CS 577: Introduction to Algorithms — Homework 6 Solutions

Problem 1

We're given a time limit x and a graph (V, E) with special vertices s and t corresponding to Madison and Seattle, respectively. Let $d(u, v)$ be the time required to go from vertex u to vertex v. Let $position(u)$ be the number assigned to the vertex u, that defines the order in which Alice wants to visit the cities. We want to find the longest $s - t$ path $(v_1, v_2, ..., v_k)$, where $v_1 = s$ and $v_k = t$, such that $\sum_{i=1}^{k-1} d(v_i, v_{i+1}) \leq x$ and each vertex in the path must be numbered higher than the previous vertex in the path.

Let $s = v_1$, $t = v_n$, and assume position(v_i) < position(v_{i+1}) for $1 \leq i < n$. We know $position(s) = 1$ and $position(t) = n$

Intuitively, suppose we arrive at v_i and we'd like to visit j vertices on the remainder of our trip (including v_i and t). Then, wherever we go next, we will still need to visit $j-1$ additional vertices. This motivates the following definition. Let $\mathrm{OPT}[i, j]$ be the smallest amount of time required to go from v_i to t along a valid path while visiting j vertices (including v_i and t). In the base cases, $\mathrm{OPT}[n, 1] = 0$ (it takes no time to visit just Seattle when already in Seattle) and $\mathrm{OPT}[i, j] = +\infty$ for all $n - i + 1 < j$ (Since we are at v_i, we have $n - i$ more vertices that are numbered higher than v_i. It's impossible to visit more than this $n - i + 1$ number of vertices, including v_i along the path to t). Let $k^*[j]$ be the list of vertices along the optimal valid path from v_1 to v_n visiting exactly j vertices. Then the following Bellman equation characterizes $\mathrm{OPT}[i, j]$.

$$\mathrm{OPT}[i,j] = \begin{cases} 0 & : \text{ if } i = n \text{ and } j = 1 \\ +\infty & : \text{ if } n - i + 1 < j \\ \min_{k>i}(\mathrm{OPT}[k, j-1] + d(v_i, v_k)) & : \text{ otherwise} \end{cases} \quad (1)$$

Algorithm:

1. Sort the vertices in order of increasing positions, breaking ties so that t appears last.
2. Set $k^* \leftarrow [\,]$
3. For i in n to 1, j in 1 to n
   (a) Set $k^*[j] \leftarrow \{1\}$
   (b) If $i = n$ and $j = 1$, Set $\mathrm{OPT}[i, j] \leftarrow 0$
   (c) Else If $n - i + 1 < j$, Set $\mathrm{OPT}[i, j] \leftarrow +\infty$
   (d) Else
      i. Set $minDuration \leftarrow +\infty$ and $minK \leftarrow i + 1$
      ii. For k in $i + 1$ to n
         A. $duration \leftarrow \mathrm{OPT}[k, j - 1] + d(v_i, v_k)$
         B. If $duration < minDuration$, set $minDuration \leftarrow duration$ and $minK \leftarrow k$
      iii. Set $\mathrm{OPT}[i, j] \leftarrow minDuration$ and add the vertex $minK$ to $k^*[j]$
4. For j in n to 1, find the largest j such that $\mathrm{OPT}[1, j] \leq x$ and return $k^*[j]$

The optimal path from s to t can be reconstructed by following the vertices in k^* in order.
The correctness of this algorithm follows by induction on $n - i$. Clearly the algorithm is correct when $i = n$ or $n - i + 1 < j$. We previously argued that the Bellman equation is correct for $\mathrm{OPT}[i, j]$ assuming correct values for $\mathrm{OPT}[k, j]$ where $i < k \leq n$. It takes $O(n \log n)$ time to sort the vertices; evaluating (1) takes $O(n)$ time, and there are $O(n^2)$ entries in the table, so filling it requires $O(n^3)$ time; and reconstructing the path takes $O(n)$ time. The total run time is therefore $O(n^3)$.

Problem 2

We will give a dynamic program which constructs the optimal stack using larger and larger subsets of the turtles, in a manner similar to the knapsack algorithm. That is, we build up solutions to subproblems restricted to the first i items, then increment i. For the knapsack problem, the order in which we considered items didn't matter. Here, however, the order in which turtles are added to the stack determines how many more turtles can be added, so we must carefully consider the order in which we choose turtles.

Consider any stack of k turtles σ = (t_1,t_2,...,t_k), where t_1 is on the bottom and t_k is on top. Let $s(t)$ be the strength of turtle t and $w(t)$ the weight. We use an exchange argument to show that the turtles in σ can be stacked in decreasing order by strength: suppose, for some index i, that s(t_i) < s(t_{i+1}). Then we can swap the order of t_i and t_{i+1}, because the weight carried by t_i in the new order does not increase, and the weight carried by t_{i+1} is at most the weight previously carried by t_i which, by feasibility of the stack σ, is at most s(t_i) < s(t_{i+1}). In particular, an optimal stack can be so arranged. In the rest of the solution, we assume the turtles are numbered in order of decreasing strength (as given on the input - i.e. before considering any particular stack). That is, s(t_1) $\geq$ s(t_2) $\geq$ ... $\geq$ s(t_n).

