Pattern matching and text compression algorithms

Maxime Crochemore [1] Thierry Lecroq [2]

[1]Institut Gaspard Monge, Université de Marne la Vallée, 2 rue de la Butte Verte, F-93166 Noisy-le-Grand Cedex, France, e-mail: mac@univ-mlv.fr

[2]Laboratoire d'Informatique de Rouen, Université de Rouen, Facultés des Sciences et Techniques, F-76821 Mont-Saint-Aignan Cedex, France, e-mail: lecroq@dir.univ-rouen.fr

Contents

List of Figures

Chapter 1

Processing texts efficiently

The present report describes 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 report are interesting in different aspects. First, they are basic components used in the implementations of practical softwares. 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 memorized in various ways, text remains the main form to exchange information. This is particularly evident in litterature or linguistics where data are composed of huge corpus and dictionaries. This apply as well to computer science where a large amount of data are stored in linear files. And this is also the case, for instance, in molecular biology because biological molecules can often be approximated as sequences of nucleotides or aminoacids. Moreover the quantity of available data in these fields tend 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 present two algorithms for matching patterns in images that are extensions of string-matching algorithms.

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

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

Finally, the report contains two classical text compression algorithms. Variants of these algorihms are implemented in practical compression softwares, in which they are often combined together or with other elementary methods.

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

Chapter 2

String-matching algorithms

String matching consists in 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 (finite set of symbols). All the algorithms in this section output 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 Σ and its size is equal to σ.

String-matching algorithms of the present section work as follows: they first align the left ends of the pattern and the text, then compare the characters of the text aligned with the characters of the pattern — this specific work is called an attempt — 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. 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 in checking, at all positions in the text between 0 and $n-m$, whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern exactly one position to the right. The brute force algorithm is given Figure 2.1.

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

2.1 Karp-Rabin algorithm

Hashing provides a simple method to avoid a quadratic number of symbol comparisons in most practical situations. Instead of checking at each position of the text if 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 ressemblance between these portions a hashing function is used. To be helpful for the string-matching problem the hashing function should have the following properties:

- efficiently computable,
- highly discriminating for strings,
- $hash(y[i+1 \dots i+m])$ 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 m let $hash(w)$ be defined as follows:

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

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

 /* Searching */
 i=0;
 while (i <= n-m) {
    j=0;
    while (j < m && y[i+j] == x[j]) j++;
    if (j >= m) OUTPUT(i);
    i++;                  /* shift one position to the right */
 }
}
```

Figure 2.1: The brute force string-matching algorithm.

where q is a large number. Then,

$$rehash(a, b, h) = ((h - a * 2^{m-1}) * 2 + b) \bmod q.$$

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

In the algorithm of Figure 2.2 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 2.1:
Let $x =$ `ing`.
Then $hash(x) = 105 * 2^2 + 110 * 2 + 103 = 743$ (values computed with the ASCII codes).

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

2.2 Knuth-Morris-Pratt algorithm

We present 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 characters of the pattern and consequently increases the speed of the search.

```
#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 */
 d=1;
 for (i=1; i < m; i++) d=(d<<1);
 hy=hx=0;
 for (i=0; i < m; i++) {
    hx=((hx<<1)+x[i]);
    hy=((hy<<1)+y[i]);
 }

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

Figure 2.2: The Karp-Rabin string-matching algorithm.

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 $y[i+j]$ and $x[j]$ with $1 < j < m$. 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 reasonnable to expect that a **prefix** v of the pattern match 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 border of $x[0 \dots j-1]$ followed by a character c different from $y[j]$. Then, after a shift, the comparisons can resume between characters $y[i+j]$ and $x[j - next[j]]$ without missing any occurrence of x in y, and avoiding a backtrack on the text (see Figure 2.3).

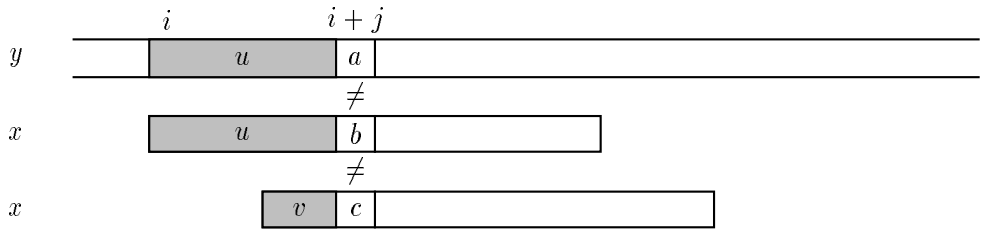


Figure 2.3: Shift in the Knuth-Morris-Pratt algorithm (v suffix of u).

```
void KMP(char *y, char *x, int n, int m)
{
 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 2.4: 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 (x[i] == x[j]) next[i]=next[j];
    else next[i]=j;
 }
}
```

Figure 2.5: Preprocessing phase of the Knuth-Morris-Pratt algorithm: computing `next`.

Example 2.2:

$y =$ . . . 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 border of `ababa` is `aba`.

The Knuth-Morris-Pratt algorithm is displayed in Figure 2.4. The table $next$ it uses can be computed in $O(m)$ time before the search phase, applying the same searching algorithm to the pattern itself, as if $y = x$ (see Figure 2.5). The worst-case running time of the algorithm is $O(m + n)$ and it requires $O(m)$ extra-space. These quantity are independent of the size of the underlying alphabet.

2.3 Boyer-Moore algorithm

The Boyer-Moore algorithm is considered as the most efficient string-matching algorithm in usual applications. A simplified version of it or the entire algorithm is often implemented in text editor for

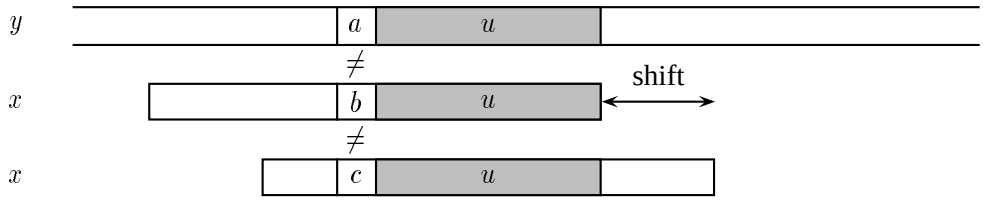


Figure 2.6: good-suffix shift, u reappears preceded by a character different from b.

