

Alignments and Approximate String Matching

Maxime Crochemore¹ and Thierry Lecroq²

¹ Department of Computer Science
King's College London
London WC2R 2LS, UK
`Maxime.Crochemore@kcl.ac.uk`
and

Université Paris Est, France
² LITIS
Université de Rouen
76821 Mont-Saint-Aignan Cedex, France
`Thierry.Lecroq@univ-rouen.fr`

The alignments constitute one of the processes used to compare strings. They allow to visualize the resemblance between strings. This chapter deals with several methods that perform the comparison of two strings in this sense. The extension to comparison methods of more than two strings is delicate, leads to algorithms whose execution time is at least exponential, and is not treated here.

The alignments are based on notions of distance or of similarity between strings. The computations are usually performed by dynamic programming. A typical example is the computation of the longest subsequence common to two strings since it shows the algorithmic techniques to implement in order to obtain an efficient computation. In particular, the reduction of the memory space obtained by one of the algorithms constitute a strategy that can often be applied in the solutions to close problems.

Section 1.1.1 describes the basic techniques for the computation of the edit (or alignment) distance and the production of the associated alignments. The chosen methodology allows to highlight a global resemblance between two strings using assumptions that simplify the computation. The search for local similarities between two strings is examined in Section 1.1.2.

The possibility of reduction of the memory space required by the computations is presented in Section 1.1.3 concerning the computation of longest common subsequences.

We are then interested Section 1.2 in the approximate search for fixed strings. More generally, approximate pattern matching consists in locating all the occurrences of factors inside a text y , of length n , that are similar to a string x , of length m . It consists in producing the positions of the factors of y that are at distance at most k from x , for a given natural integer k . We assume

in the rest that $k < m \leq n$. We consider the edit distance for measuring the approximation.

The edit distance between two strings u and v , that are not necessarily of same length, is the minimal cost of the elementary edit operations between these two strings. The method at the basis for approximate pattern matching is a natural extension of the alignment method by dynamic programming of Section 1.1. It can be improved by using a restricted notion of distance obtained by considering the minimal number of edit operations rather than the sum of their costs. With this distance, the problem is known under the name of approximate pattern matching with k differences. Section 1.2 presents several solutions.

The Hamming distance between two strings u and v of same length is the number of positions in which the two strings possess different letters. With this distance, the problem is known under the name of approximate pattern matching with k mismatches. It is treated in Section 1.3.

We examine then, in Section 1.4, the case of the search for short patterns. This gives excellent practical results and is very flexible as long as the conditions of its utilization are fulfilled. The *Shift-Or algorithm* of Section 1.4 is a method that is both very fast in practice and very easy to implement. The method is flexible enough to be adapted to a wide range of similar approximate matching problems.

1.1 Alignments

An **alignment** of two strings x and y of length m and n respectively consists in aligning their symbols on vertical lines. Formally an alignment of two strings $x, y \in V$ is a word w on the alphabet $(V \cup \{\lambda\}) \times (V \cup \{\lambda\}) \setminus \{(\lambda, \lambda)\}$ (λ is the empty word) whose projection on the first component is x and whose projection of the second component is y .

Thus an alignment $w = (\bar{x}_0, \bar{y}_0)(\bar{x}_1, \bar{y}_1) \cdots (\bar{x}_{p-1}, \bar{y}_{p-1})$ of length p is such that $x = \bar{x}_0\bar{x}_1 \cdots \bar{x}_{p-1}$ and $y = \bar{y}_0\bar{y}_1 \cdots \bar{y}_{p-1}$ with $\bar{x}_i \in V \cup \{\lambda\}$ and $\bar{y}_i \in V \cup \{\lambda\}$ for $0 \leq i \leq p-1$. The alignment is represented as follows

$$\begin{array}{cccc} \bar{x}_0 & \bar{x}_1 & \cdots & \bar{x}_{p-1} \\ \bar{y}_0 & \bar{y}_1 & \cdots & \bar{y}_{p-1} \end{array}$$

with the symbol $-$ instead of the symbol λ .

An example is presented in Fig. 1.1.

$$\begin{array}{cccc} \text{A} & \text{C} & \text{G} & - - \text{A} \\ \text{A} & \text{T} & \text{G} & \text{C} \text{T} \text{A} \end{array}$$

Fig. 1.1. Alignment of ACGA and ATGCTA.

1.1.1 Global alignment

A global alignment of two strings x and y can be obtained by computing the distance between x and y . The notion of distance between two strings is widely used to compare files. The `diff` command of UNIX operating system implements an algorithm based on this notion, in which lines of the files are treated as symbols. The output of a comparison made by `diff` gives the minimum number of operations (substitute a symbol, insert a symbol, or delete a symbol) to transform one file into the other.

Let us define the edit distance between two strings x and y as follows: it is the minimum number of elementary edit operations that enable to transform x into y . The elementary edit operations are:

- the substitution of a character of x at a given position by a character of y ,
- the deletion of a character of x at a given position,
- the insertion of a character of y in x at a given position.

A cost is associated with each elementary edit operation. For $a, b \in V$:

- $Sub(a, b)$ denotes the cost of the substitution of the character a by the character b ,
- $Del(a)$ denotes the cost of the deletion of the character a ,
- $Ins(a)$ denotes the cost of the insertion of the character a .

This means that the costs of the edit operations are independent of the positions where the operations occur. We can now define the edit distance of two strings x and y by

$$edit(x, y) = \min\{\text{cost of } \gamma \mid \gamma \in \Gamma_{x,y}\}$$

where $\Gamma_{x,y}$ is the set of all the sequences of edit operations that transform x into y , and the cost of an element $\gamma \in \Gamma_{x,y}$ is the sum of the costs of its elementary edit operations.

In order to compute $edit(x, y)$ for two strings x and y of length m and n respectively, we make use of a two-dimensional table T of $m + 1$ rows and $n + 1$ columns such that

$$T[i, j] = edit(x[0..i], y[0..j])$$

for $i = 0, \dots, m - 1$ and $j = 0, \dots, n - 1$. It follows $edit(x, y) = T[m - 1, n - 1]$.

The values of the table T can be computed by the following recurrence formula:

$$\begin{aligned} T[-1, -1] &= 0, \\ T[i, -1] &= T[i - 1, -1] + Del(x[i]), \\ T[-1, j] &= T[-1, j - 1] + Ins(y[j]), \\ T[i, j] &= \min \begin{cases} T[i - 1, j - 1] + Sub(x[i], y[j]), \\ T[i - 1, j] + Del(x[i]), \\ T[i, j - 1] + Ins(y[j]), \end{cases} \end{aligned}$$

for $i = 0, 1, \dots, m - 1$ and $j = 0, 1, \dots, n - 1$.

The value at position $[i, j]$ in the table T only depends on the values at the three neighbor positions $[i - 1, j - 1]$, $[i - 1, j]$ and $[i, j - 1]$.

The direct application of the above recurrence formula gives an exponential time algorithm to compute $T[m - 1, n - 1]$. However the whole table T can be computed in quadratic time, technique known as “dynamic programming”. This is a general technique that is used to solve the different kinds of alignments.

The computation of the table T proceeds in two steps. First it initializes the first column and first row of T , this is done by a call to a generic function MARGIN which is an argument of the algorithm and that depends on the kind of alignment that is considered. Second it computes the remaining values of T , that is done by a call to a generic function FORMULA which is an argument of the algorithm and that depends on the kind of alignment that is considered.

```

GENERIC-DP( $x, m, y, n, \text{MARGIN}, \text{FORMULA}$ )
1  MARGIN( $T, x, m, y, n$ )
2  for  $j \leftarrow 0$  to  $n - 1$  do
3    for  $i \leftarrow 0$  to  $m - 1$  do
4       $T[i, j] \leftarrow \text{FORMULA}(T, x, i, y, j)$ 
5  return  $T$ 

```

Fig. 1.2. Computation of the table T by dynamic programming.

Computing a global alignment of x and y can be done by a call to GENERIC-DP with the following arguments

($x, m, y, n, \text{GLOBAL-MARGIN}, \text{GLOBAL-FORMULA}$)
 (see Fig. 1.2, 1.3 and 1.4). The computation of all the values of the table T can thus be done in quadratic space and time: $O(m \times n)$.

```

GLOBAL-MARGIN( $T, x, m, y, n$ )
1   $T[-1, -1] \leftarrow 0$ 
2  for  $i \leftarrow 0$  to  $m - 1$  do
3     $T[i, -1] \leftarrow T[i - 1, -1] + \text{Del}(x[i])$ 
4  for  $j \leftarrow 0$  to  $n - 1$  do
5     $T[-1, j] \leftarrow T[-1, j - 1] + \text{Ins}(y[j])$ 

```

Fig. 1.3. Margin initialization for the computation of a global alignment.

An optimal alignment (with minimal cost) can then be produced by a call to the function ONE-ALIGNMENT($T, x, m - 1, y, n - 1$) (see Fig. 1.5). It consists in tracing back the computation of the values of the table T from

```

GLOBAL-FORMULA( $T, x, i, y, j$ )
1 return  $\min\{T[i-1, j-1] + Sub(x[i], y[j]),$ 
    $T[i-1, j] + Del(x[i]),$ 
    $T[i, j-1] + Ins(y[j])\}$ 

```

Fig. 1.4. Computation of $T[i, j]$ for a global alignment.

position $[m-1, n-1]$ to position $[-1, -1]$. At each cell $[i, j]$ the algorithm determines among the three values $T[i-1, j-1] + Sub(x[i], y[j])$, $T[i-1, j] + Del(x[i])$ and $T[i, j-1] + Ins(y[j])$ which has been used to produce the value of $T[i, j]$. If $T[i-1, j-1] + Sub(x[i], y[j])$ has been used it adds $(x[i], y[j])$ to the optimal alignment and proceeds recursively with the cell at $[i-1, j-1]$. If $T[i-1, j] + Del(x[i])$ has been used it adds $(x[i], -)$ to the optimal alignment and proceeds recursively with cell at $[i-1, j]$. If $T[i, j-1] + Ins(y[j])$ has been used it adds $(-, y[j])$ to the optimal alignment and proceeds recursively with cell at $[i, j-1]$. Recovering all the optimal alignments can be done by a similar technique.

