Pattern Matching and Text Compression Algorithms

Maxime Crochemore[1] Thierry Lecroq[2]

January 8, 2003

[1]University of Marne-la-Vallée
[2]University of Rouen

Chapter 8

Pattern Matching and Text Compression Algorithms

8.1 Processing Texts Efficiently

The present chapter describes a few standard algorithms used for processing texts. They apply, for example, to the manipulation of texts (text editors), to the storage of textual data (text compression), and to data retrieval systems. The algorithms of this chapter are interesting in different respects. First, they are basic components used in the implementations of practical software. Second, they introduce programming methods that serve as paradigms in other fields of computer science (system or software design). Third, they play an important role in theoretical computer science by providing challenging problems.

Although data are stored in various ways, text remains the main form of exchanging information. This is particularly evident in literature or linguistics where data are composed of huge corpora and dictionaries. This applies as well to computer science where a large amount of data are stored in linear files. And this is also the case in molecular biology where biological molecules can often be approximated as sequences of nucleotides or amino acids. Moreover, the quantity of available data in these fields tends to double every 18 months. This is the reason why algorithms should be efficient even if the speed of computers increases at a steady pace.

Pattern matching is the problem of locating a specific pattern inside raw data. The pattern is usually a collection of strings described in some formal language. Two kinds of textual patterns are presented: single strings and approximated strings. We also present two algorithms for matching patterns in images that are extensions of string-matching algorithms.

In several applications, texts need to be structured before being searched. Even if no further information is known about their syntactic structure, it is possible and indeed extremely efficient to build a data structure that supports searches. From among several existing data structures equivalent to represent indexes, we present the suffix tree, along with its construction.

The comparison of strings is implicit in the approximate pattern searching problem. Since it is sometimes required to compare just two strings (files or molecular sequences) we introduce the basic method based on longest common subsequences.

Finally, the chapter contains two classical text compression algorithms. Variants of these algorithms are implemented in practical compression software, in which they are often combined together or with other elementary methods. An example of mixing different methods is presented there.

The efficiency of algorithms is evaluated by their running times, and sometimes by the amount of memory space they require at run time as well.

```
BF(x, m, y, n)
  ▷ Searching
  for j ← 0 to n − m
      do i ← 0
         while i < m and x[i] = y[i + j]
               do i ← i + 1
         if i ≥ m
           then OUTPUT(j)
```

Figure 8.1: The brute force string-matching algorithm.

8.2 String-Matching Algorithms

String matching is the problem of finding one, or more generally, all the **occurrences** of a pattern in a text. The pattern and the text are both strings built over a finite alphabet (a finite set of symbols). Each algorithm of this section outputs all occurrences of the pattern in the text. The pattern is denoted by $x = x[0 \mathinner{..} m-1]$; its length is equal to m. The text is denoted by $y = y[0 \mathinner{..} n-1]$; its length is equal to n. The alphabet is denoted by Σ and its size is equal to σ.

String-matching algorithms of the present section work as follows: they first align the left ends of the pattern and the text, then compare the aligned symbols of the text and the pattern—this specific work is called an attempt or a scan—and after a whole match of the pattern or after a mismatch they shift the pattern to the right. They repeat the same procedure again until the right end of the pattern goes beyond the right end of the text. This is called the scan and shift mechanism. We associate each attempt with the position j in the text, when the pattern is aligned with $y[j \mathinner{..} j+m-1]$.

The brute force algorithm consists in checking, at all positions in the text between 0 and $n - m$, whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern exactly one position to the right. This is the simplest algorithm, which is described in Fig. 8.1.

The time complexity of the brute force algorithm is $O(mn)$ in the worst case but its behavior in practice is often linear on specific data.

8.2.1 Karp–Rabin Algorithm

Hashing provides a simple method for avoiding a quadratic number of symbol comparisons in most practical situations. Instead of checking at each position of the text whether the pattern occurs, it seems to be more efficient to check only if the portion of the text aligned with the pattern "looks like" the pattern. In order to check the resemblance between these portions a hashing function is used. To be helpful for the string-matching problem the hashing function should have the following properties:

- efficiently computable,
- highly discriminating for strings,
- $hash(y[j+1 \mathinner{..} j+m])$ must be easily computable from $hash(y[j \mathinner{..} j+m-1])$:
  $hash(y[j+1 \mathinner{..} j+m]) = \textsc{ReHash}(y[j], y[j+m], hash(y[j \mathinner{..} j+m-1]))$.

For a word w of length k, its symbols can be considered as digits, and we define $hash(w)$ by

$$hash(w[0 \mathinner{..} k-1]) = (w[0] \times 2^{k-1} + w[1] \times 2^{k-2} + \cdots + w[k-1]) \bmod q$$

where q is a large number. Then, $\textsc{ReHash}$ has a simple expression

$$\textsc{ReHash}(a, b, h) = ((h - a \times d) \times 2 + b) \bmod q$$

where $d = 2^{k-1}$ where q is the computer word-size (see Fig. 8.2).

During the search for the pattern x, $hash(x)$ is compared with $hash(y[j-m+1 \mathinner{..} j])$ for $m - 1 \leq j \leq n - 1$. If an equality is found, it is still necessary to check the equality $x = y[j-m+1 \mathinner{..} j]$ symbol by symbol.

```
ReHash(a, b, h)
  return ((h − a × d) << 1) + b
```

Figure 8.2: Function ReHash.

```
KR(x, m, y, n)
  ▷ Preprocessing
  d ← 1
  for i ← 1 to m − 1
        do d ← d << 1
  h_x ← 0
  h_y ← 0
  for i ← 0 to m − 1
        do h_x ← (h_x << 1) + x[i]
           h_y ← (h_y << 1) + y[i]
  ▷ Searching
  if h_x = h_y and x = y[0 . . m − 1]
    then Output(0)
  j ← m
  while j < n
        do h_y ← ReHash(y[j − m], y[j], h_y)
           if h_x = h_y and x = y[j − m + 1 . . j]
             then Output(j − m + 1)
           j ← j + 1
```

Figure 8.3: The Karp–Rabin string-matching algorithm.

In the algorithms of Fig. 8.2 and Fig. 8.3 all multiplications by 2 are implemented by shifts (operator $<<$). Furthermore, the computation of the modulus function is avoided by using the implicit modular arithmetic given by the hardware that forgets carries in integer operations. Thus, q is chosen as the maximum value of an integer of the system.

The worst-case time complexity of the Karp–Rabin algorithm is quadratic (as it is for the brute force algorithm) but its expected running time is $O(m + n)$.

Example 8.1: Let $x =$ **ing**. Then $hash(x) = 105 \times 2^2 + 110 \times 2 + 103 = 743$ (symbols are assimilated with their ASCII codes).

y	=	s	t	r	i	n	g		m	a	t	c	h	i	n	g
$hash$	=			806	797	776	743	678	585	443	746	719	766	709	736	743

8.2.2 Knuth–Morris–Pratt Algorithm

This section presents the first discovered linear-time string-matching algorithm. Its design follows a tight analysis of the brute force algorithm, and especially of the way this latter algorithm wastes the information gathered during the scan of the text.

Let us look more closely at the brute force algorithm. It is possible to improve the length of shifts and simultaneously remember some portions of the text that match the pattern. This saves comparisons between characters of the text and of the pattern, and consequently increases the speed of the search.

Consider an attempt at position j, that is, when the pattern $x[0\,..\,m-1]$ is aligned with the segment $y[j\,..\,j+m-1]$ of the text. Assume that the first mismatch (during a left to right scan) occurs between symbols $x[i]$ and $y[i+j]$ for $0 \le i < m$. Then, $x[0\,..\,i-1] = y[j\,..\,i+j-1] = u$ and $a = x[i] \neq y[i+j] = b$. When shifting, it is reasonable to expect that a **prefix** v of the pattern matches some **suffix** of the portion u of the text. Moreover, if we want to avoid another immediate mismatch, the letter following the prefix

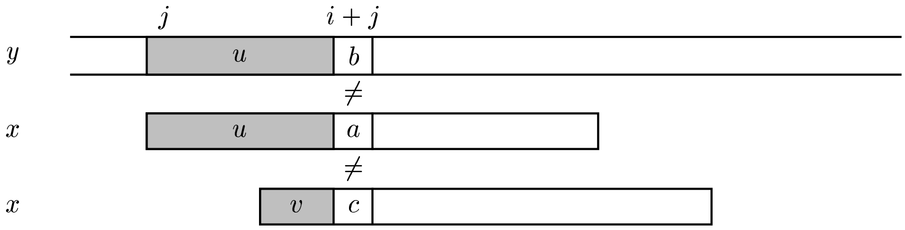


Figure 8.4: Shift in the Knuth–Morris–Pratt algorithm (v suffix of u).

```
KMP(x, m, y, n)
  ▷ Preprocessing
  next ← PreKMP(x, m)
  ▷ Searching
  i ← 0
  j ← 0
  while j < n
      do while i > −1 and x[i] ≠ y[j]
              do i ← next[i]
         i ← i + 1
         j ← j + 1
         if i ≥ m
           then OUTPUT(j − i)
                i ← next[i]
```

Figure 8.5: The Knuth–Morris–Pratt string-matching algorithm.

v in the pattern must be different from a. (Indeed, it should be expected that v matches a suffix of ub, but elaborating along this idea goes beyond the scope of the chapter.) The longest such prefix v is called the **border** of u (it occurs at both ends of u). This introduces the notation: let $next[i]$ be the length of the longest (proper) border of $x[0\,..\,i-1]$ followed by a character c different from $x[i]$. Then, after a shift, the comparisons can resume between characters $x[next[i]]$ and $y[i+j]$ without missing any occurrence of x in y and having to backtrack on the text (see Fig. 8.4).

Example 8.2: Here

$y =$	.	.	.	a	b	a	b	a	a	b	.	.	.	.	.
$x =$				$\underline{\text{a}}$	$\underline{\text{b}}$	$\underline{\text{a}}$	$\underline{\text{b}}$	$\underline{\text{a}}$	$\underline{\text{b}}$	a					
$x =$									$\underline{\text{a}}$	$\underline{\text{b}}$	a	b	a	b	a

Compared symbols are underlined. Note that the empty string is the suitable border of **ababa**. Other borders of **ababa** are **aba** and **a**.

The Knuth–Morris–Pratt algorithm is displayed in Fig. 8.5. The table *next* it uses is computed in $O(m)$ time before the search phase, applying the same searching algorithm to the pattern itself, as if $y = x$ (see Fig. 8.6). The worst-case running time of the algorithm is $O(m+n)$ and it requires $O(m)$ extra space. These quantities are independent of the size of the underlying alphabet.

8.2.3 Boyer–Moore Algorithm

The Boyer–Moore algorithm is considered as the most efficient string-matching algorithm in usual applications. A simplified version of it, or the entire algorithm, is often implemented in text editors for the search and substitute commands.

The algorithm scans the characters of the pattern from right to left beginning with the rightmost

```
PreKMP(x, m)
i ← −1
j ← 0
next[0] ← −1
while j < m
     do while i > −1 and x[i] ≠ x[j]
            do i ← next[i]
        i ← i + 1
        j ← j + 1
        if x[i] = x[j]
          then next[j] ← next[i]
          else  next[j] ← i
return next
```

Figure 8.6: Preprocessing phase of the Knuth–Morris–Pratt algorithm: computing *next*.

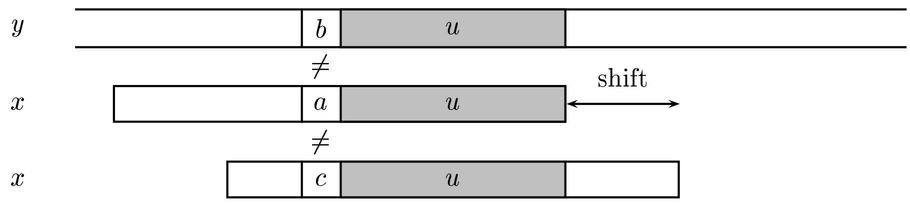


Figure 8.7: The good-suffix shift, when u reappears preceded by a character different from a.

symbol. In case of a mismatch (or a complete match of the whole pattern) it uses two precomputed functions to shift the pattern to the right. These two shift functions are called the *bad-character shift* and the *good-suffix shift*. They are based on the following observations.

Assume that a mismatch occurs between the character $x[i] = a$ of the pattern and the character $y[i+j] = b$ of the text during an attempt at position j. Then, $x[i+1\,..\,m-1] = y[i+j+1\,..\,j+m-1] = u$ and $x[i] \neq y[i+j]$. The good-suffix shift consists in aligning the **segment** $y[i+j+1\,..\,j+m-1]$ with its rightmost occurrence in x that is preceded by a character different from $x[i]$ (see Fig. 8.7). If there exists no such segment, the shift consists in aligning the longest suffix v of $y[i+j+1\,..\,j+m-1]$ with a matching prefix of x (see Fig. 8.8).

Example 8.3: Here

$y =$	.	.	.	**a**	**b**	**b**	**a**	**a**	**b**	**b**	**a**	**b**	**b**	**a**	.	.	.
$x =$	**a**	**b**	**b**	**a**	**a**	**b**	$\underline{\mathbf{b}}$	$\underline{\mathbf{a}}$	$\underline{\mathbf{b}}$	$\underline{\mathbf{b}}$	$\underline{\mathbf{a}}$						
$x =$				$\underline{\mathbf{a}}$	$\underline{\mathbf{b}}$	$\underline{\mathbf{b}}$	$\underline{\mathbf{a}}$	$\underline{\mathbf{a}}$	$\underline{\mathbf{b}}$	$\underline{\mathbf{b}}$	$\underline{\mathbf{a}}$	$\underline{\mathbf{b}}$	$\underline{\mathbf{b}}$	$\underline{\mathbf{a}}$			

The shift is driven by the suffix **abba** of x found in the text. After the shift, the segment **abba** in

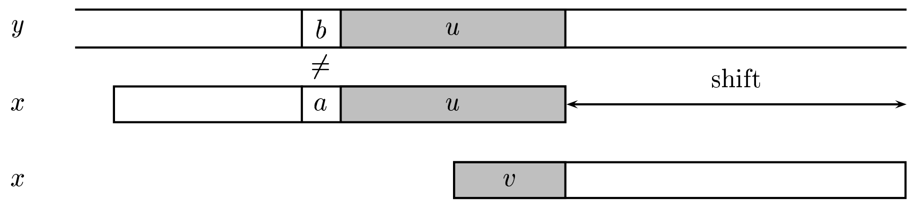


Figure 8.8: The good-suffix shift, when the situation of Fig. 8.7 does not happen. Only a suffix of u reappears as a prefix of x.