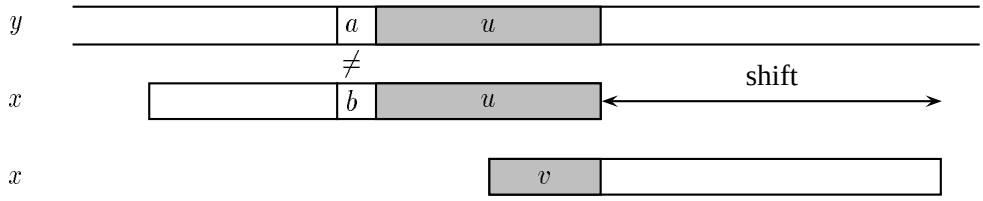


Figure 2.7: good-suffix shift, only a prefix of u reappears in x.

the "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*.

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, $y[i+j+1 \ldots i+m-1] = x[j+1 \ldots m-1] = u$ and $y[i+j] \neq x[j]$. The good-suffix shift consists in aligning the **segment** $y[i+j+1 \ldots i+m-1] = x[j+1 \ldots m-1]$ with its rightmost occurrence in x that is preceded by a character different from $x[j]$ (see Figure 2.6). If there exists no such segment, the shift consists in aligning the longest suffix v of $y[i+j+1 \ldots i+m-1]$ with a matching prefix of x (see Figure 2.7).

Example 2.3:

$y =$	.	.	.	a	b	b	a	a	b	b	a	b	b	a	.	.	.
$x =$	a	b	b	a	a	b	b	a	b	b	a						
$x =$				a	b	b	a	a	b	b	a	b	b	a			

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

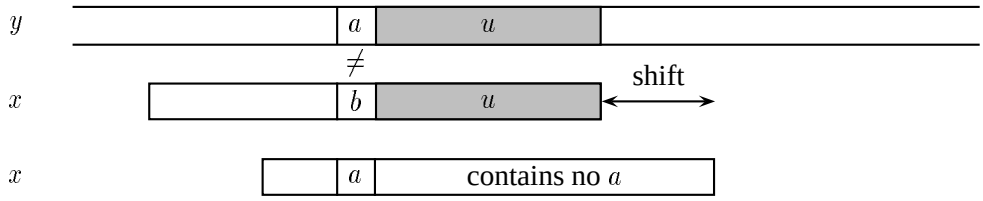


Figure 2.8: bad-character shift, a appears in x.

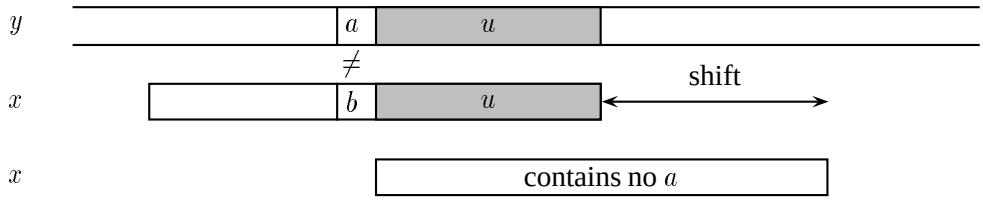


Figure 2.9: bad-character shift, a does not appear in x.

matches a segment of x. The same mismatch does not recur.

Example 2.4:

$y =$ . . . a b b a a b b a b b a b b a . .
$x =$ b b a b b a b b a
$x =$ b b a b b a b b a

The segment abba found in y partially matches a prefix of x after the shift.

The bad-character shift consists in aligning the text character $y[i+j]$ with its rightmost occurrence in $x[0 \ldots m-2]$ (see Figure 2.8). If $y[i+j]$ does not appear in the pattern x, no occurrence of x in y can include $y[i+j]$, and the left end of the pattern is aligned with the character immediately after $y[i+j]$, namely $y[i+j+1]$ (see Figure 2.9).

Example 2.5:

$y =$ . . . . . a b c d . . . .
$x =$ c d a h g f e b c d
$x =$ c d a h g f e b c d

The shift aligns the symbol a in x with the mismatch symbol a in the text.

Example 2.6:

$y =$ . . . . . a b c d . . . . . .
$x =$ c d h g f e b c d
$x =$ c d h g f e b c d

The shift positions the left end of x right after the symbol a of y.

The Boyer-Moore algorithm is shown in Figure 2.10. 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 σ and the good-suffix shift is stored in a table gs of size $m+1$. For $a \in \Sigma$:

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

Let us define two conditions:

$cond_1(j,s)$: for each k such that $j < k < m, s \geq k$ or $x[k-s] = x[k]$
$cond_2(j,s)$: if $s < j$ then $x[j-s] \neq x[j]$

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

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

```
void BM(char *y, char *x, int n, int m)
{
 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 2.10: The Boyer-Moore string-matching algorithm.

```
void PRE_BC(char *x, int m, int bc[])
{
 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 2.11: Computation of the bad-character shift.

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

Tables bc et gs can be precomputed in time $O(m + \sigma)$ before the search phase and require an extra-space in $O(m + \sigma)$ (see Figures 2.12 and 2.11). The worst-case running time of the algorithm is quadratic. However, on large alphabets (relatively 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.

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 alone produces a very efficient algorithm in practice.

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

```
void PRE_GS(char *x, int m, int gs[])
{
 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 2.12: Computation of the good-suffix shift.

bad-character shift of the current attempt. The bad-character shift of the present algorithm is slightly modified to take into account the last symbol of x as follows ($a \in \Sigma$):

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

The comparisons between text and pattern characters during each attempt can be done in any order. The algorithm of Figure 2.13 performs the comparisons from left to right. The algorithm is called Quick Search after his inventor. It has a quadratic worst-case time complexity but a good practical behaviour.

Example 2.7:

```
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.