An example of global alignment is given in Fig. 1.6.

```

ONE-ALIGNMENT( $T, x, i, y, j$ )
1 if  $i = -1$  and  $j = -1$  then
2   return  $(\lambda, \lambda)$ 
3 else if  $i = -1$  then
4   return ONE-ALIGNMENT( $T, x, -1, y, j-1$ )  $\cdot (\lambda, y[j])$ 
5 else if  $j = -1$  then
6   return ONE-ALIGNMENT( $T, x, i-1, y, -1$ )  $\cdot (x[i], \lambda)$ 
7 else if  $T[i, j] = T[i-1, j-1] + Sub(x[i], y[j])$  then
8   return ONE-ALIGNMENT( $T, x, i-1, y, j-1$ )  $\cdot (x[i], y[j])$ 
9 else if  $T[i, j] = T[i-1, j] + Del(x[i])$  then
10  return ONE-ALIGNMENT( $T, x, i-1, y, j$ )  $\cdot (x[i], \lambda)$ 
11 else return ONE-ALIGNMENT( $T, x, i, y, j-1$ )  $\cdot (\lambda, y[j])$ 

```

Fig. 1.5. Recovering an optimal alignment.

1.1.2 Local alignment

A local alignment of two strings x and y consists in finding the segment of x that is closer to a segment of y . The notion of distance used to compute global alignments cannot be used in that case since the segments of x closer to segments of y would only be the empty segment or individual characters. This is why a notion of similarity is used based on a scoring scheme for edit operations.

T	j	-1	0	1	2	3	4	5
i	$y[j]$	A	T	G	C	T	A	
-1	$x[i]$	0	1	2	3	4	5	6
0	A	1	0	1	2	3	4	5
1	C	2	1	1	2	2	3	4
2	G	3	2	2	1	2	3	4
3	A	4	3	3	2	2	3	3

Fig. 1.6. Global alignment of ACGA and ATGCTA. The values of the above table have been obtained with the following unitary costs: $Sub(a, b) = 1$ if $a \neq b$ and $Sub(a, a) = 0$, $Del(a) = Ins(a) = 1$ for $a, b \in V$.

A score (instead of a cost) is associated with each elementary edit operation. For $a, b \in V$:

- $Sub_S(a, b)$ denotes the score of substituting the character b for the character a ,
- $Del_S(a)$ denotes the score of deleting the character a ,
- $Ins_S(a)$ denotes the score of inserting the character a .

This means that the scores of the edit operations are independent of the positions where the operations occur. For two characters a and b , a positive value of $Sub_S(a, b)$ means that the two characters are close to each other, and a negative value of $Sub_S(a, b)$ means that the two characters are far apart.

We can now define the edit score of two strings x and y by

$$sco(x, y) = \max\{\text{score of } \gamma \mid \gamma \in \Gamma_{x,y}\}$$

where $\Gamma_{x,y}$ is the set of all the sequences of edit operations that transform x into y and the score of an element $\sigma \in \Gamma_{x,y}$ is the sum of the scores of its elementary edit operations.

An optimal local alignment between the strings x and y is a pair of strings (u, v) for which u is a factor of x , v is a factor of y and $sco(u, v)$ is maximal. For performing its computation, we consider a table T defined, for $i = -1, 0, \dots, m-1$ and $j = -1, 0, \dots, n-1$, by: $T[i, j]$ is the maximal similarity between a suffix of $x[0..i]$ and a suffix of $y[0..j]$. Or also

$$T[i, j] = \max\{sco(x[\ell..i], y[k..j]) \mid 0 \leq \ell \leq i \text{ and } 0 \leq k \leq j\} \cup \{0\}$$

is the score of the local alignment in $[i, j]$.

The values of the table T can be computed by the following recurrence formula:

$$\begin{aligned}
T[-1, -1] &= 0, \\
T[i, -1] &= 0, \\
T[-1, j] &= 0, \\
T[i, j] &= \max \begin{cases} T[i-1, j-1] + \text{Sub}_S(x[i], y[j]), \\ T[i-1, j] + \text{Del}_S(x[i]), \\ T[i, j-1] + \text{Ins}_S(y[j]), \\ 0, \end{cases}
\end{aligned}$$

for $i = 0, 1, \dots, m-1$ and $j = 0, 1, \dots, n-1$.

Computing the values of T for a local alignment of x and y can be done by a call to `GENERIC-DP` with the following arguments

$(x, m, y, n, \text{LOCAL-MARGIN}, \text{LOCAL-FORMULA})$
in $O(mn)$ time and space complexity (see Fig. 1.2, 1.7 and 1.8). Recovering a local alignment can be done in a way similar to what is done in the case of a global alignment (see Fig. 1.5) but the trace back procedure must start at a position of a maximal value in T rather than at position $[m-1, n-1]$.

An example of local alignment is given in Fig. 1.9.

```

LOCAL-MARGIN( $T, x, m, y, n$ )
1  $T[-1, -1] \leftarrow 0$ 
2 for  $i \leftarrow 0$  to  $m-1$  do
3    $T[i, -1] \leftarrow 0$ 
4 for  $j \leftarrow 0$  to  $n-1$  do
5    $T[-1, j] \leftarrow 0$ 

```

Fig. 1.7. Margin initialization for computing a local alignment.

```

LOCAL-FORMULA( $T, x, i, y, j$ )
1 return  $\max\{T[i-1, j-1] + \text{Sub}_S(x[i], y[j]),$ 
    $T[i-1, j] + \text{Del}_S(x[i]),$ 
    $T[i, j-1] + \text{Ins}_S(y[j]),$ 
    $0\}$ 

```

Fig. 1.8. Recurrence formula for computing a local alignment.

1.1.3 Longest Common Subsequence of Two Strings

A subsequence of a string x is obtained by deleting zero or more characters from x . More formally $w[0..i-1]$ is a subsequence of $x[0..m-1]$ if there

T	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
i		$y[j]$	E	R	D	A	W	C	Q	P	G	K	W	Y
-1	$x[i]$	0	0	0	0	0	0	0	0	0	0	0	0	0
0	E	0	1	0	0	0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	1	0	0	0	0	0	0	0	0
2	W	0	0	0	0	0	2	1	0	0	0	0	1	0
3	A	0	0	0	0	1	1	0	0	0	0	0	0	0
4	C	0	0	0	0	0	0	2	1	0	0	0	0	0
5	Q	0	0	0	0	0	0	1	3	2	1	0	0	0
6	G	0	0	0	0	0	0	0	2	1	3	2	1	0
7	K	0	0	0	0	0	0	0	1	0	2	4	3	2
8	L	0	0	0	0	0	0	0	0	0	1	3	2	1

(a)

(b) A W A C Q - G K
A W - C Q P G K

Fig. 1.9. Computation of an optimal local alignment of $x = \text{EAWACQGKL}$ and $y = \text{ERDAWCQPGKWKY}$ with scores: $Sub_S(a, a) = 1$, $Sub_S(a, b) = -3$ and $Del_S(a) = Ins_S(a) = -1$ for $a, b \in V$, $a \neq b$. (a) Values of table T . (b) The corresponding alignment.

exists an increasing sequence of integers ($k_j \mid j = 0, \dots, i - 1$) such that for $0 \leq j \leq i - 1$, $w[j] = x[k_j]$. We say that a string is an *lcs*(x, y) if it is a **longest common subsequence** of the two strings x and y . Note that two strings can have several longest common subsequences. Their common length is denoted by $llcs(x, y)$.

A brute-force method to compute an *lcs*(x, y) would consist in computing all the subsequences of x , checking if they are subsequences of y , and keeping the longest ones. The string x of length m has potentially 2^m subsequences, and so this method could take $O(2^m)$ time, which is impractical even for fairly small values of m .

However $llcs(x, y)$ can be computed with a two-dimensional table T by the following recurrence formula:

$$\begin{aligned}
 T[-1, -1] &= 0, \\
 T[i, -1] &= 0, \\
 T[-1, j] &= 0, \\
 T[i, j] &= \begin{cases} T[i - 1, j - 1] + 1 & \text{if } x[i] = y[j], \\ \max\{T[i - 1, j], T[i, j - 1]\} & \text{otherwise,} \end{cases}
 \end{aligned}$$

for $i = 0, 1, \dots, m - 1$ and $j = 0, 1, \dots, n - 1$. Then $T[i, j] = llcs(x[0..i], y[0..j])$ and $llcs(x, y) = T[m - 1, n - 1]$.

Computing $T[m - 1, n - 1]$ can be done by a call to `GENERIC-DP` with the following arguments ($x, m, y, n, \text{LOCAL-MARGIN}, \text{LCS-FORMULA}$) in $O(mn)$ time and space complexity (see Fig. 1.2, 1.7 and 1.10).

```

FORMULA-LCS( $T, x, i, y, j$ )
1 if  $x[i] = y[j]$  then
2   return  $T[i - 1, j - 1] + 1$ 
3 else return  $\max\{T[i - 1, j], T[i, j - 1]\}$ 
    
```

Fig. 1.10. Recurrence formula for computing an *lcs*.

It is possible afterward to trace back a path from position $[m - 1, n - 1]$ to exhibit an $lcs(x, y)$ in a similar way as for producing a global alignment (see Fig. 1.5). An example is presented in Fig. 1.11.

T	j	-1	0	1	2	3	4	5	6	7	8
i	$y[j]$	C	A	G	A	T	A	G	A	G	
-1	$x[i]$	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

Fig. 1.11. The value $T[4, 8] = 4$ is $llcs(x, y)$ for $x = AGCGA$ and $y = CAGATAGAG$. String $AGGA$ is an *lcs* of x and y .

1.1.4 Reducing the Space: Hirschberg Algorithm

If only the length of an $lcs(x, y)$ is required, it is easy to see that only one row (or one column) of the table T needs to be stored during the computation. The space complexity becomes $O(\min(m, n))$ as can be checked on the algorithm of Fig. 1.12. The Hirschberg algorithm computes an $lcs(x, y)$ in linear space and not only the value $llcs(x, y)$. The computation uses the algorithm of Fig. 1.12.