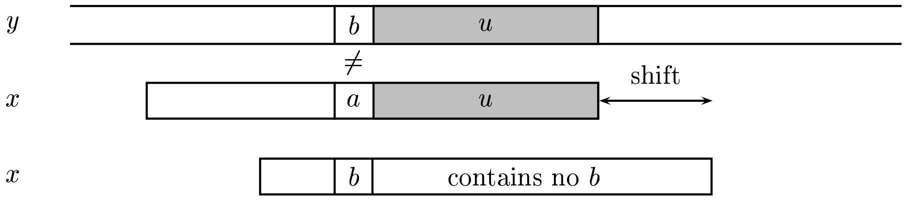


Figure 8.9: The bad-character shift, b appears in x.

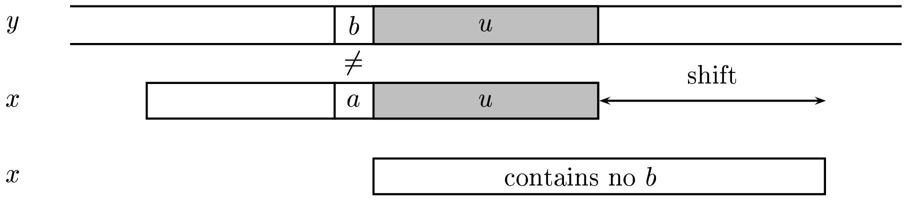


Figure 8.10: The bad-character shift, b does not appear in x (except possibly at $m-1$).

the middle of y matches a segment of x as in Fig. 8.7. The same mismatch does not recur.

Example 8.4: Here

$y=$	.	.	.	**a**	**b**	**b**	**a**	**a**	**b**	**b**	**a**	**b**	**b**	**a**	**b**	**b**	**a**	.	.
$x=$			**b**	**b**	**a**	**b**	$\underline{\mathbf{b}}$	$\underline{\mathbf{a}}$	$\underline{\mathbf{b}}$	$\underline{\mathbf{b}}$	$\underline{\mathbf{a}}$								
$x=$									$\underline{\mathbf{b}}$	$\underline{\mathbf{b}}$	$\underline{\mathbf{a}}$	$\underline{\mathbf{b}}$	$\underline{\mathbf{b}}$	$\underline{\mathbf{a}}$	$\underline{\mathbf{b}}$	$\underline{\mathbf{b}}$	$\underline{\mathbf{a}}$		

The segment **abba** found in y partially matches a prefix of x after the shift, as in Fig. 8.8.

The bad-character shift consists in aligning the text character $y[i+j]$ with its rightmost occurrence in $x[0\,..\,m-2]$ (see Fig. 8.9). If $y[i+j]$ does not appear in the pattern x, no occurrence of x in y can overlap the symbol $y[i+j]$, then the left end of the pattern is aligned with the character at position $i+j+1$ (see Fig. 8.10).

Example 8.5: Here

$y=$	.	.	.	.	.	.	**a**	**b**	**c**	**d**	.	.	.	.
$x=$	**c**	**d**	**a**	**h**	**g**	**f**	$\underline{\mathbf{e}}$	$\underline{\mathbf{b}}$	$\underline{\mathbf{c}}$	$\underline{\mathbf{d}}$				
$x=$					**c**	**d**	**a**	**h**	**g**	**f**	**e**	**b**	**c**	$\underline{\mathbf{d}}$

The shift aligns the symbol **a** in x with the mismatch symbol **a** in the text y (Fig. 8.9.).

Example 8.6: Here

$y=$	.	.	.	.	.	**a**	**b**	**c**	**d**	.	.	.	.	.	.
$x=$	**c**	**d**	**h**	**g**	**f**	$\underline{\mathbf{e}}$	$\underline{\mathbf{b}}$	$\underline{\mathbf{c}}$	$\underline{\mathbf{d}}$						
$x=$							**c**	**d**	**h**	**g**	**f**	**e**	**b**	**c**	$\underline{\mathbf{d}}$

The shift positions the left end of x right after the symbol **a** of y (Fig. 8.10.).

The Boyer–Moore algorithm is shown in Fig. 8.11. For shifting the pattern, it applies the maximum between the bad-character shift and the good-suffix shift. More formally, the two shift functions are defined as follows. The bad-character shift is stored in a table bc of size σ and the good-suffix shift is

```
BM(x, m, y, n)
▷ Preprocessing
gs ← PreGS(x, m)
bc ← PreBC(x, m)
▷ Preprocessing
j ← 0
while j ≤ n − m
     do i ← m − 1
        while i ≥ 0 and x[i] = y[i + j]
            do i ← i − 1
        if i < 0
          then Output(j)
        j ← max{gs[i + 1], bc[y[i + j] − m + i + 1]}
```

Figure 8.11: The Boyer–Moore string-matching algorithm.

```
PreBC(x, m)
for a ← firstLetter to lastLetter
     do bc[a] ← m
for i ← 0 to m − 2
     do bc[x[i]] ← m − 1 − i
return bc
```

Figure 8.12: Computation of the bad-character shift.

stored in a table gs of size $m+1$. For $a \in \Sigma$

$$bc[a] = \begin{cases} \min\{i \mid 1 \leq i < m \text{ and } x[m-1-i] = a\} & \text{if } a \text{ appears in } x, \\ m & \text{otherwise.} \end{cases}$$

Let us define two conditions,

$$\begin{cases} cond_1(i,s): & \text{for each } k \text{ such that } i < k < m,\ s \geq k \text{ or } x[k-s] = x[k], \\ cond_2(i,s): & \text{if } s < i \text{ then } x[i-s] \neq x[i]. \end{cases}$$

Then, for $0 \leq i < m$,

$$gs[i+1] = \min\{s > 0 \mid cond_1(i,s) \text{ and } cond_2(i,s) \text{ hold}\}$$

and we define $gs[0]$ as the length of the smallest period of x.
To compute the table gs, a table $suff$ is used. This table can be defined as follows: for $i = 0, 1, \ldots, m-1$,

$$suff[i] = \text{longest common suffix between } x[0\,..\,i] \text{ and } x\,.$$

It is computed in linear time and space by the function Suffixes (see Fig. 8.13).

Tables bc and gs can be precomputed in time $O(m+\sigma)$ before the search phase and require an extra space in $O(m+\sigma)$ (see Fig. 8.12 and Fig. 8.14). The worst-case running time of the algorithm is quadratic. However, on large alphabets (relative to the length of the pattern) the algorithm is extremely fast. Slight modifications of the strategy yield linear-time algorithms (see the bibliographic notes). When searching for a^m in $(a^{m-1}b)^{\lfloor n/m \rfloor}$ the algorithm makes only $O(n/m)$ comparisons, which is the absolute minimum for any string-matching algorithm in the model where the pattern only is preprocessed.

8.2.4 Quick Search Algorithm

The bad-character shift used in the Boyer–Moore algorithm is not very efficient for small alphabets, but when the alphabet is large compared with the length of the pattern, as it is often the case with

```
Suffixes(x, m)
suff[m − 1] ← m
g ← m − 1
for i ← m − 2 downto 0
    do if i > g and suff[i + m − 1 − f] ≠ i − g
          then suff[i] ← min{suff[i + m − 1 − f], i − g}
          else  if i < g
                   then g ← i
                f ← i
                while g ≥ 0 and x[g] = x[g + m − 1 − f]
                    do g ← g − 1
                suff[i] ← f − g
return suff
```

Figure 8.13: Computation of the table *suff*.

```
PreGS(x, m)
gs ← Suffixes(x, m)
for i ← 0 to m − 1
    do gs[i] ← m
j ← 0
for i ← m − 1 downto −1
    do if i = −1 or suff[i] = i + 1
          then while j < m − 1 − i
                   do if gs[j] = m
                         then gs[j] ← m − 1 − i
                      j ← j + 1
for i ← 0 to m − 2
    do gs[m − 1 − suff[i]] ← m − 1 − i
return gs
```

Figure 8.14: Computation of the good-suffix shift.

```
QS(x, m, y, n)
▷ Preprocessing
for a ← firstLetter to lastLetter
    do bc[a] ← m + 1
for i ← 0 to m − 1
    do bc[x[i]] ← m − i
▷ Searching
j ← 0
while j ≤ n − m
    do i ← 0
       while i ≥ 0 and x[i] = y[i + j]
           do i ← i + 1
       if i ≥ m
         then OUTPUT(j)
       j ← bc[y[j + m]]
```

Figure 8.15: The Quick Search string-matching algorithm.

the ASCII table and ordinary searches made under a text editor, it becomes very useful. Using it alone produces a practically very efficient algorithm that is described now.

After an attempt where x is aligned with $y[j\,..\,j+m-1]$, the length of the shift is at least equal to one. Thus, the character $y[j+m]$ is necessarily involved in the next attempt, and thus can be used for the bad-character shift of the current attempt. In the present algorithm, the bad-character shift is slightly modified to take into account the observation as follows ($a \in \Sigma$):

$$bc[a] = 1 + \begin{cases} \min\{i \mid 0 \le i < m \text{ and } x[m-1-i] = a\} & \text{if } a \text{ appears in } x, \\ m & \text{otherwise.} \end{cases}$$

Indeed, the comparisons between text and pattern characters during each attempt can be done in any order. The algorithm of Fig. 8.15 performs the comparisons from left to right. It is called Quick Search after its inventor and has a quadratic worst-case time complexity but good practical behavior.

Example 8.7: Here

$y =$	s	t	r	i	n	g	-	m	a	t	c	h	i	n	g
$x =$	$\underline{\text{i}}$	n	g												
$x =$				$\underline{\text{i}}$	$\underline{\text{n}}$	$\underline{\text{g}}$									
$x =$								$\underline{\text{i}}$	n	g					
$x =$												$\underline{\text{i}}$	n	g	
$x =$													$\underline{\text{i}}$	$\underline{\text{n}}$	$\underline{\text{g}}$

The Quick Search algorithm makes only nine comparisons to find the two occurrences of **ing** inside the text of length 15.

8.2.5 Experimental Results

In Fig. 8.16 and Fig. 8.17 we present the running times of three string-matching algorithms: the Boyer–Moore algorithm (BM), the Quick Search algorithm (QS), and the Reverse-Factor algorithm (RF). The Reverse–Factor algorithm can be viewed as a variation of the Boyer–Moore algorithm where factors (segments) rather than suffixes of the pattern are recognized. The RF algorithm uses a data structure to store all the factors of the reversed pattern: a suffix automaton or a **suffix tree** (see section 8.4).

Tests have been performed on various types of texts. In Fig. 8.16 we show the results when the text is a DNA sequence on the four-letter alphabet of nucleotides **A**, **C**, **G**, **T**. In Fig. 8.17 English text is considered.

For each pattern length, we ran a large number of searches with random patterns. The average time according to the length is shown in the two figures. The running times of both preprocessing and

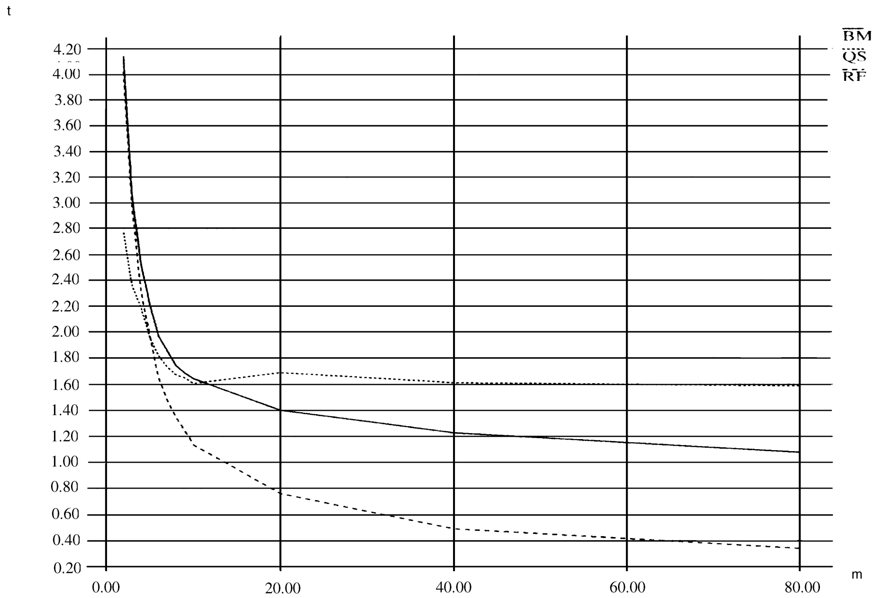


Figure 8.16: Running times for a DNA sequence.

searching phases are added. The three algorithms are implemented in a homogeneous way in order to keep the comparison significant.

For the genome, as expected, the QS algorithm is the best for short patterns. But for long patterns it is less efficient than the BM algorithm. In this latter case, the RF algorithm achieves the best results. For rather large alphabets, as is the case for an English text, the QS algorithm remains better than the BM algorithm whatever the pattern length is. In this case the three algorithms have similar behaviors; however, the QS is better for short patterns (which is typical of search under a text editor) and the RF is better for large patterns.

8.2.6 Aho–Corasick Algorithm

The UNIX operating system provides standard text (or file) facilities. Among them is the series of **grep** commands that locate patterns in files. We describe in this section the algorithm underlying the **fgrep** command of UNIX. It searches files for a finite set of strings, and can, for instance, output lines containing at least one of the strings.

If we are interested in searching for all occurrences of all patterns taken from a finite set of patterns, a first solution consists in repeating some string-matching algorithm for each pattern. If the set contains k patterns, this search runs in time $O(kn)$. The solution described in the present section and designed by Aho and Corasick runs in time $O(n \log \sigma)$. The algorithm is a direct extension of the Knuth–Morris–Pratt algorithm, and the running time is independent of the number of patterns.

Let $X = \{x_0, x_1, \ldots, x_{k-1}\}$ be the set of patterns, and let $|X| = |x_0| + |x_1| + \cdots + |x_{k-1}|$ be the total size of the set X. The Aho–Corasick algorithm first builds a **trie** $T(X)$, a digital tree recognizing the patterns of X. The trie $T(X)$ is a tree in which edges are labeled by letters and in which branches spell the patterns of X. We identify a node p in the trie $T(X)$ with the unique word w spelled by the path of $T(X)$ from its root to p. The root itself is identified with the empty word ε. Notice that if w is a node in $T(X)$ then w is a prefix of some $x_i \in X$. If w is a node in $T(X)$ and $a \in \Sigma$ then *child*(w, a) is equal to wa if wa is a node in $T(X)$; it is equal to UNDEFINED otherwise.

The function PreAC in Fig. 8.18 returns the trie of all patterns. During the second phase, where patterns are entered in the trie, the algorithm initializes an output function *out*. It associates the singleton $\{x_i\}$ with the nodes x_i $(0 \leq i < k)$, and associates the empty set with all other nodes of $T(X)$ (see Fig. 8.19).

