INF 4130 Exercises, Sept. 6 and 9, 2016 w/solutions

Exercise 1

1.1 (One may go through e.g. Exercise 1.2, 1.3, before looking at this more theoretical Exercise 1.1) We shall here look closer at the argument for why the recurrence formula given in Chapter 20.5 (and many places in the slides) is correct, so that the described algorithm will correctly solve the *edit distance problem*.

An argument for this is given in the textbook at page 644, about 12 lines from bottom. A central sentence here is: "We can assume without loss of generality that the sequence of operations involving the first *i*-1 characters of P and the first *j*-1 characters of T are operated on first". However, this is, at least to the current writer, not obvious, and we shall look at another argument for proving the formula. We shall look separately at case 1 and case 2 (see slides), and show that:

Case 1 (P[i] =T[j]): If D[i-1, j-1] is optimal (for the corresponding smaller problem), then the computed value for D[i, j] is optimal for that problem.

Case 2 (P[i] != T[j]): If D[i-1, j-1,], D[i, j-1], and D[i-1, j] are all optimal (for the corresponding smaller problems), then the computed value for D[i, j] is optimal for that problem.

As a basis for the proof, we shall assume that a transformation with as few operations as possible (that is, with *edit distance* number of steps) can always be written in the following form:

```
  l o g a r i t h m
a l   g o r i t h m
--------------------
+   -   *
```

Here '+' means an insertion,'-' means a deletion, and '*' means a substitution. Give a proof for the recurrence relation using this assumption.

Answer:

There are two cases to consider, Case 1: P[i] = T[j] and Case 2: P[i] != T[j]. We are doing case 1 below, but not Case 2. However Case 2 can be handled in much the same way, and the students are recommended to write down a proof also for that case.

A proof for Case 1(P[i] = T[j]) can run as follows: We assume that the correct ED between P[1:i-1] and T[1:j-1]) occurs in D[i-1, j-1], and from this we want to conclude that the ED between P[1: i] and T[1: j]) is also equal to D[i-1, j-1]. It is obvious that it cannot be larger (as we can easily construct a transformation between P[1: i] and T[1: j]) with this length), so the problem is to show that it cannot be smaller either.

To prove this we assume the opposite (that is, ED(P[1: i], T[1: j]) != D[i-1, j-1]), and will show that this leads to a contradiction. From this, and the above comment) we can also conclude that ED(P[1: i] ,T[1: j]), say *k*, is smaller than D[i-1, j-1]. Thus, there is a transformation T from P[1: i] to T[1: j]) with *k* steps.

As the transformation T is minimal, it can be written at the form indicated above. We will then consider two cases:

Case A: The letters P[i] and T[j] will be positioned in the same (last) column, like this:

```
. . . . . . . . P[i]
. . . . . . . . T[j]
-------------------------
+‘s, -‘s and *‘s          ←  k in total
```

Here, all the single edit-operations must happen to the left of the last column, and by using these *k* operations we could obviously also find a transformation from P[1: i-1] to T[1: j-1] with length *k*. However, k < D[i-1, j-1], and this is against the assumption that D[i-1, j-1] was the shortest transformation between P[1:i-1] and T[1:j-1] .

Case B: In a setup like above of the transformation T, P[i] and T[j] will not be positioned in the same column, but e.g. like this:

```
. . . . . . . .  x   y   z  P[i]
. . . . . . . . T[j]
-------------------------------
                 ?   -   -   -
```

Again, the number of edit-steps from P[1: i] toT[1: j]) is *k*, which is assumed to be smaller than D[i-1, j-1]. We can also see that the number of edit steps (in this special case, but we try to generalize) to the left of x and T[j] is either k-3 or k-4, depending on whether x is equal to T[j] or not. However, from the setup above, we can also find another transformation from P[1:i] to T[1:j] as follows:

```
 . . . . . . . .  x   y   z   P[i]
 . . . . . . . .              T[j]
-------------------------------------
                   -   -   -
```

Also this transformation will have *k-3* or *k-4* steps to the left of the x-column, which means that we can find a transformation from P[1:i-1] to T[1:j-1] with *k* or *k*-1 steps. However, *k* < D[i-1, j-1], and again this is against the assumption that D[i-1, j-1] was the optimal distance from P[1: i-1] to T[1: j-1] .

1.2 Run the algorithm (on paper) with two similar words, e.g., "algori" og "logari", and with two identical words.

```
          l o g a r i
        0 1 2 3 4 5 6
      --------------
    0 | 0 1 2 3 4 5 6 <- Initialization
  a 1 | 1 1 2 3 3 4 5
  l 2 | 2 1 2 3 4 4 5
  g 3 | 3 2 2 2 3 4 5
  o 4 | 4 3 2 3 3 4 5
  r 5 | 5 4 3 3 4 3 4
  i 6 | 6 5 4 4 4 4 3
      --------------
       ^
       Initialization

          l i k e
        0 1 2 3 4
      ----------
    0 | 0 1 2 3 4
  l 1 | 1 0 1 2 3
  i 2 | 2 1 0 1 2
  k 3 | 3 2 1 0 1
  e 4 | 4 3 2 1 0
      ----------
```

1.3 Show how to implement the algorithm using only one column (or row) plus a few additional variables.

Answer:

We want to calculte the values in the table is it is described above, we assume it has dimensions $D[0{:}m,0{:}n]$. We index it with $D[i,j]$, and want the value of $D[m,n]$.

