1 Edit Distance

The exact search of a query string in the text often is not the problem that we want to solve in computational biology/bioinformatics, in context of large databases, and in context of applications in general where there can be errors, inaccuracies, or uncertainties in the text and the query string. More relevant would be, for example, to find the best possible matching between the text and the query or to find a sequence of transformations which tranform the inaccurate into the desired text. These algorithms use the principle of *dynamic programming*.

Let X and Y denote two strings consisting of m and n characters, respectively, that is $X = X[1 \dots m]$ and $Y = Y[1 \dots n]$.

For two given texts X and Y we define a "distance" between X and Y as follows. The *edit distance* between X and Y, $\text{edit}(X, Y)$, is the least number of primitive "edit" operations which are needed to transform X into Y. During the transformation procedure we go over X from left to right and allow the following operations on the i-th character $X[i]$ (we consider Y to be empty at the beginning and is constructed from left to right during this procedure):

- *Copy*: Append $X[i]$ at the end of Y. Set $i \leftarrow i + 1$ after that. (This is interpreted as a simple adoption of the character $X[i]$.)
- *Insert* a: Append character a at the end of Y. i stays the same. (This is interpreted as an insertion of a in front of $X[i]$.)
- *Delete*: Character $X[i]$ is skipped (and therefore not appended at the end of Y). Set $i \leftarrow i + 1$ after that. (This is interpreted as a deletion of $X[i]$.)
- *Change to* a: Append character a at the end of Y. Set $i \leftarrow i + 1$ after that. (This is interpreted as a change of $X[i]$ to a.)

The edit distance $\text{edit}(X, Y)$ is then defined as the minimum number of *Insert*-, *Delete*- and *Change*-operations (*Copy*-operations are not counted) of a sequence of edit operations consisting of these four types such that this sequence transforms X into Y.

It is easy to see that $\text{edit}(X, Y)$ is indeed a meaningful distance since it is a metric:

- $\text{edit}(X, Y) = \text{edit}(Y, X)$ (symmetrie)
- $\text{edit}(X, Y) \leq \text{edit}(X, Z) + \text{edit}(Z, Y)$ (triangle inequality)
- $\text{edit}(X, Y) \geq 0$
- $\text{edit}(X, Y) = 0$ if and only if $X = Y$.

Remark: Instead of this definition of edit distance where we process the string X from left to right we can use the equivalent definition where we ask how many characters we have to insert into, change, or delete from X to obtain Y. For the development of an algorithm is the first definition better suited.

Here is an example, how to transform the string `Praktikum` into the string `Program`:

Operation	$X[i,m]$	Y
	`Praktikum`	
Copy	`raktikum`	`P`
Copy	`aktikum`	`Pr`
Change to `o`	`ktikum`	`Pro`
Change to `g`	`tikum`	`Prog`
Change to `r`	`ikum`	`Progr`
Change to `a`	`kum`	`Progra`
Delete	`um`	`Progra`
Delete	`m`	`Progra`
Copy		`Program`

We conclude from this sequence of edit operations edit(`Praktikum`, `Program`) $\leq$ 6. Naturally, there are many other operation sequences to transform `Praktikum` into `Programm`. In general, there can also be many different operation sequences which transform X into Y and have the minimum number of *Insert*-, *Delete*- und *Change*-operations.

2 Computation using dynamic programming

Now we want to compute the edit distance between two given texts X and Y. We consider the substrings $X[1 \ldots i]$, $o \leq i \leq m$, and $Y[1 \ldots j]$, $0 \leq j \leq n$, and try to construct a table containing all the solutions for the partial problems edit$(X[1 \ldots m], Y[1 \ldots j]$:

$$\texttt{EDIT}[i,j] = \text{edit}(X[1..i], Y[1..j])$$

If we have constructed such a table, then the solution of the original problem can directly be looked up at $\texttt{EDIT}[m,n]$.

Some parts of the `EDIT`-table can simply be initialised: it is clear that $\texttt{EDIT}[0,j] = j$ for all $j \leq n$, and $\texttt{EDIT}[i,0] = i$ for all $i \leq m$. In the first case we have to construct Y from the empty word which is done at best by using j *Insert*-operations, while in the second case we have to construct the empty word Y from X which is done at best by using i *Delete*-operations.

