Department of Mathematics and Computer Science
University of Southern Denmark

January 18, 2017
RF

Edit Distance and Friends

The *edit distance* is a measure of diffence between two strings X and Y based the number of *edit operations* needed to turn X into Y. An edit operation is one the following:

insert(i, a):	Insert the character a between position i and $i+1$ in the string.
delete(i)	Delete the character at position i in the string.
change$(i, \texttt{a})$	Change the character at position i in the string into the character `a` (required to be different from the existing character at position i.

We define edit(X, Y) to be the minimal number of such operations needed to change X into Y. One can easily argue that this is a metric (is non-negative, is zero iff $X = Y$, is symmetric in X and Y, fulfills the triangle inequality).

Consider a sequence σ of edit operations changing X into Y. Assume that characters have an ID, and that this ID is kept during change operations. Then X (the string at the start σ) consists of character i) IDs also existing in Y and ii) character IDs not existing in Y. Similarly, Y (the string at the end of the σ) consists of iii) character IDs also existing in X and iv) character IDs not existing in X. Finally, v) other character IDs may been introduced and later removed during the σ. Assume that σ is shortest possible. Then there can be no characters of type 5). Also, IDs of type ii) are simply deleted (no change operations first) and IDs of type iv) are simple inserted (no change operations afterwards). Types i) and iii) are the same IDs. These may either be unchanged, or be changed by a single change operation.

This analysis tells us that any shortest sequence (which are the ones involved in the definition of edit(X, Y)) can be represented as follows, where `abacus` is turned into `cactus`:

```
abac_us
_cactus
```

In the first column, `a` is deleted, in the second column `b` is changed to `c`, in the fifth column `t` is inserted, and in the rest of the columns there is no change. The length of this edit sequence is three, which actually is the value of edit(`abacus`, `cactus`).

Thus, when studying edit distance between two strings X and Y, we may just as well study *alignments* X and Y. An alignment of X and Y consists of some insertions of `_` symbols in both strings, subject the constraints that the two resulting strings have the same lenght and that no `_` symbols end up in the same positions in the two strings. The cost of an alignment is the number of positions not having the same character in the two strings. As is clear from above, any optimal edit sequence corresponds to an alignment of the same cost, and any alignment corresponds to an edit sequence of the same cost. In bioinformatics, the alignment type above is called *global alignment* (to separate it from another concept called local alignment). The edit distance is also known the *Levenshtein distance*.

A dynamic programming solution for edit distance (alias global alignment, alias Levenshtein distance) can be developed along the lines of the dynamic programming solution for LCS: Let edit(i, j) be the edit distance between $X[1..i]$ and $Y[1..j]$ and consider the following recursion.

$$\text{edit}(i,j) = \begin{cases} j & \text{if } i = 0 \\ i & \text{if } j = 0 \\ \min\{\text{edit}(i-1,j)+1, \text{edit}(i,j-1)+1, & \\ \qquad \text{edit}(i-1,j-1)+\delta(i,j)\} & \text{if } i,j > 0 \end{cases}$$

where $\delta(i, j)$ is 1 if $x_i \neq y_j$ and is 0 if $x_i = y_j$. The first two cases of the recursion are correct because to edit from (to) an empty string to (from) a string of length k, k character insertions (deletions) are necessary and sufficient. For the last case of the recursion, note that in an optimal alignment for $X[1..i]$ and $Y[1..j]$, the possibilities for the last pair are of the following types.

`...a`	`...a`	`...a`	`..._`
`...a`	`...b`	`..._`	`...a`
a)	b)	c)	d)

If $x_i \neq y_j$, cases b), c), and d) are possible. If $x_i = y_j$, cases a), c), and d) are possible. Correctness of the last case then follows by removing the last

column of an optimal alignment and running the usual "optimal subproblems" argument type (see LCS-notes for the written out argumentation for $\mathrm{lcs}(i,j)$).

Entirely similar to the LCS problem (see the notes on this), the edit distance between X and Y can be found as $\mathrm{edit}(m,n)$, where $m=|X|$ and $n=|Y|$, using time $O(mn)$ and space $O(\min\{m,n\})$; an actual edit sequence can be found by backtracking over the entire table (raising the space requirement to $\Theta(mn)$); and the idea of Hirschberg's algorithm can be applied to reduce this space to $O(\min\{m,n\})$. The last claim requires that one argues for a recursive formula for $\mathrm{edit}(i,j)$ of the type used in Hirschberg's algorithm.

