Sequence comparison

Christian Charras – Thierry Lecroq
LIR (Laboratoire d'Informatique de Rouen) and
ABISS (Atelier Biologie Informatique Statistique Socio-linguistique)
Faculté des Sciences et des Techniques
Université de Rouen
76821 Mont-Saint-Aignan Cedex
FRANCE
e-mails: {Christian.Charras,Thierry.Lecroq}@dir.univ-rouen.fr

Chapter 1

Basic Problem

We are interested in the notion of resemblance or similarity between two words x and y of length m and n respectively. A dual notion is to look at the distance between these two words.

A word $x = x_0x_1 \cdots x_{m-1}$ of length m is a sequence of letters from an alphabet Σ. Thus, for $0 \leq i \leq m-1$ $x_i \in \Sigma$.

The empty word denoted by ε has a null length: $|\varepsilon| = 0$.

A function d is a distance between words if:

- $\forall\, x, y \in \Sigma : d(x, x) = 0$ and $d(x, y) > 0$ for all $x \neq y$;
- $\forall\, x, y \in \Sigma : d(x, y) = d(y, x)$;
- triangular inequality: $\forall\, x, y, z \in \Sigma : d(x, y) \leq d(x, z) + d(z, y)$.

Several distances between words can be considered:

- the prefix distance:

$$d_{pref}(x, y) = |x| + |y| - 2lcp(x, y);$$

  where $lcp(x, y)$ is the length of the longest common prefix of x and y;
- the suffix distance is defined in a similar way as the prefix distance;
- id. for the substring distance;
- the Hamming distance.

We are interested in a distance which enables to transform x into y using three kinds of basic operations: the **substitution** of a letter of x by a letter of y, the **deletion** of a letter of x or the **insertion** of a letter of y. A cost is associated to each of these operations and for each letter of the alphabet:

- $Sub(a, b)$ is the cost of the substitution of the letter a by the letter b;
- $Del(a)$ is the cost of the deletion of the letter a;

- $Ins(b)$ is the cost of the insertion of the letter b;

The general problem consists of finding a sequence of such basic operations to transform x into y minimizing the total cost of the operations used. The total cost is equal to the sum of the costs of each of the basic operations. This cost is a distance on the words if Sub is a distance on the letters.

We are trying to minimize the distance between x and y which is generally the same than maximizing the similarity between these two words.

The solution is not necessarily unique.

A solution can be given as a sequence of basic operations of substitutions, deletions and insertions. It can also be given in a similar way by an **alignment**.

For two words x of length m and y of length n such as $m \leq n$. An alignment denoted by

$$z = \begin{pmatrix} \overline{x}_0 \\ \overline{y}_0 \end{pmatrix} \begin{pmatrix} \overline{x}_1 \\ \overline{y}_1 \end{pmatrix} \cdots \begin{pmatrix} \overline{x}_{p-1} \\ \overline{y}_{p-1} \end{pmatrix}$$

or

$$z = \begin{pmatrix} \overline{x}_0 & \overline{x}_1 & \cdots & \overline{x}_{p-1} \\ \overline{y}_0 & \overline{y}_1 & \cdots & \overline{y}_{p-1} \end{pmatrix}$$

of length p is such that:

- $n \leq p \leq n + m$;
- $\overline{x}_i = x_j$ or $\overline{x}_i = \varepsilon$ for $0 \leq i \leq p-1$ and $0 \leq j \leq m-1$;
- $\overline{y}_i = y_j$ or $\overline{y}_i = \varepsilon$ for $0 \leq i \leq p-1$ and $0 \leq j \leq n-1$;
- $x = \overline{x}_0\overline{x}_1 \cdots \overline{x}_{p-1}$;
- $y = \overline{y}_0\overline{y}_1 \cdots \overline{y}_{p-1}$;
- for $0 \leq i \leq p-1$, $\nexists\, i$ such that $\overline{x}_i = \varepsilon = \overline{y}_i$.

An **aligned pair** of the type $\binom{a}{b}$ with $a, b \in \Sigma$ indicates the substitution of the letter a by the letter b. An aligned pair of the type $\binom{a}{\varepsilon}$ with $a \in \Sigma$ indicates the deletion of the letter a. An aligned pair of the type $\binom{\varepsilon}{b}$ with $b \in \Sigma$ indicates the insertion of the letter b.

In an alignement or in an aligned pair, the symbol ε is often replaced by the symbol $-$.

This problem can be easily stated in terms of graph. Let $G = (V, E)$ be a labelled and weighted graph with an application $cost : E \rightarrow R$ which associates a cost to each edge of E and an application $label : E \rightarrow \Sigma \cup \{\varepsilon\} \times \Sigma \cup \{\varepsilon\}$ which associates an aligned pair to each edge of E. The graph G is defined as follows:

- V is the set of vertices defined as follows:

$$V = \{(i, j) \mid i \in [-1, m-1] \text{ and } j \in [-1, n-1]\}$$

- E is the set of edges defined as follows:
  - $((i-1,j-1),(i,j)) \in F$ for $i \in [0, m-1]$ and $j \in [0, n-1]$ and $cost((i-1,j-1),(i,j)) = Sub(x_i, y_j)$ and $label((i-1,j-1),(i,j)) = \binom{x_i}{y_j}$
  - $((i-1,j),(i,j)) \in F$ for $i \in [0, m-1]$ and $j \in [-1, n-1]$ and $cost((i-1,j),(i,j)) = Del(x_i)$ and $label((i-1,j),(i,j)) = \binom{x_i}{\varepsilon}$
  - $((i,j-1),(i,j)) \in F$ for $i \in [-1, m-1]$ and $j \in [0, n-1]$ and $cost((i,j-1),(i,j)) = Ins(y_j)$ and $label((i,j-1),(i,j)) = \binom{\varepsilon}{y_j}$

It is only necessary to find a shortest path from the vertex $(-1,-1)$ to the vertex $(m-1, n-1)$. The less the cost the less distant are the two words x and y. As the graph G is acyclic it is possible to find a shortest cost path considering one time and only one each vertex. One had just to consider them in a topological order. Such an order can be obtained by considering the vertices row by row and from left to right within each row. Dynamic programming ([3]) can solve this problem.

Let T be a two-dimensional table with $m+1$ rows and $n+1$ columns. The value of each square $T[i,j]$, for $-1 \leq i \leq m-1$ and $-1 \leq j \leq n-1$ depends only on the three squares $T[i-1,j]$, $T[i,j-1]$ and $T[i-1,j-1]$.

Then for $0 \leq i \leq m-1$ and $0 \leq j \leq n-1$, $T[i,j]$ is the minimum cost of a path from $(-1,-1)$ to (i,j). The algorithm Dynamic-Programming computes all the values of the table T.

Dynamic-Programming(x, m, y, n)
$T[-1,-1] \leftarrow 0$
for $j \leftarrow 0$ **to** $n-1$
    do $T[-1,j] \leftarrow T[-1,j-1] + Ins(y_j)$
for $i \leftarrow 0$ **to** $m-1$
    do $T[i,-1] \leftarrow T[i-1,-1] + Del(x_i)$
        for $j \leftarrow 0$ **to** $n-1$
            do $T[i,j] \leftarrow \min\{T[i-1,j-1] + Sub(x_i,y_j),$
                    $T[i-1,j] + Del(x_i),$
                    $T[i,j-1] + Ins(y_j)\}$
return $T[m-1,n-1]$

The algorithm Dynamic-Programming clearly has a $O(mn)$ time complexity.

The algorithm Dynamic-Programming only computes the cost of the transformation of x into y. To get a sequence of basic operations to transform x into y or the corresponding alignment one has to trace back in the table T from square $T[m-1,n-1]$ to square $T[-1,-1]$ taking each time the right edge (see algorithm One-Alignment).

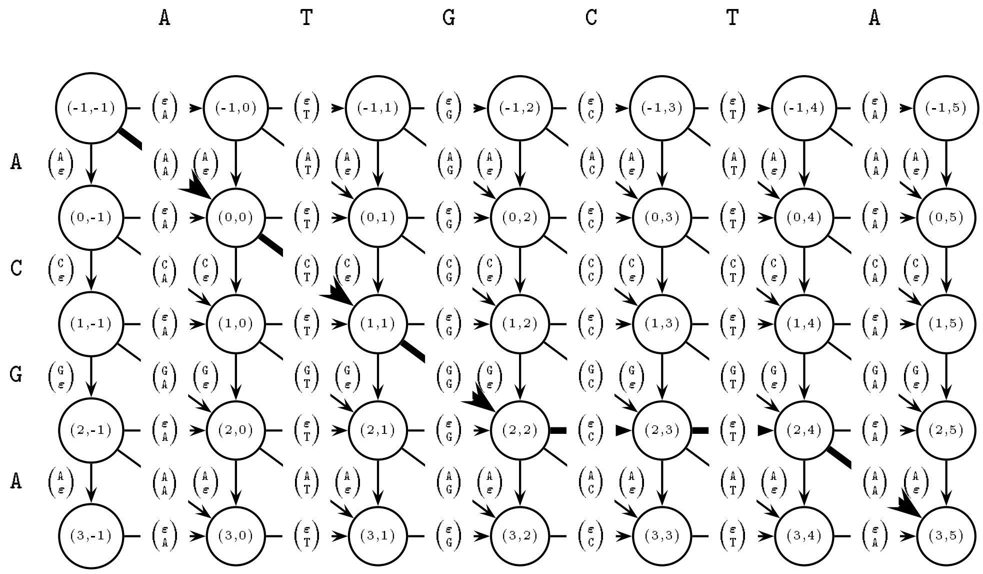

Figure 1.1: Edit graph for x = `ACGA` and y = `ATGCTA`. All the paths from $(-1,-1)$ to $(3,5)$ correspond to a different alignment between x and y. The thick edges correspond to the following alignment: $\begin{pmatrix} \mathtt{A} & \mathtt{C} & \mathtt{G} & - & - & \mathtt{A} \\ \mathtt{A} & \mathtt{T} & \mathtt{G} & \mathtt{C} & \mathtt{T} & \mathtt{A} \end{pmatrix}$.

```
One-Alignment(x, m, y, n, T)
z ← ()
i ← m − 1
j ← n − 1
while i ≠ −1 and j ≠ −1
    do if T[i, j] = T[i − 1, j − 1] + Sub(x_i, y_j)
         then z ← (x_i / y_j) · z
              i ← i − 1
              j ← j − 1
       elseif T[i, j] = T[i − 1, j] + Del(x_i)
         then z ← (x_i / ε) · z
              i ← i − 1
       else   z ← (ε / y_j) · z
              j ← j − 1
while i ≠ −1
    do z ← (x_i / ε) · z
       i ← i − 1
while j ≠ −1
    do z ← (ε / y_j) · z
       j ← j − 1
return z
```

