Homework 5

Robbie McKinstry, Jack McQuown, Cyrus Ramavarapu

12 September 2016

Greedy Problems

Problem 7:

A greedy algorithm that will minimize evictions is to select the page that will be accessed furthest in the future from the current time.

For example, assuming $k = 4$ let the current state of the system be the following:

Time	1	2	3	4	5	6	7	Fast Memory ($k = 4$)	E	D	F	B
Input	A	C	D	A	B	A	A					

In this situation, the algorithm will evict page B, since it is not used until time 5. Any other choice will result in an earlier eviction.

Proof by Exchange:

Let Alg be the process by which the above algorithm operates. Assume there exists an input I such that $Alg(I)$ is incorrect. Let $Opt(I)$ be the optimal result for I that agrees with the greatest number of steps with $Alg(I)$. $Alg(I)$ and $Opt(I)$ must have a first point of disagreement. Label this time t_i.

At time i, $Alg(I)$ must have evicted some page u and $Opt(I)$ has some page

v evicted. Since $Alg(I)$ always picks the page that will be used furthest in the future, the next time page u will be used must be further into the future than the next time page v is used. Let these times be respectively t_u and t_v.

Since $Alg(I)$ and $Opt(I)$ both agreed upto time t_i, pages u and v must be in the fast memory of $Opt(I)$ at t_i. Therefore, define $Opt'(I)$ as $Opt(I)$ except at time t_i evict page u instead of v.

To show that $Opt'(I)$ is at least as optimal as $Opt(I)$ it needs to be recognized that since $t_u > t_v$, page v will have to be brought back into memory at least as many times as page u before page u is brought back into memory.

Additionally, due to the fact that the pages in $Opt'(I)$ and $Opt(I)$ are the same except for the page needed at t_u is repalced with that at t_v, $Opt'(I)$ will not evict a page that $Opt(I)$ will not evict until at least after time t_u. This means that there will be no more evictions in $Opt'(I)$ than in $Opt(I)$. Since $Opt'(I)$ is at least as optimal as $Opt(I)$ and agrees with $Alg(I)$ for 1 more step a contradiction has be reached. Therefore, there is not input for which Alg is incorrect.

Problem 17:

A greedy algorithm that will determine if there is enough information available to fairly partition the goods is as follows:

Given a set of goods $\mathcal{G} = G_1, G_2, \ldots, G_n$, and two orderings on this set, $\mathcal{H} = G_a > G_b > \cdots > G_n$ and $\mathcal{W} = G_i > G_k > \cdots > G_m$, partition the goods by initially giving $\mathcal{W}$ and $\mathcal{H}$ their maximal elements. Remove the element given to the opposing list from each list. If this is not possible because the maximal elements for both $\mathcal{W}$ and $\mathcal{H}$ are the same, conclude *that there is not enough information to guarantee a fair partitioning.*

While there are elements in $\mathcal{G}$, continue giving $\mathcal{W}$ and $\mathcal{H}$ their second most desired element. Ties are broken arbitrarily. For example, if at a given step both $\mathcal{W}$ and $\mathcal{H}$ want G_c, arbitrarily give G_c to $\mathcal{W}$ and cross it off in the ordering for $\mathcal{H}$ and give $\mathcal{H}$ the next element in the ordering.

After all elements have been assigned,first evenly parition $\mathcal{H}$ and $\mathcal{W}$ into two equal portions. If this is not possible because $|\mathcal{G}|$ is odd, conclude that *there is not enough information available to fairly partition the goods*. If the upper half of $\mathcal{H}$ and $\mathcal{W}$ is full, conclude that *there is enough information to fairly partition*.

If the upper half of $\mathcal{H}$ or $\mathcal{W}$ are not full, begin iterating through both orderings and check to see that items given alternate with items not given. Repeat this process for alternation in groups of $2,\ 3, \ldots,\ |\mathcal{G}|/2$. Every assigned element must belong to at least 1 of the alternating patterns. If an assignment was reached such that elements from $\mathcal{G}$ can be given to $\mathcal{H}$ and $\mathcal{W}$ such that both pass at least one test, conclude that *there is enough information to fairly partition the goods*. Otherwise, conclude that *there is not enough information available*.

Proof by Exchange:

This algorithm can be shown to be correct by an exchange argument. Let Alg be the process by which the above algorithm operates. Assume there exists some input I such that $Alg(I)$ is incorrect. Let $Opt(I)$ be the optimal output for input I that aggrees with $Alg(I)$ for the most number of steps. Therefore, $Alg(I)$ and $Opt(I)$ must have a first point of disagreement. Label this step s_i.

At s_i, $Alg(I)$ must have given a different pair of goods to $\mathcal{H}$ and $\mathcal{W}$ than $Opt(I)$. This could mean that $Alg(I)$ respectively gave $\mathcal{H}$ and $\mathcal{W}$ $G_a,\ G_b$ whereas $Opt(I)$ gave $G_c,\ G_d$. However, since Alg attempts to assign the maximal element at each step to $\mathcal{H}$ and $\mathcal{W}$ and since $Alg(I)$ and $Opt(I)$ agree upto step s_i, the items $G_a,\ G_b$ must be assigned in $Opt(I)$ at a step, s_j, after s_i. As a result, define $Opt'(I)$ as $Opt(I)$ except at s_i assign G_a and G_b to the same lists as $Alg(I)$ and move the items originally assigned in $Opt(I)$ at s_i to s_j.

