CPSC-320 Intermediate Algorithm Design and Analysis Winter 2000

Solutions to Homework # 4

1. This is a problem for which natural greedy approaches don't seem to work. One such approach would be to choose the node with the highest conviviality rating first, remove its parent and children, and continue until the tree is empty. Unfortunately that does not work on the following simple tree (drawn "sideways"): 3 —¿ 6 —¿ 4.

   A straightforward (exponential time) algorithm is as follows. Let N be the root of the tree. In an optimal guest list, either N is invited or N is not invited. If N is invited, then the optimal guest list consists of N plus the optimal guest lists for the trees rooted at N's grandchildren. Otherwise, the optimal guest list consists of the optimal guest lists of the trees rooted at N's children.

```
guest_ratings(T)
  if T contains 1 node, return convivality rating of that node {base case}
  else
     R-root = Rating of the root of the tree
     For each grandchild, i, of the root do
        Rroot = Rroot + guest_ratings (Tree rooted at i)

     R-no-root = 0
     For each child, i, of the root do
       R-no-root = R-no-root + guest_ratings (Tree rooted at i)

     return Max(Rroot, R-no-root)
```

   To avoid duplicate recursive calls, a one-dimensional array can be used. If the nodes of the tree are numbered from 1 to n, then the ith array entry records the rating of the optimal guest list of the tree rooted at node i. We call the array GR (for guest rating). Here is the revised algorithm:

```
for i <-- 1 to n do GR[i] = -1 {initialization}

guest_ratings(T)
  if T contains 1 node, return convivality rating of that node {base case}
  else
     R-root = Rating of the root of the tree
     For each grandchild, i, of the root do
        if GR[i] = -1 then GR[i] <-- guest_ratings(Tree rooted at i)
        Rroot = Rroot +  GR[i]

     R-no-root = 0
```

```
For each child, i,  of the root do
   if GR[i] = -1 then GR[i] <-- guest_ratings(Tree rooted at i)
   R-no-root = R-no-root + guest_ratings (Tree rooted at i)

return Max(Rroot, R-no-root)
```

This algorithm runs in linear time: There is at most one call of guest-ratings for each node of the graph and the total time for these calls is proportional to the number of edges of the tree, which is $n-1$ if the tree has n nodes.

2. Let S and T be the two input strings, each of length n. This algorithm computes the match count of pairs of substrings of length k from S and T. It does so by filling in values in a 2-D array, MC, of size $(n-k+1)$ by $(n-k+1)$. Row i of the array represents the ith substring of length k from the string S and column i of the array represents the ith substring of T of length k.

   The algorithm first computes the entries of the first row and column of the array, using k comparisons for each. Once the first row is computed, the entries in the second row can be computed with just O(1) additional work per entry. This is because for $j > 1$, the match-count of entry $[2, j]$ equals the match count of entry $[1, j-1]$ plus 1 if the last two symbols of the substrings 2 and j match, minus 1 if the first two symbols of the substrings 1 and $j-1$ match. Thus, just two more comparisons are needed to compute MC$[2, j]$ from MC$[1, j-1]$. Similarly, once row 2 of the array has been computed, row 3 can be computed with $O(1)$ additional operations per entry.

   This makes a total of about $2(n-k+1)k$ comparisons for the first row and column plus about $2(n-k+1)^2$ comparisons for the rest of the array. The total running time to compute the whole array is $O(n^2 + nk) = O(n^2)$.

   Example 2-D array showing the number of comparisons needed to compute each match count (two strings of length n, matching substrings of length k). (Note that the array entries here represent the number of comparisons, not the match counts.)

```
          ------(n-k+1)----------
       _  |_______________________|
      |   |k  k  k  k  k          |
      |   |k  2  2  2  2          |
(n-k+1)   |k  2  2  2  2          |
      |   |k  2  2  2  2          |  n
      |   |k  2  2  2  2          |
      |   |                       |
      |   |                       |
      |_  |_______________________|

                      n
```

3. a) The maximum size of (3, 2)-Hamming set is 4. An example set of size of 4 is 000 011 101 110.

   b) For instance, any sequence that starts with 000 111 or 111 000 will not return a (3, 2)-Hamming set of maximum size. There can be many more examples.

   c) In the worst case, the algorithm iterates the size of the set returned, say m, times over the size of M, which is $O(2^n)$. Hence the total running time of the algorithm is $O(2^n m)$.

4. Suppose that $n = 4, d = 2$. Let M be the list containing (in order)

   0000,0101,1100,0110,1001,1011,0001,0010,0100,0011,0111, 1000,1010,1110,1101,1111.

   Running the algorithm provided:

   - 1st iteration: the 1st string is 0000, so 0001,0010,0100,1000 are removed from M.
     Now, M is 0000,0101,1100,0110,1001,1011,0011,0111, 1010,1110,1101,1111.
   - 2nd iteration: the second string is 0101, so 0111,1101 are removed.
     Now M is 0000,0101,1100,0110,1001,1011,0011,1010,1110,1111.
   - 3rd iteration: the third string is 1100, so 1110 is removed.
     Now, M is 0000,0101,1100,0110,1001,1011,0011,1010, 1111.
   - 4th iteration: the fourth string is 0110, so no strings are removed at all (M-old == M).
     Therefore, M is output as the (4,2)- Hamming set.

   At the end of the algorithm, M is the list

   0000,0101,1100,0110,1001,1011,0011,1010,1111.

   But $d(1001, 1011) = 1 < d = 2$. Therefore, the algorithm is incorrect.