Homework 10

Robbie McKinstry, Jack McQuown, Cyrus Ramavarapu

26 September 2016

Problem 19:

0.1 Part A

First, we will give our recursive solution, and then we will explain how it works.

```
Function: Maximum Matching Sequence (MMS)
Input: $T = t_1, \ldots, t_n, P = p_1, \ldots, p_n, c_i \forall t_i \in T$ Output: Integer
if $P$ is Empty then
    Return: 0
if $S$ is Empty then
    Return: 0
Let $subproblems = ZeroedArray[n]$
foreach $t_i$ in $T$ do
    if $t_i = P_n$ then
        $subproblem[i] = MMS(t[: i], p[: n - 1]) + c_i$
Return: $Max(subproblems)$
```

We consider the final letter of P. That letter must match some letter in T, since it must be the last letter of the sequence. Thus, we search T for all of the sites that could end the sequence. Each of those sites is a subproblem. The subproblem consists of all of the letters up to and excluding the matched letter in T (as it was the final letter of the sequence thus far) and all of the letters in P save the final one, which was just matched. The return value is the maximum cost subproblem, as defined in the problem specification.

To convert this into a recursive solution, create a 2D array $A[n][n]$. Populate A by iterating through the rows, and then the columns. The cell $A[i, j]$ represents the subproblem with $T_{i,j} = T[: i]$ and $P_{i,j} = P[: j]$. Thus, each cell is populated by looking at $c_i + A[i-1, j-1]$ if $T_i = Pj$. The output is the final cell, and the sequence can be found by tracing back through the array and taking a letter from T when value in the cell changes.

0.2 Part B

Notice first that the solution is obviously subcollection of T. Thus, one can identify a solution by search the space of all subcollections of T until finding a match on P.

To do this, for each t_i in T, starting with the empty root node, populate the ith level of the tree by taking all nodes from the $i-1$ level and giving them two children, one for the sequence that concatenates the empty string, and one for the sequence that concatenates t_i (the nodes of the tree store the subsequence). Next, prune any nodes on the same level that do not contain a prefix of P. If they do not contain a prefix of P, then they will never match P. Finally, prune the minimum-valued node if there are two nodes on the same level of the same length. (Since they are the same length and are prefixes, by the 1st rule, they are the same string.)

C:

Instead of enumerating the possible subsequences of $\mathcal{T}$, the possible subsequences of $\mathcal{P}$ can be considered. Unlike the tee developed for the subsequences of $\mathcal{T}$, this tree will have a variable branching factor depending on how many ways a letter in $\mathcal{P}$ can be made using a letter in $\mathcal{T}$

To create this tree, initially consider the first letter in $\mathcal{P}$. Label this letter p_0. Find all the instances where p_0 occurs in in $\mathcal{T}$. These will be the branches from the first node p_0. Next consider all possible ways to make the second letter in $\mathcal{P}$, p_1. These will be the branches from the second level. This process is repeated until all possible ways to make $\mathcal{P}$ are enumerated.

However, not all of these sequences may represent a valid subsequence in $\mathcal{T}$ due to ordering. This gives rise to the first pruning rule.

1. If a child node occurs at an index less than its parent in $\mathcal{P}$, prune it.

Additionally, given this pruning rule, two branches may have the same value. In this case, another pruning rule can be developed.

1. If two branches have teh same value, prune the one ending in the lowest index.

Given these two pruning rules, the following algorithm can be developed.

```
Function: $\mathcal{P}enumerationcostmax$
for $i \leftarrow 0$ to $len(\mathcal{P})$ do
    for $j \leftarrow 0$ to $len(\mathcal{T})$ do
        if $A[i,j]$ is defined and $A[i,j] < j$ then
            $A[i, j + c_j] = \min(A[i, j + c_j], j)$
Return: $Lowest\ value\ of\ k\ for\ which\ A[len(\mathcal{P}), k]\ is\ defined$
```

[H] This algorithm will return the minimum value for the subsequence in $\mathcal{T}$. Looking at the index value in A, will give the last index. By subtracting the value and then looking from the point k in $\mathcal{T}$ backwards allows for the correct subsequence to be reconstructed.

Problem 22:

The solution for this problem is entirely dependent on which trips the Bahncard is bought before. Therefore, the tree of possible solutions will look a little different than the typical trees that have been discussed both in class and in previous homework problems. Instead of including or not including the node, we will be including or not including a Bahncard before each trip T_i. Because of this, there will be many different possible solutions on a certain level of the tree of possible solutions which have a Bahncard bought right before a trip. In this problem, we assume that if a Bahncard is to be bought before a trip it will be purchased on the day of that trip.

Which leads to the following pruning rule:

1. If two nodes at the same depth in the tree have a Bahncard bought trip for trip T_i, prune the node with the higher overall cost.

This pruning rule was motivated by the idea that if a ticket is bought, there is some sequence of ticket purchases upto that point that has a minimum value. Since the price of a Bahnkarte is constant, it only makes sense to buy it for the sequence that has the lowest cost. As a result, for every trip t_i there will be only one instance in which a Bahnkarte will be bought right prior to the trip.

Using this pruning rule will give the following polynomial time algorithm:
The data structure will be a 3-D array $A[t, d, b]$ where $b = 0, 1$
t is the trip count
d is the date at which the last Bahncard was bought

```
Function: Minimum Trip Cost
for $t \leftarrow 0$ to $n$ do
    for $d \leftarrow 0$ to $n$ do
        if $A[t, d, 0]$ is defined then
            $A[t+1, d, 0] = A[t, d, 0] + \mathcal{P}(trip_t)$
            for $k \leftarrow t$ to 0 do
                if $A[k, d, 1]$ is defined then
                    if $date_t - date_k < \mathcal{L}$ then
                        $A[t+1, d, 0] =$
                          $\min A[t+1, d, 0], A[t, d, 0] + .5 \times price_t$
                    else
                        $A[t+1, d, 1] =$
                          $\min A[t+1, d, 1], A[t, d, 1] + price_b + .5 \times price_t$
                    break
return $\min_{0 \leq s \leq n} (A[n, d, *])$
```

The result of the algorithm will be the minimum trip cost possible after all trips have been completed. To recover the dates at which a Bahnkarte was purchased, the index of the minimum cost needs to be back traced until $A[t, d, 1]$ is defined. This will be last time a Bahnkarte was purchased. This processed can be repeated to discover the previous time a Bahnkarte was purchased until all the possible dates are covered.

The runtime of this algorithm is $\mathcal{O}(n^3)$, where n is the number of trips, because all trips have to be considered, and each trip could possibly have a Bahnkarte purchased at the time of the trip, and a search needs to be done to find the last time a Bahnkarte was bought.