Finally, the last phase of function PreAC (Fig. 8.18) consists in building the failure link of each node

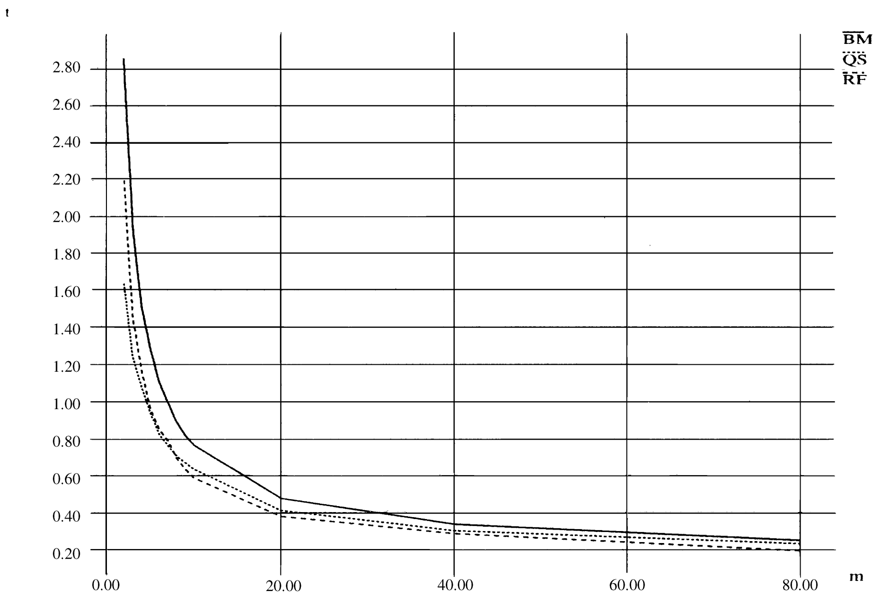


Figure 8.17: Running times for an English text.

```
PreAC(X, k)
  Create a new node root
  ▷ creates a loop on the root of the trie
  for a ∈ Σ
      do child(root, a) ← root
  ▷ enters each pattern in the trie
  for i ← 0 to k − 1
      do Enter(X[i], root)
  ▷ completes the trie with failure links
  Complete(root)
  return root
```

Figure 8.18: Preprocessing phase of the Aho–Corasick algorithm.

```
Enter(x, root)
  r ← root
  i ← 0
  ▷ follows the existing edges
  while i < |x| and child(r, x[i]) ≠ UNDEFINED and child(r, x[i]) ≠ root
      do r ← child(r, x[i])
         i ← i + 1
  ▷ creates new edges
  while i < |x|
      do Create a new node s
         child(r, x[i]) ← s
         r ← s
         i ← i + 1
  out(r) ← {x}
```

Figure 8.19: Construction of the trie.

```
Complete(root)
q ← empty queue
ℓ ← list of the edges (root, a, p) for any character a ∈ Σ and any node p ≠ root
while the list ℓ is not empty
     do (r, a, p) ← First(ℓ)
        ℓ ← Next(ℓ)
        Enqueue(q, p)
        fail(p) ← root
while the queue q is not empty
     do r ← Dequeue(q)
        ℓ ← list of the edges (root, a, p) for any character a ∈ Σ and any node p
        while the list ℓ is not empty
             do (r, a, p) ← First(ℓ)
                ℓ ← Next(ℓ)
                Enqueue(q, p)
                s ← fail(r)
                while child(s, a) = UNDEFINED
                     do s ← fail(s)
                fail(p) ← child(s, a)
                out(p) ← out(p) ∪ out(child(s, a))
```

Figure 8.20: Completion of the output function and construction of failure links.

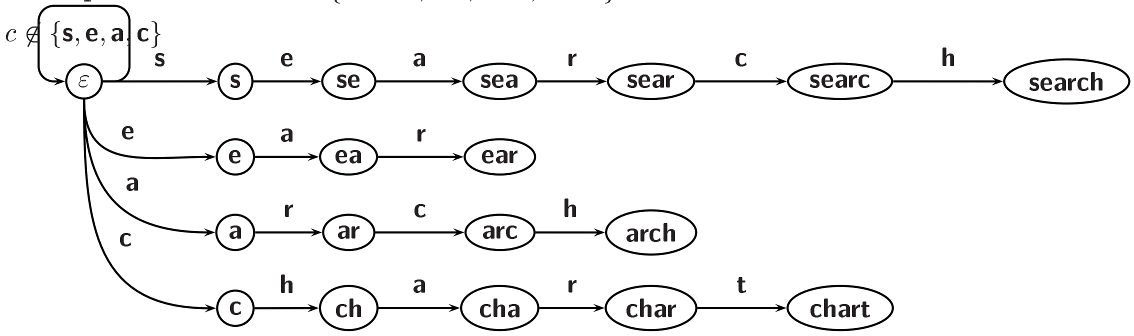
of the trie, and simultaneously completing the output function. This is done by the function Complete in Fig. 8.20. The failure function *fail* is defined on nodes as follows (w is a node):

$$\textit{fail}(w) = u \text{ where } u \text{ is the longest proper suffix of } w \text{ that belongs to } T(X)\,.$$

Computation of failure links is done during a breadth-first traversal of $T(X)$. Completion of the output function is done while computing the failure function *fail* using the following rule:

$$\text{if } \textit{fail}(w) = u \text{ then } \textit{out}(w) = \textit{out}(w) \cup \textit{out}(u)\,.$$

Example 8.8: Here $X = \{\textsf{search}, \textsf{ear}, \textsf{arch}, \textsf{chart}\}$



nodes	ε	s	se	sea	sear	searc	search	e	ea	ear
fail	ε	ε	e	ea	ear	arc	arch	ε	a	ar

nodes	a	ar	arc	arch	c	ch	cha	char	chart	
fail	ε	ε	c	ch	ε	ε	a	ar	ε	

nodes	sear	search	ear	arch	chart
out	ear	{search, arch}	ear	arch	chart

To stop going back with failure links during the computation of the failure links, and also to overpass

```
AC(X, k, y, n)
▷ Preprocessing
r ← PreAC(X, k)
▷ Searching
for j ← 0 to n − 1
     do while child(r, y[j]) = UNDEFINED
             do r ← fail(r)
        r ← child(r, y[j])
        if out(r) ≠ ∅
          then Output((out(r), j))
```

Figure 8.21: The complete Aho–Corasick algorithm.

text characters for which no transition is defined from the root, a loop is added on the root of the trie for these symbols. This is done at the first phase of function PreAC.

After the preprocessing phase is completed, the searching phase consists in parsing all the characters of the text y with $T(X)$. This starts at the root of $T(X)$ and uses failure links whenever a character in y does not match any label of outgoing edges of the current node. Each time a node with a nonempty output is encountered, this means that the patterns of the output have been discovered in the text, ending at the current position. Then, the position is output.

An implementation of the Aho–Corasick algorithm from the previous discussion is shown in Fig. 8.21. Note that the algorithm processes the text in an on-line way, so that the buffer on the text can be limited to only one symbol. Also note that the instruction $r \leftarrow \mathit{fail}(r)$ in Fig. 8.21 is the exact analogue of instruction $i \leftarrow \mathit{next}[i]$ in Fig. 8.5. A unified view of both algorithms exists but is beyond the scope of the chapter.

The entire algorithm runs in time $O(|X| + n)$ if the *child* function is implemented to run in constant time. This is the case for any fixed alphabet. Otherwise a $\log \sigma$ multiplicative factor comes from access to the children nodes.

8.3 Two-Dimensional Pattern Matching Algorithms

In this section we consider only two-dimensional arrays. Arrays may be thought of as bit map representations of images, where each cell of arrays contains the codeword of a pixel. The string-matching problem finds an equivalent formulation in two dimensions (and even in any number of dimensions), and algorithms of section 8.2 can be extended to operate on arrays.

The problem now is to locate all occurrences of a two-dimensional pattern $X = X[0\,..\,m_1-1, 0\,..\,m_2-1]$ of size $m_1 \times m_2$ inside a two-dimensional text $Y = Y[0\,..\,n_1-1, 0\,..\,n_2-1]$ of size $n_1 \times n_2$. The brute force algorithm for this problem is given in Fig. 8.22. It consists in checking at all positions of $Y[0\,..\,n_1 - m_1, 0\,..\,n_2 - m_2]$ if the pattern occurs. This algorithm has a quadratic (with respect to the size of the problem) worst-case time complexity in $O(m_1 m_2 n_1 n_2)$. We present in the next sections two more efficient algorithms. The first one is an extension of the Karp–Rabin algorithm (previous section). The second one solves the problem in linear time on a fixed alphabet; it uses both the Aho–Corasick and the Knuth–Morris–Pratt algorithms.

8.3.1 Zhu–Takaoka Algorithm

As for one-dimensional string matching, it is possible to check if the pattern occurs in the text only if the *aligned* portion of the text looks like the pattern. To do that, the idea is to use vertically the hash function method proposed by Karp and Rabin. To initialize the process, the two-dimensional arrays X and Y are translated into one-dimensional arrays of numbers x and y. The translation from X to x is done as follows ($0 \le i < m_2$):

$$x[i] = \mathit{hash}(X[0,i]X[1,i]\cdots X[m_1-1,i])$$

```
BF2D(X, m1, m2, Y, n1, n2)
▷ Searching
for j1 ← 0 to n1 − m1
     do for j2 ← 0 to n2 − m2
             do i ← 0
                 while i < m1 and x[i, 0 . . m2 − 1] = y[j1 + i, j2 . . j2 + m2 − 1]
                      do i ← i + 1
                 if i ≥ m1
                   then OUTPUT(j1, j2)
```

Figure 8.22: The brute force two-dimensional pattern matching algorithm.

and the translation from Y to y is done by ($0 \leq i < m_2$):

$$y[i] = hash(Y[0, i]Y[1, i] \cdots Y[m_1 - 1, i]).$$

The fingerprint y helps to find occurrences of X starting at row $j = 0$ in Y. It is then updated for each new row in the following way ($0 \leq i < m_2$):

$$\begin{aligned} & hash(Y[j+1, i]Y[j+2, i] \cdots Y[j+m_1, i]) \\ & \quad = \textsc{ReHash}(Y[j, i], Y[j+m_1, i], hash(Y[j, i]Y[j+1, i] \cdots Y[j+m_1-1, i])) \end{aligned}$$

(functions *hash* and ReHash are described in the section on the Karp–Rabin algorithm).

Example 8.9:

$X =$

a	a	a
b	b	a
a	a	b

$Y =$

a	b	a	b	a	b	b
a	a	a	a	b	b	b
b	b	b	a	a	a	b
a	a	a	b	b	a	a
b	b	a	a	a	b	b
a	a	b	a	b	a	a

$x =$

681	681	680

$y =$

680	684	680	683	681	685	686

Next value of y is

681	681	681	680	684	683	685

. The occurrence of x at position 1 on y corresponds to an occurrence of X at position $(1, 1)$ on Y.

Since the alphabet of x and y is large, searching for x in y must be done by a string-matching algorithm for which the running time is independent of the size of the alphabet: the Knuth–Morris–Pratt suits this application perfectly. Its adaptation is shown in Fig. 8.23.

When an occurrence of x is found in y, then we still have to check if an occurrence of X starts in Y at the corresponding position. This is done naively by the procedure of Fig. 8.24.

The Zhu–Takaoka algorithm as explained above is displayed in Fig. 8.25. The search for the pattern is performed row by row starting at row 0 and ending at row $n_1 - m_1$.

8.3.2 Bird/Baker Algorithm

The algorithm designed independently by Bird and Baker for the two-dimensional pattern matching problem combines the use of the Aho–Corasick algorithm and the Knuth–Morris–Pratt (KMP) algorithm. The pattern X is divided into its m_1 rows $R_0 = X[0, 0 \,.\,.\, m_2 - 1]$ to $R_{m_1-1} = x[m_1 - 1, 0 \,.\,.\, m_2 - 1]$. The rows are preprocessed into a trie as in the Aho–Corasick algorithm described earlier.

```
KMP-IN-LINE(X, m_1, m_2, Y, n_1, n_2, x, y, next, j_1)
i_2 ← 0
j_2 ← 0
while j_2 < n_2
    do while i_2 > −1 and x[i_2] ≠ y[j_2]
           do i_2 ← next[i_2]
       i_2 ← i_2 + 1
       j_2 ← j_2 + 1
       if i_2 ≥ m_2
         then DIRECT-COMPARE(X, m_1, m_2, Y, n_1, n_2, j_1, j_2 − 1)
              i_2 ← next[m_2]
```

Figure 8.23: Search for x in y using KMP algorithm.

```
DIRECT-COMPARE(X, m_1, m_2, Y, row, column)
j_1 ← row − m_1 + 1
j_2 ← column − m_2 + 1
for i_1 ← 0 to m_1 − 1
    do for i_2 ← 0 to m_2 − 1
           do if X[i_1, i_2] ≠ Y[i_1 + j_1, i_2 + j_2]
                then return
OUTPUT(j_1, j_2)
```

Figure 8.24: Naive check of an occurrence of x in y at position (*row, column*).

```
ZT(X, m_1, m_2, Y, n_1, n_2)
▷ Preprocessing
▷ Computes x
for i_2 ← 0 to m_2 − 1
    do x[i_2] ← 0
       for i_1 ← 0 to m_1 − 1
           do x[i_2] ← (x[i_2] << 1) + X[i_1, i_2]
▷ Computes the first value of y
for j_2 ← 0 to n_2 − 1
    do y[j_2] ← 0
       for j_1 ← 0 to m_1 − 1
           do y[j_2] ← (y[j_2] << 1) + Y[j_1, j_2]
d ← 1
for i ← 1 to m_1 − 1
    do d ← d << 1
next ← PREKMP(X′, m2)
▷ Searching
j_1 ← m_1 − 1
while j_1 < n_1
    do KMP-IN-LINE(X, m_1, m_2, Y, n_1, n_2, x, y, next, j_2)
       if j_1 < n_1 − 1
         then for j_2 ← 0 to n_2 − 1
                  do y[j_2] ← REHASH(Y[j_1 − m_1 + 1, j_2], Y[j_1 + 1, j_2], y[j_2])
       j_1 ← j_1 + 1
```

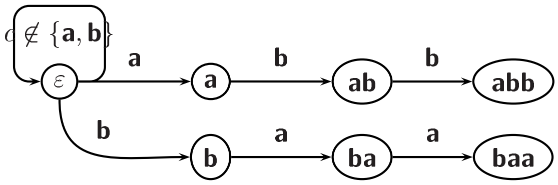
Figure 8.25: The Zhu-Takaoka two-dimensional pattern matching algorithm.

```
Pre-KMP-for-B(X, m1, m2)
i ← 0
next[0] ← −1
j ← −1
while i < m1
     do while j > −1 and X[i, 0 . . m2 − 1] ≠ X[j, 0 . . m2 − 1]
             do j ← next[j]
        i ← i + 1
        j ← j + 1
        if X[i, 0 . . m2 − 1] ≠ X[j, 0 . . m2 − 1]
          then next[i] ← next[j]
          else next[i] ← j
return next
```

Figure 8.26: Computes the function *next* for rows of X.