Let us define

$$\begin{aligned}
 T^*[i, n] &= T^*[m, j] = 0, & \text{for } 0 \leq i \leq m & \text{ and } 0 \leq j \leq n \\
 T^*[m - i, n - j] &= llcs((x[i..m - 1])^R, (y[j..n - 1])^R) \\
 & & \text{for } 0 \leq i \leq m - 1 & \text{ and } 0 \leq j \leq n - 1
 \end{aligned}$$

and

$$M(i) = \max_{0 \leq j < n} \{T[i, j] + T^*[m - i, n - j]\}$$

where the string w^R is the reverse (or mirror image) of the string w . The following property is the key observation to compute an $lcs(x, y)$ in linear space:

```

LLCS( $x, m, y, n$ )
1  for  $i \leftarrow -1$  to  $m - 1$  do
2     $C[i] \leftarrow 0$ 
3  for  $j \leftarrow 0$  to  $n - 1$  do
4     $last \leftarrow 0$ 
5    for  $i \leftarrow -1$  to  $m - 1$  do
6      if  $last > C[i]$  then
7         $C[i] \leftarrow last$ 
8      else if  $last < C[i]$  then
9         $last \leftarrow C[i]$ 
10     else if  $x[i] = y[j]$  then
11        $C[i] \leftarrow C[i] + 1$ 
12        $last \leftarrow last + 1$ 
13  return  $C$ 

```

Fig. 1.12. $O(m)$ -space algorithm to compute $llcs(x, y)$.

```

HIRSCHBERG( $x, m, y, n$ )
1  if  $m = 0$  then
2    return  $\lambda$ 
3  else if  $m = 1$  then
4    if  $x[0] \in y$  then
5      return  $x[0]$ 
6    else return  $\lambda$ 
7  else  $j \leftarrow \lfloor n/2 \rfloor$ 
8     $C \leftarrow \text{LLCS}(x, m, y[0..j-1], j)$ 
9     $C^* \leftarrow \text{LLCS}(x^R, m, y[j..n-1]^R, n-j)$ 
10    $k \leftarrow m - 1$ 
11    $M \leftarrow C[m-1] + C^*[m-1]$ 
12   for  $j \leftarrow -1$  to  $m - 2$  do
13     if  $C[j] + C^*[j] > M$  then
14        $M \leftarrow C[j] + C^*[j]$ 
15        $k \leftarrow j$ 
16   return  $\text{HIRSCHBERG}(x[0..k-1], k, y[0..j-1], j)$ .
       $\text{HIRSCHBERG}(x[k..m-1], m-k, y[j..n-1], n-j)$ 

```

Fig. 1.13. $O(\min(m, n))$ -space computation of $lcs(x, y)$.

$$M(i) = T[m-1, n-1], \quad \text{for } 0 \leq i < m.$$

In the algorithm shown in Fig. 1.13 the integer j is chosen as $n/2$. After $T[i, j-1]$ and $T^*[m-i, n-j]$ ($0 \leq i < m$) are computed, the algorithm finds an integer k such that $T[i, k] + T^*[m-i, n-k] = T[m-1, n-1]$. Then, recursively, it computes an $lcs(x[0..k-1], y[0..j-1])$ and an $lcs(x[k..m-1], y[j..n-1])$, and concatenates them to get an $lcs(x, y)$.

The running time of the Hirschberg algorithm is still $O(mn)$ but the amount of space required for the computation becomes $O(\min(m, n))$ instead of being quadratic when computed by dynamic programming.

1.2 Approximate String Matching with Differences

Approximate string matching is the problem of finding all approximate occurrences of a pattern x of length m in a text y of length n . Approximate occurrences of x are segments of y that are close to x according to a specific distance: the distance between segments and x must be not greater than a given integer k . With the edit distance (or Levenshtein distance), the problem is known as approximate string matching with k differences. The standard solutions to solve this problem consist in using the dynamic programming technique introduced in Section 1.1. We describe three variations around this technique.

Dynamic programming

We first examine a problem a bit more general for which the cost of the edit operations is not necessarily one unit. Aligning x with a factor of y amounts to align x with a prefix of y considering that the insertion of any number of letters of x at the beginning of x is not penalizing. With the table T of Section 1.1.1 we check that, to solve the problem, it is sufficient then to initialize to zero the values of the first line of the table. The positions of the occurrences are then associated with all the values of the last line of the table that are less than k .

To perform the search for approximate factors, we utilize the table R defined by

$$R[i, j] = \min\{\text{edit}(x[0..i], y[\ell..j]) \mid \ell = 0, 1, \dots, j + 1\},$$

for $i = -1, 0, \dots, m - 1$ and $j = -1, 0, \dots, n - 1$, where edit is the edit distance of Section 1.1. The computation of the values of the table R utilizes the recurrence relations that follow.

For $i = 0, 1, \dots, m - 1$ and $j = 0, 1, \dots, n - 1$, we have:

$$\begin{aligned} R[-1, -1] &= 0, \\ R[i, -1] &= R[i - 1, -1] + \text{Del}(x[i]), \\ R[-1, j] &= 0, \\ R[i, j] &= \min \begin{cases} R[i - 1, j - 1] + \text{Sub}(x[i], y[j]), \\ R[i - 1, j] + \text{Del}(x[i]), \\ R[i, j - 1] + \text{Ins}(y[j]). \end{cases} \end{aligned} \quad (1.1)$$

```

K-DIFF-DP( $x, m, y, n, k$ )
1  $R[-1, -1] \leftarrow 0$ 
2 for  $i \leftarrow 0$  to  $m - 1$  do
3    $R[i, -1] \leftarrow i + Del(x[i])$ 
4 for  $j \leftarrow 0$  to  $n - 1$  do
5    $R[-1, j] \leftarrow 0$ 
6   for  $i \leftarrow 0$  to  $m - 1$  do
7      $R[i, j] \leftarrow \min \begin{cases} R[i - 1, j - 1] + Sub(x[i], y[j]) \\ R[i - 1, j] + Del(x[i]) \\ R[i, j - 1] + Ins(y[j]) \end{cases}$ 
8   if  $R[m - 1, j] \leq k$  then
9     OUTPUT( $j$ )
    
```

Fig. 1.14. Approximate string matching with k differences by dynamic programming.

(a)	R	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
	i	$y[j]$	C	A	G	A	T	A	A	G	A	G	A	A	A
	-1	$x[i]$	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

G A T A A		G A T A A
C A G A T - A A G A G A A		C A G A T A A G A G A A

G A T A A		- G A T A A
C A G A T A - A G A G A A		C A G A T A A G A G A A

G A T A A		G A T A A -
C A G - A T A A G A G A A		C A G A T A A G A G A A

G A T A A
C A G A T A A G A G A A

Fig. 1.15. Search for $x = GATAA$ in $y = CAGATAAGAGAA$ with one difference, considering unit costs for the edit operations. (a) Values of table R . (b) The seven alignments of x with factors of y ending at positions 5, 6, 7 and 11 on y . We note that the fourth and sixth alignments give no extra information comparing to the second.

The search algorithm K-DIFF-DP whose code is given in Fig. 1.14 and that translates the recurrence of the previous proposition performs the approximate search. An example is given in Fig. 1.15.

We note that the space used by the algorithm K-DIFF-DP can be reduced to a single column by reproducing the technique of Section 1.1.3. Besides, this technique is implemented by the algorithm K-DIFF-CUT-OFF (see Fig. 1.16). As a conclusion we get the following result.

The operation K-DIFF-DP(x, m, y, n, k) that finds the factors u of y for which $edit(u, x) \leq k$ ($edit$ edit distance with any costs) executes in time $O(m \times n)$ and can be realized in space $O(m)$.

Diagonal monotony

In the rest of the section, we consider that the costs of the edit operations are unitary. This is a simple case for which we can describe more efficient computation strategies than those described above. The restriction allows to state a property of monotony on the diagonals that is at the basis of the presented variations.

Since we assume that $Sub(a, b) = Del(a) = Ins(b) = 1$ for $a, b \in V$, $a \neq b$, the recurrence relation 1.1 simplifies and becomes

$$\begin{aligned} R[-1, -1] &= 0, \\ R[i, -1] &= i + 1, \\ R[-1, j] &= 0, \\ R[i, j] &= \min \begin{cases} R[i-1, j-1] & \text{if } x[i] = y[j], \\ R[i-1, j-1] + 1 & \text{if } x[i] \neq y[j], \\ R[i-1, j] + 1, \\ R[i, j-1] + 1. \end{cases} \end{aligned} \quad (1.2)$$

for $i = 0, 1, \dots, m-1$ and $j = 0, 1, \dots, n-1$.

A diagonal d of the table R consists of the positions $[i, j]$ for which $j - i = d$ ($-m \leq d \leq n$). The property of diagonal monotony expresses that the sequence of values on each diagonal of the table R increases with i and that the difference between two consecutive values is at most one (see Fig. 1.15). Before formally stating the property, we give intermediate results. The first result means that two adjacent values on a column of the table R differ from at most one. The second result is symmetrical to the first one for the lines of R .

For each position j on the string y , we have

$$-1 \leq R[i, j] - R[i-1, j] \leq 1$$

for $i = 0, 1, \dots, m-1$.

For each position i on the string x , we have

$$-1 \leq R[i, j] - R[i, j - 1] \leq 1$$

for $j = 0, 1, \dots, n - 1$.

We now can state the result concerning the property of monotony on the diagonals announced above:

For $i = 0, 1, \dots, m - 1$ and $j = 0, 1, \dots, n - 1$, we have:

$$R[i - 1, j - 1] \leq R[i, j] \leq R[i - 1, j - 1] + 1.$$

Partial computation

The property of monotony on the diagonals is exploited in the following way to avoid to compute some values in the table R that are greater than k , the maximal number of allowed differences. The values are still computed column by column, in the increasing order of the positions on y and for each column in the increasing order of the positions on x , as done by the algorithm K-DIFF-DP. When a value equal to $k + 1$ is found in a column, it is useless to compute the next values in the same diagonal since those latter are all strictly greater than k . For pruning the computation, we keep, in each column, the largest position at which is found an admissible value. If q_j is this position, for a given column j , only the values of lines -1 to $q_j + 1$ are computed in the next column (of index $j + 1$).

The algorithm K-DIFF-CUT-OFF, given in Fig. 1.16, realizes this method.

```

K-DIFF-CUT-OFF( $x, m, y, n, k$ )
1  for  $i \leftarrow -1$  to  $k - 1$  do
2     $C_1[i] \leftarrow i + 1$ 
3   $p \leftarrow k$ 
4  for  $j \leftarrow 0$  to  $n - 1$  do
5     $C_2[-1] \leftarrow 0$ 
6    for  $i \leftarrow 0$  to  $p$  do
7      if  $x[i] = y[j]$  then
8         $C_2[i] \leftarrow C_1[i - 1]$ 
9      else  $C_2[i] \leftarrow \min\{C_1[i - 1], C_2[i - 1], C_1[i]\} + 1$ 
10    $C_1 \leftarrow C_2$ 
11   while  $C_1[p] > k$  do
12      $p \leftarrow p - 1$ 
13   if  $p = m - 1$  then
14     OUTPUT( $j$ )
15    $p \leftarrow \min\{p + 1, m - 1\}$ 

```

Fig. 1.16. Approximate string matching with k differences by partial computation.

