CS 180: Homework 4

Jonathan Woong

804205763

Winter 2017

Discussion 1B

Wednesday 22nd February, 2017

Problem 1

When their respective sport is not in season, UCLA's student-athletes are very involved in their community, helping people and spreading goodwill for the school. Unfortunately, NCAA regulations limit each student-athlete to at most one community service project per quarter, so the athletic department is not always able to help every deserving charity. For the upcoming quarter, we have S student-athletes who want to volunteer their time, and B buses to help get them between campus and the location of their volunteering. There are F projects under consideration; project i requires s_i student-athletes and b_i buses to accomplish, and will generate $g_i > 0$ units of goodwill for the university.

Use dynamic programming to produce an algorithm to determine which projects the athletic department should undertake to maximize goodwill generated. For full-credit, your algorithm should run in time $O(SBF)$ but you don't have to prove its correctness or analyze the time complexity.

S student-athletes $= \{s_1, s_2, \ldots, S\}$

B buses $= \{b_1, b_2, \ldots, B\}$

F projects $= \{1, 2, \ldots, F\}$

G goodwill $= \{g_1, g_2, \ldots, G\}$

Goal: maximize G

Let O be an optimal solution calculated using $OPT(project, student, bus)$:

If project n is not in O, we must solve the problem for projects $\{1, \ldots, n-1\}$ using $OPT(n-1, s, b)$

If project n is in O, we must solve the problem for projects $\{1, \ldots, n-1\}$, but we know that $OPT(n, s_n, b_n) = g_n + OPT(n-1, S-s_n, B-b_n)$

MAX-GOODWILL-COMPUTE-OPT:

1. Set $OPT(0, s, b) = 0$ for all students $s = \{s_1, s_2, \ldots, S\}$ and all buses $b = \{b_1, b_2, \ldots, B\}$.

Set $OPT(i, 0, 0) = 0$ for all $i = \{1, \ldots, n\}$.

2. For $i = \{1, \ldots, F\}$: $O(F)$

    For $s = \{s_1, \ldots, S\}$: $O(S)$

        For $b = \{b_1, \ldots, B\}$: $O(B)$

            If $s_i > S$ or $b_i > B$:

                $OPT(i, s, b) = OPT(i-1, s, b)$

            Else:

$$OPT(i,s,b) = \max\begin{cases} OPT(i-1,s,b) \\ g_i + OPT(i-1, s-s_i, b-b_i) \end{cases}$$

3. Output $OPT(F,S,B)$.

In order to find the subset that maximizes goodwill, we use the following algorithm:

FIND-SUBSET(i,s,b):

1. If $s_i > s$ or $b_i > b$:

   RETURN FIND-SUBSET$(i-1,s,b)$.

2. Else:

   If $g_i + OPT(i-1, s-s_i, b-b_i) > OPT(i-1,s,b)$:

      RETURN $\{i\}\cup$ FIND-SUBSET$(i-1, s-s_i, b-b_i)$.

   Else:

      RETURN FIND-SUBSET$(i-1,s,b)$.

Problem 2

You have a knapsack of total weight capacity W and there are n items with weights $w_1, \ldots, w_n$ respectively. Give an algorithm to compute the number of different subsets that you can pack safely into the knapsack. In other words, given integers $w_1, \ldots, w_n, W$ as input, give an algorithm to compute the number of different subsets $S \subseteq [n]$ such that $\sum_{i \in S} w_i \leq W$. For full-credit, your algorithm should run in time $O(nW)$ but you don't have to prove its correctness or analyze the time complexity.

Let M be the set of items to be put in the knapsack with total weight $\leq W$.

Let $FEASIBLE(i, w)$ return the set of items in the knapsack with total weight $\leq W$.

If $w_n > W$:

  $FEASIBLE(n, w) = FEASIBLE(n-1, w)$

Else:

  If n not in M:

    $FEASIBLE(n, w) = FEASIBLE(n-1, w)$

  Else n in M:

    $FEASIBLE(n, w) = n \cup FEASIBLE(n-1, w - w_n)$

The algorithm to compute all possible subsets is therefore:

1. Set $FEASIBLE(0, w) = 0$ for all weights $(w_1, \ldots, w_n)$
2. Initialize stack S to be empty.
3. For $i = 1, \ldots, n$:

  For $w = (w_1, \ldots, w_n)$:

    Compute $FEASIBLE(i, w)$ using the recurrence.

