Computer Science C73
Scarborough Campus

November 11, 2021
University of Toronto

Solutions for Homework Assignment #6

Answer to Question 1.

a. Suppose we have three orders with the deadlines, lengths, and profits indicated in the able below:

i	d_i	ℓ_i	p_i
1	2	2	1.1
2	2	1	1
3	2	1	1

The greedy algorithm would schedule only order 1, for a total profit of 1.1, whereas we can schedule jobs 2 and 3 for a total profit of 2.

b. *Definition of the subproblems that our dynamic programming algorithm will solve:* Without loss of generality, assume that $d_1 \leq d_2 \leq \ldots \leq d_n$ (rename the orders, if needed — the order is important for the recursive formula). For each i, $0 \leq i \leq n$, and each t, $0 \leq t \leq d_n$:

$$P(i,t) = \text{the maximum profit achievable to schedule a subset of orders } 1..i \text{ so that they all finish by time } t \tag{*}$$

Recursive formula to compute each subproblem: Let $t' = \min(t, d_i)$; this represents the latest time by which order i must finish to meet its deadline in a schedule where all filled orders finish by time t.

$$P(i,t) = \begin{cases} 0, & \text{if } i = 0 \text{ or } t = 0 \\ P(i-1,t), & \text{if } i > 0,\ t > 0, \text{ and } t' < \ell_i \\ \max\big(P(i-1,t), P(i-1,t'-\ell_i) + p_i\big), & \text{otherwise} \end{cases} \tag{†}$$

Justification why (†) is a correct formula to compute (∗): For the base case, if there are no orders ($i = 0$) or there is no time to complete any order ($t = 0$), the maximum possible profit is obviously 0. Therefore, $P(i,t) = 0$ in this case, as wanted.

For $i > 0$ and $j > 0$, if $t' < \ell_i$, there is no way to schedule order i so that it finishes by its deadline and by t. So, the maximum profit to schedule orders $1..i$ up to time t is the same as the maximum profit to schedule orders $1..i-1$ up to time t. Therefore, $P(i,t) = P(i-1,t)$ in this case, as wanted.

Finally, if $i > 0$, $j > 0$, and $t' \geq \ell_i$, there are two possibilities.

- The optimal schedule S for orders $1..i$ up to time t does ***not*** include order i. In this case S is also an optimal schedule for orders $1..i-1$ up to time t, so $P(i,t) = P(i-1,t)$.
- The optimal schedule S for orders $1..i$ up to time t includes order i. Since $\ell_i \leq t' \leq d_i$, order i can be scheduled in the interval $(t'-\ell_i, t')$. Furthermore, all other orders that are scheduled in S can be scheduled before time $t'-\ell$: This is because, if in fact order i is scheduled at an earlier interval in S, we can move it to the interval $(t'-\ell_i, t')$, this creating room to move all orders that were originally scheduled after order i to earlier non-overlapping intervals, so that they still meet their deadlines. So, without loss of generality, we can assume that in S orders $1..i-1$ are scheduled before time $t'-\ell_i$ and order i is scheduled in the interval $(t'-\ell_i, t')$. Let S' be the schedule that schedules orders $1..i-1$ as S but it does not schedule order i — i.e., $S'(i) = \infty$. So, the total profit of S is the total profit of S' plus the profit of order i. By a cut-and-paste argument S' is an optimal schedule for orders $1..i-1$ up to time $t'-\ell_i$. So, in this case, $P(i,t) = P(i-1,t'-\ell_i) + p_i$.

From the above two cases, we conclude that, if $i > 0$, $j > 0$, and $t' \geq \ell_i$, $P(i,t) = \max\bigl(P(i-1,t), P(i-1,t'-\ell_i)+p_i\bigr)$, as wanted.

Solving the original problem: By the definition of the subproblems $(*)$, the required return value is $P(n, d_n)$.

Pseudocode:

MaxProfit($\{(d_i, \ell_i, p_i) : 1 \leq i \leq n\}$)
1 sort the orders by non-decreasing deadline
2 **for** $t := 0$ **to** d_n **do** $P(0,t) := 0$
3 **for** $i := 1$ **to** n do $P(i,0) := 0$
4 **for** $i := 1$ **to** n **do**
    for $t := 0$ **to** d_n **do**
5         $t' = \min(t, d_i)$
6   **if** $t' < \ell_i$ **then** $P(i,t) := P(i-1,t)$
7   **else** $P(i,t) := \max\bigl(P(i-1,t), P(i-1,t'-\ell_i)+p_i\bigr)$
8 **return** $P(n, d_n)$

Running time: The running time of the algorithm is $\Theta(nd_n)$. This is pseudopolynomial, because it is polynomial in the input ***value***, but not in the input ***size***. In particular, d_n is exponentially larger than the size of its representation.