Example 8.10: Pattern X and the trie of its rows:

$$X = \begin{array}{|c|c|c|}\hline \mathsf{b} & \mathsf{a} & \mathsf{a} \\ \hline \mathsf{a} & \mathsf{b} & \mathsf{b} \\ \hline \mathsf{b} & \mathsf{a} & \mathsf{a} \\ \hline \end{array}$$



The search proceeds as follows. The text is read from the upper left corner to the bottom right corner, row by row. When reading the character $Y[j_1, j_2]$ the algorithm checks whether the portion $Y[j_1, j_2 - m_2 + 1 \,..\, j_2] = R$ matches any of $R_0, \ldots, R_{m_1-1}$ using the Aho–Corasick machine. An additional one-dimensional array a of size n_1 is used as follows: $a[j_2] = k$ means that the $k-1$ first rows $R_0, \ldots, R_{k-2}$ of the pattern match, respectively, the portions of the text: $Y[j_1 - k + 1, j_2 - m_2 + 1 \,..\, j_2], \ldots, Y[j_1 - 1, j_2 - m_2 + 1 \,..\, j_2]$. Then, if $R = R_{k-1}$, $a[j_2]$ is incremented to $k + 1$. If not, $a[j_2]$ is set to $s + 1$ where s is the maximum i such that

$$R_0 \cdots R_i = R_{k-s+1} \cdots R_{k-2} R\,.$$

The value s is computed using the KMP algorithm vertically (in columns). If there exists no such s, $a[j_2]$ is set to 0. Finally, if at some point $a[j_2] = m_1$ an occurrence of the pattern appears at position $(j_1 - m_1 + 1, j_2 - m_2 + 1)$ in the text.

The Bird/Baker algorithm is presented in Figs. 8.26 and 8.27. It runs in time $O((n_1 n_2 + m_1 m_2) \log \sigma)$.

8.4 Suffix Trees

The suffix tree $S(y)$ of a string y is a trie (as described earlier) containing all the suffixes of the string, and having the properties described subsequently. This data structure serves as an index on the string: it provides a direct access to all segments of the string, and gives the positions of all their occurrences in the string.

Once the suffix tree of a text y is built, searching for x in y remains to spell x along a branch of the tree. If this walk is successful the positions of the pattern can be output. Otherwise, x does not occur in y.

Any kind of trie that represents the suffixes of a string can be used to search it. But the suffix tree has additional features which imply that its size is linear. The suffix tree of y is defined by the following properties:

```
B(X, m1, m2, Y, n1, n2)
▷ Preprocessing
for i ← 0 to m2 − 1
     do a[i] ← 0
root ← PreAC(m1)
next ← Pre-KMP-for-B(X, m1, m2)
for j1 ← 0 to n1 − 1
     do r ← root
        for j2 ← 0 to n2 − 1
            do while child(r, Y[j1, j2]) = undefined
                   do r ← fail(r)
               r ← child(r, Y[j1, j2])
               if out(r) ≠ ∅
                 then k ← a[j2]
                      while k > 0 and X[k, 0 . . m2 − 1] = out(r)
                          do k ← next[k]
                      a[j2] ← k + 1
                      if k ≥ m1 − 1
                        then Output(j1 − m1 + 1, j2 − m2 + 1)
                 else a[j2] ← 0
```

Figure 8.27: The Bird/Baker two-dimensional pattern matching algorithm.

```
Suffix-tree(y, n)
T_{−1} ← one node tree
for j ← 0 to n − 1
     do T_j ← Insert(T_{j−1}, y[j . . n − 1])
return T_{n−1}
```

Figure 8.28: Construction of a suffix tree for y.

- All branches of $S(y)$ are labeled by all suffixes of y.
- Edges of $S(y)$ are labeled by strings.
- Internal nodes of $S(y)$ have at least two children (when y is not empty).
- Edges outgoing an internal node are labeled by segments starting with different letters.
- The preceding segments are represented by their starting positions on y and their lengths.

Moreover, it is assumed that y ends with a symbol occurring nowhere else in it (the dollar sign is used in examples). This avoids marking nodes, and implies that $S(y)$ has exactly n leaves (number of nonempty suffixes). The other properties then imply that the total size of $S(y)$ is $O(n)$, which makes it possible to design a linear-time construction of the trie. The algorithm described in the present section has this time complexity provided the alphabet is fixed, or with an additional multiplicative factor $\log \sigma$ otherwise.

The algorithm inserts all nonempty suffixes of y in the data structure from the longest to the shortest suffix, as shown in Fig. 8.28. We introduce two definitions to explain how the algorithm works:

- $head_j$ is the longest prefix of $y[j\,..\,n-1]$ which is also a prefix of $y[i\,..\,n-1]$ for some $i < j$.
- $tail_j$ is the word such that $y[j\,..\,n-1] = head_j tail_j$.

The strategy to insert the ith suffix in the tree is based on these definitions and described in Fig. 8.29.

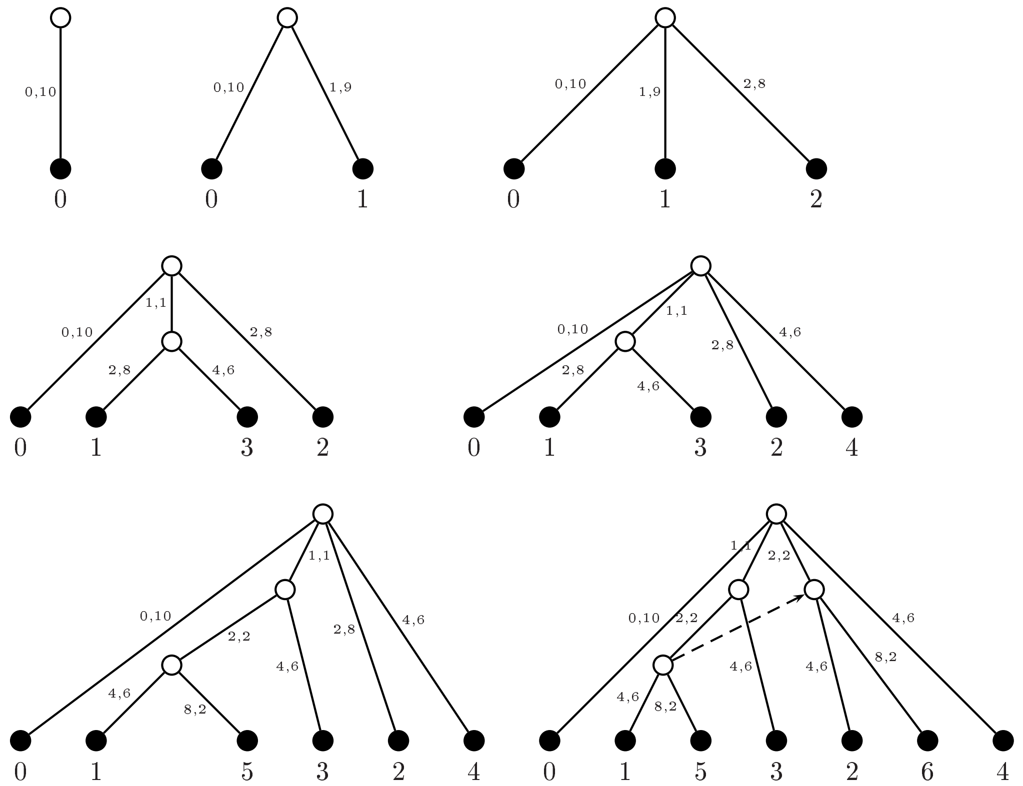
The second step of the insertion (Fig. 8.29) is clearly performed in constant time. Thus, finding the node h is critical for the overall performance of the algorithm. A brute-force method to find it consists

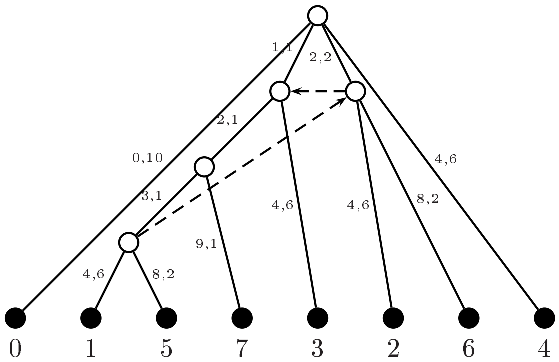
INSERT($T_{j-1}, y[j \,..\, n-1]$)
1 locate the node h associated with $head_j$ in T_{j-1}, possibly breaking an edge
2 add a new edge labeled $tail_j$ from h to a new leaf representing $y[j \,..\, n-1]$
3 **return** the modified tree

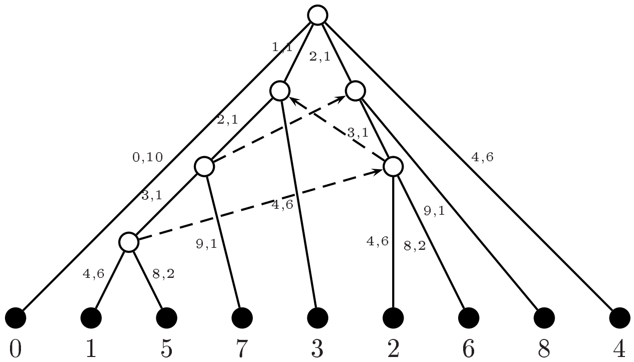
Figure 8.29: Insertion of a new suffix in the tree.

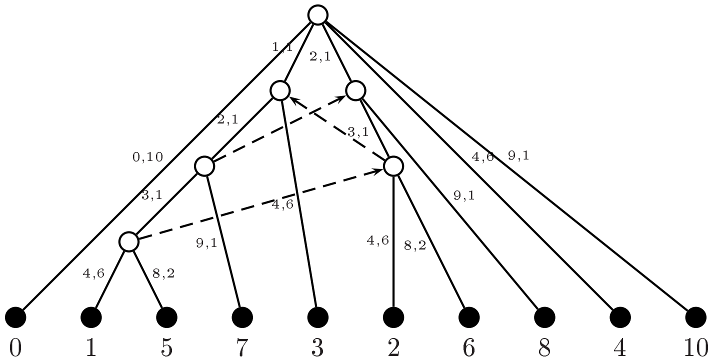
in spelling the current suffix $y[j \,..\, n-1]$ from the root of the tree, giving an $O(|head_j|)$ time complexity for the insertion at step j, and an $O(n^2)$ running time to build $S(y)$. Adding short-cut links leads to an overall $O(n)$ time complexity, although there is no guarantee that insertion at step j is realized in constant time.

Example 8.11: The different tries during the construction of the suffix tree of y = **CAGATAGAG**. Leaves are black and labeled by the position of the suffix they represent. Plain arrows are labeled by pairs: the pair (j, ℓ) stands for the segment $y[j \,..\, j+\ell-1]$. Dashed arrows represent the nontrivial suffix links.









8.4.1 McCreight Algorithm

The key to get an efficient construction of the suffix tree $S(y)$ is to add links between nodes of the tree: they are called *suffix links*. Their definition relies on the relationship between $head_{j-1}$ and $head_j$: if $head_{j-1}$ is of the form az ($a \in \Sigma$, $z \in \Sigma^*$), then z is a prefix of $head_j$. In the suffix tree the node associated with z is linked to the node associated with az. The suffix link creates a shortcut in the tree that helps with finding the next head efficiently. The insertion of the next suffix, namely, $head_j tail_j$, in the tree reduces to the insertion of $tail_j$ from the node associated with $head_j$.

The following property is an invariant of the construction: in T_j, only the node h associated with $head_j$ can fail to have a valid suffix link. This effectively happens when h has just been created at step j. The procedure to find the next head at step j is composed of two main phases:

A Rescanning: Assume that $head_{j-1} = az$ ($a \in \Sigma$, $z \in \Sigma^*$) and let d' be the associated node. If the suffix link on d' is defined, it leads to a node d from which the second step starts. Otherwise, the suffix link on d' is found by rescanning as follows. Let c' be the parent of d', and let (j, ℓ) be the label of edge (c', d'). For the ease of the description, assume that $az = av(y[j \,..\, j+\ell-1])$ (it may

```
M(y, n)
root ← Init(y, n)
head ← root
tail ← child(root, y[0])
n ← n − 1
while n > 0
      do if head = root                                    ▷ Phase A (rescanning)
            then d ← root
                  (j, ℓ) ← label(tail)
                  γ ← (j + 1, ℓ − 1)
            else  γ ← label(tail)
                  if link(head) ≠ UNDEFINED
                     then d ← link(head)
                     else  (j, ℓ) ← label(head)
                           if parent(head) = root
                              then d ← Rescan(root, j + 1, ℓ − 1))
                              else  d ← Rescan(link(parent(head)), j, ℓ))
                           link(head) ← d
         (head, γ) ← Scan(d, γ)                            ▷ Phase B (scanning)
         create a new node tail
         parent(tail) ← head
         label(tail) ← γ
         (j, ℓ) ← γ
         child(head, y[j]) ← tail
         n ← n − 1
return root
```

Figure 8.30: Suffix tree construction.

```
INIT(y, n)
create a new node root
create a new node c
parent(root) ← UNDEFINED
parent(c) ← root
child(root, y[0]) ← c
label(root) ← UNDEFINED
label(c) ← (0, n)
return root
```

Figure 8.31: Initialization procedure.

```
RESCAN(c, j, ℓ)
(k, m) ← label(child(c, y[j]))
while ℓ > 0 and ℓ ≥ m
     do c ← child(c, y[j])
        ℓ ← ℓ − m
        j ← j + m
        (k, m) ← label(child(c, y[j]))
if ℓ > 0
   then return BREAK-EDGE(child(c, y[j]), ℓ)
   else return c
```

Figure 8.32: The crucial rescan operation.

happen that $az = y[j \,..\, j + \ell - 1]$). There is a suffix link defined on c' and going to some node c associated with v. The crucial observation here is that $y[j \,..\, j + \ell - 1]$ is the prefix of the label of some branch starting at node c. Then, the algorithm rescans $y[j \,..\, j + \ell - 1]$ in the tree: let e be the child of c along that branch, and let (k, m) be the label of edge (c, e). If $m < \ell$, then a recursive rescan of $q = y[j + m \,..\, j + \ell - 1]$ starts from node e. If $m > \ell$, the edge (c, e) is broken to insert a new node d; labels are updated correspondingly. If $m = \ell$, d is simply set to e. If the suffix link of d' is currently undefined, it is set to d.

B Scanning: A downward search starts from d to find the node h associated with $head_j$. The search is dictated by the characters of $tail_{j-1}$ one at a time from left to right. If necessary a new internal node is created at the end of the scanning.

After the two phases A and B are executed, the node associated with the new head is known, and the tail of the current suffix can be inserted in the tree.

