

HANDBOOK OF COMPUTER SCIENCE AND  
ENGINEERING

Chapter 6

Pattern matching and text compression algorithms

# Contents

<b>6</b>	<b>Pattern matching and text compression algorithms</b>	<b>5</b>
6.1	Processing texts efficiently	5
6.2	String-matching algorithms	6
6.2.1	Karp-Rabin algorithm	6
6.2.2	Knuth-Morris-Pratt algorithm	8
6.2.3	Boyer-Moore algorithm	9
6.2.4	Quick Search algorithm	13
6.2.5	Experimental results	14
6.2.6	Aho-Corasick algorithm	14
6.3	Two-dimensional pattern matching algorithms	18
6.3.1	Zhu-Takaoka algorithm	19
6.3.2	Bird/Baker algorithm	22
6.4	Suffix trees	25
6.4.1	McCreight algorithm	27
6.5	Longest common subsequence of two strings	28
6.5.1	Dynamic programming	31
6.5.2	Reducing the space: Hirschberg algorithm	33
6.6	Approximate string matching	34
6.6.1	Shift-Or algorithm	36
6.6.2	String matching with $k$ mismatches	37
6.6.3	String matching with $k$ differences	38
6.6.4	Wu-Manber algorithm	40
6.7	Text compression	41
6.7.1	Huffman coding	41
6.7.2	LZW Compression	48
6.7.3	Experimental results	52
6.8	Research Issues and Summary	52
6.9	Defining Terms	53
6.10	References	54
6.11	Further Information	55



# List of Figures

6.1	The brute force string-matching algorithm. . . . .	6
6.2	The Karp-Rabin string-matching algorithm. . . . .	7
6.3	Shift in the Knuth-Morris-Pratt algorithm ( $v$ suffix of $u$ ). . . . .	8
6.4	The Knuth-Morris-Pratt string-matching algorithm. . . . .	9
6.5	Preprocessing phase of the Knuth-Morris-Pratt algorithm: computing $next$ . . . . .	9
6.6	good-suffix shift, $u$ reappears preceded by a character different from $b$ . . . . .	10
6.7	good-suffix shift, only a suffix of $u$ reappears as a prefix of $x$ . . . . .	10
6.8	bad-character shift, $a$ appears in $x$ . . . . .	10
6.9	bad-character shift, $a$ does not appear in $x$ . . . . .	11
6.10	The Boyer-Moore string-matching algorithm. . . . .	12
6.11	Computation of the bad-character shift. . . . .	12
6.12	Computation of the good-suffix shift. . . . .	13
6.13	The Quick Search string-matching algorithm. . . . .	14
6.14	Running times for a DNA sequence. . . . .	15
6.15	Running times for an english text. . . . .	16
6.16	Preprocessing phase of the Aho-Corasick algorithm. . . . .	16
6.17	Construction of the trie. . . . .	17
6.18	Completion of the output function and construction of failure links. . . . .	17
6.19	The complete Aho-Corasick algorithm. . . . .	19
6.20	The brute force two-dimensional pattern matching algorithm. . . . .	19
6.21	Search for $x'$ in $y'$ using KMP algorithm. . . . .	21
6.22	Naive check of an occurrence of $x$ in $y$ at position $(row, column)$ . . . . .	21
6.23	The Zhu-Takaoka two-dimensional pattern matching algorithm. . . . .	22
6.24	Computes the function $next$ for rows of $X$ . . . . .	23
6.25	The Bird/Baker two-dimensional pattern matching algorithm. . . . .	24
6.26	Construction of a suffix tree for $y$ . . . . .	25
6.27	Insertion of a new suffix in the tree. . . . .	26
6.28	Suffix tree construction. . . . .	29
6.29	Initialization procedure. . . . .	29
6.30	The crucial rescan operation. . . . .	30
6.31	Breaking an edge. . . . .	30
6.32	The scan operation. . . . .	31
6.33	Dynamic programming algorithm to compute $llcs(x, y) = L[m, n]$ . . . . .	32
6.34	Production of an $lcs(x, y)$ . . . . .	32
6.35	$O(\min(m, n))$ -space algorithm to compute $llcs(x, y)$ . . . . .	33
6.36	Computation of $L^*$ . . . . .	34
6.37	$O(\min(m, n))$ -space computation of $lcs(x, y)$ . . . . .	35
6.38	Meaning of vector $R_i^0$ . . . . .	36

6.39	If $R_{i-1}^0[j-1] = 0$ then $R_i^1[j] = 0$ .	38
6.40	$R_i^1[j] = R_{i-1}^1[j-1]$ if $y[i] = x[j]$ .	38
6.41	If $R_{i-1}^0[j-1] = 0$ then $R_i^1[j] = 0$ .	39
6.42	$R_i^1[j] = R_{i-1}^1[j-1]$ if $y[i] = x[j]$ .	39
6.43	If $R_i^0[j-1] = 0$ then $R_i^1[j] = 0$ .	40
6.44	$R_i^1[j] = R_{i-1}^1[j-1]$ if $y[i] = x[j]$ .	40
6.45	Wu-Manber approximate string-matching algorithm.	42
6.46	Counts the character frequencies.	43
6.47	Builds the coding tree.	44
6.48	Builds the character codes from coding tree.	45
6.49	Memorizes the coding tree in the compressed file.	45
6.50	Encodes the characters in the compressed file.	45
6.51	Complete function for Huffman coding.	45
6.52	Rebuilds the tree read from the compressed file.	47
6.53	Reads the compressed text and produces the uncompressed text.	47
6.54	Complete function for decoding.	48
6.55	LZW compression algorithm.	50
6.56	LZW decompression algorithm.	51
6.57	Sizes of texts compressed with three algorithms.	52

## Chapter 6

# Pattern matching and text compression algorithms

MAXIME CROCHEMORE, Gaspard Monge Institute, University of Marne-la-Vallée, France  
THIERRY LECROQ, Laboratoire d'Informatique de Rouen, University of Rouen, France

### 6.1 Processing texts efficiently

The present chapter describes a few standard algorithms used for processing texts. They apply, for example, to the manipulation of texts (word editors), to the storage of textual data (text compression), and to data retrieval systems. The algorithms of the 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 eighteen months. This is the reason why algorithms should be efficient even if the speed of computers increases regularly.

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 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

```

void BF(char *y, char *x, int n, int m) {
    int i, j;

    /* Searching */
    for (i=0; i <= n-m; i++) {
        j=0;
        while (j < m && y[i+j] == x[j]) j++;
        if (j >= m) OUTPUT(i);
    }
}

```

Figure 6.1: The brute force string-matching algorithm.

other elementary methods.

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

## 6.2 String-matching algorithms

String matching consists 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  $x = x[0 \dots m - 1]$ ; its length is equal to  $m$ . The text is denoted by  $y = y[0 \dots n - 1]$ ; its length is equal to  $n$ . The alphabet is denoted by  $\Sigma$  and its size is equal to  $\sigma$ .

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  $i$  in the text when the pattern is aligned with  $y[i \dots i + m - 1]$ .

The brute force algorithm consists of 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 Figure 6.1.

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.

### 6.2.1 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,

```

#define REHASH(a, b, h) (((h-a*d)<<1)+b)

void KR(char *y, char *x, int n, int m) {
    int hy, hx, d, i;

    /* Preprocessing */
    /* computes d = 2^(m-1) with the left-shift operator */
    d=1;
    for (i=1; i < m; i++) d<<=1;

    hy=hx=0;
    for (i=0; i < m; i++) {
        hx=((hx<<1)+x[i]);
        hy=((hy<<1)+y[i]);
    }

    /* Searching */
    for (i=m; i <= n; i++) {
        if (hy == hx && strncmp(y+i-m, x, m) == 0) OUTPUT(i-m);
        hy=REHASH(y[i-m], y[i], hy);
    }
}

```

Figure 6.2: The Karp-Rabin string-matching algorithm.

- $hash(y[i + 1 \dots i + m])$  must be easily computable from  $hash(y[i \dots i + m - 1])$ :  
 $hash(y[i + 1 \dots i + m]) = rehash(y[i], y[i + m], hash(y[i \dots i + m - 1]))$ .

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

$$hash(w[0 \dots k - 1]) = (w[0] * 2^{k-1} + w[1] * 2^{k-2} + \dots + w[k - 1]) \bmod q,$$

where  $q$  is a large number. Then,  $rehash$  has a simple expression

$$rehash(a, b, h) = ((h - a * d) * 2 + b) \bmod q,$$

where  $d = 2^{k-1}$ .

During the search for the pattern  $x$ , it is enough to compare  $hash(x)$  with  $hash(y[i \dots i + m - 1])$  for  $0 \leq i \leq n - m$ . If an equality is found, it is still necessary to check the equality  $x = y[i \dots i + m - 1]$  symbol by symbol.

In the algorithm of Figure 6.2 all the multiplications by 2 are implemented by shifts. 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. So,  $q$  is chosen as the maximum value of an integer.

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

**Example 6.1:**

Let  $x = \text{ing}$ .

Then  $hash(x) = 105 * 2^2 + 110 * 2 + 103 = 743$  (symbols are assimilated with their ASCII codes).

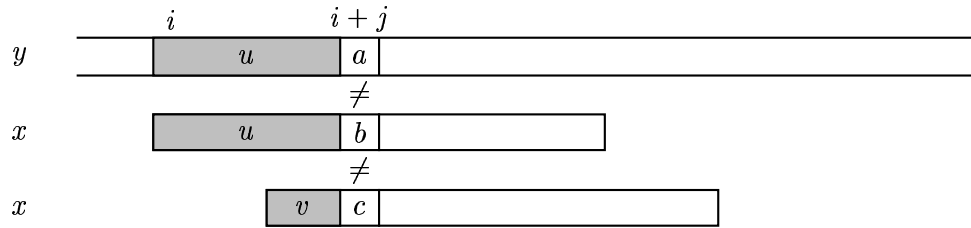


Figure 6.3: Shift in the Knuth-Morris-Pratt algorithm ( $v$  suffix of  $u$ ).

```

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

```

## 6.2.2 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 on 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  $i$ , that is, when the pattern  $x[0 \dots m-1]$  is aligned with the window  $y[i \dots i+m-1]$  on the text. Assume that the first mismatch occurs between symbols  $y[i+j]$  and  $x[j]$  for  $1 < j < m$ . Then,  $y[i \dots i+j-1] = x[0 \dots j-1] = u$  and  $a = y[i+j] \neq x[j] = b$ . 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  $v$  in the pattern must be different from  $b$ . The longest such prefix  $v$  is called the **border** of  $u$  (it occurs at both ends of  $u$ ). This introduces the notation: let  $next[j]$  be the length of the longest (proper) border of  $x[0 \dots j-1]$  followed by a character  $c$  different from  $x[j]$ . Then, after a shift, the comparisons can resume between characters  $y[i+j]$  and  $x[next[j]]$  without missing any occurrence of  $x$  in  $y$ , and avoiding a backtrack on the text (see Figure 6.3).

```

void KMP(char *y, char *x, int n, int m) {
    /* XSIZE is the maximum size of a pattern */
    int i, j, next[XSIZE];

    /* Preprocessing */
    PRE_KMP(x, m, next);

    /* Searching */
    i=j=0;
    while (i < n) {
        while (j > -1 && x[j] != y[i]) j=next[j];
        i++; j++;
        if (j >= m) { OUTPUT(i-j); j=next[m]; }
    }
}