This problem is actually NP-hard, so it is unlikely that it has a (truly) polynomial-time algorithm.

c. The idea is to "step" through $P(-,-)$ backwards starting at $P(n, d_n)$, and use the value of $P(i,t)$ to determine whether order i is included in the optimal schedule (this is the case if $P(i,t) \neq P(i-1,t)$, which means that $P(i,t) = P(i-1, \min(t,d_i) - \ell_i) + p_i$), and if so we schedule it to finish at time $\min(t, d_i)$. In the above pseudocode we replace line 8 by the following:

$i := n$; $t := d_n$
while $i \neq 0$ **do**
    if $P(i,t) = P(i-1,t)$ **then** $S(i) := \infty$; $i := i-1$
    else $S(i) := \min(t, d_i)$; $i := i-1$; $t := \min(t, d_i) - \ell_i$
return S

Answer to Question 2. Let $B[1..n, 1..n]$ be the given $n \times n$ array of 0s and 1s. For a square-of-ones (i, j, ℓ) in B, define its lower-right corner (LRC) to be $(i+\ell, j+\ell)$ and its size to be ℓ.

Definition of the subproblems that our dynamic programming algorithm will solve: For $1 \leq i, j \leq n$

$$S(i,j) = \text{the maximum size of a square-of-ones among all squares-of-ones whose LRC is } (i,j) \qquad (*)$$

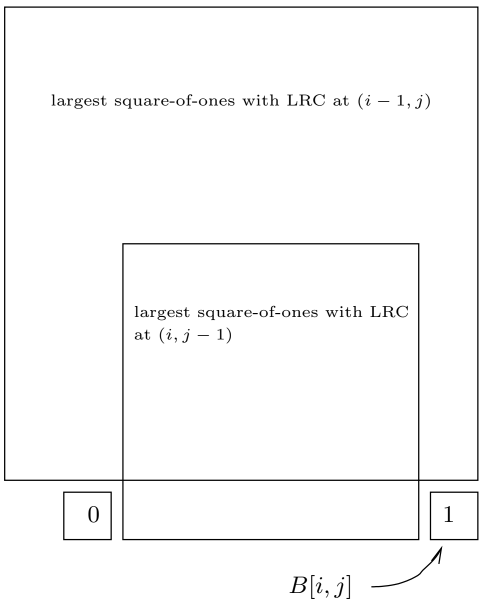
Recursive formula to compute each subproblem:

$$S(i,j) = \begin{cases} B[i,j], & \text{if } i = 1 \text{ or } j = 1 \\ 0, & \text{if } i > 1,\ j > 1, \text{ and } B[i,j] = 0 \\ m + B[i-m, j-m], & \text{if } i > 1,\ j > 1, \text{ and } B[i,j] = 1, \\ & \text{where } m = \min\bigl(S(i-1,j), S(i,j-1)\bigr) \end{cases} \qquad (\dagger)$$

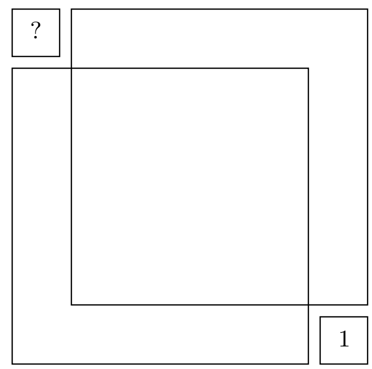
Justification why $(\dagger)$ is a correct formula to compute $(*)$: The first two cases (i.e., when $i = 1$ or $j = 1$, and when $B[i,j] = 0$ are immediate from the definition of $S(i,j)$.

The third case requires explanation, and the figure below may help clarify the discussion. First, by a cut-and-paste argument, $S(i,j) \leq m+1$: otherwise the entire $(m+2) \times (m+2)$ square whose LRC is (i,j) would consist entirely of 1s; therefore, the two $(m+1) \times (m+1)$ squares whose LRCs are $(i-1,j)$ and $(i, j-1)$ would both consist entirely of 1s. Thus, $\min\bigl(S(i-1,j), S(i,j-1)\bigr) \geq m+1$, contradicting the definition of m.