To analyze the time complexity of the entire algorithm we mainly have to evaluate the total time of all scannings, and the total time of all rescannings. We assume that the alphabet is fixed, so that branching from a node to one of its children can be implemented to take constant time. Thus, the time spent for all scannings is linear because each letter of y is scanned only once. The same holds true for rescannings because each step downward (through node e) increases strictly the position of the segment of y considered there, and this position never decreases.

An implementation of McCreight's algorithm is shown in Fig. 8.30. The next figures (Figs. 8.31–8.34) give the procedures used by the algorithm, especially procedures RESCAN and SCAN.

We use the following notation:

- *parent*(c) is the parent node of the node c,
- *label*(c) is the pair (i, l) if the edge from the parent node of c to c itself is associated with the factor $y[i \,..\, i + l - 1]$,
- *child*(c, a) is the only node that can be reached from the node c with the character a,

```
Break-edge(c, k)
create a new node g
parent(g) ← parent(c)
(j, ℓ) ← label(c)
child(parent(c), y[j]) ← g
label(g) ← (j, k)
parent(c) ← g
label(c) ← (j + k, ℓ − k)
child(g, y[j + k]) ← c
link(g) ← UNDEFINED
return g
```

Figure 8.33: Breaking an edge.

```
Scan(d, γ)
(j, ℓ) ← γ
while child(d, y[j]) ≠ UNDEFINED
      do g ← child(d, y[j])
         k ← 1
         (s, lg) ← label(g)
         s ← s + 1
         ℓ ← ℓ − 1
         j ← j + 1
         while k < lg and y[j] = y[s]
             do j ← j + 1
                s ← s + 1
                k ← k + 1
                ℓ ← ℓ − 1
         if k < lg
           then return (Break-edge(g, k), (j, ℓ))
         d ← g
return (d, (j, ℓ))
```

Figure 8.34: The scan operation.

- *link*(c) is the suffix node of the node c.

8.5 Alignment

Alignments are used to compare strings. They are widely used in computational molecular biology. They constitute a mean to visualize resemblance between strings. They are based on notions of distance or similarity. Their computation is usually done by dynamic programming. A typical example of this method is the computation of the longest common subsequence of two strings. The reduction of the memory space presented on it can be applied to similar problems. We consider three different kinds of alignment of two strings x and y: global alignment (that consider the whole strings x and y), local alignment (that enable to find the segment of x that is closer to a segment of y) and the longest common subsequence of x and y.

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 $x, y \in \Sigma$ is a word w on the alphabet $(\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\}) \setminus (\{(\varepsilon, \varepsilon)\}$ (ε is the empty word) whose projection on the first component is x and whose projection of the second component is y.

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

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

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

Example 8.12:

$$\begin{array}{cccccc}\mathsf{A} & \mathsf{C} & \mathsf{G} & - & - & \mathsf{A} \\ \mathsf{A} & \mathsf{T} & \mathsf{G} & \mathsf{C} & \mathsf{T} & \mathsf{A}\end{array}$$

is an alignment of **ACGA** and **ATGCTA**.

8.5.1 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 than 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 to each elementary edit operation. For $a, b \in \Sigma$:

- $\mathit{Sub}(a, b)$ denotes the cost of the substitution of the character a by the character b,
- $\mathit{Del}(a)$ denotes the cost of the deletion of the character a,
- $\mathit{Ins}(a)$ 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

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

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 $\mathit{edit}(x, y)$ 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

$$T[i, j] = \mathit{edit}(x[i], y[j])$$

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

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

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

GENERIC-DP$(x, m, y, n, \text{MARGIN}, \text{FORMULA})$

MARGIN(T, x, m, y, n)
for $j \leftarrow 0$ **to** $n-1$
  do for $i \leftarrow 0$ **to** $m-1$
    do $T[i, j] \leftarrow$ FORMULA(T, x, i, y, j)
return T

Figure 8.35: Computation of the table T by dynamic programming.

MARGIN-GLOBAL(T, x, m, y, n)

$T[-1, -1] \leftarrow 0$
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 $T[-1, j] \leftarrow T[-1, j-1] + Ins(y[j])$

Figure 8.36: Margin initialization for the computation of a global alignment.

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

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 a parameter 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 a parameter of the algorithm and that depends on the kind of alignment that is considered. Computing a global alignment of x and y can be done by a call to GENERIC-DP with the following parameters $(x, m, y, n, \text{MARGIN-GLOBAL}, \text{FORMULA-GLOBAL})$ (see Fig. 8.35, 8.36 and 8.37). The computation of all the values of the table T can thus be done in quadratic space and time: $O(m \times n)$.

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. 8.38). It consists in tracing back the computation of the values of the table T from 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.

Example 8.13:

FORMULA-GLOBAL(T, x, i, y, j)

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

Figure 8.37: Computation of $T[i, j]$ for a global alignment.

```
ONE-ALIGNMENT(T, x, i, y, j)
if i = −1 and j = −1
   then return (ε, ε)
   else if i = −1
           then return ONE-ALIGNMENT(T, x, −1, y, j − 1) · (ε, y[j])
         elseif j = −1
           then return ONE-ALIGNMENT(T, x, i − 1, y, −1) · (x[i], ε)
         else    if T[i, j] = T[i − 1, j − 1] + Sub(x[i], y[j])
                    then return ONE-ALIGNMENT(T, x, i − 1, y, j − 1) · (x[i], y[j])
                 elseif T[i, j] = T[i − 1, j] + Del(x[i])
                    then return ONE-ALIGNMENT(T, x, i − 1, y, j) · (x[i], ε)
                 else      return ONE-ALIGNMENT(T, x, i, y, j − 1) · (ε, y[j])
```

Figure 8.38: Recovering an optimal alignment.

T	j	-1	0	1	2	3	4	5
i		$y[j]$	**A**	**T**	**G**	**C**	**T**	**A**
-1	$x[i]$	$\boxed{0}$	1	2	3	4	5	6
0	**A**	1	$\boxed{0}$	$\boxed{1}$	$\boxed{2}$	3	4	5
1	**C**	2	1	1	2	$\boxed{2}$	3	4
2	**G**	3	2	2	1	2	$\boxed{3}$	4
3	**A**	4	3	3	2	2	3	$\boxed{3}$

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 $a, b \in \Sigma$.

8.5.2 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.

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

- $Sub_S(a, b)$ 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.

In order to compute $sco(x, y)$ 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

$$T[i, j] = sco(x[i], y[j])$$

for $i = 0, \ldots, m - 1$ and $j = 0, \ldots, n - 1$. Therefore $sco(x, y) = T[m - 1, n - 1]$.

```
MARGIN-LOCAL(T, x, m, y, n)
T[-1, -1] ← 0
for i ← 0 to m - 1
      do T[i, -1] ← 0
for j ← 0 to n - 1
      do T[-1, j] ← 0
```

Figure 8.39: Margin initialization for computing a local alignment.

FORMULA-LOCAL(T, x, i, y, j)

return $\max \begin{cases} T[i-1, j-1] + Sub_S(x[i], y[j]) \\ T[i-1, j] + Del_S(x[i]) \\ T[i, j-1] + Ins_S(y[j]) \\ 0 \end{cases}$

Figure 8.40: Recurrence formula for computing a local alignment.

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

$$\begin{aligned} T[-1,-1] &= 0\,, \\ T[i,-1] &= 0\,, \\ T[-1,j] &= 0\,, \\ T[i,j] &= \max \begin{cases} T[i-1, j-1] + Sub_S(x[i], y[j])\,, \\ T[i-1, j] + Del_S(x[i])\,, \\ T[i, j-1] + Ins_S(y[j])\,, \\ 0\,, \end{cases} \end{aligned}$$

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

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 parameters $(x, m, y, n,$ MARGIN-LOCAL, FORMULA-LOCAL$)$ in $O(mn)$ time and space complexity (see Fig. 8.35, 8.39 and 8.40). Recovering a local alignment can be done in a way similar to what is done in the case of a global alignment (see Fig. 8.38) but the trace back procedure must start at a position of a maximal value in T rather than at position $[m-1, n-1]$.

Example 8.14: Computation of an optimal local alignment of
$x =$ **EAWACQGKL** and $y =$ **ERDAWCQPGKWY** with scores:
$Sub_S(a, a) = 1$, $Sub_S(a, b) = -3$ and $Del_S(a) = Ins_S(a) = -1$ for $a, b \in \Sigma$, $a \neq b$.

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	$\boxed{0}$	0	0	0	0	0	0	0	0	0
1	**A**	0	0	0	0	$\boxed{1}$	0	0	0	0	0	0	0	0
2	**W**	0	0	0	0	0	$\boxed{2}$	1	0	0	0	0	1	0
3	**A**	0	0	0	0	1	$\boxed{1}$	0	0	0	0	0	0	0
4	**C**	0	0	0	0	0	0	$\boxed{2}$	1	0	0	0	0	0
5	**Q**	0	0	0	0	0	0	1	$\boxed{3}$	$\boxed{2}$	1	0	0	0
6	**G**	0	0	0	0	0	0	0	2	1	$\boxed{3}$	2	1	0
7	**K**	0	0	0	0	0	0	0	1	0	2	$\boxed{4}$	3	2
8	**L**	0	0	0	0	0	0	0	0	0	1	3	2	1

The corresponding optimal local alignment is:

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

Figure 8.41: Recurrence formula for computing an *lcs*.

A	**W**	**A**	**C**	**Q**	**-**	**G**	**K**
A	**W**	**-**	**C**	**Q**	**P**	**G**	**K**

8.5.3 Longest Common Subsequence of Two Strings

A subsequence of a word 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 exists an increasing sequence of integers $(k_j \mid j = 0, \ldots, i-1)$ such that for $0 \leq j \leq i-1$, $w[j] = x[k_j]$. We say that a word is an lcs(x, y) if it is a **longest common subsequence** of the two words x and y. Note that two strings can have several longest common subsequence. 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 one. The word x of length m has 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 $i = 0, 1, \ldots, m-1$ and $j = 0, 1, \ldots, n-1$. Then $T[i, j] = \text{llcs}(x[0 \,.\,. i], y[0 \,.\,. j])$ and $\text{llcs}(x, y) = T[m-1, n-1]$.

Computing $T[m-1, n-1]$ can be done by a call to GENERIC-DP with the following parameters $(x, m, y, n, \text{MARGIN-LOCAL}, \text{FORMULA-LCS})$ in $O(mn)$ time and space complexity (see Fig. 8.35, 8.39 and 8.41).

It is possible afterward to trace back a path from position $[m-1, n-1]$ in order to exhibit an lcs(x, y) in a similar way as for producing a global alignment (see Fig. 8.38).

Example 8.15: 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.

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

```
LLCS(x, m, y, n)
for i ← −1 to m − 1
      do C[i] ← 0
for j ← 0 to n − 1
      do last ← 0
          for i ← −1 to m − 1
                do if last > C[i]
                        then C[i] ← last
                     elseif last < C[i]
                        then last ← C[i]
                     elseif x[i] = y[j]
                        then C[i] ← C[i] + 1
                                 last ← last + 1
return C
```

Figure 8.42: $O(m)$-space algorithm to compute $\mathrm{llcs}(x, y)$.

```
HIRSCHBERG(x, m, y, n)
if m = 0
   then return ε
   else  if m = 1
              then if x[0] ∈ y
                        then return x[0]
                        else  return ε
              else  j ← ⌊n/2⌋
                     C ← LLCS(x, m, y[0 . . j − 1], j)
                     C* ← LLCS(x^R, m, y[j . . n − 1]^R, n − j)
                     k ← m − 1
                     M ← C[m − 1] + C*[m − 1]
                     for j ← −1 to m − 2
                           do if C[j] + C*[j] > M
                                   then M ← C[j] + C*[j]
                                            k ← j
                     return HIRSCHBERG(x[0 . . k − 1], k, y[0 . . j − 1], j)·
                                HIRSCHBERG(x[k . . m − 1], m − k, y[j . . n − 1], n − j)
```

Figure 8.43: $O(\min(m, n))$-space computation of $\mathrm{lcs}(x, y)$.

8.5.4 Reducing the Space: Hirschberg Algorithm

If only the length of an $\mathrm{lcs}(x, y)$ 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. 8.42. Indeed, the Hirschberg algorithm computes an $\mathrm{lcs}(x, y)$ in linear space and not only the value $\mathrm{llcs}(x, y)$. The computation uses the algorithm of Fig. 8.42.

Let us define

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

and

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

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

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

In the algorithm shown in Fig. 8.43 the integer j is chosen as $n/2$. After $T[i, j-1]$ and $T^*[m-i, n-j]$ ($0 \le i < m$) 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 $\mathrm{lcs}(x[0\,..\,k-1], y[0\,..\,j-1])$ and an $\mathrm{lcs}(x[k\,..\,m-1], y[j\,..\,n-1])$, and concatenates them to get an $\mathrm{lcs}(x, y)$.

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.

8.6 Approximate String Matching

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. We consider two distances in this section, the **Hamming distance** and the **Levenshtein distance**.

With the Hamming distance, the problem is also known as approximate string matching with k mismatches. With the Levenshtein distance (or edit distance), the problem is known as approximate string matching with k differences.

The Hamming distance between two words w_1 and w_2 of the same length is the number of positions with different characters. The Levenshtein distance between two words w_1 and w_2 (not necessarily of the same length) is the minimal number of differences between the two words. A difference is one of the following operations:

- A substitution: a character of w_1 corresponds to a different character in w_2.
- An insertion: a character of w_1 corresponds to no character in w_2.
- A deletion: a character of w_2 corresponds to no character in w_1.

The *Shift-Or algorithm* of the next section is a method that is both very fast in practice and very easy to implement. It solves the Hamming distance and the Levenshtein distance problems. We initially describe the method for the exact string-matching problem and then we show how it can handle the cases of k mismatches and of k differences. The method is flexible enough to be adapted to a wide range of similar approximate matching problems.

8.6.1 Shift-Or Algorithm

We first present an algorithm to solve the exact string-matching problem using a technique different from those developed in section 8.2, but which extends readily to the approximate string-matching problem.