```

Figure 6.4: The Knuth-Morris-Pratt string-matching algorithm.

```

void PRE_KMP(char *x, int m, int next[]) {
    int i, j;

    i=0; j=next[0]=-1;
    while (i < m) {
        while (j > -1 && x[i] != x[j]) j=next[j];
        i++; j++;
        if (i < m && x[i] == x[j]) next[i]=next[j];
        else next[i]=j;
    }
}

```

Figure 6.5: Preprocessing phase of the Knuth-Morris-Pratt algorithm: computing *next*.**Example 6.2:**

```

y = . . . a b a b a a . . . . .
x =     a b a b a b a
x =           a 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 Figure 6.4. 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 Figure 6.5). 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.

**6.2.3 Boyer-Moore algorithm**

The Boyer-Moore algorithm is considered 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

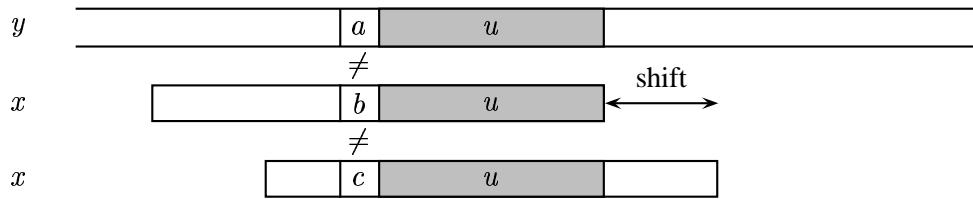


Figure 6.6: good-suffix shift,  $u$  reappears preceded by a character different from  $b$ .

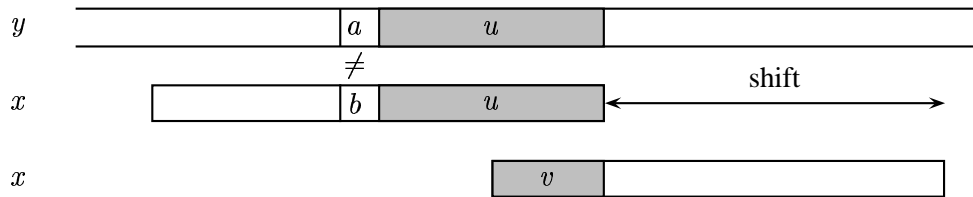


Figure 6.7: good-suffix shift, only a suffix of  $u$  reappears as a prefix of  $x$ .

“search” and “substitute” commands.

The algorithm scans the characters of the pattern from right to left beginning with the rightmost 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  $x[j] = b$  of the pattern and the character  $y[i+j] = a$  of the text during an attempt at position  $i$ . Then,  $y[i+j+1 \dots i+m-1] = x[j+1 \dots m-1] = u$  and  $y[i+j] \neq x[j]$ . The good-suffix shift consists of aligning the **segment**  $y[i+j+1 \dots i+m-1] = x[j+1 \dots m-1]$  with its rightmost occurrence in  $x$  that is preceded by a character different from  $x[j]$  (see Figure 6.6). If there exists no such segment, the shift consists of aligning the longest suffix  $v$  of  $y[i+j+1 \dots i+m-1]$  with a matching prefix of  $x$  (see Figure 6.7).

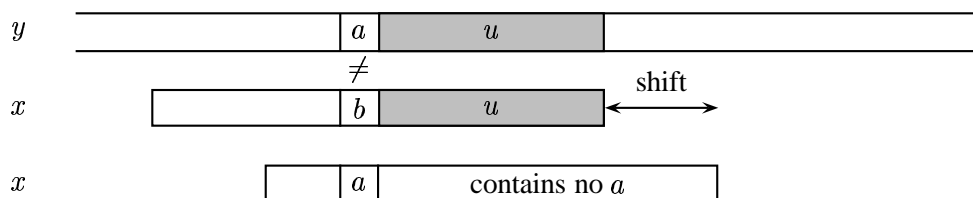


Figure 6.8: bad-character shift,  $a$  appears in  $x$ .

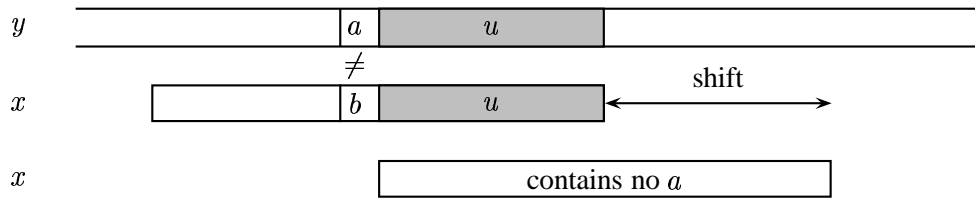


Figure 6.9: bad-character shift,  $a$  does not appear in  $x$ .

**Example 6.3:**

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

The shift is driven by the suffix  $abba$  of  $x$  found in the text. After the shift, the segment  $abba$  in the middle of  $y$  matches a segment of  $x$  as in Figure 6.6. The same mismatch does not recur.

**Example 6.4:**

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

The segment  $abba$  found in  $y$  partially matches a prefix of  $x$  after the shift, like in Figure 6.7.

The bad-character shift consists of aligning the text character  $y[i + j]$  with its rightmost occurrence in  $x[0 . . m - 2]$  (see Figure 6.8). 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 Figure 6.9).

**Example 6.5:**

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

The shift aligns the symbol  $a$  in  $x$  with the mismatch symbol  $a$  in the text  $y$  (Figure 6.8).

**Example 6.6:**

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

The shift positions the left end of  $x$  right after the symbol  $a$  of  $y$  (Figure 6.9).

The Boyer-Moore algorithm is shown in Figure 6.10. 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  $\sigma$  and the good-suffix shift is stored in a table  $gs$  of size  $m + 1$ . For  $a \in \Sigma$ :

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

```

void BM(char *y, char *x, int n, int m) {
    /* XSIZE is the maximum size of a pattern */
    /* ASIZE is the size of the alphabet      */
    int i, j, gs[XSIZE], bc[ASIZE];

    /* Preprocessing */
    PRE_GS(x, m, gs);
    PRE_BC(x, m, bc);

    /* Searching */
    i=0;
    while (i <= n-m) {
        j=m-1;
        while (j >= 0 && x[j] == y[i+j]) j--;
        if (j < 0) OUTPUT(i);
        i+=MAX(gs[j+1], bc[y[i+j]]-m+j+1);      /* shift */
    }
}

```

Figure 6.10: The Boyer-Moore string-matching algorithm.

```

void PRE_BC(char *x, int m, int bc[]) {
    /* ASIZE is the size of the alphabet */
    int j;

    for (j=0; j < ASIZE; j++) bc[j]=m;
    for (j=0; j < m-1; j++) bc[x[j]]=m-j-1;
}

```

Figure 6.11: Computation of the bad-character shift.

Let us define two conditions:

$$\begin{aligned}
 \text{cond}_1(j, s) &: \text{ for each } k \text{ such that } j < k < m, s \geq k \text{ or } x[k-s] = x[k] \\
 \text{cond}_2(j, s) &: \text{ if } s < j \text{ then } x[j-s] \neq x[j]
 \end{aligned}$$

Then, for  $0 \leq j < m$ :

$$gs[j+1] = \min\{s > 0 / \text{cond}_1(j, s) \text{ and } \text{cond}_2(j, s) \text{ hold}\}$$

and we define  $gs[0]$  as the length of the smallest period of  $x$ .

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 Figures 6.12 and 6.11). 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  $a^{m-1}b$  in  $a^n$  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.

```

void PRE_GS(char *x, int m, int gs[]) {
    /* XSIZE is the maximum size of a pattern */
    int i, j, p, f[XSIZE];

    for (i=0; i <= m; i++) gs[i]=0;
    f[m]=j=m+1;
    for (i=m; i > 0; i--) {
        while (j <= m && x[i-1] != x[j-1]) {
            if (!gs[j]) gs[j]=j-i;
            j=f[j];
        }
        f[i-1]=--j;
    }
    p=f[0];
    for (j=0; j <= m; j++) {
        if (!gs[j]) gs[j]=p;
        if (j == p) p=f[p];
    }
}

```

Figure 6.12: Computation of the good-suffix shift.

### 6.2.4 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 the ASCII table and ordinary searches made under a text editor, it becomes very useful. Using it only produces a very efficient algorithm in practice that is described now.

After an attempt where  $x$  is aligned with  $y[i \dots i + m - 1]$ , the length of the shift is at least equal to one. So, the character  $y[i + m]$  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 ( $a \in \Sigma$ ):

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

Indeed, the comparisons between text and pattern characters during each attempt can be done in any order. The algorithm of Figure 6.13 performs the comparisons from left to right. It is called Quick Search after its inventor and has a quadratic worst-case time complexity but a good practical behavior.

#### Example 6.7:

```

y = s t r i n g - m a t c h i n g
x = i n g
x =           i n g
x =                   i n g
x =                               i n g
x =                                     i n g

```

Quick Search algorithm makes 9 comparisons to find the two occurrences of `ing` inside the text of length 15.

```

void QS(char *y, char *x, int n, int m) {
    /* ASIZE is the size of the alphabet */
    int i, j, bc[ASIZE];

    /* Preprocessing */
    for (j=0; j < ASIZE; j++) bc[j]=m;
    for (j=0; j < m; j++) bc[x[j]]=m-j-1;

    /* Searching */
    i=0;
    while (i <= n-m) {
        j=0;
        while (j < m && x[j] == y[i+j]) j++;
        if (j >= m) OUTPUT(i);
        i+=bc[y[i+m]]+1;           /* shift */
    }
}

```

Figure 6.13: The Quick Search string-matching algorithm.

### 6.2.5 Experimental results

In Figures 6.14 and 6.15 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 6.4).

Tests have been performed on various types of texts. In Figure 6.14 we show the result when the text is a DNA sequence on the four-letter alphabet of nucleotides {A, C, G, T}. In Figure 6.15 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 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 it 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.

### 6.2.6 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 of repeating some string-matching algorithm for each pattern. If the set contains

Figure 6.14: Running times for a DNA sequence.

$k$  patterns, this search runs in time  $O(kn)$ . The solution described in the present section and designed by Aho and Corasick run 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  $X = \{x_0, x_1, \dots, x_{k-1}\}$  be the set of patterns, and let  $|X| = |x_0| + |x_1| + \dots + |x_{k-1}|$  be the total size of the set  $X$ . The Aho-Corasick algorithm first consists of building a **trie**  $T(X)$ , 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  $\epsilon$ . 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 PRE-AC in Figure 6.16 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 Figure 6.17).

Finally, the last phase of function PRE-AC (Figure 6.16) consists of building the failure link of each node of the trie, and simultaneously completing the output function. This is done by the function COMPLETE in Figure 6.18. The failure function  $fail$  is defined on nodes as follows ( $w$  is a node):

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

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:

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

Figure 6.15: Running times for an english text.

```

PRE-AC ( $X, k$ )
1  create a new node  $root$ 
   /* creates loops on the root of the trie */
2  for each  $a \in \Sigma$ 
3    do  $child(root, a) \leftarrow root$ 
   /* enters each pattern in the trie */
4  for  $i \leftarrow 0$  to  $k - 1$ 
5    do ENTER ( $X[i], root$ )
   /* completes the trie with failure links */
6  COMPLETE ( $root$ )
7  return  $root$ 

```

Figure 6.16: Preprocessing phase of the Aho-Corasick algorithm.