2.5 Experimental results

In Figures 2.14 and 2.15 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. RF algorithm uses a data structure to store all the factors of the reversed pattern: a suffix automaton or a **suffix tree** (see Chapter 4).

```
void QS(char *y, char *x, int n, int m)
{
 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 2.13: The Quick Search string-matching algorithm.

Tests have been performed on various types of texts. In Figure 2.14 we show the result when the text is a DNA sequence on the four-letter alphabet of nucleotides $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. In Figure 2.15 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 is the two Figures. Both, preprocessing and searching phases running times are totalized. The three algorithms are implemented in a homogeneous way in order to keep the comparison significant.

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

2.6 Aho-Corasick algorithm

The UNIX operating provides standard texts (or files) facilities. Among them is the series of `grep` command 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 outputs lines containing at least one of the strings.

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

Let $X = \{x_1, x_2, \ldots, x_k\}$ be the set of patterns, and let $|X| = |x_1| + |x_2| + \cdots + |x_k|$ be the total size of the set X. The Aho-Corasick algorithm first consists in building a **trie** $T(X)$, digital tree recognizing the patterns of X. The trie $T(X)$ is a tree which edges are labelled by letters and which

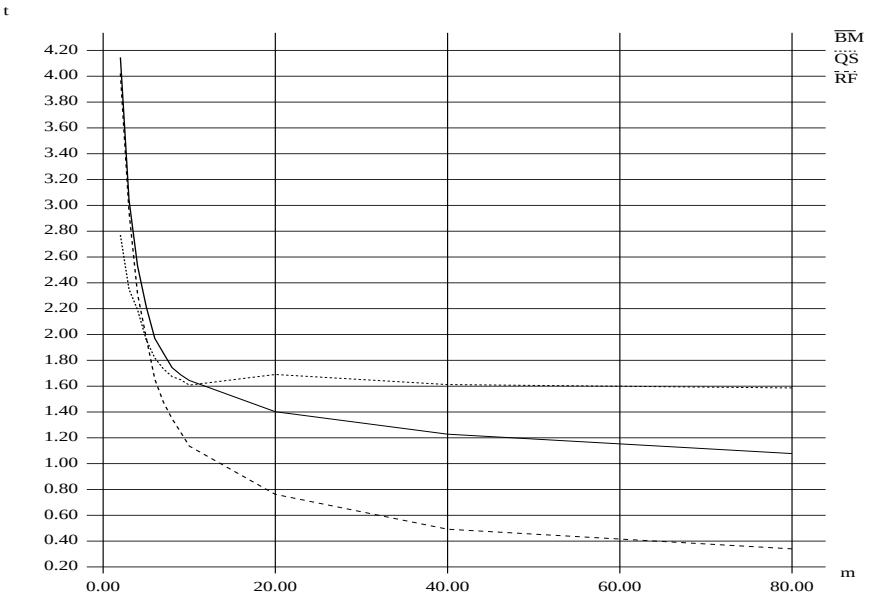


Figure 2.14: Running times for a DNA sequence.

branches spell the patterns of X. We identify a node p in the trie $T(X)$ with the unique word w spelled by the path of $T(X)$ from its root to p. The root itself is identified with the empty word: ε. Notice that if w is a node in $T(X)$ then w is a prefix of some $x_i \in X$.

The function PRE_AC in Figure 2.16 returns the trie of all patterns. During the second phase where pattern are entered in the trie the algorithm initializes an output function out. It associates the singleton $\{x_i\}$ with the nodes x_i ($1 \leq i \leq k$), and associates the empty set with all other nodes of $T(X)$ (see Figure 2.17).

The last phase of function PRE_AC (Figure 2.16) consists in 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 2.18. The failure function f is defined on nodes as follows (w is a node):

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

Computation of failure links is done using a breadth-first search of $T(X)$. Completion of the output function is done while computing the failure function f in the following way:

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

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, it is necessary to add a loop on the root of the trie for these characters. This is done by the first phase of function PRE_AC.

After the preprocessing phase is completed, the searching phase consists in parsing all the characters of the text y with $T(X)$. This starts at the root of $T(X)$ and uses failure links whenever a character of y does not match any label of edges outgoing the current node. Each time a node with a non-empty value

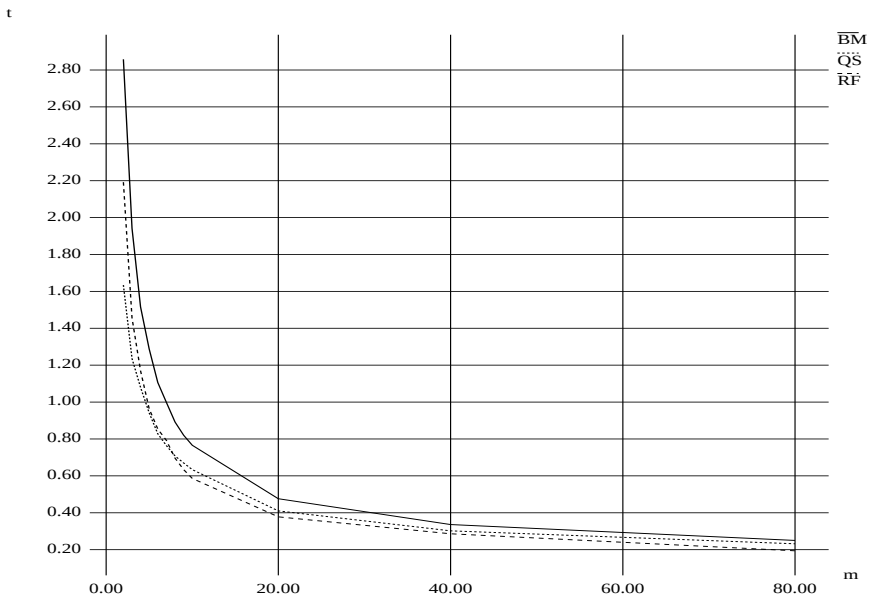


Figure 2.15: Running times for an english text.

```
NODE PRE_AC(char X[KSIZE][XSIZE], int k)
{
 NODE root;  int i, c;

 root=ALLOCATE_NODE();

 /* creates loops on the root of the trie */
 for (c=0; c < ASIZE; c++) PUT_EDGE(root, c, root);

 /* enters each pattern in the trie */
 for (i=0; i < k; ++i) ENTER(X[i], root);

 /* completes the trie with failure links */
 COMPLETE(root);

 return(root);
}
```

Figure 2.16: Preprocessing phase of the Aho-Corasick algorithm.