To check that $Opt'(I)$ is at least as optimal as $Opt(I)$, the case in which $Opt(I)$ determines that there is enough information to partition the goods and the case in which it finds the information insufficient need to be considered separately. In the first case, the condition for enough information requires the sequence of assigned goods and unassigned goods follow an alternating pattern. Since $Alg(I)$ and $Opt(I)$ agreed until step s_i alternation must have already started and to continue, $G_a,\ G_b$ and $G_c,\ G_d$ must have been in alternating positions. Exchanging these values therefore does not change the pattern, giving the same result.

In the case that $Opt(I)$ finds the ordering information insufficient to fairly partition the goods, there will be an element that does not belong to any alternating pattern of the elements assigned to $\mathcal{H}$ and $\mathcal{W}$. Since the elements assigned by $Opt(I)$ at s_i are not the maximal elements available, swaping $G_a,\ G_b$ instead of $G_c,\ G_b$, will not change the patterning since the elements are ordered. As a result, $Opt'(I)$ will also conclude that there is not enough information available.

Since in both cases $Opt'(I)$ is at least as optimal as $Opt(I)$ and agrees with $Alg(I)$ for one more step than $Opt(I)$, a contradiction has been reached. Therefore there does not exist an input for which $Alg(I)$ is incorrect.

Dynamic Programming

Problem 2:

The longest common subsequence between the three can be initially defined recursively by considering the last letter in common between the three strings and then seeing if this letter is in the longest common subsequence of one letter shorter.

$$S_A = A_1, A_2, A_3, \ldots, A_{i-1}, A_i$$
$$S_B = B_1, B_2, B_3, \ldots, B_{j-1}, B_j$$
$$S_C = C_1, C_2, C_3, \ldots, C_{k-1}, C_k$$

Reading right to left, the first letter common to each string will be the last letter in the longest common subsequence. As a result, once this letter is found, problem can be redefined in terms of the shorter substrings formed by ignoring the first common point and all suceeding letters.

This analysis leads to the following recursive algorithm:

Function: *LCS*
Input: *int i, int j, int k*
if $i \equiv j \equiv k \equiv 0$ **then**
| *return* 0
end
if $A_i \equiv B_j \equiv C_k$ **then**
| $LCS(i-1, j-1, k-1) + A_i$
else if $(A_i \equiv B_j) \neq C_k$ **then**
| $\max(LCS(i-1,\ j-1,\ k),\ LCS(i,\ j,\ k-1))$
else if $(A_i \equiv C_k) \neq B_j$ **then**
| $\max(LCS(i-1,\ j,\ k-1),\ LCS(i, j-1, k))$
else if $A_i \neq (B_j \equiv C_k)$ **then**
| $\max(LCS(i,\ j-1,\ k-1),\ LCS(i-1,\ j,\ k))$
else
| $\max(LCS(i,\ j-1,\ k-1), LCS(i-1,\ j,\ k-1), LCS(i-1,\ j-1,\ k))$
end

Although this algorithm produces the longest common subsequence for three given strings, it runs in exponential time due to the numerous recursive calls that operate on a problem of only 1 letter smaller.

By moving to an array based solution and changing the recursive calls to array look-ups, a polynomial runtime algorithm can be developed.

Function: *Array LCS*
Input: *string A, String B, String C*
Initialization:
Array[ ][ ][ ] LCS
for $i \leftarrow 0$ *to* $len(A)$ **do**
| $LCS[i][0][0] = 0$
end
for $j \leftarrow 0$ *to* $len(B)$ **do**
| $LCS[0][j][0] = 0$
end
for $k \leftarrow 0$ *to* $len(C)$ **do**
| $LCS[0][0][k] = 0$
end
Array Calculations:
for $i \leftarrow 0$ *to* $len(A)$ **do**
  for $j \leftarrow 0$ *to* $len(B)$ **do**
    for $k \leftarrow 0$ *to* $len(C)$ **do**
      if $A_i \equiv B_j \equiv C_k$ **then**
        $LCS[i][j][k] = LCS[i-1][j-1][k-1] + 1$
      else if $(A_i \equiv B_j) \neq C_k$ **then**
        $LCS[i][j][k] = \max(LCS[i-1][j-1][k],\ LCS[i][j][k-1])$
      else if $(A_i \equiv C_k) \neq B_j$ **then**
        $\max(LCS[i][j][k] = LCS[i-1][j][k-1],\ LCS[i][j-1][k])$
      else if $A_i \neq (B_j \equiv C_k)$ **then**
        $LCS[i][j][k] = \max(LCS[i][j-1][k-1],\ LCS[i-1][j],[k])$
      else
        $LCS[i][j][k] = \max(LCS[i][j-1][k-1],$
        $LCS[i-1][j][k-1],\ LCS[i-1][j-1][k])$
      end
    end
  end
end

By tracing backwards from $LCS[len(A)-1][len(B)-1][len(C)-1]$ following the path of the largest lengths, the *String* value of the longest common subsequence can be recovered.