```

ENTER ( $x, root$ )
1   $r \leftarrow root$ 
2   $i \leftarrow 0$ 
   /* follows the existing edges */
3  while  $i < length(x)$  and  $child(r, x[i]) \neq \text{UNDEFINED}$  and  $child(r, x[i]) \neq root$ 
4      do  $r \leftarrow child(r, x[i])$ 
5           $i \leftarrow i + 1$ 
   /* creates new edges */
6  while  $i < length(x)$ 
7      do create a new node  $s$ 
8           $child(r, x[i]) \leftarrow s$ 
9           $r \leftarrow s$ 
10          $i \leftarrow i + 1$ 
11  $out(r) \leftarrow x$ 

```

Figure 6.17: Construction of the trie.

```

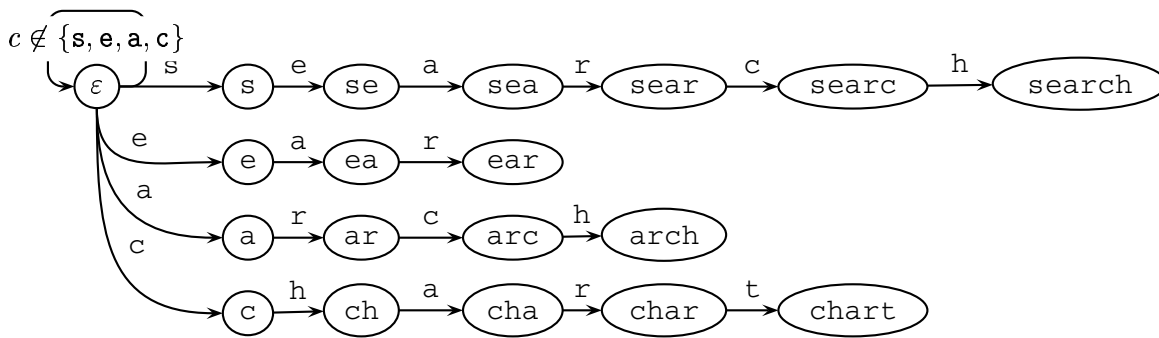
COMPLETE ( $root$ )
1   $q \leftarrow$  empty queue
2   $l \leftarrow$  list of the edges ( $root, a, p$ ) for any character  $a \in \Sigma$  and any node  $p \neq root$ 
3  while the list  $l$  is not empty
4      do ( $r, a, p$ )  $\leftarrow$  FIRST ( $l$ )
5           $l \leftarrow$  NEXT ( $l$ )
6          ENQUEUE ( $q, p$ )
7           $fail(p) \leftarrow root$ 
8  while the queue  $q$  is not empty
9      do  $r \leftarrow$  DEQUEUE ( $q$ )
10          $l \leftarrow$  list of the edges ( $r, a, p$ ) for any character  $a \in \Sigma$  and any node  $p$ 
11         while the list  $l$  is not empty
12             do ( $r, a, p$ )  $\leftarrow$  FIRST ( $l$ )
13                  $l \leftarrow$  NEXT ( $l$ )
14                 ENQUEUE ( $q, p$ )
15                  $s \leftarrow fail(r)$ 
16                 while  $child(s, a) = \text{UNDEFINED}$ 
17                     do  $s \leftarrow fail(s)$ 
18                  $fail(p) \leftarrow child(s, a)$ 
19                  $out(p) \leftarrow out(p) \cup out(child(s, a))$ 

```

Figure 6.18: Completion of the output function and construction of failure links.

**Example 6.8:**

$$X = \{\text{search, ear, arch, chart}\}$$



nodes	$\epsilon$	s	se	sea	sear	searc	search	e	ea	ear
fail	$\epsilon$	$\epsilon$	e	ea	ear	arc	arch	$\epsilon$	a	ar
nodes	a	ar	arc	arch	c	ch	cha	char	chart	
fail	$\epsilon$	$\epsilon$	c	ch	$\epsilon$	$\epsilon$	a	ar	$\epsilon$	

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

In order to stop going back with failure links during the computation of the failure links, and also in order to pass 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 PRE-AC.

After the preprocessing phase is completed, the searching phase consists of 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 non-empty 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 Figure 6.19. 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  $r \leftarrow \text{fail}(r)$  in Figure 6.19 is the exact analogue of instruction  $j = \text{next}[j]$  in Figure 6.4. A unified view of both algorithms exists but is out of 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 the access to children of nodes.

### 6.3 Two-dimensional pattern matching algorithms

In this section only we consider two-dimensional arrays. Arrays may be thought of as bitmap 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 6.2 can be extended to operate on arrays.

```

AC (y, n, X, k)
  /* Preprocessing */
1  r ← PRE-AC (X, k);
  /* Searching */
2  for i ← 0 to n − 1
3    do while child(r, y[i]) = UNDEFINED
4      do r ← fail(r);
5      r ← child(r, y[i]);
6      if out(r) ≠ ∅
7        then OUTPUT (out(r), i);

```

Figure 6.19: The complete Aho-Corasick algorithm.

```

/* YSIZE is the maximum size for a BIG IMAGE */
/* XSIZE is the maximum size for a SMALL IMAGE */
typedef char BIG_IMAGE [YSIZE] [YSIZE];
typedef char SMALL_IMAGE [XSIZE] [XSIZE];

void BF_2D(BIG_IMAGE y, SMALL_IMAGE x, int n1, int n2, int m1, int m2) {
  int i, j, k;

  /* Searching */
  for (i=0; i <= n1-m1; i++)
    for (j=0; j <= n2-m2; j++) {
      k=0;
      while (k < m1 && strcmp(&y[i+k][j], x[k], m2) == 0) k++;
      if (k >= m1) OUTPUT(i, j);
    }
}

```

Figure 6.20: The brute force two-dimensional pattern matching algorithm.

The problem is now to locate all occurrences of a two-dimensional pattern  $x = x[0 \dots m_1 - 1, 0 \dots m_2 - 1]$  of size  $m_1 \times m_2$  inside a two-dimensional text  $y = [0 \dots n_1 - 1, 0 \dots n_2 - 1]$  of size  $n_1 \times n_2$ . The brute force algorithm for this problem is given in Figure 6.20. It consists of checking at all positions of  $y[0 \dots n_1 - m_1, 0 \dots n_2 - m_2]$  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 (Section 6.2.1). 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.

### 6.3.1 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. The idea to do that 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 ( $0 \leq i < m_2$ ):

$$x'[i] = \text{hash}(x[0, i]x[1, i] \dots x[m_1 - 1, i])$$

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

$$y'[i] = \text{hash}(y[0, i]y[1, i] \dots y[m_1 - 1, i]).$$

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 ( $0 \leq i < m_2$ ):

$$\begin{aligned} &\text{hash}(y[j + 1, i]y[j + 2, i] \dots y[j + m_1, i]) = \\ &\text{rehash}(y[j, i], y[j + m_1, i], \text{hash}(y[j, i]y[j + 1, i] \dots y[j + m_1 - 1, i])) \end{aligned}$$

(functions *hash* and *rehash* are described in Section 6.2.1).

```

void KMP_IN_LINE(BIG_IMAGE Y, SMALL_IMAGE X, int YB[], int XB[],
                int n2, int m1, int m2, int next[], int row) {
    int i, j;

    i=j=0;
    while (j < n2) {
        while (i > -1 && XB[i] != YB[j]) i=next[i];
        i++; j++;
        if (i >= m2) {
            DIRECT_COMPARE(Y, X, m1, m2, row, j-1);
            i=next[m2];
        }
    }
}

```

Figure 6.21: Search for  $x'$  in  $y'$  using KMP algorithm.

```

void DIRECT_COMPARE(BIG_IMAGE Y, SMALL_IMAGE X, int m1, int m2,
                  int row, int column) {
    int i, j, i0, j0;

    i0=row-m1+1;
    j0=column-m2+1;
    for (i=0; i < m1; i++)
        for (j=0; j < m2; j++)
            if (X[i][j] != Y[i0+i][j0+j]) return;
    OUTPUT(i0, j0);
}

```

Figure 6.22: Naive check of an occurrence of  $x$  in  $y$  at position  $(row, column)$ .**Example 6.9:**

$x =$	a	a	a	$y =$	a	b	a	b	a	b	b
	b	b	a		a	a	a	a	b	b	b
	a	a	b		b	b	a	a	a	a	b
					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
--------	-----	-----	-----	--------	-----	-----	-----	-----	-----	-----	-----

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 Figure 6.21.

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 Figure 6.22.

```

#define REHASH(a,b,h) (((h-a*d)<<1)+b)

void ZT(BIG_IMAGE Y, SMALL_IMAGE X, int n1, int n2, int m1, int m2) {
    int YB[YSIZE], XB[XSIZE], next[XSIZE], j, i, row, d;

    /* Preprocessing */
    /* Computes the first value of y' */
    for (j=0; j < n2; j++) {
        YB[j]=0;
        for (i=0; i < m1; i++) YB[j]=(YB[j]<<1)+Y[i][j];
    }

    /* Computes x' */
    for (j=0; j < m2; j++) {
        XB[j]=0;
        for (i=0; i < m1; i++) XB[j]=(XB[j]<<1)+X[i][j];
    }

    row=m1-1;
    /* computes d=2^(m1-1) using the left shift operator */
    d=1;
    for (j=1; j < m1; j++) d<<=1;

    PRE_KMP(XB, m2, next);

    /* Searching */
    while (row < n1) {
        KMP_IN_LINE(Y, X, YB, XB, n2, m1, m2, next, row);
        if (row < n1-1)
            for (j=0; j < n2; j++)
                YB[j]=REHASH(Y[row-m1+1][j], Y[row+1][j], YB[j]);
        row++;
    }
}

```

Figure 6.23: The Zhu-Takaoka two-dimensional pattern matching algorithm.

The Zhu-Takaoka algorithm as explained above is displayed in Figure 6.23. The search for the pattern is performed row by row starting at row 0 and ending at row  $n_1 - m_1$ .

### 6.3.2 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 algorithm. The pattern  $x$  is divided into its  $m_1$  rows  $R_0 = x[0, 0 \dots m_2 - 1]$  to  $R_{m_1-1} = x[m_1 - 1, 0 \dots m_2 - 1]$ . The rows are preprocessed into a trie as in the Aho-Corasick algorithm (Section 6.2.6).

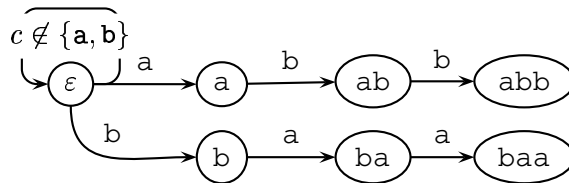
#### Example 6.10:

The trie of rows of pattern  $x$ .