```
void ENTER(char *x, NODE root)
{
 int m, i;
 NODE r, s;

 m=strlen(x);
 r=root; i=0;

 /* follows the existing edges */
 while (i < m && (s=GET_NODE(r, x[i])) != UNDEFINED && s != r) {
      r=s; i++;
 }

 /* creates new edges */
 while (i < m) {
      s=ALLOCATE_NODE();
      PUT_EDGE(r, x[i], s);
      r=s; i++;
 }

 PUT_OUTPUT(r, x);
}
```

Figure 2.17: Construction of the trie.

for the output function is encountered, this means that the patterns contained in the output function of this node have been discovered in the text, ending at the current position. The position is then output.

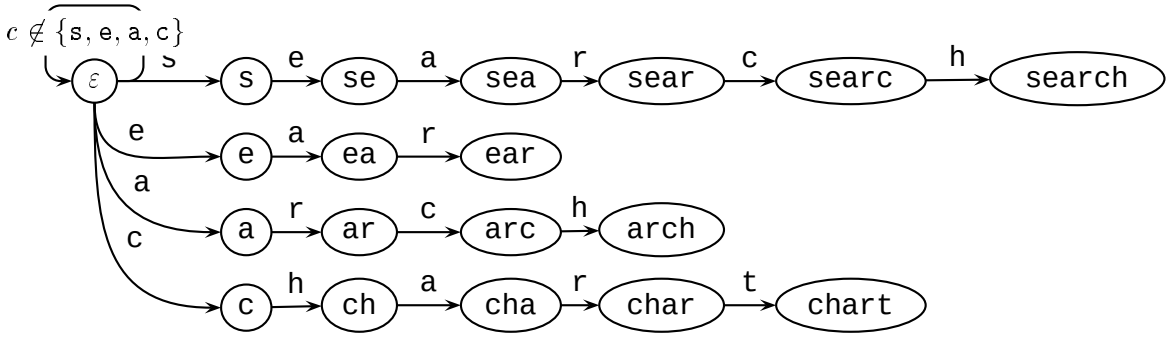
The Aho-Corasick algorithm implementing the previous discussion is shown in Figure 2.19. Note that the algorithm processes the text in an on-line way, so that the buffer on the text can be limited to only one symbol. Also note that the instruction `r=GET_FAIL(r)` in Figure 2.19 is the exact analogue of instruction `j=next[j]` in Figure 2.4. A unified view of both algorithms exist but is out of the scope of the report.

The entire algorithm runs in time $O(|X| + n)$ if the `GET_NODE` 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 execution of `GET_NODE`.

```
void COMPLETE(NODE root)
{
 QUEUE q;
 LIST l;
 NODE p, r, s, u;
 char c;

 q=EMPTY_QUEUE();
 l=GET_SONS(root);

 while (!LIST_EMPTY(l)) {
      r=FIRST_N(l);        /* r is a son of the root */
      l=NEXT(l);
      ENQUEUE(q, r);
      PUT_FAIL(r, root);
 }

 while (!QUEUE_EMPTY(q)) {
      r=DEQUEUE(q);
      l=GET_SONS(r);
      while (!LIST_EMPTY(l)) {
           p=FIRST_N(l);
           c=FIRST_L(l);  /* (r,p) is a edge labelled with c */
           l=NEXT(l);
           ENQUEUE(q, p);
           s=GET_FAIL(r);
           while ((u=GET_NODE(s, c)) == UNDEFINED) s=GET_FAIL(s);
           PUT_FAIL(p, u);
           ADD_OUTPUT(p, u);
      }
 }
}
```

Figure 2.18: Completion of the ouput function and construction of failure links.

Example 2.8:
$X = \{\texttt{search}, \texttt{ear}, \texttt{arch}, \texttt{chart}\}$



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

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

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

```
int AC(char *y, char X[KSIZE][XSIZE], int n, int k)
{
 NODE r, s;
 int i;

 /* Preprocessing */
 r=PRE_AC(X, k);

 /* Searching */
 for (i=0; i < n; ++i) {
    while ((s=GET_NODE(r, y[i])) == UNDEFINED) r=GET_FAIL(r);
    r=s;
    OUTPUT(r, i);
 }
}
```

Figure 2.19: The Aho-Corasick algorithm.

Chapter 3

Two-dimensional pattern matching algorithms

In this section only we consider two-dimensional arrays. Arrays may be thought as bitmap representations of images, where each cell of the 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 Chapter 2 extends to arrays.

The problem is now to find all occurrences of a two-dimensional pattern $x = x[0 \ldots m_1 - 1, 0 \ldots m_2 - 1]$ of size $m_1 \times m_2$ inside a two-dimensional text $y = [0 \ldots n_1 - 1, 0 \ldots n_2 - 1]$ of size $n_1 \times n_2$. The brute force algorithm for this problem is given in Figure 3.1. It consists in checking at all positions of $y[0 \ldots n_1 - m_1, 0 \ldots n_2 - m_2]$ if the pattern occurs. The brute force 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 2.1). The second solves the problem in linear-time on a fixed alphabet; it uses both the Aho-Corasick and the Knuth-Morris-Pratt algorithms.

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 is to use the hash function method proposed by Karp and Rabin vertically. First, the two-dimensionnal arrays of characters, x and y, are translated into one-dimensional arrays of numbers, x' and y'. The translation from x to x' is done as follows ($0 \leq i < m_2$):

$$x'[i] = hash(x[0, i]x[1, i] \ldots x[m_1 - 1, i])$$

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

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

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 ($0 \leq i < m_2$):

$$hash(y[j+1, i]y[j+2, i] \ldots y[j+m_1, i]) = rehash(y[j, i], y[j+m_1, i], hash(y[j, i]y[j+1, i] \ldots y[j+m_1-1, i]))$$

(functions $hash$ and $rehash$ are used in the Karp-Rabin algorithm of Section 2.1).

```
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 && strncmp(&y[i+k][j], x[k], m2) == 0) k++;
       if (k >= m1) OUTPUT(i,j);
    }
}
```

Figure 3.1: The brute force two-dimensional pattern matching algorithm.

Example 3.1:

$x =$

a	a	a
b	b	a
a	a	b

$y =$

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

$x' =$

681	681	680

$y' =$

680	684	680	683	681	685	686

Since the alphabet of x' and y' is large, searching for x' in y' must be done by a string-matching algorithm which running time is independent of the size of the alphabet: the Knuth-Morris-Pratt suits perfectly for this application. Its adaptation is shown in Figure 3.2.

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 naïvely by the procedure of Figure 3.3.

The Zhu-Takaoka algorithm working as explained above is dispayed in Figure 3.4. The search for the pattern is performed row by row beginning at row 0 and ending at row $n_1 - m_1$.

3.2 Bird/Baker algorithm

The algorithm designed independtly 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: $R_0 = x[0, 0 \ldots m_2 - 1]$ to $R_{m_1-1} = x[m_1 - 1, 0 \ldots m_2 - 1]$. The rows are preprocessed into a trie as in the Aho-Corasick algorithm.