If all the optimal alignments are required then a call to All-Alignments$(x, m-1, y, n-1, \varepsilon, T)$ will produced them.

```
All-Alignment(x, i, y, j, z, T)
if i ≠ −1
  then if j ≠ −1
         then if T[i, j] = T[i − 1, j − 1] + Sub(x_i, y_j)
                then All-Alignment(x, i − 1, y, j − 1, (x_i / y_j) · z, T)
              if T[i, j] = T[i − 1, j] + Del(x_i)
                then All-Alignment(x, i − 1, y, j, (x_i / ε) · z, T)
              if T[i, j] = T[i, j − 1] + Ins(y_j)
                then All-Alignment(x, i, y, j − 1, (ε / y_j) · z, T)
         else All-Alignment(x, i − 1, y, −1, (x_i / ε) · z, T)
  else if j ≠ −1
         then All-Alignment(x, −1, y, j − 1, (ε / y_j) · z, T)
         else Report(z)
```

In this case it is necessary to store all the values of the table T then this problem can be solve in $O(mn)$ space complexity.

When only the cost of the transformation of x into y is needed it is easy to see that a space in $O(\min(m, n))$ is sufficient since the computation of a row (respectively a column) only needs the values of the previous row (respectively column).

Furthermore it is possible to compute an alignment in linear space using a "divide and conquer" method ([9], [11] and [13]).

Chapter 2

Levenshtein Distance

If the following values are considered:

- $Sub(a, a) = 0$ for $a \in \Sigma$;
- $Sub(a, b) = 1$ for $a, b \in \Sigma$ and $a \neq b$;
- $Del(a) = Ins(a) = 1$ for $a \in \Sigma$.

Then $T[m-1, n-1]$ represents the **Levenshtein distance** ([10]) between x and y.

Example :
$x =$ `YWCQPGK` and $y =$ `LAWYQQKPGKA`

		-1	0	1	2	3	4	5	6	7	8	9	10
			L	A	W	Y	Q	Q	K	P	G	K	A
-1		0	1	2	3	4	5	6	7	8	9	10	11
0	Y	1	1	2	3	3	4	5	6	7	8	9	10
1	W	2	2	2	2	3	4	5	6	7	8	9	10
2	C	3	3	3	3	3	4	5	6	7	8	9	10
3	Q	4	4	4	4	4	3	4	5	6	7	8	9
4	P	5	5	5	5	5	4	4	5	5	6	7	8
5	G	6	6	6	6	6	5	5	5	6	5	6	7
6	K	7	7	7	7	7	6	6	5	6	6	5	6

This gives the six following alignments:

$$\begin{pmatrix} \texttt{Y} & - & \texttt{W} & \texttt{C} & \texttt{Q} & - & - & \texttt{P} & \texttt{G} & \texttt{K} & - \\ \texttt{L} & \texttt{A} & \texttt{W} & \texttt{Y} & \texttt{Q} & \texttt{Q} & \texttt{K} & \texttt{P} & \texttt{G} & \texttt{K} & \texttt{A} \end{pmatrix}$$

$$\begin{pmatrix} \texttt{Y} & - & \texttt{W} & \texttt{C} & - & \texttt{Q} & - & \texttt{P} & \texttt{G} & \texttt{K} & - \\ \texttt{L} & \texttt{A} & \texttt{W} & \texttt{Y} & \texttt{Q} & \texttt{Q} & \texttt{K} & \texttt{P} & \texttt{G} & \texttt{K} & \texttt{A} \end{pmatrix}$$

$$\begin{pmatrix} \mathtt{Y} & - & \mathtt{W} & - & \mathtt{C} & \mathtt{Q} & - & \mathtt{P} & \mathtt{G} & \mathtt{K} & - \\ \mathtt{L} & \mathtt{A} & \mathtt{W} & \mathtt{Y} & \mathtt{Q} & \mathtt{Q} & \mathtt{K} & \mathtt{P} & \mathtt{G} & \mathtt{K} & \mathtt{A} \end{pmatrix}$$

$$\begin{pmatrix} - & \mathtt{Y} & \mathtt{W} & \mathtt{C} & \mathtt{Q} & - & - & \mathtt{P} & \mathtt{G} & \mathtt{K} & - \\ \mathtt{L} & \mathtt{A} & \mathtt{W} & \mathtt{Y} & \mathtt{Q} & \mathtt{Q} & \mathtt{K} & \mathtt{P} & \mathtt{G} & \mathtt{K} & \mathtt{A} \end{pmatrix}$$

$$\begin{pmatrix} - & \mathtt{Y} & \mathtt{W} & \mathtt{C} & - & \mathtt{Q} & - & \mathtt{P} & \mathtt{G} & \mathtt{K} & - \\ \mathtt{L} & \mathtt{A} & \mathtt{W} & \mathtt{Y} & \mathtt{Q} & \mathtt{Q} & \mathtt{K} & \mathtt{P} & \mathtt{G} & \mathtt{K} & \mathtt{A} \end{pmatrix}$$

$$\begin{pmatrix} - & \mathtt{Y} & \mathtt{W} & - & \mathtt{C} & \mathtt{Q} & - & \mathtt{P} & \mathtt{G} & \mathtt{K} & - \\ \mathtt{L} & \mathtt{A} & \mathtt{W} & \mathtt{Y} & \mathtt{Q} & \mathtt{Q} & \mathtt{K} & \mathtt{P} & \mathtt{G} & \mathtt{K} & \mathtt{A} \end{pmatrix}$$

which correspond to the six minimum cost paths between $(-1,-1)$ and $(6,10)$:

		-1	0	1	2	3	4	5	6	7	8	9	10
			L	A	W	Y	Q	Q	K	P	G	K	A
-1		0	1	2	3	4	5	6	7	8	9	10	11
0	Y	1	1	2	3	3	4	5	6	7	8	9	10
1	W	2	2	2	2	3	4	5	6	7	8	9	10
2	C	3	3	3	3	3	4	5	6	7	8	9	10
3	Q	4	4	4	4	4	3	4	5	6	7	8	9
4	P	5	5	5	5	5	4	4	5	5	6	7	8
5	G	6	6	6	6	6	5	5	5	6	5	6	7
6	K	7	7	7	7	7	6	6	5	6	6	5	6

Chapter 3

Approximate string matching with k differences

If we are interested in finding all the substrings of y which are at a distance less or equal to a given value k of x then it is enough to initialize all the values of the first line of the table with 0 (this means that the cost of insertions of letters of y at the beginning of x is null). The solutions are then given by all the values of the last row of T which are less or equal to k. This problem is know as **approximate string matching with k differences**.

```
K-DIFFERENCES(x, m, y, n, k)
for j ← −1 to n − 1
    do T[−1, j] ← 0
for i ← −1 to m − 1
    do T[i, −1] ← i + 1
for j ← 0 to n − 1
    do for i ← 0 to m − 1
           do if x_i = y_j
                 then p ← 0
                 else  p ← 1
              T[i, j] ← min{T[i − 1, j − 1] + p,
                            T[i, j − 1] + 1,
                            T[i − 1, j] + 1}
       if T[n, j] ≤ k
         then REPORT(j)
```

Example:
$x =$ `GATAA`, $y =$ `CAGATAAGAGAA` and $k = 1$

		-1	0	1	2	3	4	5	6	7	8	9	10	11
			C	A	G	A	T	A	A	G	A	G	A	A
-1		0	0	0	0	0	0	0	0	0	0	0	0	0
0	G	1	1	1	0	1	1	1	1	0	1	0	1	1
1	A	2	2	1	1	0	1	1	1	1	0	1	0	1
2	T	3	3	2	2	1	0	1	2	2	1	1	1	1
3	A	4	4	3	3	2	1	0	1	2	2	2	1	1
4	A	5	5	4	4	3	2	1	0	1	2	3	2	1

Which gives the seven following alignments:

$$\begin{pmatrix} & & \mathtt{G} & \mathtt{A} & \mathtt{T} & \mathtt{A} & \mathtt{A} & & & & & & \\ \mathtt{C} & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{T} & - & \mathtt{A} & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{A} \end{pmatrix}$$

$$\begin{pmatrix} & & \mathtt{G} & \mathtt{A} & \mathtt{T} & \mathtt{A} & \mathtt{A} & & & & & & \\ \mathtt{C} & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{T} & \mathtt{A} & - & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{A} \end{pmatrix}$$

$$\begin{pmatrix} & & \mathtt{G} & \mathtt{A} & \mathtt{T} & \mathtt{A} & \mathtt{A} & & & & & \\ \mathtt{C} & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{T} & \mathtt{A} & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{A} \end{pmatrix}$$

$$\begin{pmatrix} & - & \mathtt{G} & \mathtt{A} & \mathtt{T} & \mathtt{A} & \mathtt{A} & & & & & \\ \mathtt{C} & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{T} & \mathtt{A} & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{A} \end{pmatrix}$$

$$\begin{pmatrix} & & & \mathtt{G} & \mathtt{A} & \mathtt{T} & \mathtt{A} & \mathtt{A} & & & & & \\ \mathtt{C} & \mathtt{A} & \mathtt{G} & - & \mathtt{A} & \mathtt{T} & \mathtt{A} & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{A} \end{pmatrix}$$

$$\begin{pmatrix} & & \mathtt{G} & \mathtt{A} & \mathtt{T} & \mathtt{A} & \mathtt{A} & - & & & & \\ \mathtt{C} & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{T} & \mathtt{A} & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{A} \end{pmatrix}$$

$$\begin{pmatrix} & & & & & & & \mathtt{G} & \mathtt{A} & \mathtt{T} & \mathtt{A} & \mathtt{A} \\ \mathtt{C} & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{T} & \mathtt{A} & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{A} \end{pmatrix}$$

Chapter 4

Longest common subsequences

If we set $Sub(a, a) = 0$ for $a \in \Sigma$ and $Sub(a, b) > Del(a) + Ins(b)$ for $a, b \in \Sigma$ and $a \neq b$. Then $T[m-1, n-1]$ represents the **edit distance** between x and y where a deletion of a followed by an insertion of b is always prefered to a substitution of a by b. The solution to the dual problem gives the length of a **longest common subsequence** of x and y.