```

PRE-KMP-FOR-B ( $X, m_1, next$ )
1   $i \leftarrow 0$ 
2   $next[0] \leftarrow -1$ 
3   $j \leftarrow -1$ 
4  while  $i < m_1$ 
5      do while  $j > -1$  and  $X[i, 0 \dots m_2 - 1] \neq X[j, 0 \dots m_2 - 1]$ 
6          do  $j \leftarrow next[j]$ 
7           $i \leftarrow i + 1$ 
8           $j \leftarrow j + 1$ 
9          if  $X[i, 0 \dots m_2 - 1] = X[j, 0 \dots m_2 - 1]$ 
10             then  $next[i] \leftarrow next[j]$ 
11             else  $next[i] \leftarrow j$ 

```

Figure 6.24: Computes the function  $next$  for rows of  $X$ .
$$x = \begin{array}{|c|c|c|} \hline b & a & a \\ \hline a & b & b \\ \hline b & a & a \\ \hline \end{array}$$


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  $y[i, j]$  the algorithm checks whether the portion  $y[i, j - m_2 + 1 \dots j] = R$  matches any of  $R_0, \dots, R_{m_1 - 1}$  using the Aho-Corasick machine. An additional one-dimensional array  $a$  of size  $n_1$  is used as follows:

$a[j] = k$  means that the  $k - 1$  first rows  $R_0, \dots, R_{k-2}$  of the pattern match respectively the portions of the text:  $y[i - k + 1, j - m_2 + 1 \dots j], \dots, y[i - 1, j - m_2 + 1 \dots j]$ .

Then, if  $R = R_{k-1}$ ,  $a[j]$  is incremented to  $k + 1$ . If not,  $a[j]$  is set to  $s + 1$  where  $s$  is the maximum  $i$  such that:

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

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

The Bird/Baker algorithm is presented in Figures 6.24 and 6.25. It runs in time  $O((n_1 n_2 + m_1 m_2) \log \sigma)$ .

```

B ( $Y, n_1, n_2, X, m_1, m_2$ )
  /* Preprocessing */
1  for  $i \leftarrow 0$  to  $n_2 - 1$ 
2    do  $a[i] \leftarrow 0$ 
3   $root \leftarrow$  PRE-AC (set of lines of  $X, m_1$ )
4  PRE-KMP-FOR-B ( $X, m_1, next$ )
  /* Searching */
5  for  $row \leftarrow 0$  to  $n_1 - 1$ 
6    do  $r \leftarrow root$ 
7      for  $column \leftarrow 0$  to  $n_2 - 1$ 
8        do while  $child(r, Y[row, column]) =$  UNDEFINED
9          do  $r \leftarrow fail(r)$ 
10          $r \leftarrow child(r, Y[row, column])$ 
11         if  $out(r) \neq \emptyset$ 
12           then  $k \leftarrow a[column]$ 
13             while  $k > 0$  and  $X[k, 0 \dots m_2 - 1] = out(r)$ 
14               do  $k \leftarrow next[k]$ 
15                $a[column] \leftarrow k + 1$ 
16               if  $a[column] = m_1$ 
17                 then OUTPUT ( $row - m_1 + 1, column - m_2 + 1$ )
18             else  $a[column] \leftarrow 0$ 

```

Figure 6.25: The Bird/Baker two-dimensional pattern matching algorithm.

```

SUFFIX-TREE ( $y, n$ )
1   $T_{-1} \leftarrow$  one-node tree
2  for  $i \leftarrow 0$  to  $n - 1$ 
3      do  $T_i \leftarrow$  INSERT ( $T_{i-1}, y[i \dots n - 1]$ )
4  return  $T_{n-1}$ 

```

Figure 6.26: Construction of a suffix tree for  $y$ .

## 6.4 Suffix trees

The suffix tree  $S(y)$  of a string  $y$  is a trie (see Section 6.2.6) containing all the suffixes of the string, and having the properties described below. 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:

- 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 above segments are represented by their starting positions and their lengths in  $y$ .

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 non-empty 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 non-empty suffixes of  $y$  in the data structure from the longest to the shortest suffix as shown in Figure 6.26. We introduce two definitions in order to explain how the algorithm works:

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

The strategy to insert the  $i$ -th suffix in the tree is based on these definitions and described in Figure 6.27.

The second step of the insertion (Figure 6.27) 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 of spelling the current suffix  $y[i \dots n - 1]$  from the root of the tree, giving an  $O(|head_i|)$  time complexity for the insertion at step  $i$ , and an  $O(n^2)$  running time to build  $S(y)$ . Adding ‘short-cut’

INSERT ( $T_{i-1}, y[i \dots n - 1]$ )

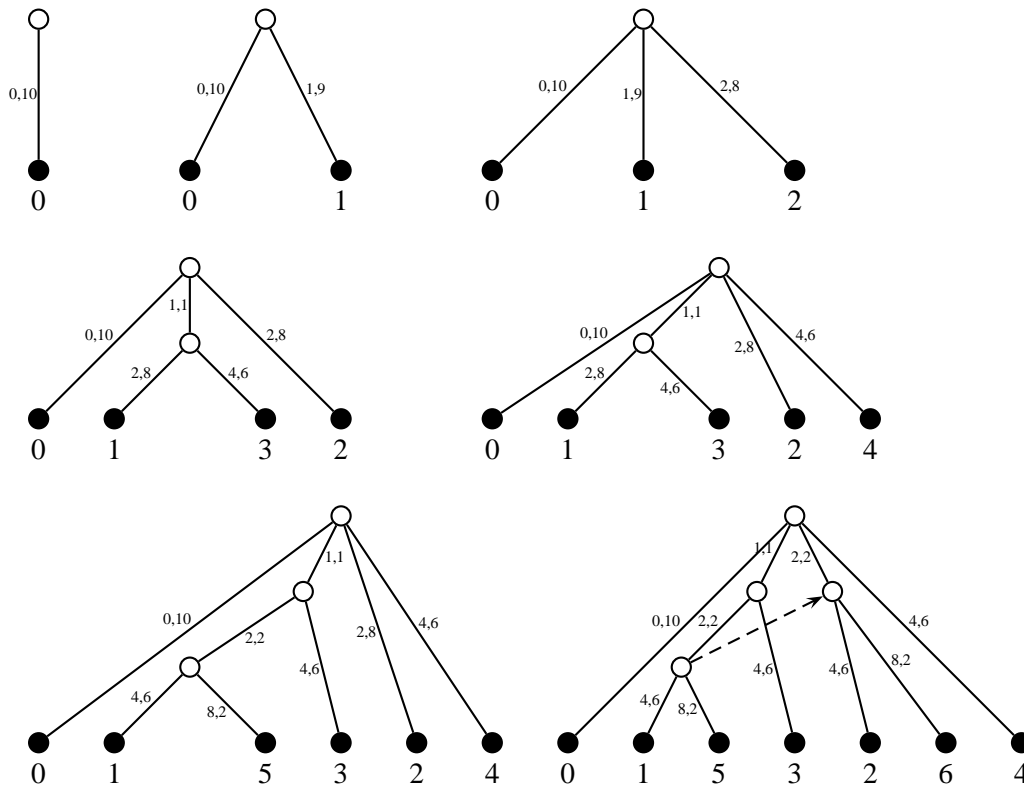
- 1 locate the node  $h$  associated with  $head_i$  in  $T_{i-1}$ , possibly breaking an edge
- 2 add a new edge labeled  $tail_i$  from  $h$  to a new leaf representing suffix  $y[i \dots n - 1]$
- 3 **return** the modified tree

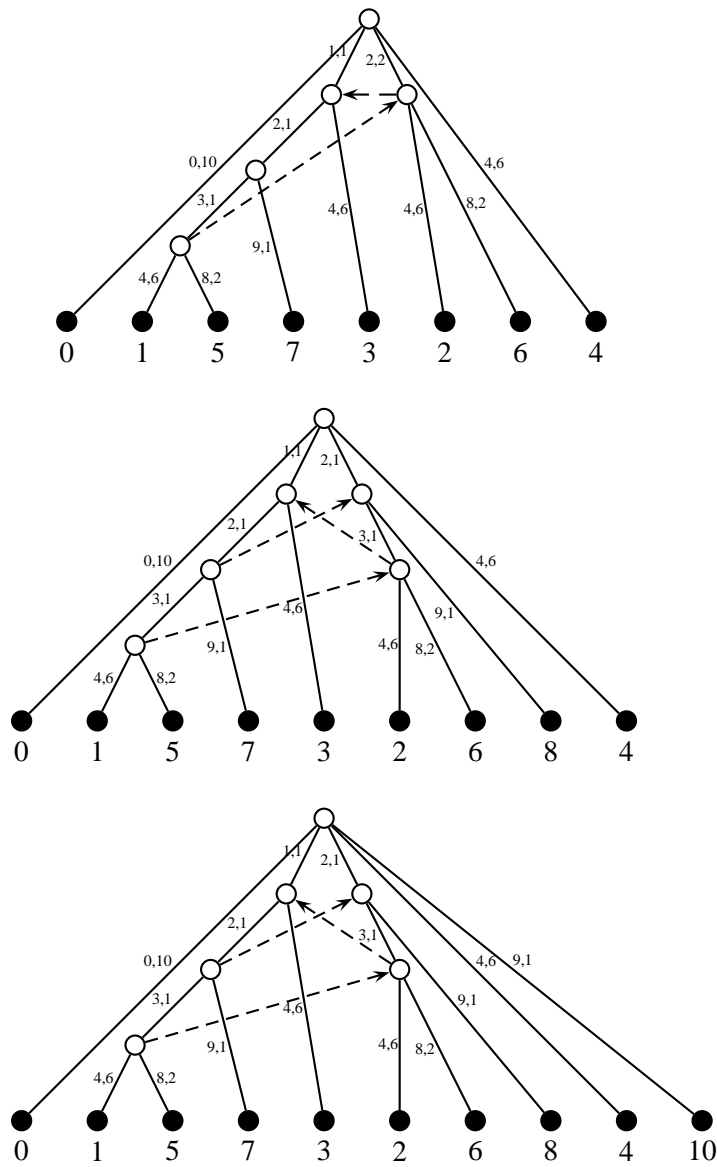
Figure 6.27: Insertion of a new suffix in the tree.

links leads to an overall  $O(n)$  time complexity, although there is no guaranty that insertion at step  $i$  is realized in constant time.

**Example 6.11:**

The different tries during the construction of the suffix tree of  $y = \text{CAGATAGAG}\$$ . Leaves are black and labeled by the position of the suffix they represent. Plain arrows are labeled by pairs: the pair  $(i, l)$  stands for the segment  $y[i \dots i + l - 1]$ . Dashed arrows represent the non-trivial suffix links.





### 6.4.1 McCreight algorithm

The clue 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  $head_{i-1}$  and  $head_i$ :

if  $head_{i-1}$  is of the form  $az$  ( $a \in \Sigma$ ,  $z \in \Sigma^*$ ),  
then  $z$  is a prefix of  $head_i$ .

In the suffix tree the node associated with  $z$  is linked to the node associated with  $az$ . The suffix link creates a short-cut in the tree that helps in finding the next head efficiently. The insertion of the next suffix, namely  $head_i tail_i$ , in the tree reduces to the insertion of  $tail_i$  from the node associated with  $head_i$ .

The following property is an invariant of the construction: in  $T_i$ , only the node  $h$  associated with  $head_i$  can fail to have a valid suffix link. This effectively happens when  $h$  has just been created at step

*i*. The procedure to find the next head at step *i* is composed of two main phases:

#### A Rescanning

Assume that  $head_{i-1} = az$  ( $a \in \Sigma, z \in \Sigma^*$ ) 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  $(i, l)$  be the label of edge  $(c', d')$ . For the ease of the description, assume that  $az = av(y[i \dots i + l - 1])$  (it may happen that  $az = y[i \dots i + l - 1]$ ). There is a suffix link defined on  $c'$  and going to some node  $c$  associated with  $v$ . The crucial observation here is that  $y[i \dots i + l - 1]$  is the prefix of the label of some branch starting at node  $c$ . Then, the algorithm rescans  $y[i \dots i + l - 1]$  in the tree: let  $e$  be the child of  $c$  along that branch, and let  $(j, m)$  be the label of edge  $(c, e)$ . If  $m < l$  then a recursive rescan of  $q = y[i + m \dots i + l - 1]$ , starts from node  $e$ . If  $m > l$ , the edge  $(c, e)$  is broken to insert a new node  $d$ ; labels are updated correspondingly. If  $m = l$ ,  $d$  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  $head_i$ . The search is dictated by the characters of  $tail_{i-1}$  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 Figure 6.28. The next figures 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 a pair  $(i, l)$  if node  $c$  is associated with the factor  $y[i \dots i + l - 1]$ ,
- $child(c, a)$  is the only node that can be reached from the node  $c$  with the character  $a$ ,
- $link(c)$  is the suffix node of the node  $c$ .

## 6.5 Longest common subsequence of two strings

The notion of a longest common subsequence of two strings is widely used to compare files. The `diff` command of UNIX operating system implements an algorithm based of 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 (insert a symbol, or delete a symbol) to transform one file into the other, which introduces what is known as the **edit distance** between the strings (see Section 6.6). The comparison of