The column -1 is initialized until line $k - 1$ that corresponds to the value k . For the next columns of index $j = 0, 1, \dots, n - 1$, the values are computed until line

$$p_j = \min \begin{cases} 1 + \max\{i \mid 0 \leq i \leq m - 1 \text{ and } R[i, j - 1] \leq k\}, \\ m - 1. \end{cases}$$

The table R is implemented with the help of two tables C_2 and C_1 that allow to memorize respectively the values of the column during the computation and the values of the previous column. The process is similar to the one that is used in the algorithm LLCS of Section 1.1.4. At each iteration of the loop Lines 7–10, we have:

$$\begin{aligned} C_1[i - 1] &= R[i - 1, j - 1], \\ C_2[i - 1] &= R[i - 1, j], \\ C_1[i] &= R[i, j - 1]. \end{aligned}$$

We compute then the value $C_2[i]$ that is also $R[i, j]$. We find thus at this line an implementation of Relation 1.2. An example of computation is given in Fig. 1.17.

R	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
i	$y[j]$	C	A	G	A	T	A	A	G	A	G	A	A	A
-1	$x[i]$	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	1	1	0	1	1	1	1	0	1	0	1
2	T				2	1	0	1	2	2	1	1	1	1
3	A						1	0	1	2	2	2	1	1
4	A							1	0	1	2			1

Fig. 1.17. Pruning of the computation of the dynamic programming table for the search for $x = \text{GATAA}$ in $y = \text{CAGATAAGAGAA}$ with one difference (see Figure 1.15). We notice that seventeen values of table R (those that are not shown) are not useful for the computation of occurrences of approximate factors of x in y .

We note that the memory space used by the algorithm $K\text{-DIFF-CUT-OFF}$ is $O(m)$. Indeed, only two columns are memorized. This is possible since the computation of the values for one column only needs those of the previous column.

Diagonal computation

The variant of search with differences that we consider now consists in computing the values of the table R according to the diagonals and by taking into account the property of monotony. The interesting positions on the diagonals are those where changes of values happen. These changes are incrementation because of the chosen distance.

For a number q of differences and a diagonal d , we denote by $L[q, d]$ the index i of the line on which $R[i, j] = q$ for the last time on the diagonal

R	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
i	$y[j]$	C	A	G	A	T	A	A	G	A	G	A	A	
-1	$x[i]$						0							
0	G							1						
1	A								1					
2	T									2				
3	A										2			
4	A											3		

Fig. 1.18. Values of table R on diagonal 5 for the approximate search for $x = \text{GATAA}$ in $y = \text{CAGATAAGAGAA}$. The last occurrences of each value on the diagonal are in gray. The lines where they occur are stored in table L by the algorithm of diagonal computation. We thus have $L[0, 5] = -1$, $L[1, 5] = 1$, $L[2, 5] = 3$, $L[3, 5] = 4$.

$j - i = d$. The idea of the definition of $L[q, d]$ is shown in Fig. 1.18. Formally, for $q = 0, 1, \dots, k$ and $d = -m, -m + 1, \dots, n - m$, we have

$$L[q, d] = i$$

if and only if i is the maximal index, $-1 \leq i < m$, for which there exists an index j , $-1 \leq j < n$, with

$$R[i, j] \leq q \text{ and } j - i = d.$$

In other words, for fixed q , the values $L[q, d]$ mark the lowest borderline of the values less than q in the table R (gray values in Fig. 1.19).

The definition of $L[q, d]$ implies that q is the smallest number of differences between $x[0..L[q, d]]$ and a factor of the text ending at position $d + L[q, d]$ on y . It moreover implies that the letters $x[L[q, d] + 1]$ and $y[d + L[q, d] + 1]$ are different when they are defined.

The values $L[q, d]$ are computed by iteration on d , for q going from 0 to $k + 1$. The principle of the computation relies on Recurrence 1.2 and the above statements. A simulation of the computation on the table R is presented in Fig. 1.19.

For the approximate pattern matching with k differences problem, only the values $L[q, d]$ for which $q \leq k$ are necessary. If $L[q, d] = m - 1$, it means that there is an occurrence of the string x at the diagonal d with at most q differences. The occurrence ending at position $d + m - 1$, this is only valid if $d + m \leq n$. We get another approximate occurrences at the end of y when $L[q, d] = i$ and $d + i = n - 1$; in this case the number of differences is $q + m - 1 - i$.

The algorithm K-DIFF-DIAG , given in Fig. 1.21 performs the approximate search for x in y by computing the values $L[q, d]$. It uses the function lcp where $lcp(u, v)$ gives the length of the longest common prefix of two strings u and v . Let us note that the first possible occurrence of an approximate factor of x in y can end at position $m - 1 - k$ on y , this corresponds to diagonal $-k$. The last possible occurrence starts at position $n - m + k$ on y , this corresponds

(a)	R	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
	i	$y[j]$	C	A	G	A	T	A	A	G	A	G	A	A	A
	-1	$x[i]$	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	G			0					0					
	1	A				0						0			
	2	T					0								
	3	A						0							
	4	A							0						

(b)	R	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
	i	$y[j]$	C	A	G	A	T	A	A	G	A	G	A	A	A
	-1	$x[i]$	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	A		1	1	0	1	1	1	1	0				
	2	T				1	0	1			1	1			
	3	A					1	0	1					1	
	4	A						1	0	1					1

Fig. 1.19. Simulation of the diagonal computation for the search for $x = \text{GATAA}$ in $y = \text{CAGATAAGAGAA}$ with one difference (see Figure 1.15). (a) Values computed during the first step (Lines 8–13 for $q = 0$ of Algorithm L-DIFF-DIAG); they detect the occurrence of x at right position 6 on y (since $R[4, 6] = 0$). (b) Values computed during the second step (Lines 8–13 for $q = 1$); they indicate the approximate factors of x with one difference at right positions 5, 7 and 11 on y (since $R[4, 5] = R[4, 7] = R[4, 11] = 1$).

to diagonal $n - m + k$. Thus only diagonals going from $-k$ to $n - m + k$ are considered during the computation (the initialization is also done on the diagonals $-k - 1$ and $n - m + k + 1$ to simplify the writing of the algorithm). Fig. 1.20 shows the table L obtained on the example of Fig. 1.15.

d	-2	-1	0	1	2	3	4	5	6	7	8	9
$q = -1$		-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
$q = 0$	-1	-1	-1	-1	4	-1	-1	-1	-1	1	-1	
$q = 1$		0	1	4	4	4	1	1	2	4		

Fig. 1.20. Values of table L of the diagonal computation when $x = \text{GATAA}$, $y = \text{CAGATAAGAGAA}$ and $k = 1$. Lines $q = 0$ and $q = 1$ correspond to a state of the computation simulated on table R in Figure 1.19. Values $4 = |\text{GATAA}| - 1$ on line $q = 1$ indicate the presence of occurrences of x with at most one difference ending at positions $1 + 4$, $2 + 4$, $3 + 4$ and $7 + 4$ on y .

The algorithm K-DIFF-DIAG computes the table L .

For every string x of length m , every string y of length n and every integer k such that $k < m \leq n$, the operation $\text{K-DIFF-DIAG}(x, m, y, n, k)$ computes the approximate occurrences of x in y with at most k differences.

```

K-DIFF-DIAG( $x, m, y, n, k$ )
1  for  $d \leftarrow -1$  to  $n - m + k + 1$  do
2     $L[-1, d] \leftarrow -2$ 
3  for  $q \leftarrow 0$  to  $k - 1$  do
4     $L[q, -q - 1] \leftarrow q - 1$ 
5     $L[q, -q - 2] \leftarrow q - 1$ 
6  for  $q \leftarrow 0$  to  $k$  do
7    for  $d \leftarrow -q$  to  $n - m + k - q$  do
8       $\ell \leftarrow \max \begin{cases} L[q - 1, d - 1] \\ L[q - 1, d] + 1 \\ L[q - 1, d + 1] + 1 \end{cases}$ 
9       $\ell \leftarrow \min\{\ell, m - 1\}$ 
10      $L[q, d] \leftarrow \ell + |\text{lcp}(x[\ell + 1..m - 1], y[d + \ell + 1..n - 1])|$ 
11     if  $L[q, d] = m - 1$  or  $d + L[q, d] = n - 1$  then
12       OUTPUT( $d + m - 1$ )

```

Fig. 1.21. Approximate string matching with k differences by diagonals.

In the way that the algorithm K-DIFF-DIAG is described, the memory space for the computation is principally used by the table L . We note that it is sufficient to memorize a single line to correctly perform the computation, this gives an implementation in space $O(n)$. It is however possible to reduce the space to $O(m)$ obtaining a space comparable to algorithm K-DIFF-CUT-OFF.

If the computation of $\text{lcp}(u, v)$ is realized in time $O(|\text{lcp}(u, v)|)$, the algorithm K-DIFF-DIAG executes in time $O(m \times n)$. But it is possible to prepare the strings x and y in such a way that any $\text{lcp}(u, v)$ query is answered in constant time. For this, we utilize the suffix tree, of the string $z = x\$y$ where $\$ \notin \text{alph}(y)$. The string

$$w = \text{lcp}(x[\ell + 1..m - 1], y[d + \ell + 1..n - 1])$$

is nothing else but the string $\text{lcp}(x[\ell + 1..m - 1]\$, y[d + \ell + 1..n - 1])$ since $\$ \notin \text{alph}(y)$. Let f and g be the external nodes of the suffix tree associated with suffixes of $x[\ell + 1..m - 1]\$y$ and $y[d + \ell + 1..n - 1]$ of the string z . Their common prefix of maximal length is then the label of the path leading from the initial state to the lowest node that is a common ancestor to f and g . This reduces the computation of w to the computation of this node.

The problem of the common ancestor that we are interested in here is the one for which the tree is static. A linear preprocessing of the tree allows to get a response in constant time to the queries (see notes). The consequence of this result is that on a fixed alphabet, after preparation of the strings x and y in linear time, it is possible to execute the algorithm K-DIFF-DIAG in time $O(k \times n)$.

1.3 Approximate String Matching with Mismatches

In this section, we are interested in the search for all the occurrences of a string x of length m in a string y of length n with at most k mismatches ($k \in \mathbb{N}$, $k < m \leq n$). The Hamming distance between two strings u and v of same length is the number of mismatches between u and v and is defined by:

$$\text{Ham}(u, v) = \text{card}\{i \mid u[i] \neq v[i], i = 0, 1, \dots, |u| - 1\}.$$

The problem can then be expressed as the search for all the positions $j = 0, 1, \dots, n - m$ on y that satisfy the inequality $\text{Ham}(x, y[j..j + m - 1]) \leq k$.