```
void KMP_IN_LINE(BIG_IMAGE Y, SMALL_IMAGE X, int YB[], int XB[],
         int n1, 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, n1, n2, m1, m2, row, j-1);
      i=next[m2];
    }
 }
}
```

Figure 3.2: Search for x' in y' using KMP algorithm.

```
void DIRECT_COMPARE(BIG_IMAGE Y, SMALL_IMAGE X, int n1, int n2,
                    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 3.3: Naive check of an occurrence of x in y at position $(row, column)$.

```
#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 first value 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;
 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, n1, 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++;
 }
}
```

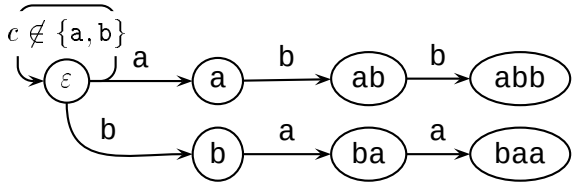
Figure 3.4: The Zhu-Takaoka two-dimensional pattern matching algorithm.

```
void PRE_KMP(SMALL_IMAGE X, int m1, int m2, int next[])
{
 int i, j;

 i=0;
 next[0]=j=(-1);
 while (i < m1) {
    while (j > -1 && strncmp(X[i], X[j], m2) != 0) j=next[j];
    i++; j++;
    if (strncmp(X[i], X[j], m2) == 0) next[i]=next[j];
    else next[i]=j;
 }
}
```

Figure 3.5: Computes the failure function of Knuth-Morris-Pratt for X.

Example 3.2:
The trie of rows of pattern x.

$$x = \begin{array}{|c|c|c|}\hline \mathtt{b} & \mathtt{a} & \mathtt{a} \\ \hline \mathtt{a} & \mathtt{b} & \mathtt{b} \\ \hline \mathtt{b} & \mathtt{a} & \mathtt{a} \\ \hline \end{array}$$



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 $y[i, j - m_2 + 1 \ldots j] = R$ matches any of $R_1, \ldots R_{m_1-1}$ using the Aho-Corasick machine. An additional one-dimensional array a of size $O(n_1)$ is used as follows:

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

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:

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

The value s is computed using KMP vertically (in columns). If there exists no such s, $a[j]$ is set to 0. Finally, if $a[j] = m_1$ 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 3.5 and 3.6. It runs in time $O((n_1 n_2 + m_1 m_2) \log \sigma)$.

```
void B(BIG_IMAGE Y, SMALL_IMAGE X, int n1, int n2, int m1, int m2)
{
 int next[XSIZE], a[TSIZE], row, column, k;
 NODE root, r, s;
 char *x;

 /* Preprocessing */
 memset(a, 0, n2*sizeof(int));
 root=PRE_AC(X, m1, m2);
 PRE_KMP(X, m1, m2, f);

 /* Searching */
 for (row=0; row < n1; row++) {
    r=root;
    for (column=0; column < n2; column++) {
       while ((s=GET_NODE(r, Y[row][column])) == UNDEFINED) r=GET_FAIL(r);
       r=s;
       if ((x=GET_OUTPUT(r)) != UNDEFINED) {
          k=a[column];
          while (k>0 && strncmp(X[k], x, m2) != 0) k=f[k];
          a[column]=k+1;
          if (k >= m1-1) OUTPUT(row-m1+1, column-m2+1);
       }
       else a[column]=0;
    }
 }
}
```

Figure 3.6: The Bird/Baker two-dimensional pattern matching algorithm.

Chapter 4

Suffix trees

The suffix tree $S(y)$ of a string y is a trie (see Section 2.6) containing all the suffixes of the string, and having properties that are 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 trie to represent 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 labelled by all suffixes of y,
- edges of $S(y)$ are labelled by strings,
- internal nodes of $S(y)$ have at least two sons (when y is non empty),
- edges outgoing an internal node are labelled by segments starting with different letters,
- the above segments are represented by their starting and ending positions 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 $n + 1$ leaves. 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 the suffixes of y in the data structure as follows:

$T_{-1} \leftarrow$ UNDEFINED;
for $i \leftarrow 0$ **to** $n - 1$ **do**
    $T_i \leftarrow$ INSERT($T_{i-1}, y[i \ldots n-1]$);
endfor

The last tree T_{n-1} is the suffix tree $S(y)$.

We introduce two definitions to explain how the algorithm works:

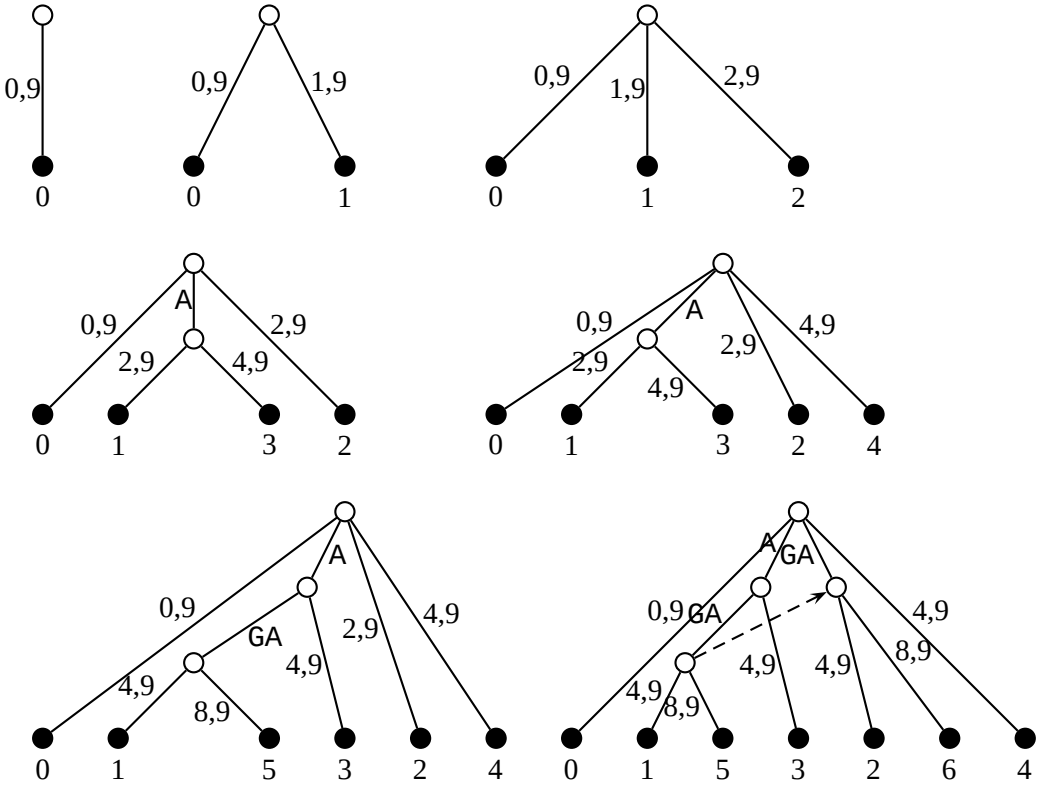
- $head_i$ is the longest prefix of $y[i \ldots n-1]$ which is also a prefix of $y[j \ldots n-1]$ for some $j < i$,
- $tail_i$ is the word such that $y[i \ldots n-1] = head_i tail_i$.

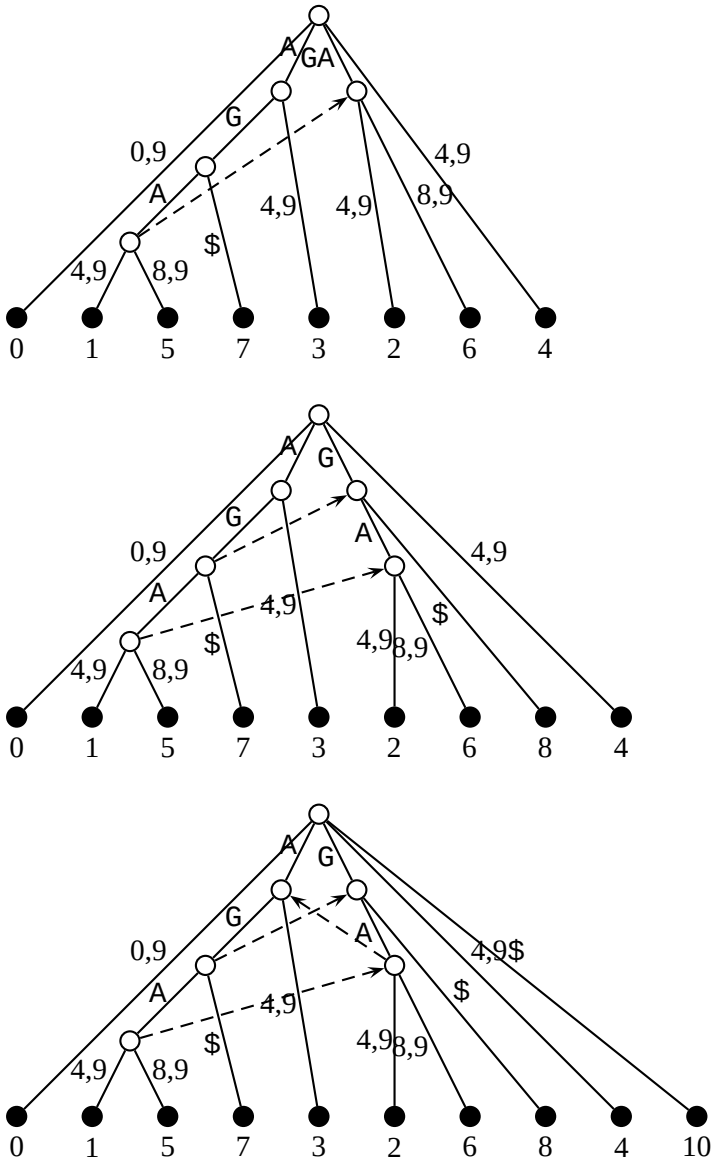
The strategy to insert the i-th suffix in the tree is based on the previous definitions:

INSERT(T_{i-1}, $y[i \ldots n-1]$)
locate the node h associated with $head_i$ in T_{i-1}, possibly breaking an edge;
add a new edge labelled $tail_i$ from h to a new leaf representing suffix $y[i \ldots n-1]$;
return the modified tree;

The second step of the insertion is performed in constant time. It is thus finding the node h that is critical for the overall performance of the algorithm. A brute force method to find it consists in spelling the current suffix $y[i \ldots n-1]$ from the root of the tree, giving an $O(|head_i|)$ time complexity for insertion at step i, and an $O(n^2)$ running time to built $S(y)$. Adding 'short-cut' links leads to an overall $O(n)$ time complexity, although insertion at step i is not realized in constant time.

Example 4.1:
The different trees during the construction of the suffix tree of $y =$ CAGATAGAG$. Leaves are black and labelled by the position of the suffix they represent. Dashed arrows represent the non trivial suffix links.





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 finding the next head efficiently. To insert 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 happen 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 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 father of d', and let w be the label of edge (c', d'). For the ease of the description, assume that $az = avw$ (it may happen that $az = w$). There is a suffix link defined on c' and going to some node c associated with av. The crucial observation here is that w is prefix of the label of a branch starting at node c. The algorithm rescans w in the tree: let e be the child of c along that branch, and let p be the label of edge (c, e). If $|p| < |w|$ then a recursive rescan of q, where $w = pq$, starts from node e. If $|p| > |w|$, the edge (c, e) is broken to insert a new node d; labels are updated correspondingly. If $|p| = |w|$, 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, the node associated with the new head is known, and the tail of the current suffix can be inserted in the tree.

To analyse 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 sons 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 word w considered there, and this position never decreases.