Note that viewing the edit-table as an oriented grid graph by connecting entry (i,j) by edges to entries $(i+1,j)$, $(i,j+1)$ and $(i+1,j+1)$, we arrive at a DAG. If we put weights one on horizontal and on vertical edges, and put weight $\delta(i,j)$ on diagonal edges, alignments of X and Y and paths from $(0,0)$ to (m,n) in this graph are in a natural one-to-one correspondence (each step of the path generates a column of the alignment, and vice versa). Hence, shortest paths algorithms may be used to find $\mathrm{edit}(m,n)$ (the distance from $(0,0)$ to (m,n) in the graph) and a corresponding edit sequence (a corresponding shortest path). In particular, the $O(V+E)$ time shortest path algorithm for DAGs from the course DM507 may be used. It gives the same time and space results as those above (when not using Hirschberg's algorithm), and actually is essentially the same as the dynamic programming solution described.

The edit distance defined above is easily generalized to charging different costs for insertion, deletion and change operations (even to a cost for the change operation which depends on the two characters in question, which may be desirable in biology for DNA strings). The only difference is that the value 1 at appropriate places is changed to some other value in the recursion.

For edit distance with no changes (corresponding to a cost for changes of ∞), we have the following relation:

$$2\,\mathrm{lcs}(m,n) = m+n-\mathrm{edit}(m,n)$$

which shows that the LCS problem is a special case of the edit distance problem. This relation is seen by noting that for an alignment with no character changes, the only possibilities for pairs in the alignment are the following.

```
...a...        ...a...        ..._...
...a...        ..._...        ...a...
```

Hence, any alignment (with no character changes) corresponds to a common subsequence, and any common subsequence corresponds to an alignment (with no character changes). Here is an illustration of such an alignment and its corresponding common subsequence of length four:

```
_abac_us
ca__ctus
```

```
_abac_us
 |  | ||
ca__ctus
```

We see that total number of characters is twice the number of pairs of characters (hence, twice the length of the common subsequence) plus the remaining number of alignment columns (each with _ as a member of the column, hence this number is exactly the cost of the alignment). On the other hand, the number of characters is of course $m + n$. Therefore, for any pair of corresponding alignment (with no character changes) and common subsequence, of costs a and s, respectively, we have $2s + a = m + n$. Thus, the maximum possible value of s will be in a pair with the minimum possible value for a. This shows $2\,\mathrm{lcs}(m, n) + \mathrm{edit}(m, n) = m + n$.

In DNA sequences in cells, entire pieces are often inserted and deleted in one go, leading to stretches of the _ char. It is a more realistic cost measure if such events incur a cost closer to a single insertion or deletion, rather than to the length of the piece, as in the standard edit distance. To accomodate that, the edit distance can be generalized to assign a cost to such stretches by any desired function of the stretch length. Formally, we define a *gap* in an alignment to be any maximal (that is, terminated in both ends by a char different from _ or by the end of the string) stretch of _ characters in either of the two strings. Let $g(k)$ be the cost that we assign to a gap of length k.

A recursive description of the edit distance can in this case be given using the following four values:

$$
\begin{array}{lcl}
S(i,j) & = & \text{cost of an optimal alignment for } X[1..i] \text{ and } Y[1..j] \\
 & & \text{of type a) or b)} \\
D(i,j) & = & \text{cost of an optimal alignment for } X[1..i] \text{ and } Y[1..j] \\
 & & \text{of type c)} \\
I(i,j) & = & \text{cost of an optimal alignment for } X[1..i] \text{ and } Y[1..j] \\
 & & \text{of type d)} \\
T(i,j) & = & \text{cost of an optimal alignment for } X[1..i] \text{ and } Y[1..j] \\
 & & \text{of any type}
\end{array}
$$

These values fulfill the interdependent recursions below. The recursion for T is:

$$
T(i,j) = \begin{cases}
0 & \text{if } i,j = 0 \\
g(i) & \text{if } i > 0 \text{ and } j = 0 \\
g(j) & \text{if } i = 0 \text{ and } j > 0 \\
\min\{S(i,j), D(i,j), I(i,j)\} & \text{if } i,j > 0
\end{cases}
$$

Here, correctness of the first and the last case is fairly obvious. For correctness of the second case, note that for empty X, the only alignment possible by the definition of alignments is the one shown below, which has cost $g(j)$.

```
x1 x2 .... xj
-  -  .... -
```

A similar argument applies to the third case.

