CPSC-500 Fundamentals of Algorithm Design and Analysis Winter 2013

Solutions to Homework # 4

1. Let $T = x_1 + x_2 + \ldots + x_n$ be the total cost of the treats. The treats can be split fairly if and only if there is a subset S of the treats whose values sum to $T/2$. A recursive algorithm proceeds as follows. When considering the last treat, x_n, there are are two possibilities: we could add x_n to the subset S or not. If x_n is added to the subset, we then need to solve a smaller problem, namely to determine whether there is a subset of the treats $x_1, x_2, \ldots, x_{n-1}$ that sum to $T/2 - x_n$. If x_n is not added, then again we need to solve a smaller problem, namely to determine whether there is a subset of the treats $x_1, x_2, \ldots, x_{n-1}$ that sum to $T/2$.

   For $0 \leq i \leq n$ and $0 \leq t \leq \lceil T/2 \rceil$, let $P[i,t]$ be true if and only if it is possible to find a subset of the treats $x_1, \ldots, x_i$ that sum to t. Then if $i > 0$,

   $$P[i,t] = P[i-1, t-x_i] \text{ OR } P[i-1,t].$$

   This leads to the following algorithm. We assume that the algorithm has access to an array $x[1...n]$ of the treat values, which are all positive. Our goal is to compute $P[n, T/2]$. The following dynamic programming algorithm does this.

```
P[0,0] <-- true
for t = 1 to T/2 do P[0,t] <-- false

for i = 1 to n do
    for t = 1 to T/2 do
        if t - x[i] >= 0 then
            P[i,t] <--  P[i-1, t-x[i]] OR P[i-1,t]
      else // t - x[i] < 0
           P[i,t] <-- P[i-1,t]

return P[n,T/2]
```

   The overall running time is $O(nT)$.

2. Let G be an acyclic network and, as in the paper, let the nodes of the network be numbered from 1 to N and let 1 and N be a source and destination node, respectively. For simplicity, we'll also assume that the nodes are numbered in topological order. The algorithm is as follows:

```
main(G,e)
  // input G is a dag with nodes 1 through N in topological order;
  // there is a weight t(x,y) associated with each arc (x,y)
  // input e is a positive number
```

```
  // first calculate f(i), the length of the shortest
  // path from node i to node N:
  for ( i = N-1 to 1 )
      {
      f(i) = min{ t(i,j) + f(j): (i,j) an arc}
      }

  // then use depth first search to find output the desired paths:
  { dfs-find-paths(1,{empty path},0) }


// this procedure does the real work, and relies on input e in
// addition to the parameters that are passed to it
procedure dfs-find-paths(x,P,d)
   // given as input a node x of G with x <> N and a path
      P from 1 to x of distance d, this procedure outputs
      all paths from 1 to N that start with P whose length
      is at most f(1) + e

   for all y such that (x,y) is an arc of G
      {
      if ( d + t(x,y) + f(y) <= f(1) + e )
         // P can be extended to a desired path via node y
         {
         push (x,y,t(x,y))
         }

   while ( top == (x,*,*) )
      // the leftmost entry on the top of the stack is x
      {
      pop(x,y,t);
      if ( y = N ) then { output P,y}
                   else { dfs-find-paths(next,P,y,d + t) }
      }
end // procedure dfs-find-paths(x,P,d)

end // algorithm main(G,e)
```

3. (a) The string "abababababab" 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) Following is an algorithm to compute all tandem arrays:

```
// this function is 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] )
}
```

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.

4. A straightforward approach is to use dynamic programming to align all pairs of circular shifts. There are $O(mn)$ such pairs, and each global alignment takes $O(mn)$ time, resulting in a $O(n^2m^2)$-time algorithm.

   An improved algorithm is obtained by noticing that the best global alignment of all pairs of circular shifts of input strings x and y is in fact the best global alignment of $x_1 \ldots x_n$ with all circular shifts $y_{j+1} \ldots y_m y_1 \ldots y_j$ of y, for $0 \leq j \leq m-1$. To see why, note that any other global alignment with score s can be circularly shifted so that the leftmost symbol of the circular shift of x is x_1, while maintaining score s. This reduces the number of needed global alignments to m, resulting in a $O(n^2m)$ algorithm. (Award 7 points for this approach.)

   There are better algorithms; here is a sketch of one approach. Suppose that $n < m$; otherwise swap x and y in what follows. First, construct a grid graph G with $(n+1) \times (2m+1)$ nodes. The rows of the graph correspond to prefixes of x, including the empty prefix, and the columns correspond to prefixes of yy. In what follows let $y' = y'_1 y'_2 \ldots y'_n = yy$. The weighted edges of G are of three types:

   $([i,j],[i+1,j])$, with weight $\delta(x_{i+1}, -)$

   $([i,j],[i,j+1])$, with weight $\delta(-, y'_{j+1})$

   $([i,j],[i+1,j+1])$, with weight $\delta(x_{i+1}, y'_{j+1})$

   These edges and weights are chosen so that any shortest path from $[0,j]$ to $[n,j+m]$ corresponds to a best global alignment between $x_1 x_2 \ldots x_n$ and the circular shift $y_{j+1} y_{j+2} \ldots y_m y_1 \ldots y_j$. Call such a shortest path $P(j)$, for $0 \leq j \leq m-1$.

   The key idea is to find paths $P(0), P(1), \ldots P(m-1)$; if $P(j)$ is the shortest of all of these paths then the overall best global alignment of a circular shift of y to x is that circular shift $y_{j+1} y_{j+2} \ldots y_m y_1 \ldots y_j$.

How can these paths be found efficiently? A second key point is that it is sufficient to find *non-crossing* paths. Paths $P(k)$ and $P(k')$ with $j < j'$ are non-crossing if for all i, if $[i, j]$ is a node of $P(k)$ and $[i, j']$ is a node of $P(k')$ then $j \leq j'$. Using this assertion, it is possible to find non-crossing paths in a recursive fashion. First find $P(0)$ and $P(m-1)$; use the standard global alignment for these. Then find a path $P(\lceil m - 1/2 \rceil)$ that does not cross $P(0)$ and $P(m-1)$. This path partitions grid G into two parts and a divide and conquer approach can be used to find further paths in these two parts.

(Award 10 points for any correct solution that uses asymptotically less than $O(n^2 m)$ time, and 8 or 9 points to any solution that explores promising ideas to do this.)