```

M(y, n)
1  root ← INIT(y, n)
2  head ← root
3  tail ← child(root, y[0])
4  n ← n - 1
5  while n > 0
6      do /* Phase A (rescanning) */
7          if head = root
8              then d ← root
9                   (i, l) ← label(tail)
10                  γ ← (i + 1, l - 1)
11             else γ ← label(tail)
12                  if link(head) ≠ UNDEFINED
13                     then d ← link(head)
14                     else (i, l) ← label(head)
15                         if parent(head) = root
16                            then d ← RESCAN(root, i + 1, l - 1)
17                            else d ← RESCAN(link(parent(head)), i, l)
18                  link(head) ← d
19             /* Phase B (scanning) */
20             (head, γ) ← SCAN(d, γ)
21             create a new node tail
22             parent(tail) ← head
23             label(tail) ← γ
24             (i, l) ← γ
25             child(head, y[i]) ← tail
26             n ← n - 1
27  return root

```

Figure 6.28: Suffix tree construction.

```

INIT(y, n)
1  create a new node root
2  create a new node c
3  parent(root) ← UNDEFINED
4  parent(c) ← root
5  child(root, y[0]) ← c
6  label(root) ← UNDEFINED
7  label(c) ← (0, n)
8  return root

```

Figure 6.29: Initialization procedure.

```

RESCAN ( $c, i, l$ )
1  ( $j, m$ )  $\leftarrow$  label( $child(c, y[i])$ )
2  while  $l > 0$  and  $l \geq m$ 
3      do  $c \leftarrow child(c, y[i])$ 
4           $l \leftarrow l - m$ 
5           $i \leftarrow i + m$ 
6          ( $j, m$ )  $\leftarrow$  label( $child(c, y[i])$ )
7  if  $l > 0$ 
8      then return BREAK-EDGE ( $child(c, y[i]), l$ )
9      else return  $c$ 

```

Figure 6.30: The crucial rescan operation.

```

BREAK-EDGE ( $c, k$ )
1  create a new node  $g$ 
2   $parent(g) \leftarrow parent(c)$ 
3  ( $i, l$ )  $\leftarrow$  label( $c$ )
4   $child(parent(c), y[i]) \leftarrow g$ 
5   $label(g) \leftarrow (i, k)$ 
6   $parent(c) \leftarrow g$ ;
7   $label(c) \leftarrow (i + k, l - k)$ 
8   $child(g, y[i + k]) \leftarrow c$ 
9   $link(g) \leftarrow$  UNDEFINED
10 return  $g$ 

```

Figure 6.31: Breaking an edge.

```

SCAN ( $d, \gamma$ )
1  ( $i, l$ )  $\leftarrow \gamma$ 
2  while  $child(d, y[i]) \neq \text{UNDEFINED}$ 
3      do  $g \leftarrow child(d, y[i])$ 
4           $k \leftarrow 1$ 
5           $(s, lg) \leftarrow label(g)$ 
6           $s \leftarrow s + 1$ 
7           $l \leftarrow l - 1$ 
8           $i \leftarrow i + 1$ 
9          while  $k < lg$  and  $y[i] = y[s]$ 
10             do  $i \leftarrow i + 1$ 
11                  $s \leftarrow s + 1$ 
12                  $k \leftarrow k + 1$ 
13                  $l \leftarrow l - 1$ 
14         if  $k < lg$ 
15             then return (BREAK-EDGE ( $g, k$ ), ( $i, l$ ))
16          $d \leftarrow g$ 
17 return ( $d, (i, l)$ )

```

Figure 6.32: The scan operation.

molecular sequences is basically done with a closed concept, the alignment of strings, which consists of aligning their symbols on vertical lines. This is related to an edit distance with the additional operation of substitution.

A subsequence of a word  $x$  is obtained by deleting zero or more characters from  $x$ . More formally  $w[0 \dots i - 1]$  is a subsequence of  $x[0 \dots m - 1]$  if there exists an increasing sequence of integers ( $k_j / j = 0, \dots, i - 1$ ) 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  $lcs(x, y)$ . Their common length is denoted by  $llcs(x, y)$ .

A brute-force method to compute an  $lcs(x, y)$  would consist of 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, so this method could take  $O(2^m)$  time, which is impracticable even for fairly small values of  $m$ .

### 6.5.1 Dynamic programming

The commonly-used algorithm to compute an  $lcs(x, y)$  is a typical application of the dynamic programming method. Decomposing the problem into subproblems produces wide overlaps between them. So memorization of intermediate values is necessary to avoid recomputing them many times. Using dynamic programming it is possible to compute an  $lcs(x, y)$  in  $O(mn)$  time and space. The method naturally leads to computing  $lcs$ 's for longer and longer prefixes of the two words. To do so, we consider the two-dimensional table  $L$  defined by:

$$L[i, 0] = L[0, j] = 0, \text{ for } 0 \leq i \leq m \text{ and } 0 \leq j \leq n, \text{ and}$$

$$L[i + 1, j + 1] = llcs(x[0 \dots i], y[0 \dots j]), \text{ for } 0 \leq i \leq m - 1 \text{ and } 0 \leq j \leq n - 1.$$

```

int LCS(char *x, char *y, int m, int n, int L[YSIZE][YSIZE]) {
    int i, j;

    for (i=0; i <= m; i++) L[i][0]=0;
    for (j=0; j <= n; j++) L[0][j]=0;

    for (i=0; i < m; i++)
        for (j=0; j < n; j++)
            if (x[i] == y[j]) L[i+1][j+1]=L[i][j]+1;
            else L[i+1][j+1]=MAX(L[i+1][j], L[i][j+1]);
    return L[m][n];
}

```

Figure 6.33: Dynamic programming algorithm to compute  $llcs(x, y) = L[m, n]$ .

```

char *TRACE(char *x, char *y, int m, int n, int L[YSIZE][YSIZE]) {
    int i, j, l;
    char z[YSIZE];

    i=m; j=n; l=L[m][n];
    z[l--]='\0';
    while (i > 0 && j > 0) {
        if (L[i][j] == L[i-1][j-1]+1 && x[i-1] == y[j-1]) {
            z[l--]=x[i-1];
            i--; j--;
        }
        else if (L[i-1][j] > L[i][j-1]) i--;
        else j--;
    }
    return(z);
}

```

Figure 6.34: Production of an  $lcs(x, y)$ .

Computing  $llcs(x, y) = L[m, n]$  relies on a basic observation that yields the simple recurrence relation ( $0 \leq i < m, 0 \leq j < n$ ):

$$L[i+1, j+1] = \begin{cases} L[i, j] + 1 & \text{if } x[i] = y[j], \\ \max(L[i, j+1], L[i+1, j]) & \text{otherwise.} \end{cases}$$

The relation is used by the algorithm of Figure 6.33 to compute all the values from  $L[0, 0]$  to  $L[m, n]$ . The computation takes  $O(mn)$  time and space. It is afterward possible to trace back a path from  $L[m, n]$  to exhibit an  $lcs(x, y)$  (see Figure 6.34).

```

int LLCS(char *x, char *y, int m, int n, int *L) {
    int i, j, last;

    for (j=0; j <= n; j++) L[j]=0;
    for (i=0; i < m; i++) {
        last=0;
        for (j=0; j < n; j++)
            if (last > L[j+1]) L[j+1]=last;
            else if (last < L[j+1]) last=L[j+1];
            else if (x[i] == y[j]) {
                L[j+1]++;
                last++;
            }
    }
    return L[n];
}

```

Figure 6.35:  $O(\min(m, n))$ -space algorithm to compute  $llcs(x, y)$ .**Example 6.12:**

The value  $L[5, 9] = 4$  is  $llcs(x, y)$  for  $x = AGCGA$  and  $y = CAGATAGAG$ . String AGGA is an lcs of  $x$  and  $y$ .

$x$		C	A	G	A	T	A	G	A	G	
$y$	0	0	1	2	3	4	5	6	7	8	9
A	0	0	0	0	0	0	0	0	0	0	0
G	1	0	0	1	1	1	1	1	1	1	1
C	2	0	0	1	2	2	2	2	2	2	2
G	3	0	1	1	2	2	2	2	2	2	2
A	4	0	1	1	2	2	2	2	3	3	3
	5	0	1	2	2	3	3	3	3	4	4

**6.5.2 Reducing the space: Hirschberg algorithm**

If only the length of an  $lcs(x, y)$  is required, it is easy to see that only one row (or one column) of the table  $L$  needs to be stored during the computation. The space complexity becomes  $O(\min(m, n))$  as it can be checked on the algorithm of Figure 6.35. Indeed, the Hirschberg algorithm computes an  $lcs(x, y)$  in linear space and not only the value  $llcs(x, y)$ . The computation uses the algorithm of Figure 6.35.

Let us define:

$$L^*[i, n] = L^*[m, j] = 0, \text{ for } 0 \leq i \leq m \text{ and } 0 \leq j \leq n, \text{ and}$$

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