We calculate row by row from the top down, in our algorithm we now use an array DR[0:*n*] that we initialize with 0, 1, 2, ..., *n*. During execution this array will contain values from row *i* in $DR[0{:}j]$, and values from row *i*-1 in $DR[j{+}1{:}n]$

We also need two new variables, "newDij" og "prevoius". The program looks like this:

```
for j = 0 to n do { DR[j] = j }  // Initializing DR (row zero)
previous = 0                     // In general: the value of D[i-1, j-1]
for i = 1 to m do {
   DR[0] = i                     // Initialization of column zero
   for j = 1 to m do {
      if P[i] == T[j] then newDij = previous
      else newDij = min(DR[j], previous, DR[j-1])
      previous = DR[j]
      DR[i] = newDij
   }
}
```

1.4 Solve the problem given in the last sentence of section 20.5 on page 645. That is: In the slides we originally wanted to find an algorithm for searching through a string T, and look for substrings S = T[p], T[p+1], … , T[q] of T similar to a given string P. We can assume that we want to find the first substring of T whose edit distance to P is less than or equal to a given K (or report that no such substring occurs).

Answer:

The trick is to initialize row zero (along the direction of T) with only zeroes. This has the effect that we allow a new substring S of T with small enough ED to P to start anywhere in T (but see below). We look at the following example:

```
T = a b a e g b c d b a c d g a . . .
P = a b c d
K = 1
       a  b  a  e  g  b  c  d  b  a  c  d  g  a  .  .  .
    ---------------------------------------------------
    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  .  .  .
                   |              \
a   0  0  1  0  1  1  1  1  1  1  0  1  1  1  0  .  .  .
                      \              |
b   1  1  0  1  1  2  1  2  2  1  1  1  2  2  1  .  .  .
                         \              \
c   2  2  1  1  2  2  2  1  1  2  2  1  2  3  2  .  .  .
                            \              \
d   3  3  2  2  2  3  3  2  1  2  3  2  1  2  3  .  .  .
```

We can here observe that we get 1 (≤K) two times in the last row, and for these we can find the corresponding subsequence S of T by going backwards from each of the 1-values in the last row, as shown in the piucture (and as we did for the simple edit distance case). Thus we see that these are $S = gbcd$ and $S = acd$ respectively, and we can also see what the correct edit operation is (even if this requires a little thinking!).

One might object to the above argument for initializing the top row with only zeroes (which was: "we then allow a new substring of T to start anywhere in T") by saying that we might then get false small values in the bottom row, as the top row along the found substring is only zeroes, instead of 0, 1, 2, … as we usually have when computing the edit distance. However, there can be no such influence as the backwards path we found from the lower row describes the influence we have used, and this path do not reach the top row until the start of S.

When executing this algorithm it is natural to fill column after column (starting each time with a zero at the top), and when we get K or less in the last row entry and we only want the first occurrence in T, we can stop and find the corresponding substring S of T.

We can obviously also do this with only one array of the same lengh as P plus a few variables, as in Exercise 1.3 above. If we then want all occurences of legal S-strings in T, we could then, during

the search, simply remember at what indices in T we get edit distance ≤ K, and then afterwards go back to these places in T and find the corresponding substrings S.

Example, time usage: How much time would this algorithm use to search through our entire genome (about $3*10^9$ letters), for a string that is e.g. 100 = 10^2 letters long. Then we would have to compute the recurrence formula $3*10^{11}$ times. Assuming a machine (with caching etc.) using an average of 10 ns to fetch data from the store, we may assume 100 ns = 10^{-7} seconds for each computation of the recurrence formula. Thus, a full search would take $3*10^{11} * 10^{-7} = 3*10^4$ seconds, which is about eight hours. Thus, this is doable, but the biologists usually also need some extra "weight values" in the recurrence formula, and they usually want to search for longer strings than 100 letters (often more than 1000 letters). Thus, doing it straight-forward as above usually takes too much time. On can to same extent optimize the above algorithm, but for many real cases one still has to introduce special tricks to speed up the process, which usually also has the bad effect that the search becomes approximate.

For more information, see e.g. https://en.wikipedia.org/wiki/Human_genome. We will also later have a guest lecture by Torbjørn Rognes from the Bio-Informatics group about the algorithms they are using.

Exercise 2

Look into memoization – using a table as in standard dynamic programming, but with an algorithm following the recursive formula top-down. The trick is now that each recursive call first looks into the table, and checks if the answer to the current sub-problem is already calculated. If it is, this value is used, otherwise we have to do recursive calls to solve the necessary smaller problems.

Write such an algorithm for exercise 20.19.

The array $D[0:m,0:n]$ is just like in 20.19, initialize it the same way, initialize the rest of the array to -1 to indicate that no value is calculated for this sub-problem (0 is a possible calculated value).

```
function EdDist(i,j): int {              // Called from outside with (m,n)
   if D[i,j] >= 0 then return D[i,j]
   else {
      if P[i] == T[j] then D[i,j] = EdDist[i-1,j-1]
      else D[i,j] = min(EdDist[i-1,j], EdDist[i-1,j-1], EdDist[i,j-1]) + 1
      return D[i,j]
   }
}
```

Note that the recursion always stops because of the initialization.