The recursion for S is

$$
S(i,j) = T(i-1,j-1) + \delta(i,j) \text{ if } i,j > 0,
$$

correctness of which is easily argued by removing the last column of an optimal alignment of type a) or b) and then running the usual "optimal subproblems" type argument (see LCS-notes for the written out argumentation for $\text{lcs}(i,j)$).

The recursion for D is

$$
D(i,j) = \begin{cases}
g(i) & \text{if } i > 0 \text{ and } j = 0 \\
\min\{\min_{1 \le k \le i-1}\{S(i-k,j) + g(k)\}, & \\
\qquad \min_{1 \le k \le i}\{I(i-k,j) + g(k)\}\} & \text{if } i,j > 0
\end{cases}
$$

The argument for correctness of the first case is similar to that for the second case of the recursion for T (note that the single allowed alignment really is

of type c), as required for D). For correctness of the last case, consider an optimal alignment of type c). It ends in a gap for Y of some length k, where $1 \leq k$ (to be type c)) and $k \leq i$ (the gap is opposite k characters of X, by the definition of alignments). The gap is preceeded by the last character of Y, and opposite that gap are the last k characters of X. This is illustrated below, where $t = i - k + 1$.

```
.... ?  xt .... xi
.... yj _  .... _
```

The position `?` may be the $i - k$'th character of X or it may be `_`. In the former case, which can only happen for $k \leq i - 1$ (or there will be no characters of X left outside the gap area), the part of the alignment up to this position is a type a) or b) alignment of $X[1..(i-k)]$ and $Y[1..j]$. In the latter case, the part of the alignment up to this position is a type d) alignment of $X[1..(i - k)]$ and $Y[1..j]$. We do not know k, but we try all possibilities and take minimums. Correctness of this case now follows by an "optimal subproblems" type argument (see the notes for Hirschberg's algorithm for a written out version of a similar argumentation for Hirschberg's recursion formula).

A symmetric argument proves correctness of the following recursion for I:

$$I(i,j) = \begin{cases} g(j) & \text{if } i = 0 \text{ and } j > 0 \\ \min\{\min_{1 \leq k \leq i-1}\{S(i-k,j) + g(k)\}, & \\ \qquad \min_{1 \leq k \leq i}\{D(i-k,j) + g(k)\}\} & \text{if } i,j > 0 \end{cases}$$

The dynamic programming solution will fill out the four tables for T, S, D, and I simultaneously, using the same traversal pattern for all tables (for instance rows top-down, and each row left-to-right, as usual). Filling out a table entry may take $\Theta(m + n)$ time, due to the minimization over k in the last cases of the recursions for D and I. As the total table size is still $\Theta(mn)$, the running time becomes $\Theta(mn(m + n))$. As always, an actual alignment can be found by backtracing.

For the restricted case of $g(k) = ak + b$ (that is, g a linear[1] function), a better analysis can be made. This is still a generalization of the basic edit

[1] In biological settings, the word affine is often used here (which is equally correct in terms of math terminology)

distance (which can be said to use $g(k) = k$), in a way relevant for DNA sequence analysis where a larger b (gap opening cost) and smaller a (gap extention cost) is often used. The better analysis is realizing that a type c) alignment with gap length $k \geq 2$ can be seen as an extension of a type c) having gap length $k-1$, hence an increase in cost of a. If $k = 1$, it is an extension of a type a) or b) or d) alignment by a new gap of length one, which has an increase in cost of $a + b$. These possibilities are illustrated below:

```
.... xi-1 xi          .... ?  xi
....  _   _           .... yj _
```

With the usual type of argument, the recursion for D becomes

$$D(i,j) = \begin{cases} g(i) & \text{if } i > 0 \text{ and } j = 0 \\ \min\{D(i-1,j) + a, & \\ \quad S(i-1,j) + a + b, & \\ \quad I(i-1,j) + a + b\} & \text{if } i,j > 0 \end{cases}$$

A symmetric argument gives a symmetric (D and I, and i and j, switch roles) recursion for I, while those for T and S are not changed. Due to the removed minimization over k, the time is back to $O(mn)$.