    If $FEASIBLE(i, w)$ returns a set that is not currently in S:

      Push $FEASIBLE(i, w)$ onto S.

    Else:

      Do nothing.

4. RETURN the total number of elements in S.

Just like the knapsack algorithm, this algorithm runs in $O(nW)$.

Problem 3

Given two strings $X = x_1x_2\cdots x_m$ and $Y = y_1y_2\cdots y_n$, let $deleteScore(X,Y)$ be the least number of characters you have to delete from X, Y so that you get two same strings. For example, if $X = goodman$ and $Y = goldmann$, then $deleteScore(X,Y) = 3$ (you can delete 'o' from X, 'l' and one 'n' from Y to get the same string - "godman").

Give an algorithm that given two strings X, Y computes $deleteScore(X,Y)$. For full-credit, your algorithm should run in polynomial time.

We define the recurrence relation as:

$$deleteScore(X,Y) = \min \begin{cases} case\ 1: 2 + deleteScore(X-1, Y-1)\ if\ (X_m \neq Y_n) \\ case\ 2: 0 + deleteScore(X-1, Y-1)\ if\ (X_m = Y_n) \\ case\ 3: 1 + deleteScore(m-1, n) \\ case\ 4: 1 + deleteScore(m, n-1) \end{cases}$$

This works much like the edit distance algorithm.

Case 1: Instead of incrementing the cost by 1 when $X_m \neq Y_n$, we increment it by 2. This is because the cost to create two matching strings requires the deletion of X_m and Y_n.

Case 2: When X_m and Y_n match, we don't need to delete anything.

Case 3: When we insert a blank for Y_n in the edit distance algorithm, this is equivalent to deleting X_m.

Case 4: When we insert a blank for X_m in the edit distance algorithm, this is equivalent to deleting Y_n.

Problem 4

There are four types of brackets: (,), <, and >. We define what it means for a string made up of these four characters to be *well-nested* in the following way:

(a) The empty string is well-nested.

(b) If A is well-nested, then so are <A> and (A).

(c) If S, T are both well-nested, then so is their concatenation ST.

Devise an algorithm that takes as input a string $s = (s_1, s_2, ..., s_n)$ of length n made up of these four types of characters. The output should be the length of the shortest well-nested string that contains s as a subsequence.

We use the same algorithm as RNA sequencing shown in class, but do not have the condition that a character can only be paired with another character more than 4 spaces away. Instead, we allow a closing bracket to be paired with a corresponding opening bracket at any prefix position.

Let $OPT(i, j)$ be the length of the shortest well-nested string that contains s.

$$OPT(i,j) = \min \begin{cases} case\ 1 : 2 + OPT(i, j-1) \\ case\ 2 : 1 + OPT(i, t-1) + OPT(t+1, j-1) \end{cases}$$

Case 1: s_i is either a closing bracket that is not paired to an opening bracket or an opening bracket that was never paired with a closing bracket, so we +1 for the unpaired bracket and +1 for inserting the bracket that completes the pair.

Case 2: s_i is a closing bracket that is paried to an opening bracket s_t, so we +1 for the closing bracket and calculate the optimal solution for the two substrings $(s_i, \ldots, s_{t-1})$ and $(s_{t+1}, \ldots, s_{j-1})$.

We evaluate the subproblems in the order of small difference $(j - i)$ to bigger difference, just like the RNA sequencing algorithm:

ALGORITHM:

1. Initialize $OPT(i, k) = 0$ for $i = 0$ and $k = 0$.
2. For $k = 1, \ldots, n-1$:

    For $i = 1, \ldots, n-k$:

        $j = i + k$

        Compute $OPT(i, j)$ using the recurrence.
3. RETURN $OPT(1, n)$.

Just like the RNA sequencing algorithm, this algorithm runs in $O(n^3)$.