Algorithms Fall 2016 Homework 1 Solution

September 28, 2016

PROBLEM 1: (30 POINTS)

We can define c[i, j] to be the optimal cost of breaking the string between positions $L[i]$ (exclusive) and $L[j]$ (inclusive). We have the following recursive formula for the Breaking a String Problem:

$$c[i,j] = \begin{cases} 0 & \text{if } j \le i+1 \\ \min\limits_{i<k<j} (c[i,k] + c[k,j]) + (L[j] - L[i]) & \text{if } j > i+1 \end{cases}$$

Based on the above recursive formulation, we have the pseudo code for computing the optimal cost L as follows:

```
BREAKING-A-STRING(L)
 1  m = L.length
 2  let c[1...m, 1...m] be a new table
 3  let p[1...m, 1...m] be another table
 4  for i ← 1 to m − 1
 5          c[i, i + 1] = 0
 6          p[i, i + 1] = 0
 7  for l ← 2 to m − 1
 8          for i ← 1 to m − l
 9                  j = i + l
10                  p[i, j] = 0
11                  c[i, j] = c[i, i + 1] + c[i + 1, j] + (L[j] − L[i])
12                  for k ← i + 1 to j − 1
13                          q = c[i, k] + c[k, j] + (L[j] − L[i])
14                          if q < c[i][j]
15                                  c[i][j] = q
16                                  p[i][j] = k
17  return c and p
```

We can construct a sequence of breaks that makes the least cost according to the result p-table. We start the algorithm from $p[1][m]$ and retrieve the optimal cut positions recursively until $P[i][j] = 0 (or\ j = i+1)$. The pseudo-code is as follows.

```
PRINT-BREAKS(p, i, j)
1  if j > i + 1
2          print L[p[i][j]]
3          PRINT-BREAKS(p, i, p[i][j])
4          PRINT-BREAKS(p, p[i][j], j)
```

The running time of BREAKING-A-STRING is $O(m^3)$ since there are three levels of loops(line 7 to line 16) . To print the optimal sequence of break points for L, we just need to call the procedure $PRINT-BREAKS(p, 1, m)$. The running time of constructing the optimal sequence is $O(m)$ since the procedure PRINT-BREAKS is called at $m-2$ times and each time it prints one position.

PROBLEM 2: (20 POINTS)

We create two virtual activities a_0 with $s_0 = f_0 = 0$ and a_{n+1} with $s_{n+1} = f_{n+1} = \infty$. Then selecting a maximum-size subset of mutually compatible activities from S is equivalent to calculating $A_{0,n+1}$. Besides table $c[0..n+1, 0..n+1]$, we define table $activities[0..n+1, 0..n+1]$ in which $activities[i, j]$ is the activity k we select to put into A_{ij}.
Since both $S_{i,i}$ and $S_{i,i+1}$ are empty set, then $c[i, i] = 0$ for all i and $c[i, i+1] = 0$ for $0 \le i \le n$.

DYNAMIC-ACTIVITY-SELECTOR(s, f, n)

```
1  let c[0...n+1, 0...n+1] and activities[0...n+1, 0...n+1] be new tables
2  for i = 0 to n
3          c[i, i] = 0
4          c[i, i+1] = 0
5  c[n+1, n+1] = 0
6  for l = 2 to n+1
7          for i = 0 to n − l + 1
8                  j = i + 1
9                  c[i, j] = 0
10                 k = j − 1
11                 while k > i
12                         if f[i] ≤ s[k] and f[k] ≤ s[j] and c[i, k] + c[k, j] + 1 > c[i, j]
13                                 c[i, j] = c[i, k] + c[k, j] + 1
14                                 activities[i, j] = k
15                         k = k − 1
16 activities_set = ∅
17 GET-ACTIVITIES(c, activities, 0, n+1, activities_set)
18 return activities_set
```

GET-ACTIVITIES$(c, activities, i, j, set)$

```
1  if c[i, j] > 0
2          k = activities[i, j]
3          set = set ∪ {k}
4          GET-ACTIVITIES(c, activities, i, k, set)
5          GET-ACTIVITIES(c, activities, k, j, set)
```

The running time is $O(n^3)$.

PROBLEM 3: (20 POINTS)

Intuitively, to get the maximum value, we should take items with more value but with less weight. Since in this problem, the more value the item is, the lesser it weighs, we can greedily pick the most valuable item each time which the knapsack could hold. The iteration ends when the knapsack can't hold any more items.
To prove this greedy algorithm is correct, we prove the following theorem:
Theorem. Suppose there is an item set $< i_1, i_2, ..., i_n >$ with worth $v_1 \ge v_2 \ge ... \ge v_n$ and weight $w_1 \le w_2 \le ... \le w_n$ and the knapsack has a capacity W. We define the knapsack problem as a tuple (I, v, w, W).Then we claim:

1. Optimal Substructure:If $I_{st} =< i_s, ..., i_t >$ is an optimal solution for (I, v, w, W),i.e it is one of the most valuable set of items that the knapsack with capacity W can hold, then if we pick out i_k from I_{st}, $I_{st} \backslash i_k$ must be an optimal solution for $(I \backslash i_k, v \backslash v_k, w \backslash w_k, W - w_k)$.

2. Greedy Choice:There must exist an optimal solution of (I, v, w, W) which include i1.

Proof.

1. The optimal substructure property is trivial. We can prove that by contradiction.

2. Suppose $I_{st} =< is, ..., it >$ is an optimal solution for (I, v, w, W), also $vs \geq ... \geq v_t$, $w_s \leq ... \leq w_t$ and $s \neq 1$. Then we substitute i_1 for i_s to get a set $I' =< i1, ..., it >$ the value of which is no less than I since $v1 \geq vi$ and $w1 \leq wi$. Since I is an optimal solution, I' must also be an optimal solution.

PROBLEM 4: (30 POINTS)

We can prove this by contradiction. Suppose we can find an optimal code of the alphabet whose codeword lengths are not monotonically increasing. Let d be the optimal code whose codeword lengths are not monotonically increasing. It means there exists two characters, let's say a_i and a_j, such that $a_i.freq < a_j.freq$ and $d_i.length < d_j.length$. Suppose the length of document coded by d is L. We can create a new code d' by swapping the codeword of a_i and a_j. That is $d'_i = d_j$ and $d'_j = d_i$. The length of the document coded by d'. Suppose the length of document coded by d' is L', we have:

$$L - L' = a_i.freq * (d_i.length - d'_i.length) + a_j.freq * (d_j.length - d'_j.length)$$
$$= (a_i.freq - a_j.freq) * (d_i.length - d_j.length) > 0$$

This contradicts the assumption that d is an optimal code since there exists another code which generates shorter coded document.