The notion of longest common subsequence of two strings is used to compare file. The `diff` command of the UNIX system is based on this notion where the lines of the files are considered as the symbols of the alphabet.

A subsequence of a word x is obtained by deleting zero, one or several letters of x. More formally $w_0w_1 \cdots w_{i-1}$ is a subsequence of $x_0x_1 \cdots x_{m-1}$ if there exists a strictly increasing sequence of integers $(k_0, k_1, \ldots k_{i-1})$ such that $0 \leq j \leq i-1$, $w_j = x_{k_j}$.

A word w is a longest common subsequence of x and y if w is a subsequence of x, a subsequence of y and its length is maximal.

Let us remark that two words x and y can have several different longest common subsequences. The set of the longest common subsequences of x and y is denoted by $Lcs(x, y)$.

The (unique) length of the elements of $Lcs(x, y)$ is denoted by $lcs(x, y)$.

A straightforward method to compute $lcs(x, y)$ consists in considering all the subsequences of x, checking if they are subsequences of y and keeping the longest. The word x of length m has 2^m subsequences, thus this method is not applicable for high values of m.

Let x a word of length m and y a word of length n. Using dynamic programming yields to compute $lcs(x, y)$ in $O(mn)$ time and space complexity. The method computes length of longest common subsequences of longer and longer prefixes of the two words x and y.

For that we used a two-dimensional table T of size $(m+1) \times (n+1)$ defined as follows:

$$T[i,-1] = T[-1,j] = 0, \text{ for } -1 \le i \le m-1 \text{ and } -1 \le j \le n-1,$$

and for $0 \le i \le m-1$ and $0 \le j \le n-1$

$$T[i,j] = lcs(x_0 x_1 \cdots x_i, y_0 y_1 \cdots y_j)$$

The computation of $lcs(x,y) = T[m-1, n-1]$ if based on a simple observation which leads to the following recurrence formula: for $0 \le i \le m-1$ and $0 \le j \le n-1$

$$T[i,j] = \begin{cases} T[i-1,j-1]+1 & \text{if } x_i = y_j, \\ \max(T[i-1,j], T[i,j-1]) & \text{otherwise.} \end{cases}$$

PROPOSITION For $0 \le i \le m-1$ and $0 \le j \le n-1$:

$$w = \begin{cases} zx_i & \text{if } x_i = y_j, \text{ such that } z \in Lcs(x_0 x_1 \cdots x_{i-1}, y_0 y_1 \cdots y_{j-1}), \\ z' & \text{if } T[i-1,j] \ge T[i,j-1], \\ & \text{such that } z' \in Lcs(x_0 x_1 \cdots x_{i-1}, y_0 y_1 \cdots y_j), \\ z'' & \text{such that } z'' \in Lcs(x_0 x_1 \cdots x_i, y_0 y_1 \cdots y_{j-1}) \text{ otherwise.} \end{cases}$$

PROOF Let $z \in Lcs(x_0 x_1 \cdots x_{i-1}, y_0 y_1 \cdots y_{j-1})$, $z' \in Lcs(x_0 x_1 \cdots x_{i-1}, y_0 y_1 \cdots y_j)$ and $z'' \in Lcs(x_0 x_1 \cdots x_i, y_0 y_1 \cdots y_{j-1})$. z is a longest common subsequence of $x_0 x_1 \cdots x_{i-1}$ and $y_0 y_1 \cdots y_{j-1}$, z' is a longest common subsequence of $x_0 x_1 \cdots x_{i-1}$ and $y_0 y_1 \cdots y_j$ and z'' is a longest common subsequence of $x_0 x_1 \cdots x_i$ and $y_0 y_1 \cdots y_{j-1}$. $|z| = T[i-1,j-1]$, $|z'| = T[i-1,j]$ and $|z''| = T[i,j-1]$. Then clearly if $x_i = y_j$, $T[i,j] = 1 + T[i-1,j-1]$ and $w = zx_i$ is a longest common subsequence of $x_0 x_1 \cdots x_i$ and $y_0 y_1 \cdots y_j$.

If $x_i \ne y_j$ then let us consider the flowing two cases:

- $T[i-1,j] \ge T[i,j-1]$ thus $T[i,j] = T[i-1,j]$ and z' which is a longest common subsequence of $x_0 x_1 \cdots x_{i-1}$ and $y_0 y_1 \cdots y_j$ of length $T[i-1,j]$ is also a longest common subsequence of $x_0 x_1 \cdots x_i$ and $y_0 y_1 \cdots y_j$ of length $T[i,j]$;

- $T[i-1,j] < T[i,j-1]$ thus $T[i,j] = T[i,j-1]$ and z'' which is a longest common subsequence of $x_0 x_1 \cdots x_i$ and $y_0 y_1 \cdots y_{j-1}$ of length $T[i,j-1]$ is also a longest common subsequence of $x_0 x_1 \cdots x_i$ and $y_0 y_1 \cdots y_j$ of length $T[i,j]$.