```

void LLCS_REVERSE(char *x, char *y, int a, int m, int n, int *Lstar) {
    int i, j, last;

    for (j=0; j <= n; j++) Lstar[j]=0;
    for (i=m-1; i >= a; i--) {
        last=0;
        for (j=n-1; j >= 0; j--)
            if (last > Lstar[n-j]) Lstar[n-j]=last;
            else if (last < Lstar[n-j]) last=Lstar[n-j];
            else if (x[i] == y[j]) {
                Lstar[n-j]++;
                last++;
            }
    }
}

```

Figure 6.36: Computation of  $L^*$ .

and

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

where the word  $w^R$  is the reverse (or mirror image) of the word  $w$ . The algorithm of Figure 6.36 compute the table  $L^*$ . The following property is the key observation to compute an  $lcs(x, y)$  in linear space:

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

In the algorithm shown in Figure 6.37 the integer  $i$  is chosen as  $m/2$ . After  $L[i, j]$  and  $L^*[m - i, n - j]$  ( $0 \leq j < m$ ) are computed, the algorithm finds an integer  $k$  such that  $L[i, k] + L^*[m - i, n - k] = L[m, n]$ . Then, recursively, it computes an  $lcs(x[0 \dots i], y[0 \dots k])$  and an  $lcs(x[i + 1 \dots m - 1], y[k + 1 \dots n - 1])$ , and concatenate them to get an  $lcs(x, y)$ .

The running time of the Hirschberg algorithm is still  $O(mn)$  but the amount of space required for the computation becomes  $O(\min(m, n))$  instead of being quadratic as in Section 6.5.1.

## 6.6 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 counts 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  $w_2$ ,

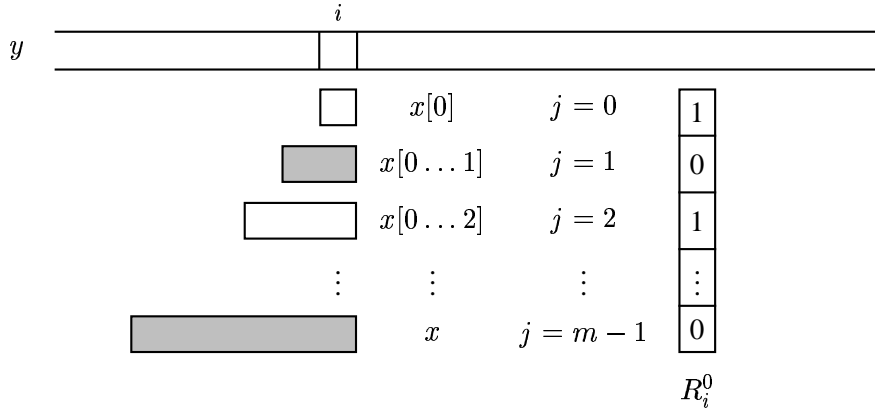
```

char *HIRSCHBERG(char *x, char *y, int m, int n) {
    int i, j, k, M;
    static char z[YSIZE];
    static int L[YSIZE], Lstar[YSIZE];
    static int count=0;

    if (m == 0) z[count]='\0';
    else if (m == 1) {
        for (i=0; i < n; i++)
            if (x[0] == y[i]) {
                z[count++]=x[0];
                z[count]='\0';
                return(z);
            }
        z[count]='\0';
    }
    else {
        i=m/2;
        LLCS(x, y, i, n, L);
        LLCS_REVERSE(x, y, i, m, n, Lstar);
        k=n;
        M=L[n]+Lstar[0];
        for (j=n-1; j >= 0; j--)
            if (L[j]+Lstar[n-j] >= M) {
                M=L[j]+Lstar[n-j];
                k=j;
            }
        HIRSCHBERG(x, y, i, k);
        HIRSCHBERG(x+i, y+k, m-i, n-k);
        z[count]='\0';
    }
    return(z);
}

```

Figure 6.37:  $O(\min(m, n))$ -space computation of  $lcs(x, y)$ .

Figure 6.38: Meaning of vector  $R_i^0$ .

- 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 two above 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  $k$  insertions, deletions, or substitutions. The method is flexible enough to be adapted to a wide range of similar approximate matching problems.

### 6.6.1 Shift-Or algorithm

We first present an algorithm to solve the exact string-matching problem using a technique different from those developed in Section 6.2, but which extends to the approximate string-matching problem.

Let  $R^0$  be a bit array of size  $m$ . Vector  $R_i^0$  is the value of the entire array  $R^0$  after text character  $y[i]$  has been processed (see Figure 6.38). It contains information about all matches of prefixes of  $x$  that end at position  $i$  in the text ( $0 \leq j \leq m - 1$ ):

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

Therefore,  $R_i^0[m - 1] = 0$  is equivalent to say that an (exact) occurrence of the pattern  $x$  ends at position  $i$  in  $y$ .

The vector  $R_i^0$  can be computed after  $R_{i-1}^0$  by the following recurrence relation:

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

and

$$R_i^0[0] = \begin{cases} 0 & \text{if } x[0] = y[i], \\ 1 & \text{otherwise.} \end{cases}$$

The transition from  $R_{i-1}^0$  to  $R_i^0$  can be computed very fast as follows. For each  $a \in \Sigma$ , let  $S_a$  be a bit array of size  $m$  defined by:

$$\text{for } 0 \leq j \leq m - 1, \quad S_a[j] = 0 \text{ iff } x[j] = a.$$

The array  $S_a$  denotes the positions of the character  $a$  in the pattern  $x$ . Each  $S_a$  can be preprocessed before the search. And the computation of  $R_i^0$  reduces to two operations, *SHIFT* and *OR*:

$$R_i^0 = \text{SHIFT}(R_{i-1}^0) \text{ OR } S_{y[i]}.$$

**Example 6.13:**

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

	$S_A$	$S_C$	$S_G$	$S_T$
	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

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

- There is an exact match on the first  $j$  characters of  $x$  up to  $y[i-1]$  (i.e.  $R_{i-1}^0[j-1] = 0$ ). Then, substituting  $y[i]$  to  $x[j]$  creates a match with one substitution (see Figure 6.39). Thus,

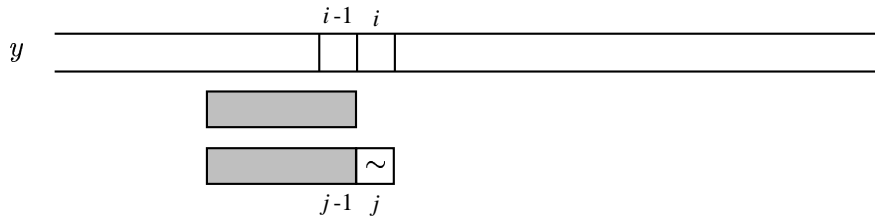
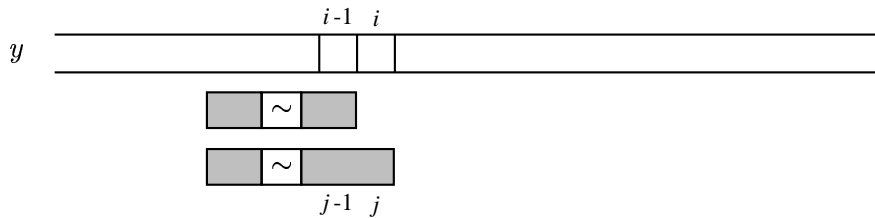
$$R_i^1[j] = R_{i-1}^0[j-1].$$

- There is a match with one substitution on the first  $j$  characters of  $x$  up to  $y[i-1]$  and  $y[i] = x[j]$ . Then, there is a match with one substitution of the first  $j+1$  characters of  $x$  up to  $y[i]$  (see Figure 6.40). Thus,

$$R_i^1[j] = \begin{cases} R_{i-1}^1[j-1] & \text{if } y[i] = x[j], \\ 1 & \text{otherwise.} \end{cases}$$

This implies that  $R_i^1$  can be updated from  $R_{i-1}^1$  by the relation:

$$R_i^1 = (\text{SHIFT}(R_{i-1}^1) \text{ OR } S_{y[i]}) \text{ AND } \text{SHIFT}(R_{i-1}^0)$$

Figure 6.39: If  $R_{i-1}^0[j-1] = 0$  then  $R_i^1[j] = 0$ .Figure 6.40:  $R_i^1[j] = R_{i-1}^1[j-1]$  if  $y[i] = x[j]$ .**Example 6.14:**

String  $x = \text{GATAA}$  occurs at positions 2 and 7 in  $y = \text{CAGATAAGAGAA}$  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

**6.6.3 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 to the case of only one deletion. The method is based on the following elements.

**One insertion allowed:** here, vector  $R_{i-1}^1$  indicates all matches with at most one insertion up to text character  $y[i-1]$ .

$R_{i-1}^1[j-1] = 0$  if the first  $j$  characters of  $x$  ( $x[0 \dots j-1]$ ) match  $j$  symbols of the last  $j+1$  text characters up to  $y[i-1]$ .

Array  $R^0$  is maintained as before, and we show how to maintain array  $R^1$ . Two cases can arise:

- There is an exact match on the first  $j$  characters of  $x$  ( $x[0 \dots j-1]$ ) up to  $y[i-1]$ . Then inserting  $y[i]$  creates a match with one insertion up to  $y[i]$  (see Figure 6.41). Thus

$$R_i^1[j] = R_{i-1}^0[j-1].$$

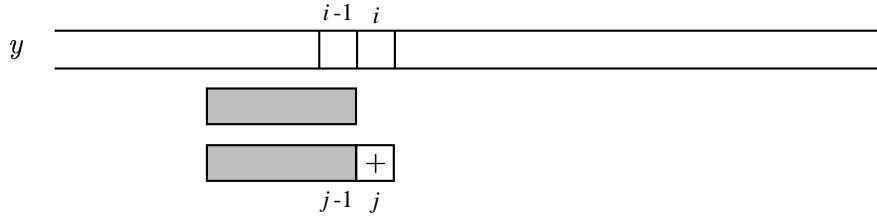


Figure 6.41: If  $R_{i-1}^0[j - 1] = 0$  then  $R_i^1[j] = 0$ .

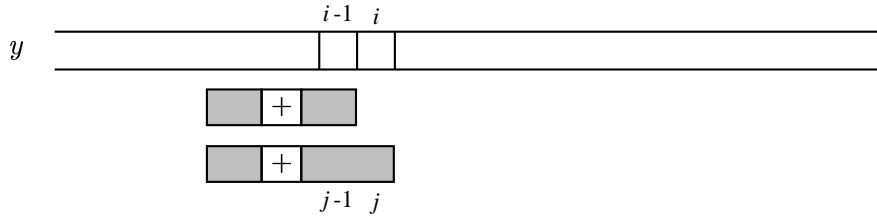


Figure 6.42:  $R_i^1[j] = R_{i-1}^1[j - 1]$  if  $y[i] = x[j]$ .

- There is a match with one insertion on the  $j$  first characters of  $x$  up to  $y[i - 1]$ . Then if  $y[i] = x[j]$  there is a match with one insertion on the first  $j + 1$  characters of  $x$  up to  $y[j]$  (see Figure 6.42). Thus,

$$R_i^1[j] = \begin{cases} R_{i-1}^1[j - 1] & \text{if } y[i] = x[j], \\ 1 & \text{otherwise.} \end{cases}$$

The above shows that  $R_i^1$  can be updated from  $R_{i-1}^1$  with the formula:

$$R_i^1 = (\text{SHIFT}(R_{i-1}^1) \text{ OR } S_{y[i]}) \text{ AND } R_{i-1}^0.$$

**Example 6.15:**

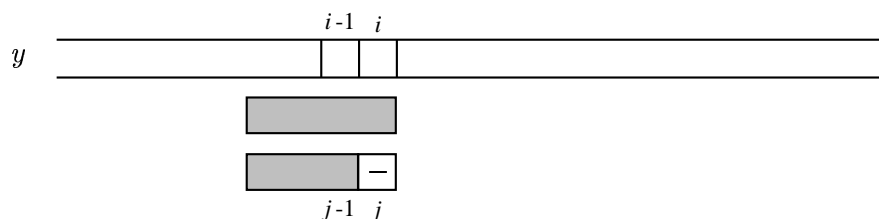
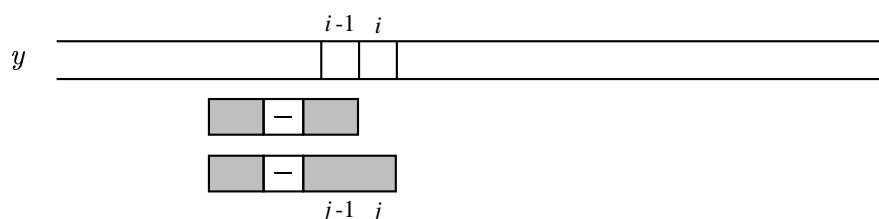
GATAAG is an occurrence of  $x = \text{GATAA}$  with one insertion in  $y = \text{CAGATAAGAGAA}$

	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 allowed:** We assume here that  $R_{i-1}^1$  indicates all possible matches with at most one deletion up to  $y[i - 1]$ . As in previous problems, two cases arise:

- There is an exact match on the first  $j + 1$  characters of  $x$  ( $x[0 \dots j]$ ) up to  $y[i]$  (i.e.  $R_i^0[j] = 0$ ). Then, deleting  $x[j]$  creates a match with one deletion (see Figure 6.43). Thus,

$$R_i^1[j] = R_i^0[j].$$

Figure 6.43: If  $R_i^0[j-1] = 0$  then  $R_i^1[j] = 0$ .Figure 6.44:  $R_i^1[j] = R_{i-1}^1[j-1]$  if  $y[i] = x[j]$ .

- There is a match with one deletion on the first  $j$  characters of  $x$  up to  $y[i-1]$  and  $y[i] = x[j]$ . Then, there is a match with one deletion on the first  $j+1$  characters of  $x$  up to  $y[i]$  (see Figure 6.44). Thus,

$$R_i^1[j] = \begin{cases} R_{i-1}^1[j-1] & \text{if } y[i] = x[j], \\ 1 & \text{otherwise.} \end{cases}$$

The discussion provides the following formula used to update  $R_i^1$  from  $R_{i-1}^1$ :

$$R_i^1 = (\text{SHIFT}(R_{i-1}^1) \text{ OR } S_{y[i]}) \text{ AND } \text{SHIFT}(R_i^0).$$

### Example 6.16:

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

	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

## 6.6.4 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 previously. The algorithm maintains  $k+1$  bit arrays  $R^0, R^1, \dots, R^k$  that are described now. The vector  $R^0$  is maintained similarly as in the exact matching case (Section 6.6.1). The other vectors

are computed with the formula ( $1 \leq j \leq k$ ):

$$R_i^j = (\text{SHIFT}(R_{i-1}^j) \text{ OR } S_{y[i]}) \\ \text{AND } \text{SHIFT}(R_i^{j-1}) \\ \text{AND } \text{SHIFT}(R_{i-1}^{j-1}) \\ \text{AND } R_{i-1}^{j-1},$$

which can be rewritten into:

$$R_i^j = (\text{SHIFT}(R_{i-1}^j) \text{ OR } S_{y[i]}) \\ \text{AND } \text{SHIFT}(R_i^{j-1} \text{ AND } R_{i-1}^{j-1}) \\ \text{AND } R_{i-1}^{j-1}.$$

**Example 6.17:**

$x = \text{GATAA}$  and  $y = \text{CAGATAAGAGAA}$  and  $k = 1$  The following 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 Figure 6.45. 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  $O(kn)$ .

## 6.7 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 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 by just a pointer to their first occurrences.

### 6.7.1 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 bits, for instance). Coding the text is just replacing each symbol (more