1.3.1 Search automaton

A natural solution to this problem consists in using an automaton that recognizes the language $V^*\{w \mid \text{Ham}(x, w) \leq k\}$. To do this, we can consider the non-deterministic automaton defined as follows:

- each state is a pair (ℓ, i) where ℓ is the level of the state and i is its depth, with $0 \leq \ell \leq k$, $-1 \leq i \leq m - 1$ and $\ell \leq i + 1$;
- the initial state is $(0, -1)$;
- the terminal states are of the form $(\ell, m - 1)$ with $0 \leq \ell \leq k$;
- the transitions are, for $0 \leq \ell \leq k$, $0 \leq i < m - 1$ and $a \in V$, either of the form $((0, -1), a, (0, -1))$, or of the form $((\ell, i), x[i + 1], (\ell, i + 1))$, or of the form $((\ell, i), a, (\ell + 1, i + 1))$ if $a \neq x[i + 1]$ and $0 \leq \ell \leq k - 1$.

The automaton possesses $k + 1$ levels, each level ℓ allowing to recognize the prefixes of x with ℓ mismatches. The transitions of the form $((\ell, i), a, (\ell, i + 1))$ correspond to the equality of letters while those of the form $((\ell, i), a, (\ell + 1, i + 1))$ correspond to the inequality of letters. The loop on the initial state allows to find all the occurrences of the searched factors. During the analysis of the text with the automaton, if a terminal state $(\ell, m - 1)$ is reached, this indicates the presence of an occurrence of x with exactly ℓ mismatches.

It is clear that the automaton possesses $(k + 1) \times (m + 1 - \frac{k}{2})$ states and that it can be built in time $O(k \times m)$. An example is shown in Fig. 1.22. Unfortunately, the total number of states obtained by determinizing the automaton is

$$\Theta(\min\{m^{k+1}, (k + 1)!(k + 2)^{m-k+1}\}).$$

We can check that a direct simulation of the automaton produces a search algorithm whose execution time is $O(m \times n)$ using the dynamic programming as in the previous section. Actually by using a method adapted to the problem we get, in the rest, an algorithm that performs the search in time $O(k \times n)$. This produces a solution of same complexity as the one of algorithm K-DIFF-DIAG that nevertheless solves a more general problem. But the solution that follows is based on a simple management of lists without using a search algorithm for common ancestor.

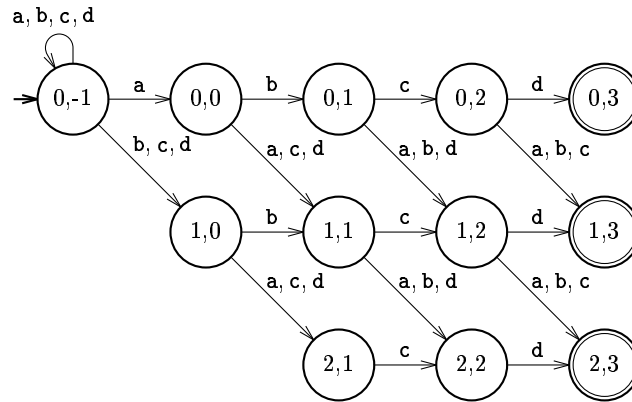


Fig. 1.22. The (non-deterministic) automaton of approximate pattern matching with two mismatches for the string $abcd$ on the alphabet $V = \{a, b, c, d\}$.

1.3.2 Specific implementation

We show how to reduce the execution time of the simulation of the previous automaton. To obtain the desired time, we utilize during the search a queue F of positions that stores detected mismatches. Its update is done by letter comparisons, but also by merging with queues associated with string x . The sequences that they represent are defined as follows.

For a shift q of x , $1 \leq q \leq m - 1$, $G[q]$ is the increasing sequence, of maximal length $2k + 1$, of the positions on x of the leftmost mismatches between $x[q..m - 1]$ and $x[0..m - q - 1]$. The sequences are determined during a preprocessing phase that is described at the end of the section.

The searching phase consists in performing attempts at all the positions $j = 0, 1, \dots, n - m$ on y . During the attempt at position j , we scan the factor $y[j..j + m - 1]$ of the text and the generic situation is the following (see Fig. 1.23): the prefix $y[j..g]$ of the window has already been scanned during

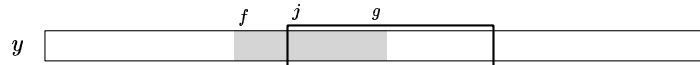


Fig. 1.23. Variables of Algorithm K-MISMATCHES. During the attempt at position j , variables f and g spot a previous attempt. The mismatches between $y[f..g]$ and $x[0..g - f]$ are stored in the queue F .

a previous attempt at position f , $f < j$, and no comparison already happens on the suffix $y[g + 1..n - 1]$ of the text. During the comparison of the already scanned part of the text, $y[j..g]$, around k tests can be necessary. Fig. 1.24 shows a computation example.


```

K-MISMATCHES( $x, m, G, y, n, k$ )
1  $F \leftarrow$  EMPTY-QUEUE()
2  $(f, g) \leftarrow (-1, -1)$ 
3 for  $j \leftarrow 0$  to  $n - m$  do
4   if LENGTH( $F$ ) > 0 and HEAD( $F$ ) =  $j - f - 1$  then
5     DEQUEUE( $F$ )
6   if  $j \leq g$  then
7      $J \leftarrow$  MIS-MERGE( $f, j, g, F, G[j - f]$ )
8   else  $J \leftarrow$  EMPTY-QUEUE()
9   if LENGTH( $J$ )  $\leq k$  then
10     $F \leftarrow J$ 
11     $f \leftarrow j$ 
12  do
13     $g \leftarrow g + 1$ 
14    if  $x[g - j] \neq y[g]$  then
15      ENQUEUE( $F, g - j$ )
16  while LENGTH( $F$ )  $\leq k$  and  $g < j + m - 1$ 
17  if LENGTH( $F$ )  $\leq k$  then
18    OUTPUT( $j$ )

```

Fig. 1.25. Approximate string matching with k mismatches.

positions by the quantity $j - f$. This is realized in the algorithm MIS-MERGE during the addition of a position in the output queue.

If the merge realized by the algorithm MIS-MERGE executes in linear time, the execution time of the algorithm K-MISMATCHES is $O(k \times n)$ in space $O(k \times m)$.

1.3.3 Merge

The aim of the operation MIS-MERGE($f, j, g, F, G[j - f]$) (Line 8 of the algorithm K-MISMATCHES) is to produce the sequence of positions of the mismatches between the strings $x[0..g-j]$ and $y[j..g]$, relying on the knowledge of the mismatches stored in the queues F and $G[j - f]$. This algorithm is given in Fig. 1.28.

The positions p in F mark the mismatches between $x[0..g-f]$ and $y[f..g]$, but only those that satisfy the inequality $f + p \geq j$ (by definition of F we already have $f + p \leq g$) are useful to the computation. The objective of the test in Line 5 of the algorithm K-MISMATCHES is precisely to delete from F the useless values. The positions q of $G[j - f]$ denote the mismatches between $x[j-f..m-1]$ and $x[0..m-j+f-1]$. Those that are useful must satisfy the inequality $f + q \leq g$ (we already have $f + q \geq j$). The test in Line 19 of the algorithm MIS-MERGE takes into account this constraint. Fig. 1.27 illustrates the merge (see also Fig. 1.24).

Let us consider a position p on x such that $j \leq f + p \leq g$. If p occurs in F , this means that $y[f + p] \neq x[p]$. If p is in $G[j - f]$, this means that

	j	$y[j]$	F
	0	a	$\langle 3, 4, 5 \rangle$
	1	b	$\langle 0, 1, 2, 5 \rangle$
	2	a	$\langle 2, 3 \rangle$
	3	b	$\langle 0, 1, 2, 3 \rangle$
	4	c	$\langle 0, 2, 3 \rangle$
	5	b	$\langle 0, 3, 4, 5 \rangle$
	6	b	$\langle 0, 1, 2, 3 \rangle$
	7	a	$\langle 3, 4, 6, 7 \rangle$
	8	b	$\langle 0, 1, 2, 3 \rangle$
	9	a	$\langle 3, 4, 5, 6 \rangle$
	10	b	$\langle 0, 1 \rangle$
	11	a	$\langle 1, 2, 3, 4 \rangle$
	12	a	$\langle 1, 2, 3 \rangle$
	13	c	$\langle 3, 4, 5, 7 \rangle$
	14	b	$\langle 0, 1, 2, 3 \rangle$
	15	a	$\langle 3, 4, 5, 7 \rangle$
	16	b	$\langle 0, 1, 2, 3 \rangle$
	17	a	$\langle 3, 5, 6, 7 \rangle$

i	$x[i]$	$G[i]$
0	a	$\langle \rangle$
1	b	$\langle 1, 2, 3, 4, 5, 6, 7 \rangle$
2	a	$\langle 3, 4, 5 \rangle$
3	c	$\langle 3, 6, 7 \rangle$
4	b	$\langle 4, 5, 6, 7 \rangle$
5	a	$\langle \rangle$
6	b	$\langle 6, 7 \rangle$
7	a	$\langle \rangle$

(a)
(b)

Fig. 1.26. Queues used for the approximate search with three mismatches of $x = abacbaba$ in $y = ababcbbabababababbbab$. **(a)** Values of table G for string $abacbaba$. The queue $G[3]$ for instance contains 3, 6 and 7, positions on x of the mismatches between its suffix $cbaba$ and its prefix $abacb$. **(b)** Successive values of queue F of the mismatches computed by Algorithm K-MISMATCHES. The values at positions 0, 2, 4, 10 and 12 on y possess less than three elements, which reveals the presence of occurrences of x with at most three mismatches at these positions. At position 0, for instance, the factor $ababcbbba$ of y possesses exactly three mismatches with x : they are at positions 3, 4 and 5 on x .