This formula is used by the algorithm LCS to compute all the values of T.

```
LONGEST-COMMON-SUBSEQUENCE(x, m, y, n)
for i ← −1 to m − 1
     do T[i, −1] ← 0
for j ← −1 to n − 1
     do T[−1, j] ← 0
for i ← 0 to m − 1
     do for j ← 0 to n − 1
            do if x_i = y_j
                  then T[i, j] ← T[i − 1, j − 1] + 1
                  else  T[i, j] ← max(T[i, j − 1],
                                      T[i − 1, j])
return T
```

This computation is evidently in $O(mn)$ time and space complexity. It is the possible to exhibit a longest common subsequence of x and y tracing back in the table from $T[m-1, n-1]$ to $T[-1, -1]$.

```
TRACE-BACK(x, m, y, n, T)
i ← m − 1
j ← n − 1
k ← T[m − 1, n − 1] − 1
while i > 0 and j > 0
     do if T[i, j] = T[i − 1, j − 1] + 1 et x_i = y_j
          then w_k ← x_i
               i ← i − 1
               j ← j − 1
               k ← k − 1
        elseif T[i − 1, j] > T[i, j − 1]
          then i ← i − 1
        else   j ← j − 1
return w
```

Example :
`AGGA` is a longest common subsequence of x and y.

		-1	0	1	2	3	4	5	6	7	8
			C	A	G	A	T	A	G	A	G
-1		0	0	0	0	0	0	0	0	0	0
0	A	0	0	1	1	1	1	1	1	1	1
1	G	0	0	1	2	2	2	2	2	2	2
2	C	0	1	1	2	2	2	2	2	2	2
3	G	0	1	1	2	2	2	2	3	3	3
4	A	0	1	2	2	3	3	3	3	4	4

This corresponds to the four following alignments:

$$\begin{pmatrix} - & \mathtt{A} & \mathtt{G} & \mathtt{C} & - & - & - & \mathtt{G} & \mathtt{A} & - \\ \mathtt{C} & \mathtt{A} & \mathtt{G} & - & \mathtt{A} & \mathtt{T} & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{G} \end{pmatrix}$$

$$\begin{pmatrix} - & \mathtt{A} & \mathtt{G} & - & \mathtt{C} & - & - & \mathtt{G} & \mathtt{A} & - \\ \mathtt{C} & \mathtt{A} & \mathtt{G} & \mathtt{A} & - & \mathtt{T} & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{G} \end{pmatrix}$$

$$\begin{pmatrix} - & \mathtt{A} & \mathtt{G} & - & - & \mathtt{C} & - & \mathtt{G} & \mathtt{A} & - \\ \mathtt{C} & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{T} & - & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{G} \end{pmatrix}$$

$$\begin{pmatrix} - & \mathtt{A} & \mathtt{G} & - & - & - & \mathtt{C} & \mathtt{G} & \mathtt{A} & - \\ \mathtt{C} & \mathtt{A} & \mathtt{G} & \mathtt{A} & \mathtt{T} & \mathtt{A} & - & \mathtt{G} & \mathtt{A} & \mathtt{G} \end{pmatrix}$$

Chapter 5

Cost depending on the length of the gaps

It is possible to penalize the length of the gaps instead of penalizing the deletion or the insertion of individual letters. Let us use a function $\lambda : N \rightarrow R$. $\lambda(k)$ indicates the cost of a gap of length k. In this case the computation of an optimal alignment is done using the following recurrence formula:

$$\left\{ \begin{array}{lcl} D(i,j) & = & \min\{T[k,j] + \lambda(i-k) \mid k \in [0, i-1]\} \\ I(i,j) & = & \min\{T[i,k] + \lambda(j-k) \mid k \in [0, j-1]\} \\ T[i,j] & = & \min\{T[i-1,j-1] + Sub(x_i, y_j), D(i,j), I(i,j)\} \end{array} \right.$$

$D(i,j)$ indicates the score of an optimal alignment between $x_0x_1 \cdots x_i$ and $y_0y_1 \cdots y_j$ ending with deletions of letters of x. $I(i,j)$ indicates the score of an optimal alignment between $x_0x_1 \cdots x_i$ and $y_0y_1 \cdots y_j$ ending with insertions of letters of y. $T[i,j]$ indicates the score of an optimal alignment between $x_0x_1 \cdots x_i$ and $y_0y_1 \cdots y_j$.

Without restriction on the function λ the problem can be solve in $O(mn(m+n))$ time.

If the function λ is affine: $\lambda(k) = g + h(k-1)$ which corresponds to penalize the opening of a gap with g and widening a gap is penalized with h then the problem can be solve in $O(mn)$ time ([7]):

$$\left\{ \begin{array}{lcl} D(i,j) & = & \min\{D(i-1,j) + h, T[i-1,j] + g\} \\ I(i,j) & = & \min\{I(i,j-1) + h, T[i,j-1] + g\} \\ T[i,j] & = & \min\{T[i-1,j-1] + Sub(x_i, y_j), D(i,j), I(i,j)\} \end{array} \right.$$

```
GAP(x, m, y, n)
for i ← −1 to m − 1
     do D[i, −1] ← ∞
        I[i, −1] ← ∞
for i ← −1 to n − 1
     do D[−1, i] ← ∞
        I[−1, i] ← ∞
T[−1, −1] ← 0
T[−1, 0] ← g
T[0, −1] ← g
for i ← 1 to m − 1
     do T[i, −1] ← T[i − 1, −1] + h
for i ← 1 to n − 1
     do T[−1, i] ← T[−1, i − 1] + h
for i ← 0 to m − 1
     do for j ← 0 to n − 1
             do D[i, j] ← min{D[i − 1, j] + h, T[i − 1, j] + g}
                I[i, j] ← min{I[i, j − 1] + h, T[i, j − 1] + g}
                T[i, j] ← min{T[i − 1, j − 1] + Sub(x_i, y_j),
                              D[i, j], I[i, j]}
```

It is easy to see that the tables D, I and T can be reduce to a linear size if only the value of $T[m-1, n-1]$ is needed..

