CPSC-320 Intermediate Algorithm Design and Analysis Winter 2000

Solutions to Practice Homework # 4

1. A natural greedy approach to this problem is to always apply whichever of the six operations is applicable and is cheapest. For example, in converting "algorithm" to "altruistic," suppose that the *insert* operation is cheaper than anything else. The greedy approach would apply *insert* "a". Then, the "a" in "algorithm" must be deleted (or killed). If delete is an expensive operation, it may be that it would have been better to do a *copy*, rather than an *insert*, in the first place.

   Unfortunately, it does not seem possible to make a decision on which operation to apply first without taking into account what effect the choice of first operation has on the cost of the remaining subproblem.

   A simple and correct (but expensive!) algorithm for the problem is as follows. For each of the six possible operations, say op, calculate the cost of applying op and add this to the cost of solving the remaining subproblem (this cost can be calculated recursively). For any given subproblem, some operations may not be applicable. (For example, the *twiddle* operation is not applicable at the start of the algorithm on the above example input, since *twiddle* applied to "al" is "la" which is not the start of "altruistic".) If an operation is not applicable on string pair S[i...n], T[j...m] then the cost of applying that operation is set to infinity.

   The following algorithm returns the cost of the cheapest sequence of operations that can be applied to convert S[i..n] to T[j..m]. In order to focus on the important ideas, the details of determining if a particular operation is "applicable" to strings S[1..n] and T[1..m] are not included. (The "insert" operation is applicable if and only if $j \leq m$ because in this case there is still a letter that needs to be inserted in the target string. The "copy" operation is applicable if and only if $i \leq n, j \leq m$, and S[i] = T[j]. And so on.)

```
edit_distance(S[i..n], T[j...m])
   if i>n and j>m then return 0 {this is the base case;
                                 both S and T are empty}

   else {1. compute the overall cost of first applying insert}
        if insert is not applicable then T-insert <-- infinity
        else
           T_insert <-- cost(insert) + edit_distance(S[i...n], T[j+1...m])


        {2. compute the overall cost of first applying delete}
        if delete is not applicable then T_delete <-- infinity
        else
        T_delete <-- cost(delete) + edit_distance(S[i+1...n], T[j...m])

        {3. compute the overall cost of first applying copy}
        if copy is not applicable then T_copy <--  infinity
        else
           T_copy <-- cost(copy) + edit_distance(S[i+1...n], T[j+1...m])

        {4. compute the overall cost of first applying replace}
        if replace is not applicable then T_replace <-- infinity
        else
           T_replace <-- cost(replace) + edit_distance(S[i+1...n], T[j+1...m])
        {5. compute the overall cost of first applying twiddle}
        if twiddle is not applicable then T_twiddle <-- infinity
        else
           T_twiddle <-- cost(twiddle) + edit_distance(S[i+2...n], T[j+2...m])

        {6. compute the overall cost of first applying kill}
        if kill is not applicable the T_kill <-- infinity
        else
           T_kill <-- cost(kill) + edit_distance(S[n+1,n], T[j...m])

        return min(T_insert, T_delete, T_replace, T_copy, T_twiddle, T_kill)
```