$x[p] \neq x[p - j + f]$. Four situations can arise for a position p whether it occurs or not in F and $G[j - f]$. (see Fig. 1.24 and 1.27):

1. The position p is neither in F nor in $G[j - f]$. We have $y[f + p] = x[p]$ and $x[p] = x[p - j + f]$, thus $y[f + p] = x[p - j + f]$.
2. The position p is in F but not in $G[j - f]$. We have $y[f + p] \neq x[p]$ and $x[p] = x[p - j + f]$, thus $y[f + p] \neq x[p - j + f]$.
3. The position p is in $G[j - f]$ but not in F . We have $y[f + p] = x[p]$ and $x[p] \neq x[p - j + f]$, thus $y[f + p] \neq x[p - j + f]$.
4. The position p is in F and in $G[j - f]$. We have $y[f + p] \neq x[p]$ and $x[p] \neq x[p - j + f]$, this does not allow to conclude on the equality between $y[f + p]$ and $x[p - j + f]$.

Among the enumerated cases, only the last three can lead to a mismatch between the letters $y[f + p]$ and $x[p - j + f]$. Only the last case requires an

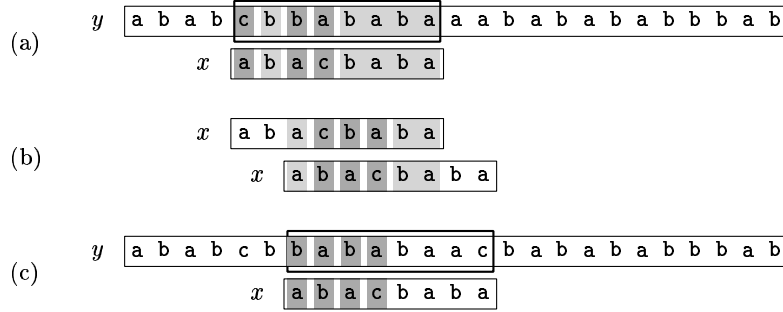


Fig. 1.27. Merge during the search with three mismatches of $x = abacbaba$ in $y = ababcbbabababababababab$. (a) Occurrence of x at position 4 on y with three mismatches at positions 0, 2 and 3 on x ; $F = \{0, 2, 3\}$. (b) There are three mismatches between $x[2..7]$ and $x[0..5]$; $G[2] = \{3, 4, 5\}$. (c) The sequences conserved for the merge are $\{2, 3\}$ and $\{3, 4, 5\}$, and this latter produces the sequence $\{2, 3, 4, 5\}$ of positions of the four first mismatches between x and $y[6..13]$. A single letter comparison is necessary at position 3, between $x[1]$ and $y[7]$, since the other positions only occur in one of the two sequences.

extra comparison of letters. They are processed in this respective order at Lines 7–8, 10–11 and 12–15 of the algorithm of merge.

The algorithm MIS-MERGE (see Fig. 1.28) executes in linear time.

1.3.4 Correctness proof

The correctness proof of the algorithm K-MISMATCHES relies on the proof of the function MIS-MERGE. One of the main arguments of the proof is a property of the Hamming distance that is stated below.

Let u , v and w be three strings of same length. Let us set $d = Ham(u, v)$, $d' = Ham(v, w)$, and assume $d' \leq d$. We then have:

$$d - d' \leq Ham(u, w) \leq d + d'.$$

When the operation MIS-MERGE($f, j, g, F, G[j - f]$) is executed in the algorithm K-MISMATCHES, the next conditions are satisfied:

1. $f < j \leq g \leq f + m - 1$;
2. $F = \langle p \mid x[p] \neq y[f + p] \text{ and } j \leq f + p \leq g \rangle$;
3. $x[g - f] \neq y[g]$;
4. $LENGTH(F) \leq k + 1$;
5. $G = \langle p \mid x[p] \neq x[p - j + f] \text{ and } j \leq f + p \leq g' \rangle$ for an integer g' such that $j \leq g' \leq f + m - 1$.

Moreover, if $g' < f + m - 1$, $LENGTH(G) = 2k + 1$ by definition of G . By taking these conditions as assumption we get the following result.

Let $J = \text{MIS-MERGE}(f, j, g, F, G[j - f])$. If $LENGTH(J) \leq k$,

```

MIS-MERGE( $f, j, g, F, G$ )
1   $J \leftarrow$  EMPTY-QUEUE()
2  while LENGTH( $J$ )  $\leq k$  and LENGTH( $F$ )  $> 0$ 
   and LENGTH( $G$ )  $> 0$  do
3     $p \leftarrow$  HEAD( $F$ )
4     $q \leftarrow$  HEAD( $G$ )
5    if  $p < q$  then
6      DEQUEUE( $F$ )
7      ENQUEUE( $J, p - j + f$ )
8    else if  $q < p$  then
9      DEQUEUE( $G$ )
10     ENQUEUE( $J, q - j + f$ )
11    else DEQUEUE( $F$ )
12     DEQUEUE( $G$ )
13     if  $x[p - j + f] \neq y[f + p]$  then
14       ENQUEUE( $J, p - j + f$ )
15  while LENGTH( $J$ )  $\leq k$  and LENGTH( $F$ )  $> 0$  do
16    DEQUEUED( $F, p$ )
17    ENQUEUE( $J, p - j + f$ )
18  while LENGTH( $J$ )  $\leq k$  and LENGTH( $G$ )  $> 0$ 
   and HEAD( $G$ )  $\leq g - f$  do
19    DEQUEUED( $G, q$ )
20    ENQUEUE( $J, q - j + f$ )
21  return  $J$ 

```

Fig. 1.28. Algorithm for merging queues.

$$J = \langle p \mid x[p] \neq y[j + p] \text{ and } j \leq j + p \leq g \rangle,$$

and, in the contrary case,

$$\text{Ham}(y[j..g], x[0..g - j]) > k.$$

The result that follows is on the correctness of algorithm K-MISMATCHES. It assumes that the sequences $G[q]$ are computed in accordance with their definition.

If $x, y \in V^*$, $m = |x|$, $n = |y|$, $k \in \mathbb{N}$ and $k < m \leq n$, the algorithm K-MISMATCHES detects all the positions $j = 0, 1, \dots, n - m$ on y for which $\text{Ham}(x, y[j..j + m - 1]) \leq k$.

1.3.5 Preprocessing

The aim of the preprocessing phase is to compute the values of the table G that is required by the algorithm K-MISMATCHES. Let us recall that for a shift q of x , $1 \leq q \leq m - 1$, $G[q]$ is the increasing sequence of positions on x of the leftmost mismatches between $x[q..m - 1]$ and $x[0..m - q - 1]$, and that this sequence is limited to $2k + 1$ elements.

The algorithm PRE-K-MISMATCHES is given in Fig. 1.29. The computation of the sequences $G[q]$ is realized in an elementary way by the function whose code follows.

```

PRE-K-MISMATCHES( $x, m, k$ )
1 for  $q \leftarrow 1$  to  $m - 1$  do
2    $G[q] \leftarrow$  EMPTY-QUEUE()
3    $i \leftarrow q$ 
4   while LENGTH( $G[q]$ ) <  $2k + 1$  and  $i < m$  do
5     if  $x[i] \neq x[i - q]$  then
6       ENQUEUE( $G[q], i$ )
7      $i \leftarrow i + 1$ 
8 return  $G$ 

```

Fig. 1.29. Preprocessing for the approximate string matching with mismatches.

The execution time of the algorithm is $O(m^2)$, but it is possible to prepare the table in time $O(k \times m \times \log m)$.

1.4 Shift-Or Algorithm

We are interested in this Section in the case of the search for short patterns. We first present an algorithm to solve the exact string matching problem, but that extends readily to the approximate string matching problems.

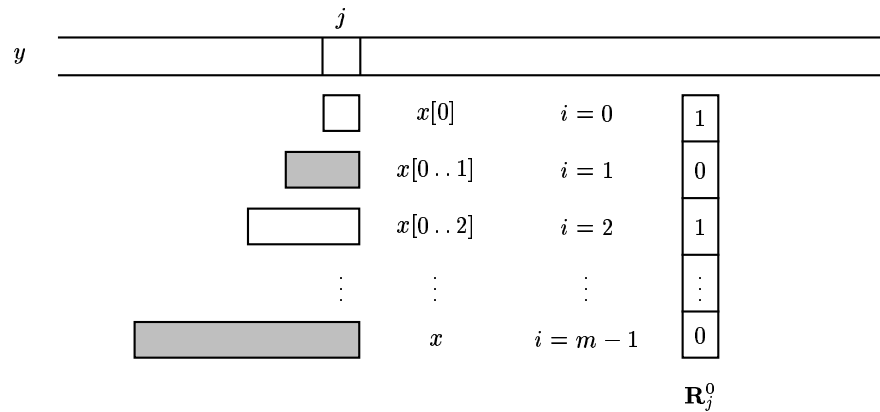


Fig. 1.30. Meaning of vector \mathbf{R}_j^0 . Each matching prefix of x is associated with value 1 in R_j^0 .

Let \mathbf{R}^0 be a bit array of size m . Vector \mathbf{R}_j^0 is the value of the entire array \mathbf{R}^0 after text character $y[j]$ has been processed (see Fig. 1.30). It contains information about all matches of prefixes of x that end at position j in the text. It is defined, for $0 \leq i \leq m - 1$, by

$$\mathbf{R}_j^0[i] = \begin{cases} 0 & \text{if } x[0..i] = y[j-i..j], \\ 1 & \text{otherwise.} \end{cases}$$

Therefore, $\mathbf{R}_j^0[m-1] = 0$ is equivalent to saying that an (exact) occurrence of the pattern x ends at position j in y .

The vector \mathbf{R}_j^0 can be computed after \mathbf{R}_{j-1}^0 by the following recurrence relation:

$$\mathbf{R}_j^0[i] = \begin{cases} 0 & \text{if } \mathbf{R}_{j-1}^0[i-1] = 0 \text{ and } x[i] = y[j], \\ 1 & \text{otherwise,} \end{cases}$$

and

$$\mathbf{R}_j^0[0] = \begin{cases} 0 & \text{if } x[0] = y[j], \\ 1 & \text{otherwise.} \end{cases}$$

The transition from \mathbf{R}_{j-1}^0 to \mathbf{R}_j^0 can be computed very fast as follows. For each $a \in V$, let S_a be a bit array of size m defined, for $0 \leq i \leq m - 1$, by

$$S_a[i] = 0 \quad \text{iff} \quad x[i] = a.$$

The array S_a denotes the positions of the character a in the pattern x . All arrays S_a are preprocessed before the search starts. And the computation of \mathbf{R}_j^0 reduces to two operations, SHIFT and OR:

$$\mathbf{R}_j^0 = \text{SHIFT}(\mathbf{R}_{j-1}^0) \quad \text{OR} \quad S_{y[j]}.$$

An example is given in Fig. 1.31.

Approximate String Matching with k Mismatches

The Shift-Or algorithm easily adapts to support approximate string matching with k mismatches. To simplify the description, we shall present the case where at most one substitution is allowed.