To exhibit an alignment a forth quadratic table has to be used to store in each square which neighbour has served to compute its value (same row and previous column, previous row same column or previous row previous column). Only three different values are possible thus only two bits are enough per square. If all the optimal alignments are to be computed then three bits are required.

Example :
$x =$ YWCQPGK, $y =$ LAWYQQKPGKA, $Sub(a, a) = 0$, $Sub(a, b) = 3$, $g = 3$ and $h = 1$.

D		-1	0	1	2	3	4	5	6	7	8	9	10
			L	A	W	Y	Q	Q	K	P	G	K	A
-1		∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
0	Y	∞	6	7	8	9	10	11	12	13	14	15	16
1	W	∞	6	8	9	8	11	12	13	14	15	16	17
2	C	∞	7	9	9	9	11	13	14	15	16	17	18
3	Q	∞	8	10	10	10	12	14	15	16	17	18	19
4	P	∞	9	11	11	11	12	14	16	17	18	19	20
5	G	∞	10	12	12	12	13	15	17	16	19	20	21
6	K	∞	11	13	13	13	14	16	18	17	16	19	20

I		-1	0	1	2	3	4	5	6	7	8	9	10
			L	A	W	Y	Q	Q	K	P	G	K	A
-1		∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
0	Y	∞	6	6	7	8	8	9	10	11	12	13	14
1	W	∞	7	8	9	9	10	11	12	13	14	15	16
2	C	∞	8	9	10	11	12	13	14	15	16	17	18
3	Q	∞	9	10	11	12	13	12	13	14	15	16	17
4	P	∞	10	11	12	13	14	15	15	16	16	17	18
5	G	∞	11	12	13	14	15	16	16	18	19	16	17
6	K	∞	12	13	14	15	16	17	17	18	19	19	16

T		-1	0	1	2	3	4	5	6	7	8	9	10
			L	A	W	Y	Q	Q	K	P	G	K	A
-1		0	3	4	5	6	7	8	9	10	11	12	13
0	Y	3	3	6	7	5	8	9	10	11	12	13	14
1	W	4	6	6	6	8	8	11	12	13	14	15	16
2	C	5	7	9	9	9	11	11	14	15	16	17	18
3	Q	6	8	10	10	10	9	11	13	14	15	16	17
4	P	7	9	11	11	11	12	12	14	13	16	17	18
5	G	8	10	12	12	12	13	15	15	16	13	16	17
6	K	9	11	13	13	13	14	16	15	17	16	13	16

We get the three following optimal alignments:

$$\begin{pmatrix} - & \mathtt{Y} & \mathtt{W} & \mathtt{C} & \mathtt{Q} & - & - & \mathtt{P} & \mathtt{G} & \mathtt{K} & - \\ \mathtt{L} & \mathtt{A} & \mathtt{W} & \mathtt{Y} & \mathtt{Q} & \mathtt{Q} & \mathtt{K} & \mathtt{P} & \mathtt{G} & \mathtt{K} & \mathtt{A} \end{pmatrix}$$

$$\begin{pmatrix} \mathtt{Y} & - & \mathtt{W} & \mathtt{C} & \mathtt{Q} & - & - & \mathtt{P} & \mathtt{G} & \mathtt{K} & - \\ \mathtt{L} & \mathtt{A} & \mathtt{W} & \mathtt{Y} & \mathtt{Q} & \mathtt{Q} & \mathtt{K} & \mathtt{P} & \mathtt{G} & \mathtt{K} & \mathtt{A} \end{pmatrix}$$

$$\begin{pmatrix} - & - & - & \mathtt{Y} & \mathtt{W} & \mathtt{C} & \mathtt{Q} & - & - & \mathtt{P} & \mathtt{G} & \mathtt{K} & - \\ \mathtt{L} & \mathtt{A} & \mathtt{W} & \mathtt{Y} & - & - & \mathtt{Q} & \mathtt{Q} & \mathtt{K} & \mathtt{P} & \mathtt{G} & \mathtt{K} & \mathtt{A} \end{pmatrix}$$

Chapter 6

Local similarity

We are here intererested in finding an optimal alignment between a part of x and a part of y.

The notion of distance is not suitable anymore to get the result thus we have to use the notion of similarity.

We use the following recurrence formula ([18]):

$$T[i,j] = \max \begin{cases} 0 \\ T[i-1,j-1] + Sub(x_i, y_j) \\ T[i-1,j] + Del(x_i) \\ T[i,j-1] + Ins(y_j) \end{cases}$$

where substitutions, deletions and insertions are given negative values.