An implementation of McCreight's algorithm is shown in Figure 4.1. The next figures give the procedures used by the algorithm, and especially procedures RESCAN and SCAN.

```
NODE M(char *y, int n)
{
 NODE root, head, tail, d;
 char *end, *gamma;

 end=y+n;

 head=root=INIT(y, n);
 tail=GET_SON(root, *y);

 while (--n) {
    /* Phase A (rescanning) */
    if (head == root) {
      d=root; gamma=GET_LABEL(tail)+1;
    } else {
      gamma=GET_LABEL(tail);
      if (GET_LINK(head) != UNDEFINED) d=GET_LINK(head);
      else {
       if (GET_FATHER(head) == root)
        d=RESCAN(root, GET_LABEL(head)+1, GET_LENGTH(head)-1);
       else
        d=RESCAN(GET_LINK(GET_FATHER(head)), GET_LABEL(head), GET_LENGTH(head));
       PUT_LINK(head, d);
      }
    }

    /* Phase B: scanning */
    head=SCAN(d, &gamma);

    tail=ALLOCATE_NODE();
    PUT_FATHER(tail, head);
    PUT_LABEL(tail, gamma);
    PUT_LENGTH(tail, (int)(end-gamma));
    PUT_SON(head, *gamma, tail);
 }
 return(root);
}
```

Figure 4.1: Suffix tree construction.

```
NODE INIT(char *y, int n)
{
 NODE root, son;

 root=ALLOCATE_NODE();
 son=ALLOCATE_NODE();
 PUT_FATHER(root, UNDEFINED);
 PUT_FATHER(son, root);
 PUT_SON(root, *y, son);
 PUT_LABEL(root, UNDEFINED);
 PUT_LABEL(son, y);
 PUT_LENGTH(root, 0);
 PUT_LENGTH(son, n);
 return(root);
}
```

Figure 4.2: Initialization procedure.

```
NODE RESCAN(NODE c, char *beta, int m)
{
 while (m > 0 && m >= GET_LENGTH(GET_SON(c, *beta))) {
    c=GET_SON(c, *beta);
    m-=GET_LENGTH(c);
    beta+=GET_LENGTH(c);
 }
 if (m > 0) return(BREAK_EDGE(GET_SON(c, *beta), m));
 else return(c);
}
```

Figure 4.3: The crucial rescan operation.

```
NODE BREAK_EDGE(NODE x, int k)
{
 NODE y;

 y=ALLOCATE_NODE();
 PUT_FATHER(y, GET_FATHER(x));
 PUT_SON(GET_FATHER(x), *GET_LABEL(x), y);
 PUT_LABEL(y, GET_LABEL(x));
 PUT_LENGTH(y, k);
 PUT_FATHER(x, y);
 PUT_LABEL(x, GET_LABEL(x)+k);
 PUT_LENGTH(x, GET_LENGTH(x)-k);
 PUT_SON(y, *GET_LABEL(x), x);
 PUT_LINK(y, UNDEFINED);
 return(y);
}
```

Figure 4.4: Breaking an edge.

```
NODE SCAN(NODE d, char **gamma)
{
 NODE f;
 int k, lg;
 char *s;

 while (GET_SON(d, **gamma) != UNDEFINED) {
    f=GET_SON(d, **gamma);
    k=1;
    s=GET_LABEL(f)+1;
    lg=GET_LENGTH(f);
    (*gamma)++;
    while (k < lg && **gamma == *s) {
       (*gamma)++;
       s++;
       k++;
    }
    if (k < lg) return(BREAK_EDGE(f, k));
    d=f;
    (*gamma)++;
 }
 return(d);
}
```

Figure 4.5: The scan operation.

Chapter 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 system implement an algorithm based of this notion where lines of the files are considered symbols. Informally, the result of a comparison gives the minimum number of operations (insert a symbol, or delete a symbol) to transform one string into the other, which introduces what is known as the **edit distance** between the strings (see Chapter 6). The comparison of molecular sequences is basically done with a closed concept, alignment of strings, which consists in 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 $w[0 \ldots i-1]$ is a subsequence of $x[0 \ldots m-1]$ if there exists an increasing sequence of integers $(k_j / j = 0, \ldots, i-1)$ such that, for $0 \leq j \leq i-1$, $w[j] = x[k_j]$. We say that a word is an $lcs(x, y)$ if it is a longest common subsequence of the two words x and y. Note that two strings can have several $lcs(x, y)$. Their (unique) length of denoted by $llcs(x, y)$.

A brute force method to compute an $lcs(x, y)$ would consist in computing all the subsequences of x, checking if they are subsequences of y and keeping the longest one. The word x of length m has 2^m subsequences, so this method is impracticable for large values of m.

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:

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

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

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

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

```
void 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; i++) 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 5.1: Dynamic programming algorithm to compute $llcs(x, y) = L[m, n]$.

The relation is used by the algorithm of Figure 5.1 to compute all the values from $L[0, 0]$ to $L[m, n]$. The computation obviously take $O(mn)$ time and space. It is afterward possible to trace back a path from $L[m, n]$ to exhibit an $lcs(x, y)$ (see Figure 5.2).

Example 5.1:
String AGGA is an lcs of x and y.

	x	C	A	G	A	T	A	G	A	G	
y		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

5.2 Reducing the space: Hirschberg algorithm

If only the length of an $lcs(x, y)$ is needed, 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 $O(\min(m, n))$ (see Figure 5.3). Indeed, Hirschberg algorithm computes an $lcs(x, y)$ in linear space (and not only the value $llcs(x, y)$) using the previous algorithm.

Let us define:

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

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

```
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] && 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 5.2: Production of an $lcs(x, y)$.

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

 for (i=0; i <= n; i++) L[i]=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 5.3: $O(\min(m, n))$-space algorithm to compute $llcs(x, y)$.