Next, we show that $S(i,j) \geq m$. The $(m+1)\times(m+1)$ square with LRC (i,j) consists entirely of 1s, with the possible exception of $B[i-m, j-m]$. This is because, by definition of m, the two $m \times m$ squares with LRCs $(i-1,j)$ and $(i,j-1)$ consist entirely of 1s. Together with the bit $B[i,j]$ which is 1 (by the hypothesis of the case), this covers the entire $(m+1)\times(m+1)$ square with LRC (i,j), with the exception of $B[i-m,j-m]$. So, in this case, if $B[i-m,j-m]=1$ then $S(i,j)=m+1$; otherwise, $S(i,j)=m$. In other words, $S(i,j) = m + B[i-m,j-m]$, as wanted.



Largest square-of-ones with LRC above and to the left of (i,j) have different sizes



Largest square-of-ones with LRC above and to the left of (i,j) have the same size

<u>*Solving the original problem:*</u> By the definition of the subproblems $(*)$, the required return value is the triple $\big(i,j,S(i,j)\big)$ such that $S(i,j)$ is maximum.

<u>*Pseudocode:*</u>

```
for i := 1 to n do S(i,1) := B[i,1]
for j := 1 to n do S(1,j) := B[1,j]
for i := 2 to n do
    for j := 2 to n do
        if B[i,j] = 0 then
            S(i,j) := 0
        else
            m := min(S(i−1,j), S(i,j−1))
            S(i,j) := m + B[i−m, j−m]
ℓ* := 0
for i := 1 to n do
    for j := 1 to n do
        if S(i,j) > ℓ* then i* := i; j* := j; ℓ* := S(i,j) then
if ℓ* = 0 then return (0,0,−1)
else return (i* − (ℓ* − 1), j* − (ℓ* − 1), ℓ* − 1)
```

<u>*Running time:*</u> The running time of this algorithm is obviously $\Theta(n^2)$.

Answer to Question 3.

(1) Run the Floyd-Warshall algorithm on the given graph G to find shortest paths between every pair of nodes. Let $D(u, v)$ be the weight of a shortest $u \to v$ path as computed by this algorithm ($D(u, v) = \infty$ if there is no such path), and $P(u, v)$ be the predecessor of v on a shortest $u \to v$ path (NIL if $u = v$).

(2) Construct a weighted graph $G' = (V', E')$ such that $(u, v) \in E'$ if and only if $D(u, v) \leq d$, with edge weight function $\mathbf{wt}' : E' \to \mathbb{R}$, where $\mathbf{wt}'(u, v) = D(u, v)$. This represents a map of only the towns with gas stations and and edge between two towns only if you can drive from one to the other in your car without refuelling.

(3) Run Dijkstra's algorithm on G' with start node s and edge weights $\mathbf{wt}'$ to find, for every node $u \in V'$, the weight $D'(u)$ of a shortest $s \to u$ path in G', and the predecessor $P'(u)$ of u on a shortest $s \to u$ path in G'. (Note that $s, t \in V'$, so s and t are nodes in G'.)

(4) If $D'(t) = \infty$ then there is no route you can follow to drive from s to t in your car. Otherwise, use P' and P to recover the desired path: First let $u_1 = s, u_2, \ldots, u_k = t$ be a shortest $s \to t$ path in G': this is the reverse of the path $t, P'(t), P'(P'(t)), \ldots, s$. Then use P to find the shortest $u_i \to u_{i+1}$ path p_i in G, for each $1 \leq i \leq k-1$: this is the reverse of the path $u_{i+1}, P(u_i, u_{i+1}), P(u_i, P(u_i, u_{i+1})), \ldots, u_i$. Finally, concatenate $p_1, p_2, \ldots, p_{k-1}$ and return the resulting path.

Let n be the number of nodes and m be the number of edges in the given graph G.

- The Floyd-Warshall algorithm (step (1)) takes $O(n^3)$ time.
- Constructing the graph G' (step (2)) takes $O(n^2)$ time. (Note that G' has $O(n)$ nodes and $O(n^2)$ edges.)
- Dijkstra's algorithm (step (3)) takes $O(n^2 \log n)$ time.
- Recovering the path (step (4)) takes $O(n)$ time.

Thus, the overall running time is $O(n^3) + O(n^2) + O(n^2 \log n) + O(n) = O(n^3)$ time.