Let $\mathbf{R}^0$ be a bit array of size m. Vector $\mathbf{R}^0_j$ is the value of the entire array $\mathbf{R}^0$ after text character $y[j]$ has been processed (see Fig. 8.44). It contains information about all matches of prefixes of x that end at position j in the text. It is defined, for $0 \le i \le m-1$, by

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

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

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

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

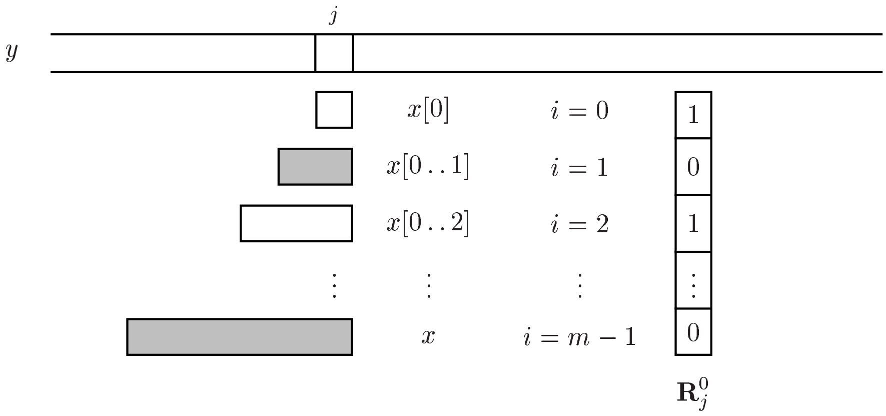


Figure 8.44: Meaning of vector $\mathbf{R}^0_j$.

and

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

The transition from $\mathbf{R}^0_{j-1}$ to $\mathbf{R}^0_j$ can be computed very fast as follows. For each $a \in \Sigma$, let S_a be a bit array of size m defined, for $0 \le i \le m-1$, 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 $\mathbf{R}^0_j$ reduces to two operations, SHIFT and OR:

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

Example 8.16: String $x =$ **GATAA** occurs at position 2 in $y =$ **CAGATAAGAGAA**.

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	1	1	1	0	1	0	1	1
A	1	1	1	0	1	1	1	1	0	1	0	1
T	1	1	1	1	0	1	1	1	1	1	1	1
A	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

8.6.2 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 $\mathbf{R}^1_{j-1}$ 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.

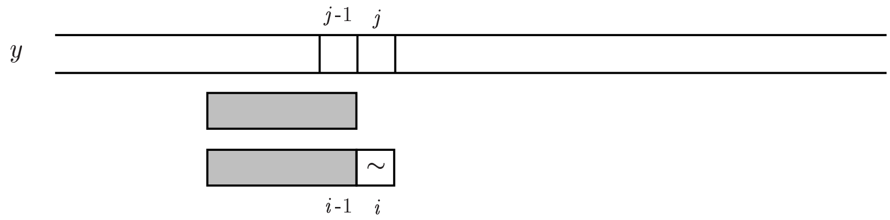


Figure 8.45: If $\mathbf{R}^0_{j-1}[i-1] = 0$ then $\mathbf{R}^1_j[i] = 0$.

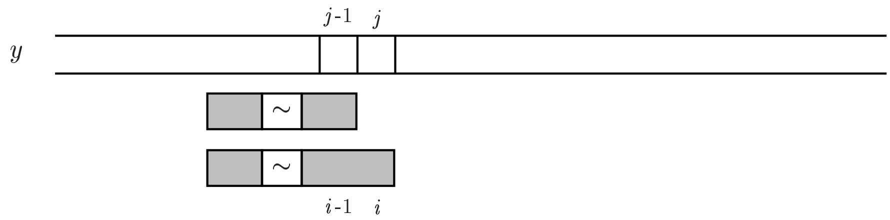


Figure 8.46: $\mathbf{R}^1_j[i] = \mathbf{R}^1_{j-1}[i-1]$ if $x[i] = y[j]$.

- There is an exact match on the first i characters of x up to $y[j-1]$ (i.e., $\mathbf{R}^0_{j-1}[i-1] = 0$). Then, substituting $x[i]$ to $y[j]$ creates a match with one substitution (see Fig. 8.45). Thus,
$$\mathbf{R}^1_j[i] = \mathbf{R}^0_{j-1}[i-1]\,.$$
- 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. 8.46). Thus,
$$\mathbf{R}^1_j[i] = \begin{cases} \mathbf{R}^1_{j-1}[i-1] & \text{if } x[i] = y[j], \\ 1 & \text{otherwise.} \end{cases}$$

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

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

Example 8.17: String x = **GATAA** occurs at positions 2 and 7 in y = **CAGATAAGAGAA** with no more than one mismatch.

	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	1	0	1	0	0	1	0	1	0	0
T	1	1	1	1	0	1	1	1	1	0	1	0
A	1	1	1	1	1	0	1	1	1	1	0	1
A	1	1	1	1	1	1	0	1	1	1	1	0

8.6.3 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 $\mathbf{R}^1_{j-1}$ indicates all matches with at most one insertion up to text character $y[j-1]$. $\mathbf{R}^1_{j-1}[i-1] = 0$ if the first i characters of x ($x[0\,..\,i-1]$) match i symbols of

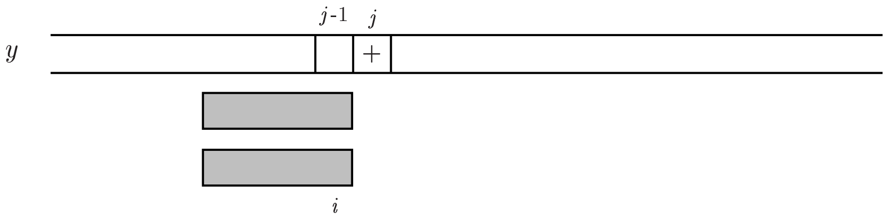


Figure 8.47: If $\mathbf{R}^0_{j-1}[i] = 0$ then $\mathbf{R}^1_j[i] = 0$.

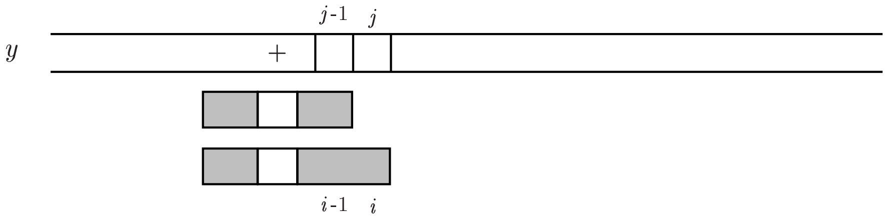


Figure 8.48: $\mathbf{R}^1_j[i] = \mathbf{R}^1_{j-1}[i-1]$ if $x[i] = y[j]$.

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

This shows that $\mathbf{R}^1_j$ can be updated from $\mathbf{R}^1_{j-1}$ with the formula
$$\mathbf{R}^1_j = \big(\text{SHIFT}\big(\mathbf{R}^1_{j-1}\big) \quad \text{OR} \quad S_{y[j]}\big) \quad \text{AND} \quad \mathbf{R}^0_{j-1}\,.$$

Example 8.18: Here **GATAAG** is an occurrence of $x =$ **GATAA** with exactly one insertion in $y =$ **CAGATAAGAGAA**

	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

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

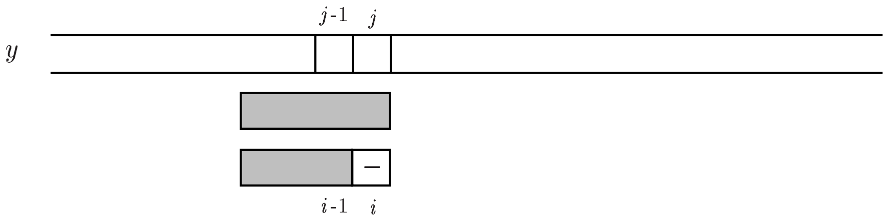


Figure 8.49: If $\mathbf{R}^0_j[i] = 0$ then $\mathbf{R}^1_j[i] = 0$.

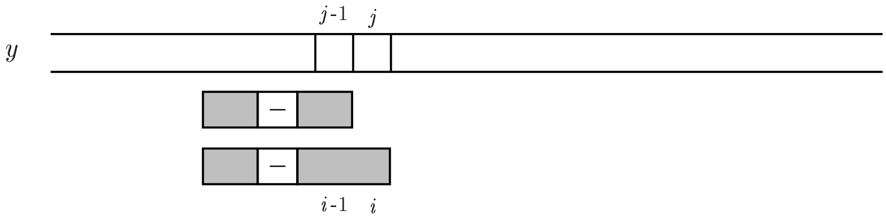


Figure 8.50: $\mathbf{R}^1_j[i] = \mathbf{R}^1_{j-1}[i-1]$ if $x[i] = y[j]$.

- There is an exact match on the first $i+1$ characters of x ($x[0\,..\,i]$) up to $y[j]$ (i.e., $\mathbf{R}^0_j[i] = 0$). Then, deleting $x[i]$ creates a match with one deletion (see Fig. 8.49). Thus,

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

- 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. 8.49). Thus,

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

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

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

Example 8.19: **GATA** and **ATAA** are two occurrences with one deletion of x = **GATAA** in y = **CAGATAAGAGAA**

	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

8.6.4 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 $\mathbf{R}^0, \mathbf{R}^1, \ldots, \mathbf{R}^k$ that are described now. The

```
WM(x, m, y, n, k)
for each character a ∈ Σ
     do S_a ← 1^m
for i ← 0 to m − 1
     do S_{x[i]}[i] ← 0
R^0 ← 1^m
for ℓ ← 1 to k
     do R^ℓ ← SHIFT(R^{ℓ−1})
for j ← 0 to n − 1
     do T ← R^0
        R^0 ← SHIFT(R^0)  OR  S_{y[j]}
        for ℓ ← 1 to k
            do T' ← R^ℓ
               R^ℓ ← (SHIFT(R^ℓ)  OR  S_{y[j]})  AND  (SHIFT((T  AND  R^{ℓ−1}))  AND  T
               T ← T'
        if R^k[m − 1] = 0
          then OUTPUT(j)
```

Figure 8.51: Wu–Manber approximate string-matching algorithm.

vector $\mathbf{R}^0$ is maintained similarly as in the exact matching case (section "Shift-Or Algorithm"). The other vectors are computed with the formula ($1 \le \ell \le k$)

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

which can be rewritten into

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

Example 8.20: Here $x =$ **GATAA** and $y =$ **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.

	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

The method, called the Wu–Manber algorithm, is implemented in Fig. 8.51. 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 $O(kn)$.

8.7 Text Compression

In this section we are interested in algorithms that compress texts. Compression serves both to save storage space and to save transmission time. We shall assume that the uncompressed text is stored in a file. The aim of compression algorithms is to produce another file containing the compressed version of the same text. Methods in this section work with no loss of information, so that decompressing the compressed text restores exactly the original text.

We apply two main strategies to design the algorithms. The first strategy is a statistical method that takes into account the frequencies of symbols to build a uniquely decipherable code optimal with respect to the compression. The code contains new codewords for the symbols occurring in the text. In this method fixed-length blocks of bits are encoded by different codewords. *A contrario* the second strategy encodes variable-length segments of the text. To put it simply, the algorithm, while scanning the text, replaces some already read segments just by a pointer to their first occurrences.

Text compression software often use a mixture of several methods. An example of that is given in Section 8.7.3 which contains in particular two classical simple compression algorithms. They compress efficiently only a small variety of texts when used alone. But they become more powerful with the special preprocessing presented there.

8.7.1 Huffman Coding

The Huffman method is an optimal statistical coding. It transforms the original code used for characters of the text (ASCII code on 8 b, for instance). Coding the text is just replacing each symbol (more exactly each occurrence of it) by its new codeword. The method works for any length of blocks (not only 8 b), but the running time grows exponentially with the length. In the following, we assume that symbols are originally encoded on 8 b to simplify the description.

The Huffman algorithm uses the notion of **prefix code**. A prefix code is a set of words containing no word that is a prefix of another word of the set. The advantage of such a code is that decoding is immediate. Moreover, it can be proved that this type of code does not weaken the compression.

A prefix code on the binary alphabet $\{0, 1\}$ can be represented by a trie (see section on the Aho–Corasick algorithm) that is a binary tree. In the present method codes are complete: they correspond to complete tries (internal nodes have exactly two children). The leaves are labeled by the original characters, edges are labeled by 0 or 1, and labels of branches are the words of the code. The condition on the code implies that codewords are identified with leaves only. We adopt the convention that, from an internal node, the edge to its left child is labeled by 0, and the edge to its right child is labeled by 1.

In the model where characters of the text are given new codewords, the Huffman algorithm builds a code that is optimal in the sense that the compression is the best possible (the length of the compressed text is minimum). The code depends on the text, and more precisely on the frequencies of each character in the uncompressed text. The more frequent characters are given short codewords, whereas the less frequent symbols have longer codewords.

Encoding

The coding algorithm is composed of three steps: count of character frequencies, construction of the prefix code, and encoding of the text.

The first step consists in counting the number of occurrences of each character in the original text (see Fig. 8.52). We use a special end marker (denoted by END), which (virtually) appears only once at the end of the text. It is possible to skip this first step if fixed statistics on the alphabet are used. In this case the method is optimal according to the statistics, but not necessarily for the specific text.

The second step of the algorithm builds the tree of a prefix code using the character frequency $freq(a)$ of each character a in the following way:

- create a one-node tree t for each character a, setting $weight(t) = freq(a)$ and $label(t) = a$,
- repeat (1), extract the two least weighted trees t_1 and t_2, and (2) create a new tree t_3 having left subtree t_1, right subtree t_2, and weight $weight(t_3) = weight(t_1) + weight(t_2)$,

```
COUNT(fin)
for each character a ∈ Σ
      do freq(a) ← 0
while not end of file fin and a is the next symbol
      do freq(a) ← freq(a) + 1
END ← 1
```

Figure 8.52: Counts the character frequencies.

```
BUILD-TREE()
for each character a ∈ Σ ∪ {END}
      do if freq(a) ≠ 0
             then create a new node t
                     weight(t) ← freq(a)
                     label(t) ← a
lleaves ← list of all the nodes in increasing order of weight
ltrees ← empty list
while LENGTH(lleaves) + LENGTH(ltrees) > 1
      do (ℓ, r) ← extract the two nodes of smallest weight (among the two nodes at the
                   beginning of lleaves and the two nodes at the beginning of ltrees)
          create a new node t
          weight(t) ← weight(ℓ) + weight(r)
          left(t) ← ℓ
          right(t) ← r
          insert t at the end of ltrees
return t
```

Figure 8.53: Builds the coding tree.

- until only one tree remains.

The tree is constructed by the algorithm BUILD-TREE in Fig. 8.53. The implementation uses two linear lists. The first list contains the leaves of the future tree, each associated with a symbol. The list is sorted in the increasing order of the weight of the leaves (frequency of symbols). The second list contains the newly created trees. Extracting the two least weighted trees consists in extracting the two least weighted trees among the two first trees of the list of leaves and the two first trees of the list of created trees. Each new tree is inserted at the end of the list of the trees. The only tree remaining at the end of the procedure is the coding tree.

After the coding tree is built, it is possible to recover the codewords associated with characters by a simple depth-first search of the tree (see Fig. 8.54; *codeword*(a) is then the binary code associated with the character a.