```

void WM(char *y, char *x, int n, int m, int k) {
    unsigned int j, last1, last2, lim, mask, S[ASIZE], R[KSIZE];
    int i;

    /* Preprocessing */
    for (i=0; i < ASIZE; i++) S[i]=~0;
    lim=0;
    for (i=0, j=1; i < m; i++, j<<=1) {
        S[x[i]]&=~j;
        lim|=j;
    }
    lim=~(lim>>1);
    R[0]=~0;
    for (j=1; j <= k; j++) R[j]=R[j-1]>>1;

    /* Search */
    for (i=0; i < n; i++) {
        last1=R[0];
        mask=S[y[i]];
        R[0]=(R[0]<<1)|mask;
        for (j=1; j <= k; j++) {
            last2=R[j];
            R[j]=((R[j]<<1)|mask)&((last1&R[j-1])<<1)&last1;
            last1=last2;
        }
        if (R[k] < lim) OUTPUT(i);
    }
}

```

Figure 6.45: Wu-Manber approximate string-matching algorithm.

```

COUNT (fin)
1  for each character  $a \in \Sigma$ 
2      do  $freq(a) \leftarrow 0$ 
3  while not end of file fin and  $a$  is the next symbol
4      do  $freq(a) \leftarrow freq(a) + 1$ 
5   $freq(\text{END}) \leftarrow 1$ 

```

Figure 6.46: Counts the character frequencies.

exactly each occurrence of it) by its new codeword. The method works for any length of blocks (not only 8 bits), but the running time grows exponentially with the length. In the following, we assume that symbols are originally encoded on 8 bits 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 6.2.6) 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 a 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 while 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, encoding of the text.

The first step consists of counting the number of occurrences of each character in the original text (see Figure 6.46). 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
  - Extract the two least weighted trees  $t_1$  and  $t_2$ ,
  - Create a new tree  $t_3$  having left subtree  $t_1$ , right subtree  $t_2$ , and weight  $weight(t_3) = weight(t_1) + weight(t_2)$
- until only one tree remains.

## BUILD-TREE

```

1  for each  $a \in \Sigma \cup \{\text{END}\}$ 
2      do if  $\text{freq}(a) \neq 0$ 
3          then create a new node  $t$ 
4               $\text{weight}(t) \leftarrow \text{freq}(a)$ 
5               $\text{label}(t) \leftarrow a$ 
6   $l\text{leaves} \leftarrow$  list of all the nodes in increasing order of weight
7   $l\text{trees} \leftarrow$  empty list
8  while  $\text{LENGTH}(l\text{leaves}) + \text{LENGTH}(l\text{trees}) > 1$ 
9      do  $(l, r) \leftarrow$  extract the two nodes of smallest weight (among the two nodes at the beginning
           of  $l\text{leaves}$  and the two nodes at the beginning of  $l\text{trees}$ )
10     create a new node  $t$ 
11      $\text{weight}(t) \leftarrow \text{weight}(l) + \text{weight}(r)$ 
12      $\text{left}(t) \leftarrow l$ 
13      $\text{right}(t) \leftarrow r$ 
14     insert  $t$  at the end of  $l\text{trees}$ 
15      $\text{size} \leftarrow \text{size} - 1$ 
16 return  $t$ 

```

Figure 6.47: Builds the coding tree.

The tree is constructed by the algorithm BUILD-TREE in Figure 6.47. The implementation uses two linear lists. The first list contains the leaves of the future tree associated each with a symbol. The list is sorted in the increasing order of the weight of the leaves (symbol frequency of symbols). The second list contains the newly created trees. Extracting the two least weighted trees consists of 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 Figure 6.48);  $\text{codeword}(a)$  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. These information are placed in a header of the compressed file, to be 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 bits (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 Figure 6.49.

After the header of the compressed file is computed, the encoding of the original text is realized by the algorithm of Figure 6.50.

A complete implementation of the Huffman algorithm, composed of the three steps described above, is given in Figure 6.51.

```

BUILD-CODE (t, length)
1  if t is not a leaf
2      then   temp[length] ← 0
3              BUILD-CODE (left(t), length + 1)
4              temp[length] ← 1
5              BUILD-CODE (right(t), length + 1)
6      else   codeword(label(t)) ← temp[0...length - 1]

```

Figure 6.48: Builds the character codes from coding tree.

```

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

```

Figure 6.49: Memorizes the coding tree in the compressed file.

```

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

```

Figure 6.50: Encodes the characters in the compressed file.

```

CODING (fin, fout)
1  COUNT (fin)
3  t ← BUILD-TREE
3  BUILD-CODE (t, 0)
4  CODE-TREE (fout, t)
5  CODE-TEXT (fin, fout)

```

Figure 6.51: Complete function for Huffman coding.

**Example 6.18:**

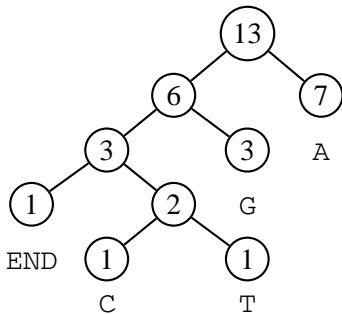
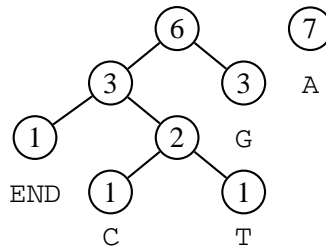
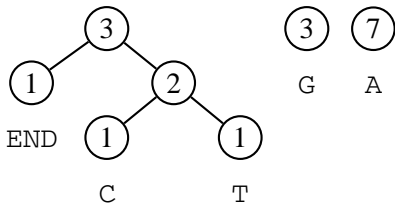
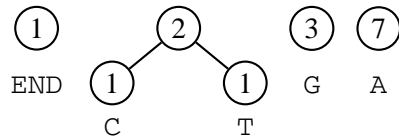
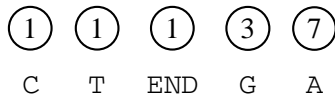
$y = \text{CAGATAAGAGAA}$

Length of  $y = 12 \times 8 = 96$  bits (assuming an 8-bit code)

Character frequencies:

A	C	G	T	END
7	1	3	1	1

Different steps during the construction of the coding tree:



character codewords:

A	C	G	T	END
1	0010	01	0011	000

Encoded tree: 0001binary(END,9)01binary(C,9)1 binary(T,9)1binary(G,9)1binary(A,9),  
which produces a header of length 54 bits:

0001 100000000 01 001000011 1 001010100 1 001000111 1 001000001

Encoded text: 0010 1 01 1 0011 1 1 01 1 01 1 1 000,  
of length 24 bits

Total length of the compressed file: 78 bits

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 time linear in the sum of the sizes of the original and compressed texts.

```

REBUILD-TREE (fin, t)
1  b ← read a bit from the file fin
2  if b = 1    /* leaf */
3      then    left(t) ← NIL
4              right(t) ← NIL
5              label(t) ← symbol corresponding to the 9 next bits in the file fin
6  else       create a new node l
7              left(t) ← l
8              REBUILD-TREE (fin, l)
9              create a new node r
10             right(t) ← r
11             REBUILD-TREE (fin, r)

```

Figure 6.52: Rebuilds the tree read from the compressed file.

```

DECODE-TEXT (fin, fout, root)
1  t ← root
2  while label(t) ≠ END
3      do if t is a leaf
4          then    write label(t) in the file fout
5                  t ← root
6          else    b ← read a bit from the file fin
7                  if b = 1
8                      then    t ← right(t)
9                      else    t ← left(t)

```

Figure 6.53: Reads the compressed text and produces the uncompressed text.

## Decoding

Decoding a file containing a text compressed by Huffman algorithm is a mere programming exercise. First the coding tree is rebuilt by the algorithm of Figure 6.52. 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 Figure 6.53.

The complete decoding program is given in Figure 6.54. It calls the preceding functions. The running time of the decoding program is linear in the sum of the sizes of the texts it manipulates.