We now modify the algorithm so that duplicate recursive calls are avoided. Each recursive call has parameters of the form S[i...n] and T[j...m] where $1 \leq i \leq n+1$ and $1 \leq j \leq m+1$. Hence there are a total of $(m+1)(n+1)$ possible recursive calls. We can store the results of these in a table as they are computed. Call the table Edit-Table. Initially, all entries of the table are set to -1. Here is the modified algorithm. It has running time $O(mn)$.

```
edit_distance(S[i..n], T[j...m])

     if i>n and j>m then return 0 {base case; both S and T are empty}

     else {1. compute the overall cost of first applying insert}
        if insert is not applicable then T-insert <-- infinity
        else
           if Edit-Table[i,j+1] = -1 then
              Edit-Table[i,j+1] <-- edit_distance(S[i...n], T[j+1...m])
           T_insert <-- cost(insert) + Edit-Table[i,j+1]

        {2. compute the overall cost of first applying delete}
        if delete is not applicable then T_delete <-- infinity
        else
           if Edit-Table[i+1,j] = -1 then
              Edit-Table[i+1,j] <-- edit_distance(S[i+1...n], T[j...m])
           T_delete <-- cost(delete) + Edit-Table[i+1,j]

        {3. compute the overall cost of first applying copy}
        if copy is not applicable then T_copy <--  infinity
        else
           if Edit-Table[i+1,j+1] = -1 then
           Edit-Table[i+1,j+1] <-- edit_distance(S[i+1...n], T[j+1...m])
           T_copy <-- cost(copy) + Edit-Table[i+1,j+1]

        {4. compute the overall cost of first applying replace}
        if replace is not applicable then T_replace <-- infinity
        else
           if Edit-Table[i+1,j+1] = -1 then
              Edit-Table[i+1,j+1] <-- edit_distance(S[i+1...n], T[j+1...m])
           T_replace <-- cost(replace) + Edit-Table[i+1,j+1]

        {5. compute the overall cost of first applying twiddle}
```

```
          if twiddle is not applicable then T_twiddle <-- infinity
         else
             if Edit-Table[i+2,j+2] = -1 then
                Edit-Table[i+2,j+2] <-- edit_distance(S[i+2...n], T[j+2...m])
             T_twiddle <-- cost(twiddle) + Edit-Table[i+2,j+2]

          {6. compute the overall cost of first applying kill}
          if kill is not applicable the T_kill <-- infinity
          else
             if Edit-Table[n+1,j] = -1 then
                Edit-Table[n+1,j] <-- edit_distance(S[n+1...n], T[j...m])
             T_kill <-- cost(kill) + Edit-Table[n+1,j]

          return min(T_insert, T_delete, T_replace, T_copy, T_twiddle, T_kill)
```

2. (a) The string 'ababababab' is an example of two maximal tandem arrays of base 'abab', that overlap. One has shift 0 and repeats twice 'abababab ab' The other has shift 2 and repeats twice 'ab abababab'
   (b)

```
// used by function tandem_array()
     function compute_pi ( P[0..m-1], pi[0..m-1] )
     {
        pi[0] = 0
        q = 0

        for i = 1 to m-1 do{       // O(m) comparisons
             while ( (q>0) and (P[i] != P[q]) )
                  q = pi[q-1]
             if ( P[i] == P[q] )
                  q = q + 1
             pi[i] = q
        }
     }

     function tandem_array ( T[0..n-1], P[0..m-1] )
     {
        ta = array of int [0..n-1]  // array initalized to all zeros
        pi = array of int [0..m-1]

        compute_pi(P,pi)
        q = 0
        for i = 0 to n-1 do{        // O(n) comparisons
             while ( (q>0) and (P[q] != T[i]) )
                  q = pi[q-1]
             if ( P[q] == T[i] )
                  q = q + 1
             if ( q == m ){         // we have a match
                  shift = (i-m+1)
                  if (shift < m)
                        // if pattern occurs in first characters,
                        // it is the first occurrence
                        ta[shift] = 1
                  else{
                        // otherwise, append it to the last one
                        ta[shift] = ta[shift-m] + 1
                        ta[shift-m] = 0
                  }
                  q = pi[q-1]
        } // end for

        // Print out all Maximal Tandem Arrays
        for i = 0 to n-1 do
             if(ta[i] > 1{
                  // tandem arrays must repeat more than once
                  print "Maximal Tandem Array occurs with shift"
                  print ( i-(ta[i]-1)*m )
                  print "and repeat of"
                  print ( ta[i] )
             }

     } // end of function tandem_array()
```

This algorithm uses the Knuth-Morris-Pratt Algorithm to find each matching pattern. But instead of printing out each match, it stores the match in an array TA[0..n]. The match is stored in the array by placing the number of repeats in the 'shift' element of the array. The number of repeats is determined by adding one to the previous m character's repeat count, even if it is zero (not a match).

Therefore, the running time is the same as the KMP algorithm which is $O(m)$ for the compute-pi() function, and $O(n)$ for the tandem-array() function.