In the third step, the original text is encoded. Since the code depends on the original text, in order to be able to decode the compressed text, the coding tree and the original codewords of symbols must be stored with the compressed text. This information is placed in a header of the compressed file, to be

```
BUILD-CODE(t, length)
if t is not a leaf
   then temp[length] ← 0
          BUILD-CODE(left(t), length + 1)
          temp[length] ← 1
          BUILD-CODE(right(t), length + 1)
   else codeword(label(t)) ← temp[0 . . length − 1]
```

Figure 8.54: Builds the character codes from the coding tree.

```
CODE-TREE(fout, t)
if t is not a leaf
  then write a 0 in the file fout
       CODE-TREE(fout, left(t))
       CODE-TREE(fout, right(t))
  else write a 1 in the file fout
       write the original code of label(t) in the file fout
```

Figure 8.55: Memorizes the coding tree in the compressed file.

```
CODE-TEXT(fin, fout)
while not end of file fin and a is the next symbol
    do write codeword(a) in the file fout
write codeword(END) in the file fout
```

Figure 8.56: Encodes the characters in the compressed file.

read at decoding time just before the compressed text. The header is made via a depth-first traversal of the tree. Each time an internal node is encountered a 0 is produced. When a leaf is encountered a 1 is produced followed by the original code of the corresponding character on 9 b (so that the end marker can be equal to 256 if all the characters appear in the original text). This part of the encoding algorithm is shown in Fig. 8.55.

After the header of the compressed file is computed, the encoding of the original text is realized by the algorithm of Fig. 8.56.

A complete implementation of the Huffman algorithm, composed of the three steps just described, is given in Fig. 8.57.

Example 8.21: Here y = **CAGATAAGAGAA**. The length of $y = 12 \times 8 = 96$ b (assuming an 8-b code). The character frequencies are

A	C	G	T	END
7	1	3	1	1

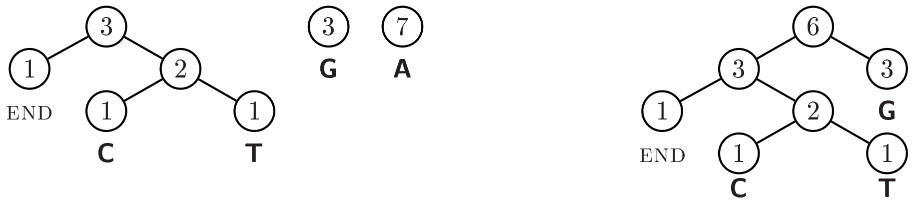
The different steps during the construction of the coding tree are

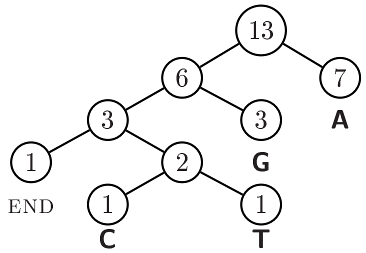

```
CODING(fin, fout)
COUNT(fin)
t ← BUILD-TREE()
BUILD-CODE(t, 0)
CODE-TREE(fout, t)
CODE-TEXT(fin, fout)
```

Figure 8.57: Complete function for Huffman coding.

```
REBUILD-TREE(fin, t)
b ← read a bit from the file fin
if b = 1                                        ▷ leaf
  then left(t) ← NIL
       right(t) ← NIL
       label(t) ← symbol corresponding to the 9 next bits in the file fin
  else create a new node ℓ
       left(t) ← ℓ
       REBUILD-TREE(fin, ℓ)
       create a new node r
       right(t) ← r
       REBUILD-TREE(fin, r)
```

Figure 8.58: Rebuilds the tree read from the compressed file.





character codewords:

A	C	G	T	END
1	0010	01	0011	000

The encoded tree is **0001** binary (END, 9)**01**binary (**C**, 9)**1**binary(**T**, 9) **1**binary (**G**, 9)**1**binary (**A**, 9), which produces a header of length 54 b,

0001 100000000 01 001000011 1 001010100 1 001000111 1 001000001
The encoded text

0010 1 01 1 0011 1 1 01 1 01 1 1 000
is of length 24 b. The total length of the compressed file is 78 b.

The construction of the tree takes $O(\sigma \log \sigma)$ time if the sorting of the list of the leaves is implemented efficiently. The rest of the encoding process runs in linear time in the sum of the sizes of the original and compressed texts.

Decoding

Decoding a file containing a text compressed by the Huffman algorithm is a mere programming exercise. First, the coding tree is rebuilt by the algorithm of Fig. 8.58. Then, the uncompressed text is recovered by parsing the compressed text with the coding tree. The process begins at the root of the coding tree and follows a left edge when a 0 is read or a right edge when a 1 is read. When a leaf is encountered, the corresponding character (in fact the original codeword of it) is produced and the parsing phase resumes at the root of the tree. The parsing ends when the codeword of the end marker is read. An implementation of the decoding of the text is presented in Fig. 8.59.

The complete decoding program is given in Fig. 8.60. It calls the preceding functions. The running

```
Decode-text(fin, fout, root)
t ← root
while label(t) ≠ END
    do if t is a leaf
          then label(t) in the file fout
               t ← root
          else b ← read a bit from the file fin
               if b = 1
                  then t ← right(t)
                  else t ← left(t)
```

Figure 8.59: Reads the compressed text and produces the uncompressed text.

```
Decoding(fin, fout)
create a new node root
Rebuild-tree(fin, root)
Decode-text(fin, fout, root)
```

Figure 8.60: Complete function for Huffman decoding.

time of the decoding program is linear in the sum of the sizes of the texts it manipulates.

8.7.2 Lempel–Ziv–Welsh (LZW) Compression

Ziv and Lempel designed a compression method using encoding segments. These segments are stored in a dictionary that is built during the compression process. When a segment of the dictionary is encountered later while scanning the original text it is substituted by its index in the dictionary. In the model where portions of the text are replaced by pointers on previous occurrences, the Ziv–Lempel compression scheme can be proved to be asymptotically optimal (on large enough texts satisfying good conditions on the probability distribution of symbols).

The dictionary is the central point of the algorithm. It has the property of being prefix closed (every prefix of a word of the dictionary is in the dictionary), so that it can be implemented as a tree. Furthermore, a hashing technique makes its implementation efficient. The version described in this section is called the Lempel–Ziv–Welsh method after several improvements introduced by Welsh. The algorithm is implemented by the **compress** command existing under the UNIX operating system.

Compression Method

We describe the scheme of the compression method. The dictionary is initialized with all the characters of the alphabet. The current situation is when we have just read a segment w in the text. Let a be the next symbol (just following w). Then we proceed as follows:

- If wa is not in the dictionary, we write the index of w to the output file, and add wa to the dictionary. We then reset w to a and process the next symbol (following a).
- If wa is in the dictionary, we process the next symbol, with segment wa instead of w.

Initially, the segment w is set to the first symbol of the source text.

Example 8.22: Here $y =$ **CAGTAAGAGAA**

C	A	G	T	A	A	G	A	G	A	A		w	written	added
	↑											**C**	67	**CA**, 257
		↑										**A**	65	**AG**, 258
			↑									**G**	71	**GT**, 259
				↑								**T**	84	**TA**, 260
					↑							**A**	65	**AA**, 261
						↑						**A**		
							↑					**AG**	258	**AGA**, 262
								↑				**A**		
									↑			**AG**		
										↑		**AGA**	262	**AGAA**, 262
											↑	**A**		
													65	
													256	

Decompression Method

The decompression method is symmetrical to the compression algorithm. The dictionary is recovered while the decompression process runs. It is basically done in this way:

- Read a code c in the compressed file.
- Write in the output file the segment w which has index c in the dictionary.
- Add to the dictionary the word wa where a is the first letter of the next segment.

In this scheme, a problem occurs if the next segment is the word which is being built. This arises only if the text contains a segment $azazax$ for which az belongs to the dictionary but aza does not. During the compression process the index of az is written into the compressed file, and aza is added to the dictionary. Next, aza is read and its index is written into the file. During the decompression process the index of aza is read while the word az has not been completed yet: the segment aza is not already in the dictionary. However, since this is the unique case where the situation arises, the segment aza is recovered taking the last segment az added to the dictionary concatenated with its first letter a.

Example 8.23: Here the decoding is 67, 65, 71, 84, 65, 258, 262, 65, 256

read	written	added
67	**C**	
65	**A**	**CA**, 257
71	**G**	**AG**, 258
84	**T**	**GT**, 259
65	**A**	**TA**, 260
258	**AG**	**AA**, 261
262	**AGA**	**AGA**, 262
65	**A**	**AGAA**, 263
256		

Implementation

For the compression algorithm shown in Fig. 8.61, the dictionary is stored in a table D. The dictionary is implemented as a tree; each node z of the tree has the three following components:

- *parent*(z) is a link to the parent node of z.

```
COMPRESS(fin, fout)
count ← −1
for each character a ∈ Σ
    do count ← count + 1
       HASH-INSERT(D, (−1, a, count))
count ← count + 1
HASH-INSERT(D, (−1, END, count))
p ← −1
while not end of file fin
    do a ← next character of fin
       q ← HASH-SEARCH(D, (p, a))
       if q = NIL
         then write code(p) on 1 + log(count) bits in fout
              count ← count + 1
              HASH-INSERT(D, (p, a, count))
              p ← HASH-SEARCH(D, (−1, a))
         else p ← q
write code(p) on 1 + log(count) bits in fout
write code(HASH-SEARCH(D, (−1, END))) on 1 + log(count) bits in fout
```

Figure 8.61: LZW compression algorithm.

- *label*(z) is a character.
- *code*(z) is the code associated with z.

The tree is stored in a table that is accessed with a hashing function. This provides fast access to the children of a node. The procedure HASH-INSERT$((D,(p,a,c)))$ inserts a new node z in the dictionary D with $parent(z) = p$, $label(z) = a$, and $code(z) = c$. The function HASH-SEARCH$((D,(p,a)))$ returns the node z such that $parent(z) = p$ and $label(z) = a$.

For the decompression algorithm, no hashing technique is necessary. Having the index of the next segment, a bottom-up walk in the trie implementing the dictionary produces the mirror image of the segment. A stack is used to reverse it. We assume that the function *string*(c) performs this specific work for a code c. The bottom-up walk follows the parent links of the data structure. The function *first*(w) gives the first character of the word w. These features are part of the decompression algorithm displayed in Fig. 8.62.

The Ziv–Lempel compression and decompression algorithms run both in linear time in the sizes of the files provided a good hashing technique is chosen. Indeed, it is very fast in practice. Its main advantage compared to Huffman coding is that it captures long repeated segments in the source file.

8.7.3 Mixing several methods

We describe simple compression methods and then an example of a combination of several of them, basis of the popular **bzip** software.

Run Length Encoding

The aim of Run Length Encoding (RLE) is to efficiently encode repetitions occurring in the input data. Let us assume that it contains a good quantity of repetitions of the form $aa \dots a$ for some character a ($a \in \Sigma$). A repetition of k consecutive occurrences of letter a is replaced by $\&ak$, where the symbol & is a new character ($\& \notin \Sigma$).

The string $\&ak$ that encodes a repetition of k consecutive occurrences of a is itself encoded on the binary alphabet $\{0,1\}$. In practice, letters are often represented by their ASCII code. Therefore, the codeword of a letter belongs to $\{0,1\}^k$ with $k = 7$ or 8. Generally there is no problem in choosing or encoding the special character &. The integer k of the string $\&ak$ is also encoded on the binary

```
UNCOMPRESS(fin, fout)
count ← −1
for each character a ∈ Σ
    do count ← count + 1
       HASH-INSERT(D, (−1, a, count))
count ← count + 1
HASH-INSERT(D, (−1, END, count))
c ← first code on 1 + log(count) bits in fin
write string(c) in fout
a ← first(string(c))
while TRUE
    do d ← next code on 1 + log(count) bits in fin
       if d > count
         then count ← count + 1
              parent(count) ← c
              label(count) ← a
              write string(c)a in fout
              c ← d
         else a ← first(string(d))
              if a ≠ END
                then count ← count + 1
                     parent(count) ← c
                     label(count) ← a
                     write string(d) in fout
                     c ← d
                else break
```

Figure 8.62: LZW decompression algorithm.

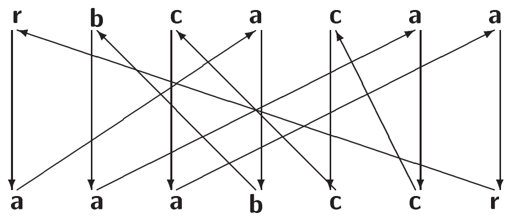


Figure 8.63: Example of text $y =$ **baccara**. Top line is $BW(y)$ and bottom line the sorted list of letters of it. Top-down arrows correspond to succession of occurrences in y. Each bottom-up arrow links the same occurrence of a letter in y. Arrows starting from equal letters do not cross. The circular path is associated with rotations of the string y. If the starting point is known, the only occurrence of letter **b** here, following the path produces the initial string y.

alphabet, but it is not sufficient to translate it by its binary representation, because we would be unable to recover it at decoding time inside the stream of bits. A simple way to cope with this is to encode k by the string $0^\ell \text{bin}(k)$, where $\text{bin}(k)$ is the binary representation of k, and ℓ is the length of it. This works well because the binary representation of k starts with a 1 so there is no ambiguity to recover ℓ by counting during the decoding phase. The size of the encoding of k is thus roughly $2 \log k$. More sophisticated integer representations are possible, but none is really suitable for the present situation. Simpler solution consists in encoding k on the same number of bits as other symbols, but this bounds values of ℓ and decreases the power of the method.

Move To Front

The Move To Front (MTF) method may be regarded as an extension of Run Length Encoding or a simplification of Ziv–Lempel compression. It is efficient when the occurrences of letters in the input text are localized into relatively short segment of it. The technique is able to capture the proximity between occurrences of symbols and to turn it into a short encoded text.

Letters of the alphabet Σ of the input text are initially stored in a list that is managed dynamically. Letters are represented by their rank in the list, starting from 1, rank that is itself encoded as described above for RLE.

Letters of the input text are processed in an on-line manner. The clue of the method is that each letter is moved to the beginning of the list just after it is translated by the encoding of its rank.

The effect of MTF is to reduce the size of the encoding of a letter that reappears soon after its preceding occurrence.

Integrated example

Most compression software combine several methods to be able to compress efficiently a large range of input data. We present an example of this strategy, implemented by the UNIX command **bzip**.

Let $y = y[0]y[1] \cdots y[n-1]$ be the input text. The k-th rotation (or conjugate) of y, $0 \leq k \leq n-1$, is the string $y_k = y[k]y[k+1] \cdots y[n-1]y[0]y[1] \cdots y[k-1]$.

We define the BW transformation as $BW(y) = y[p_0]y[p_1] \cdots y[p_{n-1}]$, where $p_i + 1$ is such that y_{p_i+1} has rank i in the sorted list of all rotations of y.

It is remarkable that y can be recovered from both $BW(y)$ and a position on it, starting position of the inverse transformation (see Figure 8.63). This is possible due to the following property of the transformation. Assume that $i < j$ and $y[p_i] = y[p_j] = a$. Since $i < j$, the definition implies $y_{p_i+1} < y_{p_j+1}$. Since $y[p_i] = y[p_j]$, transferring the last letters of y_{p_i+1} and y_{p_j+1} to the beginning of these words does not change the inequality. This proves that the two occurrences of a in $BW(y)$ are in the same relative order as in the sorted list of letters of y. Figure 8.63 illustrates the inverse transformation.