To find an optimal local alignment, it is sufficient to find the largest value in the table T and to trace back the path from the square containing this value.

Example :
x = `YWCQPGK`, y = `LAWYQQKPGKA`, $Sub(a,a) = 1$, $Sub(a,b) = -3$, $Del(a) = Ins(a) = -1$.

T		-1	0	1	2	3	4	5	6	7	8	9	10
			L	A	W	Y	Q	Q	K	P	G	K	A
-1		0	0	0	0	0	0	0	0	0	0	0	0
0	Y	0	0	0	0	1	0	0	0	0	0	0	0
1	W	0	0	0	1	0	0	0	0	0	0	0	0
2	C	0	0	0	0	0	0	0	0	0	0	0	0
3	Q	0	0	0	0	0	1	1	0	0	0	0	0
4	P	0	0	0	0	0	0	0	0	1	0	0	0
5	G	0	0	0	0	0	0	0	0	0	2	1	0
6	K	0	0	0	0	0	0	0	1	0	1	3	2

`PGK` corresponds to the highest area of similarity between `YWCQPGK` and

LAWYQQKPGKA.

Chapter 7

List of web sites

- VSNS BioComputing Division:
  - http://www.biotech.ist.unige.it/bcd/welcome.html
  - http://merlin.mbcr.bcm.tmc.edu:8001/bcd/welcome.html
  - http://www.biotech.ist.unige.it/bcd/
  - http://www.techfak.uni-bielefeld.de/bcd/welcome.html
- Introduction to Bioinformatics
  - http://sanda.cryst.bbk.ac.uk/PPS2/course/section5/
- Dynamic programming method for sequence and structure comparison and searching (Andrej Sali):
  - http://tome.cbs.univ-montp1.fr/htmltxt/Doc/manual/node137.html
  - http://www.chem.mq.edu.au/ dmoran/modeller/node137.html
  - http://www.ocms.ox.ac.uk/docs/modeller4/node146.html
- Algorithms for Molecular Biology (Dalit Naor and Ron Shamir):
  - http://www.math.tau.ac.il/ shamir/algmb.html
- Algorithms in Molecular Biology (Richard Karp, Larry Ruzzo):
  - http://www.cs.washington.edu/education/courses/590bi/98w/
- Algorithms in Molecular Biology (Richard Karp, Larry Ruzzo, Martin Tompa):
  - http://www.cs.washington.edu/education/courses/590bi/96w/
- Sequence Comparison: Some Theory and Some Practice (Imre Simon):
  - http://www.ime.usp.br/ is/papir/sctp/

Here is a list of the main papers as well as main books in the domain and introductory chapters in more general books:

Bibliography

[1] A. Apostolico. String editing and longest common subsequences. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 2 Linear Modeling: Background and Application, chapter 8, pages 361–398. Springer-Verlag, Berlin, 1997.

[2] A. Apostolico and Z. Galil, editors. *Pattern matching algorithms.* Oxford University Press, 1997.

[3] R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[4] M. Crochemore and T. Lecroq. Pattern matching and text data compression algorithms. In Allen B. Tucker Jr., editor, *The Computer Science and Engineering Handbook*, chapter 8, pages 162–202. CRC Press Inc., Boca Raton, FL, 1996.

[5] M. Crochemore and W. Rytter. *Text algorithms.* Oxford University Press, 1994.

[6] R. Giancarlo. Dynamic programming: special cases. In A. Apostolico and Z. Galil, editors, *Pattern matching algorithms*, chapter 7, pages 201–236. Oxford University Press, 1997.

[7] O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.*, 162:705–708, 1982.

[8] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge University Press, Cambridge, 1997.

[9] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975.

[10] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Sov. Phys. Dokl.*, 6:707–710, 1966.

[11] W. Miller and E. W. Myers. Sequence comparison with concave weighting functions. *Bull. Math. Biol.*, 50(2):97–120, 1988.

[12] E. W. Myers. An overview of sequence comparison algorithm in molecular biology. Report TR-91-29, Department of Computer Science, University of Arizona, Tucson, AZ, 1991.

[13] E. W. Myers and W. Miller. Optimal alignments in linear space. *Comput. Appl. Biosci.*, 4(1):11–17, 1988.

[14] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48:443–453, 1970.

[15] D. Sankoff and J. B. Kruskal. *Time warps, string edits, and macromolecules: the theory and practice of sequence comparison.* Addison-Wesley, Reading, MA, 1983.

[16] P. H. Sellers. On the theory and computation of evolutionary distance. *SIAM J. Appl. Math.*, 26:787–793, 1974.

[17] J. C. Setubal and J. Meidanis. *Introduction to Computional Molecular Biology.* PWS Publishing Company, 1997.

[18] T. F. Smith and M. S. Waterman. Identification of common molecular sequences. *J. Mol. Biol.*, 147:195–197, 1981.

[19] G. A. Stephen. *String searching algorithms.* World Scientific Press, 1994.

[20] M. S. Waterman. *Mathematical methods for DNA sequences.* CRC Press Inc., Boca Raton, FL, 1991.

[21] M. S. Waterman. *Introduction to Computational Biology.* Chapman & Hall, 1995.

[22] M. S. Waterman, T. F. Smith, and W. A. Beyer. Some biological sequence metrics. *Adv. Math*, 20:367–387, 1976.