We use arrays \mathbf{R}^0 and S as before, and an additional bit array \mathbf{R}^1 of size m . Vector \mathbf{R}_{j-1}^1 indicates all matches with at most one substitution up to the text character $y[j-1]$. The recurrence on which the computation is based splits into two cases.

- There is an exact match on the first i characters of x up to $y[j-1]$ (i.e., $\mathbf{R}_{j-1}^0[i-1] = 0$). Then, substituting $x[i]$ to $y[j]$ creates a match with one substitution (see Fig. 1.32). Thus,

$$\mathbf{R}_j^1[i] = \mathbf{R}_{j-1}^0[i-1].$$

	S_A	S_C	S_G	S_T
	1	1	0	1
	0	1	1	1
	1	1	1	0
	0	1	1	1
	0	1	1	1
	C A G A T A A G A G A A			
G	1	1	0	1
A	1	1	1	0
T	1	1	1	1
A	1	1	1	1
A	1	1	1	1

Fig. 1.31. String $x = \text{GATAA}$ occurs at position 2 in $y = \text{CAGATAAGAGAA}$.

- There is a match with one substitution on the first i characters of x up to $y[j - 1]$ and $x[i] = y[j]$. Then, there is a match with one substitution of the first $i + 1$ characters of x up to $y[j]$ (see Fig. 1.33). Thus,

$$\mathbf{R}_j^1[i] = \begin{cases} \mathbf{R}_{j-1}^1[i - 1] & \text{if } x[i] = y[j], \\ 1 & \text{otherwise.} \end{cases}$$

This implies that \mathbf{R}_j^1 can be updated from \mathbf{R}_{j-1}^1 by the relation:

$$\mathbf{R}_j^1 = (\text{SHIFT}(\mathbf{R}_{j-1}^1) \quad \text{OR} \quad S_{y[j]}) \quad \text{AND} \quad \text{SHIFT}(\mathbf{R}_{j-1}^0).$$

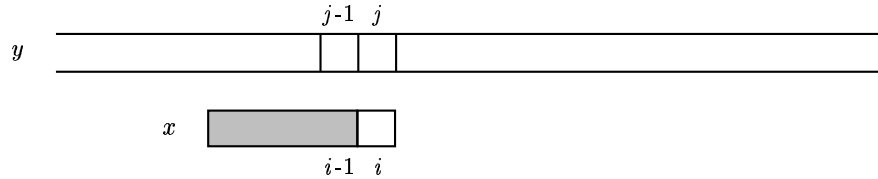


Fig. 1.32. If $\mathbf{R}_{j-1}^0[i - 1] = 0$ then $\mathbf{R}_j^1[i] = 0$.

An example is presented in Fig. 1.34.

Approximate String Matching with k Differences

We show in this section how to adapt the Shift-Or algorithm to the case of only one insertion, and then dually to the case of only one deletion. The method is based on the following elements.

One insertion is allowed: here, vector \mathbf{R}_{j-1}^1 indicates all matches with at most one insertion up to text character $y[j - 1]$. $\mathbf{R}_{j-1}^1[i - 1] = 0$ if the first i

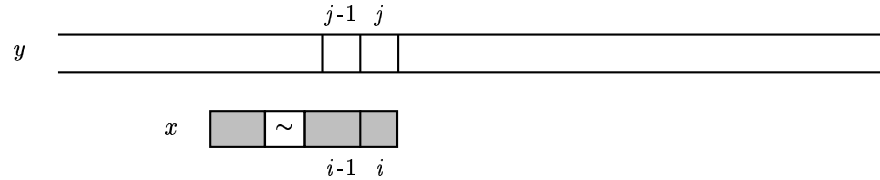


Fig. 1.33. $\mathbf{R}_j^1[i] = \mathbf{R}_{j-1}^1[i - 1]$ if $x[i] = y[j]$.

C	A	G	A	T	A	A	G	A	G	A	A
G	0	0	0	0	0	0	0	0	0	0	0
A	1	0	1	0	1	0	0	1	0	1	0
T	1	1	1	0	1	1	1	1	0	1	0
A	1	1	1	1	0	1	1	1	1	0	1
A	1	1	1	1	1	0	1	1	1	1	0

Fig. 1.34. String $x = \text{GATAA}$ occurs at positions 2 and 7 in $y = \text{CAGATAAGAGAA}$ with no more than one mismatch.

characters of x ($x[0..i - 1]$) match i symbols of the last $i + 1$ text characters up to $y[j - 1]$. Array \mathbf{R}^0 is maintained as before, and we show how to maintain array \mathbf{R}^1 . Two cases arise.

- There is an exact match on the first $i + 1$ characters of x ($x[0..i]$) up to $y[j - 1]$. Then inserting $y[j]$ creates a match with one insertion up to $y[j]$ (see Fig. 1.35). Thus,

$$\mathbf{R}_j^1[i] = \mathbf{R}_{j-1}^0[i].$$

- There is a match with one insertion on the i first characters of x up to $y[j - 1]$. Then if $x[i] = y[j]$ there is a match with one insertion on the first $i + 1$ characters of x up to $y[j]$ (see Fig. 1.36). Thus,

$$\mathbf{R}_j^1[i] = \begin{cases} \mathbf{R}_{j-1}^1[i - 1] & \text{if } x[i] = y[j], \\ 1 & \text{otherwise.} \end{cases}$$

This shows that \mathbf{R}_j^1 can be updated from \mathbf{R}_{j-1}^1 with the formula

$$\mathbf{R}_j^1 = (\text{SHIFT}(\mathbf{R}_{j-1}^1) \text{ OR } S_{y[j]}) \text{ AND } \mathbf{R}_{j-1}^0.$$

An example is given in Fig. 1.37.

One deletion is allowed: we assume here that \mathbf{R}_{j-1}^1 indicates all possible matches with at most one deletion up to $y[j - 1]$. As in the previous solution, two cases arise.

- There is an exact match on the first i characters of x ($x[0..i - 1]$) up to $y[j]$ (i.e., $\mathbf{R}_j^0[i - 1] = 0$). Then, deleting $x[i]$ creates a match with one deletion (see Fig. 1.38). Thus,

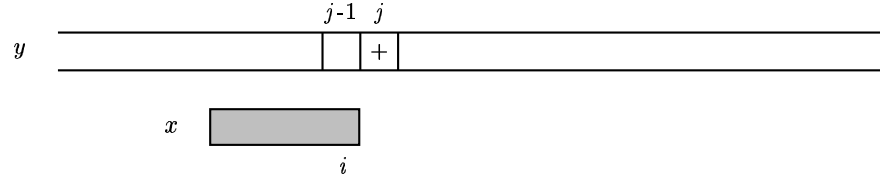


Fig. 1.35. If $\mathbf{R}_{j-1}^0[i] = 0$ then $\mathbf{R}_j^1[i] = 0$.

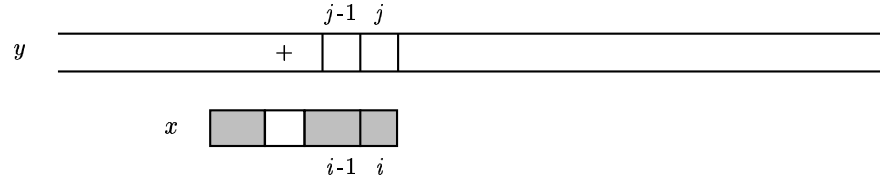


Fig. 1.36. $\mathbf{R}_j^1[i] = \mathbf{R}_{j-1}^1[i - 1]$ if $x[i] = y[j]$.

```

C A G A T A A G A G A A
G 1 1 1 0 1 1 1 1 0 1 0 1
A 1 1 1 1 0 1 1 1 1 0 1 0
T 1 1 1 1 1 0 1 1 1 1 1 1
A 1 1 1 1 1 1 0 1 1 1 1 1
A 1 1 1 1 1 1 1 0 1 1 1 1
    
```

Fig. 1.37. GATAAG is an occurrence of $x = \text{GATAA}$ with exactly one insertion in $y = \text{CAGATAAGAGAA}$.

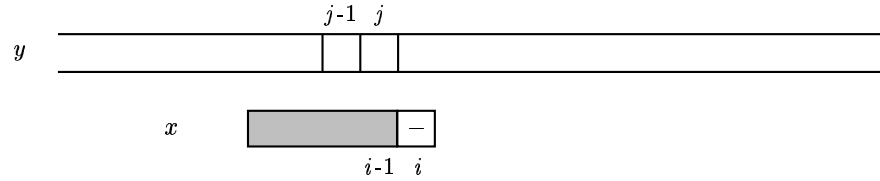


Fig. 1.38. If $\mathbf{R}_j^0[i] = 0$ then $\mathbf{R}_j^1[i] = 0$.

$$\mathbf{R}_j^1[i] = \mathbf{R}_j^0[i - 1].$$

- There is a match with one deletion on the first i characters of x up to $y[j - 1]$ and $x[i] = y[j]$. Then, there is a match with one deletion on the first $i + 1$ characters of x up to $y[j]$ (see Fig. 1.39). Thus,

$$\mathbf{R}_j^1[i] = \begin{cases} \mathbf{R}_{j-1}^1[i - 1] & \text{if } x[i] = y[j], \\ 1 & \text{otherwise.} \end{cases}$$

The discussion provides the following formula used to update \mathbf{R}_j^1 from \mathbf{R}_{j-1}^1 :

$$\mathbf{R}_j^1 = (\text{SHIFT}(\mathbf{R}_{j-1}^1) \text{ OR } S_{y[j]}) \text{ AND } \text{SHIFT}(\mathbf{R}_j^0).$$

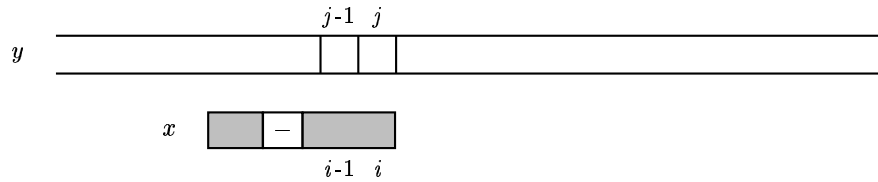


Fig. 1.39. $\mathbf{R}_j^1[i] = \mathbf{R}_{j-1}^1[i - 1]$ if $x[i] = y[j]$.

An example is presented in Fig. 1.40.

