater than or equal to the maximum slot available to schedule the remaining orders in a max-heap keyed on profit. The details are interesting and are left as an exercise.

Answer to Question 2.

Algorithm. The algorithm is to process videos in order of decreasing second phase length.

```
    Schedule($V$)
1   $L :=$ the set of triples in $V$ sorted in decreasing second phase length
2   for $i := 1$ to $n$ do $S[i] :=$ first component of $i$-th tripe in $L$
3   return $S$
```

Correctness. We will prove the optimality of the schedule output by our algorithm by using the exchange argument.

Let $A = i_1, i_2, \ldots, i_n$ be any schedule (i.e., permutation of $1..n$). The finish time of video i_k in this schedule is $(f_{i_1} + f_{i_2} + \ldots + f_{i_k}) + s_k$. The term in parentheses is the time for the first phase of all videos up to and including i_k and the last term is the time for the second phase of video i_k, which can start as soon as its first phase is completed.

Given schedule A, we define an inversion to be two videos i and j such that i appears before j in A, but $s_i < s_j$.

Our algorithm produces a schedule with no inversions. Any two schedules with no inversions differ only in the order of schedules that have the same second phase length, and therefore they have the same maximum completion time over all videos. Therefore, to prove the correctness of our algorithm it suffices to prove that a schedule with no inversions is optimal.

Let S^* be an optimal schedule with the minimum number of inversions. We will now prove that S^* has no inversions. Suppose, for contradiction, that S^* has at least one inversion. Therefore, there are two videos i and j that appear consecutively in S^* but $s_i < s_j$. Consider now the schedule $\widehat{S}$ obtained from S^* by swapping i and j (so that j appears before i).

Clearly, every video other than i and j has the same completion time in S^* as in $\widehat{S}$. Let $i_1, i_2, \ldots, i_k$ be the videos that precede i in S^* (and therefore j in $\widehat{S}$). Consider now the completion times of i and j in each of S^* and $\widehat{S}$:

Completion time of i in S^*: $C_i^* = (f_{i_1} + f_{i_2} + \ldots + f_{i_k} + f_i) + s_i$
Completion time of j in S^*: $C_j^* = (f_{i_1} + f_{i_2} + \ldots + f_{i_k} + f_i + f_j) + s_j$
Completion time of j in $\widehat{S}$: $\widehat{C}_j = (f_{i_1} + f_{i_2} + \ldots + f_{i_k} + f_j) + s_j$
Completion time of i in $\widehat{S}$: $\widehat{C}_i = (f_{i_1} + f_{i_2} + \ldots + f_{i_k} + f_j + f_i) + s_i$

Note that $\widehat{C}_j \le C_j^*$ (because C_j^* has an additional non-negative term) and $\widehat{C}_i < C_j^*$ (because the parenthesized expressions are equal but $s_i < s_j$). Therefore, $\max(\widehat{C}_i, \widehat{C}_j) \le \max(C_i^*, C_j^*)$. Since all other videos have the same completion time in $\widehat{S}$ as in S^*, the maximum completion time in $\widehat{S}$ is at most the maximum completion time in S^*. Since S^* is optimal, so is $\widehat{S}$. But $\widehat{S}$ has fewer inversions than S^*, contradicting that S^* has at least one inversion. Therefore S^* has no inversions.

Since (a) S^* is an optimal schedule with no inversions, (b) all schedules with no inversions have the same maximum completion time, and (c) our algorithm produces a schedule with no inversions, it follows that our algorithm produces an optimal schedule.

Running time. The running time of the algorithm is dominated by the time to sort V, which can be done in $O(n \log n)$ time using, say, mergesort or heapsort.

Answer to Question 3.

(1) Use Dijkstra's algorithm to find the weight $\delta(u)$ of a shortest path from s to u, for each node u.

(2) By reversing the edges, use Dijkstra's algorithm again to find the weight $\delta^R(v)$ of a shortest path from v to t, for each node v.

(3) For each potential edge (u, v), compute the sum $\delta(u) + \mathbf{wt}(u, v) + \delta^R(v)$, and keep track of a potential edge that minimizes this sum.

(4) Return the potential edge kept.

Each use of Dijkstra's algorithm in steps (1) and (2) takes $O\big((m+n)\log n\big)$ time. Reversing the edges in step 2 takes $O(m+n)$ time. So, steps (1) and (2) take $O\big((m+n)\log n\big)$ time. Step (3) takes $O(k)$ time, and step 4 takes $O(1)$ time. So, the algorithm takes $O\big(k + (m+n)\log n\big)$ time.