CS1510 Dynamic Programming Problem 20

Rebecca Negley, Sean Myers

October 5, 2011

20. The input to this problem is two sequences $T = t_1, \ldots, t_n$ and $P = p_1, \ldots, p_k$ such that $k \leq n$, and a positive integer cost c_i associated with each t_i. The problem is to find a subsequence of T that matches P with maximum aggregate cost. That is, find the sequence $i_1 < \ldots < i_k$ such that for all j, $1 \leq j \leq k$, we have $t_{i_j} = p_j$ and $\sum_{j=1}^{k} c_{i_j}$ is maximized.

    (a) Give a recursive algorithm to solve this problem. Then explain how to turn this recursive algorithm into a dynamic program.

    Solution: To construct a recursive algorithm, first consider the elements of T. If the last element of T is equal to the last element of P (so $t_n = p_k$), it is possible that the last element of T is in the optimal subsequence. If we know the subsequence $T1$ of $T' = t_1, \ldots, t_{n-1}$ that matches $P' = p_1, \ldots, p_{k-1}$ with maximum aggregate cost and the subsequence $T2$ of T' that matches P with maximum aggregate cost, we can decide whether or not t_n is in the optimal subsequence. If the cost of t_n added to the total cost of $T1$ is greater than the total cost of $T2$, we know t_n is in the optimal solution. Otherwise, t_n must not be in the optimal solution, since it does not produce the maximum aggregate cost. Since finding $T1$ and $T2$ are smaller instances of the larger problem we are trying to solve, we can find them through recursion. We also have to consider if $t_n \neq p_k$. In that case, t_n cannot be in the optimal subsequence. Since elements retain their order, t_n would be the last element in the subsequence and hence must match p_k, the last element of P, to adhere to the problem conditions. Then, $t_n \neq p_k$ gives that the optimal subsequence of T must be equal to $T2$, the subsequence of T' that matches P with maximum aggregate cost. If $k = 0$, we can return 0 cost because the empty subsequence will match P and have no cost. If $k > n$, we can return $-\infty$ because no subsequence of T will match P, leading to no viable solutions. In pseudocode, our recursive algorithm then looks as follows:

```
MAC($n$, $k$)
    if $k = 0$
        return 0
    else if $k > n$
        return $-\infty$
    else
        if $t_n$= $p_k$ then:
            return max(MAC($n-1$, $k-1$)+$c_n$, MAC($n-1$, $k$))
        else:
            return MAC($n-1$, $k$)
```

    To turn this into a dynamic programming algorithm, we construct a 2-dimensional matrix of size $(n+1) \times (k+1)$ and use the base cases of our recursive algorithm as array initialization. We can then fill in the rest of the table by traversing down the rows and going left to right, using our non-base case recursive method with array look ups instead of recursive calls. Our maximum aggregate cost will be in the lower right corner (position (n, k)) of the marix. To find its corresponding subsequence, we simply need to backtrack through the table. The algorithm then looks as follows:

```
MAC($n$, $k$)
    Mac[$n+1, k+1$]
    for $i$ from 0 to $n$ do:
        Mac[$i, 0$] = 0
        for $j$ from $i+1$ to $k$ do:
            Mac[$i, j$] = $-\infty$
    for $i$ from 1 to $n$ do:
        for $j$ from 1 to $i$ do:
            if $t_i = p_j$ then:
                Mac[$i, j$] = max(Mac[$i-1, j-1$] + $c_i$, Mac[$i-1, j$])
            else:
                Mac[$i, j$] =Mac[$i-1, j$]
    return Mac[$n, k$]
```

(b) Give a dynamic programming algorithm based on enumerating subsequences of T and using the pruning method.

Solution: We will first construct a tree where all of the subsequences of T are at the leaves. The tree will have n levels. Each level i corresponds to element t_i. Each node can choose to include t_i in the subsequence or not. Initially, there are 2^n possible subsequences at the leaves, so create two pruning rules:
(1) If a sequence of length j at some node does not match the first j elements of P (or if $j > k$), prune it.
(2) If two sequences at the same level have the same length, prune the one with the lesser aggregate cost. (Since both match P, they have identical elements, so we can extend them the same way.)

We can then construct our dynamic program from our pruned tree. We create an $(n+1) \times (k+1)$ matrix to represent the tree, where the first dimension corresponds to the level and the second corresponds to the length of the subsequence. An outer loop will iterate down the n levels of the tree. At each level, we will derive our information from the level before. If we can add the current element to a subsequence (so that it matches P for one more element), we set the value corresponding to the length of the new subsequence at the next level to the maximum of its current value and the cost of the new subsequence. Our optimal subsequence must be of length k, so our solution cost will be at position (n, k) of our matrix. We can find the corresponding subsequence going back up our table. The algorithm then looks as follows:

```
MAC($n$, $k$)
    Initialize Mac to $-\infty$
    Mac[$0, 0$] = 0
    for $i$ from 0 to $(n-1)$ do:
        for $j$ from 0 to $k$ do:
            if Mac[$i, j$] $\geq 0$:
                Mac[$i+1, j$] = max(Mac[$i+1, j$], Mac[$i, j$])
                Mac[$i+1, j+1$] = max(Mac[$i+1, j+1$], Mac[$i, j$] + $c_i$)
    return Mac[$n, k$]
```

(c) Give a dynamic programming algorithm based on enumerating subsequences of P and using the pruning method.

Solution: Consider a k-level tree, where each level i corresponds to the subsequence $p_1, \dots, p_i$ of P. Each node represents a subsequence of T that matches the subsequence of P represented by the node's level. Then, all possible optimal subsequences of T are on level k of the tree, so we simply need the maximum of the values at that level. Observe that each node can have a variable number of children, including 0 based on the position in T of its last element and how many elements follow it in T that match the next element of P. To cut down on the complexity of the tree, we construct one pruning rule:
(1) If two nodes on the same level represent subsequences of T that end at the same element, prune the one with the lesser aggregate cost. (Since both subsequences end at the same location in T, we can extend them to form an optimal subsequence in the same way, so keep the one with the greatest value up to that level.)

We then construct our dynamic programming algorithm from this tree. We create a $(k+1) \times (n+1)$ matrix to represent the tree, where the first dimension corresponds to the levels of the tree (and, hence, the length of the subsequences) and the second corresponds to the position in T of the last element of the subsequence. We create an outer loop to iterate down the k levels of the tree. We derive our information for each level from the level above it. We create a loop to iterate through the possible locations of the last value of the subsequence (the second dimension of the matrix) and determine the children of each node based on another loop. This inner loop goes through all possible locations of the next item in the subsequence. If the next element of P is equal to the element at t_m, then a new subsequence can be constructed by appending t_m to the subsequence at the parent node. The value of the subsequence gets stored in the cell corresponding the the next level and location m if it is greater than what is already at that location. When we complete our table in this manner, our maximum aggregate sum will then be the maximum of the values in the bottom row of the table. The algorithm looks as follows:

```
MAC($k$, $n$)
    Initialize Mac to $-\infty$
    Mac$[0,0] = 0$
    for $i$ from 0 to $(k-1)$ do:
        for $j$ from $i$ to $n$ do:
            if Mac$[i,j] \geq 0$:
                for $m$ from $j+1$ to $n$ do:
                    if $t_m = p_{i+1}$:
                        Mac$[i+1,m]$ = max(Mac$[i+1,m]$, Mac$[i,j] + c_m$)
    return $\max_j$ Mac$[k,j]$
```