	C	A	G	A	T	A	A	G	A	G	A	A
G	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	0	0	1	0	0	0	0	0	0	0
T	1	1	1	0	0	1	1	1	0	1	0	1
A	1	1	1	1	0	0	1	1	1	1	1	0
A	1	1	1	1	1	0	0	1	1	1	1	1

Fig. 1.40. GATA and ATAA are two occurrences with one deletion of $x = \text{GATAA}$ in $y = \text{CAGATAAGAGAA}$

Wu–Manber Algorithm

We present in this section a general solution for the approximate string matching problem with at most k differences of the types: insertion, deletion, and substitution. It is an extension of the problems presented above. The following algorithm maintains $k + 1$ bit arrays $\mathbf{R}^0, \mathbf{R}^1, \dots, \mathbf{R}^k$ that are described now. The vector \mathbf{R}^0 is maintained similarly as in the exact matching case. The other vectors are computed with the formula ($1 \leq \ell \leq k$)

$$\begin{aligned} \mathbf{R}_j^\ell = & (\text{SHIFT}(\mathbf{R}_{j-1}^\ell) \text{ OR } S_{y[j]}) \\ & \text{AND } \text{SHIFT}(\mathbf{R}_j^{\ell-1}) \\ & \text{AND } \text{SHIFT}(\mathbf{R}_{j-1}^{\ell-1}) \\ & \text{AND } \mathbf{R}_{j-1}^{\ell-1} \end{aligned}$$

which can be rewritten into

$$\begin{aligned} \mathbf{R}_j^\ell = & (\text{SHIFT}(\mathbf{R}_{j-1}^\ell) \text{ OR } S_{y[j]}) \\ & \text{AND } \text{SHIFT}(\mathbf{R}_j^{\ell-1} \text{ AND } \mathbf{R}_{j-1}^{\ell-1}) \\ & \text{AND } \mathbf{R}_{j-1}^{\ell-1}. \end{aligned}$$

```

      C A G A T A A G A G A A
      G 0 0 0 0 0 0 0 0 0 0 0 0
      A 1 0 0 0 0 0 0 0 0 0 0 0
      T 1 1 1 0 0 0 1 1 0 0 0 0
      A 1 1 1 1 0 0 0 1 1 1 0 0
      A 1 1 1 1 1 0 0 0 1 1 1 0

```

Fig. 1.41. Here $x = \text{GATAA}$ and $y = \text{CAGATAAGAGAA}$ and $k = 1$. The output 5, 6, 7, and 11 corresponds to the segments GATA , GATAA , GATAAG , and GAGAA which approximate the pattern GATAA with no more than one difference.

```

WM( $x, m, y, n, k$ )
1  for each character  $a \in V$  do
2     $S_a \leftarrow 1^m$ 
3  for  $i \leftarrow 0$  to  $m - 1$  do
4     $S_{x[i]}[i] \leftarrow 0$ 
5   $\mathbf{R}^0 \leftarrow 1^m$ 
6  for  $\ell \leftarrow 1$  to  $k$  do
7     $\mathbf{R}^\ell \leftarrow \text{SHIFT}(\mathbf{R}^{\ell-1})$ 
8  for  $j \leftarrow 0$  to  $n - 1$  do
9     $T \leftarrow \mathbf{R}^0$ 
10    $\mathbf{R}^0 \leftarrow \text{SHIFT}(\mathbf{R}^0) \text{ OR } S_{y[j]}$ 
11  for  $\ell \leftarrow 1$  to  $k$  do
12    $T' \leftarrow \mathbf{R}^\ell$ 
13    $\mathbf{R}^\ell \leftarrow (\text{SHIFT}(\mathbf{R}^\ell) \text{ OR } S_{y[j]}) \text{ AND}$ 
       $(\text{SHIFT}(T \text{ AND } \mathbf{R}^{\ell-1})) \text{ AND } T$ 
14    $T \leftarrow T'$ 
15  if  $\mathbf{R}^k[m - 1] = 0$  then
16   OUTPUT( $j$ )

```

Fig. 1.42. Wu–Manber approximate string matching algorithm.

An example is given in Fig. 1.41.

The method, called the Wu–Manber algorithm, is implemented in Fig. 1.42. It assumes that the length of the pattern is no more than the size of the memory word of the machine, which is often the case in applications.

The preprocessing phase of the algorithm takes $O(\sigma m + km)$ memory space, and runs in time $O(\sigma m + k)$. The time complexity of its searching phase is $O(kn)$.

1.5 Bibliographic notes

The techniques described in this chapter are overused in molecular biology for comparing sequences of chains of nucleic acids (DNA or RNA) or of amino acids (proteins). The books of Deonier, Tavaré and Waterman [9], Setubal

and Meidanis [28], Gusfield [10] and Böckenhauer and Bongartz [4] constitute excellent introductions to problems of the domain. The book of Sankoff and Kruskal [26] contains numerous applications of alignments. The book of Crochemore, Hancart and Lecroq [7] presents in detail, together with their correctness proofs, the algorithms for computing alignments and solving the approximate string matching problems.

The notion of longest common subsequence to two strings is used for file comparison. The command `diff` of the UNIX system implements an algorithm based on this notion by considering that the lines of the files are letters of the alphabet. Among the algorithms at the basis of this command are those of Hunt and Szymanski [14] and of Myers [20]. A general presentation of the algorithms for searching for common subsequences can be found in an article by Apostolico [1]. Wong and Chandra [32] shown that the algorithm `LCS-SIMPLE` is optimal in a model where we limit the access to letters to equality tests. Without this condition, Hirschberg [13] gave a (lower) bound $\Omega(n \times \log n)$. On a bounded alphabet, Masek and Paterson [18] gave an algorithm running in time $O(n^2/\log n)$. The extension of this result to the general computation of alignments is an open problem (see Apostolico and Giancarlo [2]). Using the Lempel-Ziv factorization of the two strings, Crochemore, Landau and Ziv-Ukelson designed an algorithm for computing alignments running in time $O(hn^2/\log n)$ where $h \leq 1$ is the entropy of the text.

The initial algorithm of global alignment, from Needleman and Wunsch [24], runs in cubic time. The algorithm of Wagner and Fischer [31], as well as the algorithm of local alignment of Smith and Waterman [29], run in quadratic time (see [10], page 234). The method of dynamic programming was introduced by Bellman (1957; see [6]). Sankoff [25] discusses the introduction of the dynamic programming in the processing of molecular sequences.

The algorithm `LCS` is from Hirschberg [12]. A generalization of this method has been proposed by Myers and Miller [21].

Charras and Lecroq created the site [5], accessible on the Web, where animations of alignment algorithms are available.

The book of Navarro and Raffinot [23] is an excellent introduction to exact and approximate string matching.

The algorithm `K-DIFF-CUT-OFF` is from to Ukkonen [30]. The algorithm `K-DIFF-DIAG` together with its implementation with the help of the computation of common ancestors was described by Landau and Vishkin [16]. Harel and Tarjan [11] presented the first algorithm running in constant time that solves the problem of the common ancestor to two nodes of a tree. An improved version is from Schieber and Vishkin [27].

Landau and Vishkin [15] conceived the algorithm `K-MISMATCHES`. The size of the automaton of Section 1.3 was established by Melichar [19].

The approximate pattern matching for short strings in the way of the algorithm `K-DIFF-SHORT-PATTERN` is from Wu and Manber [33] and also from Baeza-Yates and Gonnet [3].

A synthesis and experimental results on the approximate pattern matching is presented by Navarro [22].

References

1. A. Apostolico. String Editing and Longest Common Subsequences. G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*, 1997, 361–398. Springer-Verlag.
2. A. Apostolico and R. Giancarlo. Sequence alignment in molecular biology. *J. Comput. Bio.*, 5(2):173–196, 1998.
3. R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Comm. ACM*, 35(10):74–82, 1992.
4. H.-J. Böckenhauer and D. Bongartz. *Algorithmic Aspects of Bioinformatics*. Springer-Verlag, 2007.
5. C. Charras and T. Lecroq. *Sequence Comparison*. <http://monge.univ-mlv.fr/~lecroq/seqcomp/>
6. T. H. Cormen, C. E. Leiserson and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
7. M. Crochemore, C. Hancart and T. Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.
8. M. Crochemore, G. M. Landau and M. Ziv-Ukelson. A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. *SIAM J. Comput.*, 32(6):1654–1673, 2003.
9. R. C. Deonier, S. Tavaré and M. S. Waterman. *Computational Genome Analysis: An Introduction (Statistics for Biology & Health)*. Springer-Verlag, 2005.
10. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
11. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
12. D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Comm. ACM*, 18(6):341–343, 1975.
13. D. S. Hirschberg. An Information-Theoretic Lower Bound for the Longest Common Subsequence Problem. *Inform. Process. Lett.*, 7(1):40–41, 1978.
14. J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Comm. ACM*, 20(5):350–353, 1977.
15. G. M. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoret. Comput. Sci.*, 43(2–3):239–249, 1986.
16. G. M. Landau and U. Vishkin. Fast string matching with k differences. *J. Comput. System Sci.*, 37(1):63–78, 1988.
17. V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Sov. Phys. Dokl.*, 6:707–710, 1966.
18. W. J. Masek and M. S. Paterson. A faster algorithm for computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–13, 1980.
19. B. Melichar. Approximate String Matching by Finite Automata. *Computer Analysis of Images and Patterns*, V. Hlavác and R. Sára, editors, Lecture Notes in Computer Science, volume 970, 342–349, Springer-Verlag, Berlin, 1995.
20. E. W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

21. E. W. Myers and W. Miller. Optimal alignment in linear space. *CABIOS*, 4(1):11–17, 1988.
22. G. Navarro. A guided tour to approximate string matching. *ACM Comp. Surv.*, 33(1):31–88, 2001.
23. G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.
24. 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.
25. D. Sankoff. The early introduction of dynamic programming into computational biology. *Bioinformatics*, 16:41–47, 2000.
26. D. Sankoff and J. Kruskal, editors. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Cambridge University Press, second edition, 1999.
27. B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, 1988.
28. J. C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.
29. T. F. Smith and M. S. Waterman. Identification of common molecular sequences. *J. Mol. Biol.*, 147:195–197, 1981.
30. E. Ukkonen. Algorithms for approximate string matching. *Inform. and Control*, 64(1–3):100–118, 1985.
31. R. A. Wagner and M. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.
32. C. K. Wong and A. K. Chandra. Bounds for the string editing problem. *J. ACM*, 23(1):13–16, 1976.
33. S. Wu and U. Manber. Fast text searching allowing errors. *Comm. ACM*, 35(10):83–91, 1992.