```

DECODING (fin, fout)
1  create a new node root
2  REBUILD-TREE (fin, root)
3  DECODE-TEXT (fin, fout, root)

```

Figure 6.54: Complete function for decoding.

## 6.7.2 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-Welch method after several improvements introduced by Welch. 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 6.19:

$y = \text{CAGTAAGAGAA}$

C	A	G	T	A	A	G	A	G	A	A	<i>w</i>	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 symmetric 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 6.20:

decoding: 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		

```

COMPRESS (fin, fout)
1  count  $\leftarrow$  -1
2  for each character a  $\in$   $\Sigma$ 
3      do count  $\leftarrow$  count + 1
4          HASH-INSERT (D, (-1, a, count))
5  count  $\leftarrow$  count + 1
6  HASH-INSERT (D, (-1, END, count))
7  p  $\leftarrow$  -1
8  while not end of file fin
9      do a  $\leftarrow$  next character of fin
10         q  $\leftarrow$  HASH-SEARCH (D, (p, a))
11         if q = NIL
12             then write code(p) on  $1 + \log(\textit{count})$  bits in fout
13                 count  $\leftarrow$  count + 1
14                 HASH-INSERT (D, (p, a, count))
15                 p  $\leftarrow$  HASH-SEARCH (D, (-1, a))
16             else p  $\leftarrow$  q
17 write p on  $1 + \log(\textit{count})$  bits in fout
18 write code(HASH-SEARCH (D, (-1, END))) in  $1 + \log(\textit{count})$  bits in fout

```

Figure 6.55: LZW compression algorithm.

### Implementation

For the compression algorithm shown in Figure 6.55, 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:

- $\textit{parent}(z)$ : a link to the parent node of  $z$ ,
- $\textit{label}(z)$ : a character,
- $\textit{code}(z)$ : the code associated with  $z$ .

The tree is stored in table that is accessed with a hashing function. This provides a 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  $\textit{parent}(z) = p$ ,  $\textit{label}(z) = a$  and  $\textit{code}(z) = c$ . The function HASH-SEARCH ( $D, (p, a)$ ) returns the node  $z$  such that  $\textit{parent}(z) = p$  and  $\textit{label}(z) = a$ .

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  $\textit{string}(c)$  performs this specific work for a code  $c$ . The bottom-up walk follows the parent links of the data structure. The function  $\textit{first}(w)$  gives the first character of the word  $w$ . These features are part of the decompression algorithm displayed in Figure 6.56.

The Ziv-Lempel compression and decompression algorithms run both in time linear 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.

```

UNCOMPRESS (fin,fout)
1  count ← -1
2  for each character a ∈ Σ
3      do count ← count + 1
4          HASH-INSERT (D, (-1, a, count))
5  count ← count + 1
6  HASH-INSERT (D, (-1, END, count))
7  c ← first code on 1 + log(count) bits in fin
8  write string(c) in fout
9  a ← first(string(c))
10 repeat
11     d ← next code on 1 + log(count) in the fin
12     if d > count
13         then count ← count + 1
14             parent(count) ← c
15             label(count) ← a
16             write string(c)a in fout
17             c ← d
18         else a ← first(string(d))
19             if a ≠ END
20                 then count ← count + 1
21                     parent(count) ← c
22                     label(count) ← a
23                     write string(d) in fout
24                     c ← d
25             else exit
26 forever

```

Figure 6.56: LZW decompression algorithm.

Source texts	French	C sources	Alphabet	Random
Sizes in bytes	62816	684497	530000	70000
Huffman	53.27%	62.10%	72.65%	<b>55.58%</b>
Ziv-Lempel	<b>41.46%</b>	34.16%	2.13%	63.60%
Factor	47.43%	<b>31.86%</b>	<b>0.09%</b>	73.74%

Figure 6.57: Sizes of texts compressed with three algorithms.

### 6.7.3 Experimental results

The table of Figure 6.57 contains a sample of experimental results showing the behavior of compression algorithms on different types of texts. The table is extracted from (Zipstein, 1992).

The source files are: French text, C sources, Alphabet, and Random. Alphabet is a file containing a repetition of the line `abc . . . zABC . . . Z`. Random is a file where the symbols have been generated randomly, all with the same probability and independently of each others.

The compression algorithms reported in the table are: the Huffman algorithm of Section 6.7.1, the Ziv-Lempel algorithm of Section 6.7.2, and a third algorithm called Factor. This latter algorithm encodes segments of the source text as Ziv-Lempel algorithm does. But the segments are taken among all segments already encountered in the text before the current position. The method gives usually better compression ratio but is more difficult to implement.

The table of Figure 6.57 gives in percentage the sizes of compressed files. Results obtained by Ziv-Lempel and Factor algorithms are similar. Huffman coding gives the best result for the Random file. Finally, experience shows that exact compression methods often reduce the size of data to 30%–50% of their original size.

## 6.8 Research Issues and Summary

The algorithm for string searching by hashing was introduced by Harrison (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  $\Phi$  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 algorithm.

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, Morris and Pratt, 1977). Cole (1995) proves that the maximum number of symbol comparisons is bounded by  $3n$ , and that this bound is tight.

Knuth, Morris, and Pratt (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 alii (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, 1994).

It is known that any string searching algorithm, working with symbol comparisons, makes at least  $n + \frac{9}{4m}(n - m)$  comparisons in the worst case (see Cole et al. 1995). Some string searching algorithms make less than  $2n$  comparisons at search phase. The presently-known upper bound on the problem is  $n + \frac{8}{3(m+1)}(n - m)$ , but with a quadratic-time preprocessing step (Cole et al., 1995). With a linear-time preprocessing step, the current upper bound is  $n + \frac{4 \log m + 2}{m}(n - m)$  by Breslauer and Galil (1993). Except in few cases (patterns of length 3, for example), lower and upper bound do not meet. So, the problem of the exact complexity of string searching is open.

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 while 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 et al., 1994).

The suffix tree construction of Section 6.4 is by McCreight (1976). 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 (Myers and Manber, 1993). All these techniques are presented in (Crochemore and Rytter, 1994). The data structures implement full indexes with standard operations while applications sometimes need only incomplete indexes. The design of compact indexes is still unsolved.

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. 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 6.6 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. The command `compact` of UNIX implements this version.

Several variants of the Ziv and Lempel algorithm exist. The reader can refer to the book of Bell, Cleary, and Witten (1990) for a discussion on them. The book of Nelson (1992) presents practical implementations of various compression algorithms.

## 6.9 Defining Terms

**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), i.e.  $w = vuz$  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.

## 6.10 References

- Aho, A.V. 1990. Algorithms for finding patterns in strings. In *Handbook of Theoretical Computer Science, vol. A, Algorithms and complexity*, ed. J. van Leeuwen, p 255–300. Elsevier, Amsterdam.
- Aho, A.V., and Corasick, M.J. 1975. Efficient string matching: an aid to bibliographic search. *Comm. ACM.* 18(6):333–340.
- Baeza-Yates, R.A., and Gonnet, G.H. 1992. A new approach to text searching. *Comm. ACM.* 35(10):74–82.
- 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.
- Bell, T.C., Cleary, J.G., and Witten, I.H. 1990. *Text compression*, Prentice Hall, Englewood Cliffs, New Jersey.
- Bird, R.S. 1977. Two-dimensional pattern matching. *Inf. Process. Lett.* 6(5):168–170.
- Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., Chen, M.T., and Seiferas, J. 1985. The smallest automaton recognizing the subwords of a text. *Theoret. Comput. Sci.* 40:31–55.
- Boyer, R.S., and Moore, J.S. 1977. A fast string searching algorithm. *Comm. ACM.* 20:762–772.
- Breslauer, D., and Galil, Z. 1993. Efficient comparison based string matching. *J. Complexity.* 9(3):339–365.
- Breslauer, D., Colussi, L., and Toniolo, L. 1993. Tight comparison bounds for the string prefix matching problem. *Inf. Process. Lett.* 47(1):51–57.
- Cole, R. 1994. Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm. *SIAM J. Comput.* 23(5):1075–1091.

- Cole, R., Hariharan, R., Zwick, U., and Paterson, M.S. 1995. Tighter lower bounds on the exact complexity of string matching. *SIAM J. Comput.* 24(1):30–45.
- Colussi, L. 1994. Fastest pattern matching in strings. *J. Algorithms.* 16(2):163–189.
- Crochemore, M. 1986. Transducers and repetitions. *Theoret. Comput. Sci.* 45(1):63–86.
- Crochemore, M., and Rytter, W. 1994. *Text Algorithms*, Oxford University Press.
- Galil, Z. 1981. String matching in real time. *J. ACM.* 28(1):134–149.
- Hancart, C. 1993. On Simon’s string searching algorithm. *Inf. Process. Lett.* 47:95–99.
- Hirschberg, D.S. 1975. A linear space algorithm for computing maximal common subsequences. *Comm. ACM.* 18(6):341–343.
- Hume, A., Sunday, D.M. 1991. Fast string searching. *Software – Practice and Experience.* 21(11):1221–1248.
- Karp, R.M., Rabin, M.O. 1987. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.* 31(2):249–260.
- Knuth, D.E., Morris Jr, J.H., Pratt, V.R. 1977. Fast pattern matching in strings. *SIAM J. Comput.* 6(1):323–350.
- Lecroq, T. 1995. Experimental results on string-matching algorithms. *Software - Practice & Experience* 25(7):727–765.
- McCreight, E.M. 1976. A space-economical suffix tree construction algorithm. *J. Algorithms.* 23(2):262–272.
- Manber, U., Myers, G. 1993. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.* 22(5):935–948.
- Nelson, M. 1992. *The data compression book*, M&T Books.
- Simon, I. 1993. String matching algorithms and automata. In *First American Workshop on String Processing*, ed. Baeza-Yates and Ziviani, p 151–157. Universidade Federal de Minas Gerais.
- Stephen, G.A. 1994. *String searching algorithms*, World Scientific Press.
- Sunday, D.M. 1990. A very fast substring search algorithm. *Comm. ACM.* 33(8):132–142.
- Welch, T. 1984. A technique for high-performance data compression. *IEEE Computer.* 17(6):8–19.
- Wu, S., Manber, U. 1992. Fast text searching allowing errors. *Comm. ACM.* 35(10):83–91.
- Zipstein, M. 1992. Data compression with factor automata. *Theoret. Comput. Sci.* 92(1):213–221.
- Zhu, R.F., Takaoka, T. 1989. A technique for two-dimensional pattern matching. *Comm. ACM.* 32(9):1110–1120.

## 6.11 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 few books such as:

- Bell, T.C., Cleary, J.G., Witten, I.H. 1990. *Text compression*, Prentice Hall, Englewood Cliffs, New Jersey.
- Crochemore, M., Rytter, W. 1994. *Text algorithms*, Oxford University Press.

- Nelson, M. 1992. *The data compression book*, M&T Books.
- Stephen, G.A. 1994. *String searching algorithms*, World Scientific Press.

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

Finally, two main annual conferences present the latest advances of this field of research:

- *Combinatorial Pattern Matching*, which started in 1990 and was held in Paris (France), London (England), Tucson (Arizona), Padova (Italy), Asilomar (California), Helsinki (Finland), Laguna Beach (California).
- *Data Compression Conference*, which is regularly held at Snowbird.

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

Several books on the design and analysis of general algorithms contain a chapter 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.