Transformation BW obviously does not compress the input text y. But $BW(y)$ is compressed more efficiently with simple methods. This is the strategy applied for the command **bzip**. It is a combination of the BW transformation followed by MTF encoding and RLE encoding. Arithmetic coding, a method providing compression ratios slightly better than Huffman coding, may also be used.

Sizes in bytes	111,261	768,771	377,109	513,216	39,611	93,695	
Source Texts	bib	book1	news	pic	progc	trans	Average
pack	5.24	4.56	5.23	1.66	5.26	5.58	4.99
gzip-b	2.51	3.25	3.06	0.82	2.68	1.61	2.69
bzip2-1	2.10	2.81	2.85	0.78	2.53	1.53	2.46

Table 8.1: Compression results with three algorithms: Huffman coding (**pack**), Ziv–Lempel coding (**gzip-b**) and Burrows-Wheeler coding (**bzip2-1**). Figures give the number of bits used per character (letter). They show that **pack** is the less efficient method and that **bzip2-1** compresses a bit more than **gzip-b**.

Table 8.1 contains a sample of experimental results showing the behavior of compression algorithms on different types of texts from the Calgary Corpus: `bib` (bibliography), `book1` (fiction book), `news` (USENET batch file), `pic` (black and white fax picture), `progc` (source code in C) and `trans` (transcript of terminal session).

The compression algorithms reported in the table are: the Huffman coding algorithm implemented by **pack**, the Ziv–Lempel algorithm implemented by **gzip-b** and the compression based on the BW transform implemented by **bzip2-1**.

Additional compression results can be found at `http://corpus.canterbury.ac.nz`.

8.8 Research Issues and Summary

The algorithm for string searching by hashing was introduced by Harrison in 1971, and later fully analyzed by Karp and Rabin (1987).

The linear-time string-matching algorithm of Knuth, Morris, and Pratt is from 1976. It can be proved that, during the search, a character of the text is compared to a character of the pattern no more than $\log_\Phi(|x|+1)$ (where Φ is the golden ratio $(1+\sqrt{5})/2$). Simon (1993) gives an algorithm similar to the previous one but with a delay bounded by the size of the alphabet (of the pattern x). Hancart (1993) proves that the delay of Simon's algorithm is, indeed, no more than $1+\log_2|x|$. He also proves that this is optimal among algorithms searching the text through a window of size 1.

Galil (1981) gives a general criterion to transform searching algorithms of that type into real-time algorithms.

The Boyer–Moore algorithm was designed by Boyer and Moore (1977). The first proof on the linearity of the algorithm when restricted to the search of the first occurrence of the pattern is in Knuth et al. (1977). Cole (1994) proves that the maximum number of symbol comparisons is bounded by $3n$, and that this bound is tight.

Knuth et al. (1977) consider a variant of the Boyer–Moore algorithm in which all previous matches inside the current window are memorized. Each window configuration becomes the state of what is called the Boyer–Moore automaton. It is still unknown whether the maximum number of states of the automaton is polynomial or not.

Several variants of the Boyer–Moore algorithm avoid the quadratic behavior when searching for all occurrences of the pattern. Among the more efficient in terms of the number of symbol comparisons are: the algorithm of Apostolico and Giancarlo (1986), Turbo–BM algorithm by Crochemore et al. (1992) (the two algorithms are analyzed in Lecroq (1995)), and the algorithm of Colussi (1994).

The general bound on the expected time complexity of string matching is $O(|y|\log|x|/|x|)$. The probabilistic analysis of a simplified version of the Boyer–Moore algorithm, similar to the Quick Search algorithm of Sunday (1990) described in the chapter, was studied by several authors.

String searching can be solved by a linear-time algorithm requiring only a constant amount of memory in addition to the pattern and the (window on the) text. This can be proved by different techniques presented in Crochemore and Rytter (2002).

The Aho–Corasick algorithm is from Aho and Corasick (1975). It is implemented by the **fgrep** command under the UNIX operating system. Commentz-Walter (1979) has designed an extension of the Boyer-Moore algorithm to several patterns. It is fully described in Aho (1990).

On general alphabets the two-dimensional pattern matching can be solved in linear time, whereas the running time of the Bird/Baker algorithm has an additional $\log \sigma$ factor. It is still unknown whether the problem can be solved by an algorithm working simultaneously in linear time and using only a constant amount of memory space (see Crochemore and Rytter 2002).

The suffix tree construction of section 8.4 is by McCreight (1976). An on-line construction is given by Ukkonen (1995). Other data structures to represent indexes on text files are: direct acyclic word graph (Blumer et al., 1985), suffix automata (Crochemore, 1986), and suffix arrays (Manber and Myers, 1993). All these techniques are presented in (Crochemore and Rytter, 2002). The data structures implement full indexes with standard operations, whereas applications sometimes need only incomplete indexes. The design of compact indexes is still unsolved.

First algorithms for aligning two sequences are by Needleman and Wunsch (1970) and Wagner and Fischer (1974). Idea and algorithm for local alignment is by Smith and Waterman (1981). Hirschberg (1975) presents the computation of the lcs in linear space. This is an important result because the algorithm is classically run on large sequences. Another implementation is given in Durbin et al. (1998). The quadratic time complexity of the algorithm to compute the Levenshtein distance is a bottleneck in practical string comparison for the same reason.

Approximate string searching is a lively domain of research. It includes, for instance, the notion of regular expressions to represent sets of strings. Algorithms based on regular expression are commonly found in books related to compiling techniques. The algorithms of section 8.6 are by Baeza-Yates and Gonnet (1992) and Wu and Manber (1992).

The statistical compression algorithm of Huffman (1951) has a dynamic version where symbol counting is done at coding time. The current coding tree is used to encode the next character and then updated. At decoding time a symmetrical process reconstructs the same tree, so the tree does not need to be stored with the compressed text, see Knuth (1985). The command **compact** of UNIX implements this version.

Several variants of the Ziv and Lempel algorithm exist. The reader can refer to Bell et al. (1990) for a discussion on them. Nelson (1992) presents practical implementations of various compression algorithms. The BW transform is from Burrows and Wheeler (1994).

Defining Terms

Alignment: an alignment of two strings x and y is a word of the form $(\overline{x}_0, \overline{y}_0)(\overline{x}_1, \overline{y}_1) \cdots (\overline{x}_{p-1}, \overline{y}_{p-1})$ where each $(\overline{x}_i, \overline{y}_i) \in (\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\}) \setminus (\{(\varepsilon, \varepsilon)\}$ for $0 \leq i \leq p-1$ and both $x = \overline{x}_0\overline{x}_1 \cdots \overline{x}_{p-1}$ and $y = \overline{y}_0\overline{y}_1 \cdots \overline{y}_{p-1}$.

Border: A word $u \in \Sigma^*$ is a border of a word $w \in \Sigma^*$ if u is both a prefix and a suffix of w (there exist two words $v, z \in \Sigma^*$ such that $w = vu = uz$). The common length of v and z is a period of w.

Edit distance: The metric distance between two strings that counts the minimum number of insertions and deletions of symbols to transform one string into the other.

Hamming distance: The metric distance between two strings of same length that counts the number of mismatches.

Levenshtein distance: The metric distance between two strings that counts the minimum number of insertions, deletions, and substitutions of symbols to transform one string into the other.

Occurrence: An occurrence of a word $u \in \Sigma^*$, of length m, appears in a word $w \in \Sigma^*$, of length n, at position i if for $0 \leq k \leq m-1$, $u[k] = w[i+k]$.

Prefix: A word $u \in \Sigma^*$ is a prefix of a word $w \in \Sigma^*$ if $w = uz$ for some $z \in \Sigma^*$.

Prefix code: Set of words such that no word of the set is a prefix of another word contained in the set. A prefix code is represented by a coding tree.

Segment: A word $u \in \Sigma^*$ is a segment of a word $w \in \Sigma^*$ if u occurs in w (see occurrence), that is, $w = vuz$ for two words $v, z \in \Sigma^*$ (u is also referred to as a factor or a subword of w).

Subsequence: A word $u \in \Sigma^*$ is a subsequence of a word $w \in \Sigma^*$ if it is obtained from w by deleting zero or more symbols that need not be consecutive (u is sometimes referred to as a subword of w, with a possible confusion with the notion of segment).

Suffix: A word $u \in \Sigma^*$ is a suffix of a word $w \in \Sigma^*$ if $w = vu$ for some $v \in \Sigma^*$.

Suffix tree: Trie containing all the suffixes of a word.

Trie: Tree in which edges are labeled by letters or words.

Bibliography

[1] Aho, A. V. 1990. Algorithms for finding patterns in strings. In *Handbook of Theoretical Computer Science*, vol. A. *Algorithms and Complexity*, J. van Leeuwen, ed., pp. 255–300. Elsevier, Amsterdam.

[2] Aho, A. V. and Corasick, M. J. 1975. Efficient string matching: an aid to bibliographic search. *Comm. ACM* 18(6):333–340.

[3] Baeza-Yates, R. A. and Gonnet, G. H. 1992. A new approach to text searching. *Comm. ACM* 35(10):74–82.

[4] Baker, T. P. 1978. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM J. Comput.* 7(4):533–541.

[5] Bell, T. C., Cleary, J. G., and Witten, I. H. 1990. *Text Compression.* Prentice–Hall, Englewood Cliffs, NJ.

[6] Bird, R. S. 1977. Two-dimensional pattern matching. *Inf. Process. Lett.* 6(5):168–170.

[7] Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., Chen, M. T., and Seiferas, J. 1985. The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.* 40:31–55.

[8] Boyer, R. S. and Moore, J. S. 1977. A fast string searching algorithm. *Comm. ACM* 20(10):762–772.

[9] Breslauer, D., Colussi, L., and Toniolo, L. 1993. Tight comparison bounds for the string prefix matching problem. *Inf. Process. Lett.* 47(1):51–57.

[10] Burrows, M. and Wheeler, D. 1994. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation.

[11] Cole, R. 1994. Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm. *SIAM J. Comput.* 23(5):1075–1091.

[12] Colussi, L. 1994. Fastest pattern matching in strings. *J. Algorithms* 16(2):163–189.

[13] Crochemore, M. 1986. Transducers and repetitions. *Theor. Comput. Sci.* 45(1):63–86.

[14] Crochemore, M. and Rytter, W. 2002. *Jewels of Stringology.* World Scientific.

[15] Durbin, R., Eddy, S., and A. Krogh, A., and Mitchison G. 1998. *Biological sequence analysis probabilistic models of proteins and nucleic acids.* Cambridge University Press.

[16] Galil, Z. 1981. String matching in real time. *J. ACM* 28(1):134–149.

[17] Hancart, C. 1993. On Simon's string searching algorithm. *Inf. Process. Lett.* 47(2):95–99.

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

[19] Hume, A. and Sunday, D. M. 1991. Fast string searching. *Software—Practice Exp.* 21(11):1221–1248.

[20] Karp, R. M. and Rabin, M. O. 1987. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.* 31(2):249–260.

[21] Knuth, D. E. 1985. Dynamic Huffman coding. *J. Algorithms* 6(2):163–180.

[22] Knuth, D. E., Morris, J. H., Jr, and Pratt, V. R. 1977. Fast pattern matching in strings. *SIAM J. Comput.* 6(1):323–350.

[23] Lecroq, T. 1995. Experimental results on string-matching algorithms. *Software—Practice Exp.* 25(7):727–765.

[24] McCreight, E. M. 1976. A space-economical suffix tree construction algorithm. *J. Algorithms* 23(2):262–272.

[25] Manber, U. and Myers, G. 1993. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.* 22(5):935–948.

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

[27] Nelson, M. 1992. *The Data Compression Book*. M&T Books.

[28] Simon, I. 1993. String matching algorithms and automata. In *First American Workshop on String Processing*, Baeza-Yates and Ziviani, eds., pp. 151–157. Universidade Federal de Minas Gerais.

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

[30] Stephen, G. A. 1994. *String Searching Algorithms*. World Scientific Press.

[31] Sunday, D. M. 1990. A very fast substring search algorithm. *Comm. ACM* 33(8):132–142.

[32] Ukkonen, E. 1995. On-line construction of suffix trees. *Algorithmica* 14(3):249–260.

[33] Wagner, R. A. and Fischer, M. 1974. The string-to-string correction problem. *J. ACM* 21(1):168–173.

[34] Welch, T. 1984. A technique for high-performance data compression. *IEEE Comput.* 17(6):8–19.

[35] Wu, S. and Manber, U. 1992. Fast text searching allowing errors. *Comm. ACM* 35(10):83–91.

[36] Zhu, R. F. and Takaoka, T. 1989. A technique for two-dimensional pattern matching. *Comm. ACM* 32(9):1110–1120.

Further Information

Problems and algorithms presented in the chapter are just a sample of questions related to pattern matching. They share the formal methods used to design solutions and efficient algorithms. A wider panorama of algorithms on texts may be found in a few books such as:

Apostolico, A. and Galil, Z., editors. 1997. *Pattern Matching Algorithms*. Oxford University Press.

Bell, T. C., Cleary, J. G., and Witten, I. H. 1990. *Text Compression*. Prentice-Hall, Englewood Cliffs, NJ.

Crochemore, M. and Rytter, W. 2002. *Jewels of Stringology*. World Scientific.

Gusfield D. 1997. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press.

Navarro, G. and Raffinot M. 2002. *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press.

Nelson, M. 1992. *The Data Compression Book*. M&T Books.

Salomon, D. 2000. *Data Compression: the Complete Reference*. Springer Verlag.

Stephen, G. A. 1994. *String Searching Algorithms.* World Scientific Press.

Research papers in pattern matching are disseminated in a few journals, among which are: *Communications of the ACM*, *Journal of the ACM*, *Theoretical Computer Science*, *Algorithmica*, *Journal of Algorithms*, *SIAM Journal on Computing*, *Journal of Discrete Algorithms*.

Finally, three main annual conferences present the latest advances of this field of research: Combinatorial Pattern Matching, which started in 1990. Data Compression Conference, which is regularly held at Snowbird. The scope of SPIRE (String Processing and Information Retrieval) includes the domain of data retrieval.

General conferences in computer science often have sessions devoted to pattern matching algorithms.

Several books on the design and analysis of general algorithms contain chapters devoted to algorithms on texts. Here is a sample of these books:

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. 1990. *Introduction to Algorithms.* MIT Press.

Gonnet, G. H. and Baeza-Yates, R. A. 1991. *Handbook of Algorithms and Data Structures.* Addison-Wesley.

Animations of selected algorithms may be found at:

- `http://www-igm.univ-mlv.fr/~lecroq/string/` (Exact String Matching Algorithms),
- `http://www-igm.univ-mlv.fr/~lecroq/seqcomp/` (Alignments).