Define the strength of a stack of turtles to be the maximum weight that can be added to the stack. Given a stack σ = (t_1,....,t_k) and a particular turtle t_i in σ, let r(σ, t_i) = s(t_i) - $\sum_{j=i}^{k} w(t_j)$. So r(σ, t_i) is the difference between t_i's strength and the weight it is actually carrying in σ. The strength of the stack is the minimum of this quantity over all turtles: $\min_{t_i \in \sigma}(r(\sigma, t_i))$

Consider the weakest turtle; does it belong in an optimal stack? Suppose we decide we want to include this turtle; then we need to find the largest subset of the first $n-1$ turtles which can form a stack strong enough to support the final turtle. Otherwise, we should simply find the tallest possible stack drawn from the first $n-1$ turtles. In general, when considering the ith turtle, we want to make the highest possible stack which has strength at least s ≥ 0. Formally, let $\mathrm{OPT}[i, s]$ be the maximum number of turtles in a stack with strength at least s using at most the first i turtles; we have just sketched an argument for the following Bellman equation:

$$\mathrm{OPT}[i, s] = \max \begin{cases} \mathrm{OPT}[i-1, s] \\ \mathrm{OPT}[i-1, s + w(t_i)] + 1, \text{only if } s(t_i) - w(t_i) \geq s \end{cases} \tag{2}$$

This equation leads to an algorithm which solves the problem. But the run time would depend on the strength of the strongest turtle, making the algorithm pseudo-polynomial. We can do better by swapping the roles of the (potentially exponential) strength and the (polynomial) count: that is, by putting the strength in the table and making the count an index.

Let $\mathrm{OPT}'[i, h]$ to be the maximal strength of any stack of h turtles drawn from the first i turtles. To determine $\mathrm{OPT}'[i, h]$, we must decide whether or not to include t_i. If not, then clearly $\mathrm{OPT}'[i, h]$ is the strength of a stack of h turtles drawn from the first $i-1$. If we do include t_i, then $\mathrm{OPT}'[i, h]$ is the strength of a stack formed by placing t_i on top of an optimal stack of $h-1$ turtles drawn from the first $i-1$. The strength of this stack is at most $\mathrm{OPT}'[i-1, h-1] - w(t_i)$ (the strength of the base stack after adding t_i), or else $s(t_i) - w(t_i)$ (the capacity of t_i to carry weight beyond its own), whichever is smaller. We have derived the following Bellman equation for $\mathrm{OPT}'[i, h]$.

$$\mathrm{OPT}'[i, h] = \max \begin{cases} \mathrm{OPT}'[i-1, h] \\ \min(\mathrm{OPT}'[i-1, h-1], s_i) - w_i \end{cases} \tag{3}$$

We take as base cases when $i = 0$ (representing that no turtles are available) or $h = 0$. For these, we can set $\mathrm{OPT}'[i, 0] = +\infty$ for all $i = 0, 1, \ldots, n$, since any weight of turtles can be stacked on an empty tower, and $\mathrm{OPT}'[0, h] = -\infty$ for $h > 0$ since it is impossible to make a tower of positive height with no turtles.

Algorithm: We initialize the base cases. Then, for $i = 1, ..., n$ and $h = 1, ..., n$ we set $\mathrm{OPT}'[i, h]$ according to (3). Finally, to recover the solution, we want the largest value of h for which the strongest stack of h turtles has non-negative strength; in other words, we return $max\{h : \mathrm{OPT}'[n, h] \geq 0\}$.

1. Sort turtles by decreasing order of strength, break ties by sorting on decreasing order of weights.
2. For i in 0 to n, set $\mathrm{OPT}'[i, 0] \leftarrow +\infty$
3. For h in 1 to n, set $\mathrm{OPT}'[0, h] \leftarrow -\infty$
4. For i in 1 to n, For h in 1 to i
   (a) $weightIncludingithTurtle = \min(\mathrm{OPT}'[i-1, h-1], s_i) - w_i$
   (b) $weightExcludingithTurtle = \mathrm{OPT}'[i-1, h]$
   (c) Set $\mathrm{OPT}'[i, h] \leftarrow \max(weightIncludingithTurtle, weightExcludingithTurtle)$
5. For h in n to 0, find the largest h such that $\mathrm{OPT}'[n, h] \neq +\infty$ and return it.

Correctness follows by induction: Our algorithm clearly computes the correct value for $\mathrm{OPT}'[i, h]$ in the base cases (when at least one of i or h is 0). We argued in the paragraph preceding our Bellman equation (3) that it is correct for a given value of i, assuming we know the correct values of $\mathrm{OPT}'[i-1, h]$ for all h. Thus our algorithm finds the highest stack possible among all stacks in which the turtles are sorted by strength. Since we have previously argued that the highest such stack is optimal, our algorithm is correct.

The run time analysis is straightforward: Sorting the turtles takes $O(n \log n)$ time; initialization requires $O(n)$ time; evaluating the Bellman equation takes $O(1)$ time, and the table has $O(n^2)$ entries, so filling it out takes $O(n^2)$ time; and finding the final answer takes $O(n)$ time. The total run time is therefore $O(n^2)$.