If the last operation which transforms $X[1 \ldots i]$ into $Y[1 \ldots j$ is an *Insert*-operation, then we have $\texttt{EDIT}[i,j] = \texttt{EDIT}[i,j-1] + 1$. If the last operation is a *Delete*-operation, then we have $\texttt{EDIT}[i,j] = \texttt{EDIT}[i-1,j]+1$. If the last operation is a *Copy*-operation which can only be the case if $X[i] = Y[j]$, then we have $\texttt{EDIT}[i,j] = \texttt{EDIT}[i-1,j-1]$. And if the last operation is a *Change*-operation, then we have $\texttt{EDIT}[i,j] = \texttt{EDIT}[i-1,j-1]+1$.

We can construct the table $\texttt{EDIT}[i,j]$ by using the equality

$$\texttt{EDIT}[i,j] = \min(\texttt{EDIT}[i,j-1]+1, \texttt{EDIT}[i-1,j]+1, \texttt{EDIT}[i-1,j-1] + \delta(X[i], Y[j]))$$

where $\delta(a,b) = 1$ if $a \neq b$, und $\delta(a,b) = 0$ if $a = b$ (since the *Copy*-operation doesn't contribute to the distance).

A naive program having two nested loops has running time $\Theta(mn)$, but also uses $\Theta(mn)$ memory for the complete table. However, it is easy to see that the memory usage can be reduced to $O(\min(m,n))$ if we are only interested in finding $\texttt{EDIT}[m,n]$. (How?)

This algorithm is a typical example for *dynamic programming*: the solution of the problem is assembled from already computed and saved solutions for partial problems (of the same kind).

2.1 Computation as a shortest path in a graph

An alternative view on the method of computing the edit distance of the last section is obtained by reducing this problem to a graph problem. Let (i, j) for $0 \leq i \leq m$ and $0 \leq j \leq n$ be nodes of a graph G. The node (i, j) represents the pair of substrings $X[1 \ldots i]$ and $Y[1 \ldots j]$.

The (directed) edges of G should be chosen in such a way that each path from (i, j) to (i', j'), $i \leq i'$ and $j \leq j'$, corresponds to a sequence of edit operations which transforms $X[i + 1 \ldots i']$ into $Y[j + 1 \ldots j']$. (Remind that $X[a \ldots b]$ is, by convention, the empty string if $a > b$.)

The weight of the path (sum of the edge weights) should correspond to the number of *Insert*-, *Delete*- und *Change*-operations on this path. Correspondingly, all edges $(i - 1, j) \rightarrow (i, j)$ have weight 1, all edges $(i, j - 1) \rightarrow (i, j)$ have weight 1, and all edges $(i - 1, j - 1) \rightarrow (i, j)$ have weight $\delta(X[i], Y[j])$. The edit distance between strings X and Y, which is as defined above the minimum number of *Insert*-, *Delete*- und *Change*-operationen of a sequenz of edit operations transforming X into Y, can be computed as the length of the shortest path from $(0, 0)$ to (m, n) in G. A path from $(0, 0)$ to (m, n) corresponds to a processing of the string from left to right where at each character $X[i]$ one or more edit operations are executed.

The graph G obviously is acyclic such that this computation can be conducted in linear time with respect to the size of G (by using topological sorting, see Volker Turau. *Algorithmische Graphentheorie*. Addison-Wesley, Bonn, 1996. S. 250–254.) Since the graph has $\Theta(mn)$ nodes and edges, both the running time and memory usage to compute the edit distance using this method are in $\Theta(mn)$. Alternatively, the algorithm of Dijkstra can be used to find the shortest path from $(0, 0)$ to (m, n) (which results in a worse upper bound for the running time).

One advantage of this solution is that an optimal sequence of operations transforming X into Y can easily be reconstructed; simply follow the path from $(0, 0)$ to (m, n). It is also possible to modify the edge weights such that they reflect certain costs for the corresponding operations. Both generalizations can also easily be incorporated into a solution using dynamic programming. Here we have to be careful that each table entry $\texttt{EDIT}[i, j]$ can be computed in constant time.