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

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

Figure 5.4: Computation of L^*.

where the word w^R is the reverse (or mirror image) of the word w. The algorithm of Figure 5.4 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 5.5 the integer i is chosen as $m/2$. After $L[i, j]$ and $L^*[i, j]$ ($0 \leq j < m$) are computed, the algorithm finds an integer k such that $L[i, k] + L^*[i, k] = L[m, n]$. Then, recursively, it computes an $lcs(x[0 \ldots i], y[0 \ldots k])$ and an $lcs(x[i{+}1 \ldots m{-}1], y[k{+}1 \ldots n{-}1])$, 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 5.1.

```
char *HIRSCHBERG(char *x, char *y, int m, int n)
{
 int i, j, k, M, L1[YSIZE], L2[YSIZE];
 char z[YSIZE];

 strcpy(z, "");
 if (m == 0) return;
 else if (m == 1) {
   for (i=0; i < n; i++)
      if (x[0] == y[i]) {
        z[0]=x[0];
        z[1]='\0';
        return(z);
      }
   return(z);
 }
 else {
   i=m/2;
   LLCS(i, n, x, y, L1);
   LLCS_REVERSE(i, m, n, x, y, L2);
   k=n;
   M=L1[n]+L2[0];
   for (j=n-1; j >= 0; j--)
      if (L1[j]+L2[n-j] >= M) {
        M=L1[j]+L2[n-j];
        k=j;
      }
   strcat(z, HIRSCHBERG(i, k, x, y));
   strcat(z, HIRSCHBERG(m-i, n-k, x+i, y+k));
   return(z);
 }
}
```

Figure 5.5: $O(\min(m, n))$-space computation of $lcs(x, y)$.

Chapter 6

Approximate string matching

Approximate string matching consists in 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 must be not greater than a given integer k. We consider two distances, the **Hamming distance** and the **Levenshtein distance**.

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

The Hamming distance between two words w_1 and w_2 of 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 operation:

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

The **Shift-Or algorithm** of the next section is a method that is both very fast in practice and very easy to implement. It adapts to 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 main advantage of the method is that it can adapt to a wide range of problems.

6.1 Shift-Or algorithm

We first present an algorithm to solve the exact string-matching problem using a technique different from those developped in Chapter 2.

Let R^0 be a bit array of size m. Vectore R_i^0 is the value of the array R^0 after text character $y[i]$ has been processed (see Figure 6.1). It contains informations about all matches of prefixes of x that end at position i in the text ($0 \leq j \leq m-1$):

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

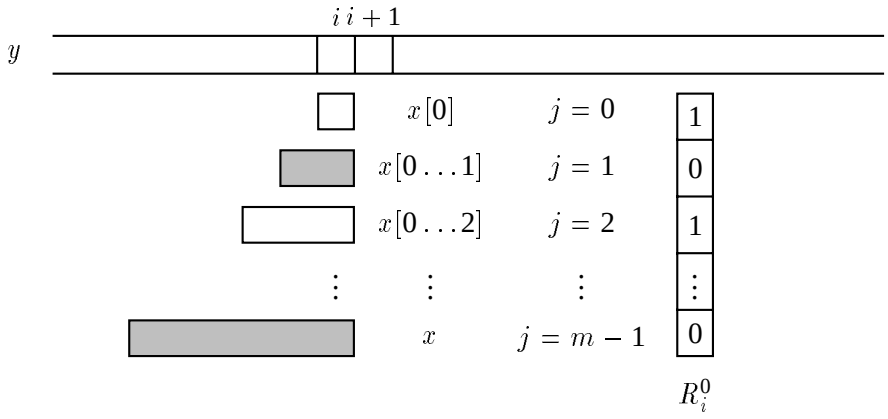


Figure 6.1: Meaning of vector R_i^0.

The vector R_{i+1}^0 can be computed after R_i^0 as follows. For each $R_i^0[j] = 0$:

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

and

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

If $R_{i+1}^0[m-1] = 0$ then a complete match can be reported.

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

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

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+1}^0 reduces to two operations, shift and or:

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

Example 6.1:
$x =$ `GATAA` and $y =$ `CAGATAAGAGAA`

$S_{\mathtt{A}}$	$S_{\mathtt{C}}$	$S_{\mathtt{G}}$	$S_{\mathtt{T}}$
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

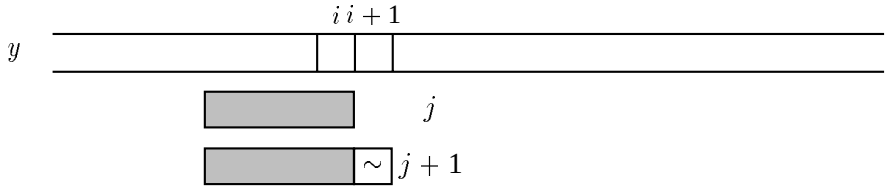


Figure 6.2: If $R_i^0[j] = 0$ then $R_{i+1}^1[j+1] = 0$.

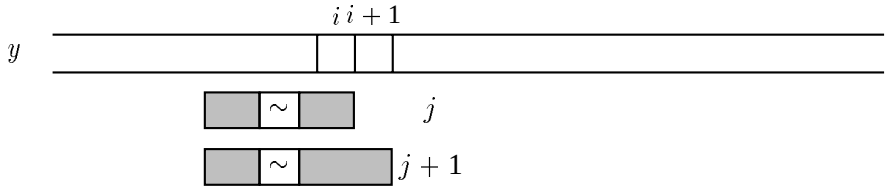


Figure 6.3: $R_{i+1}^1[j+1] = R_i^1[j]$ if $y[i+1] = x[j]$.

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 show 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 indicates all matches with at most one substitution up to the text character $y[i]$. Two cases can arise:

- There is an exact match on the first j characters of x up to $y[i]$ (i.e. $R_i^0[j] = 0$). Then, substituting $y[i+1]$ to $x[j]$ creates a match with one substitution (see Figure 6.2). Thus,

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

- There is a match with one substitution on the first j characters of x up to $y[i]$ and $y[i+1] = x[j]$. Then, there is a match with one substitution of the first $j+1$ characters of x up to $y[i+1]$ (see Figure 6.3). Thus,

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

Finally R_{i+1}^1 can be updated from R_i^1 as follows:

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

Example 6.2:
$x =$ `GATAA` and $y =$ `CAGATAAGAGAA`

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

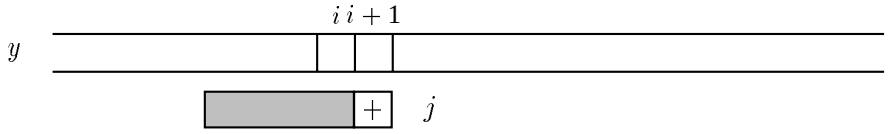


Figure 6.4: If $R_i^0[j] = 0$ then $R_{i+1}^1[j] = 0$.

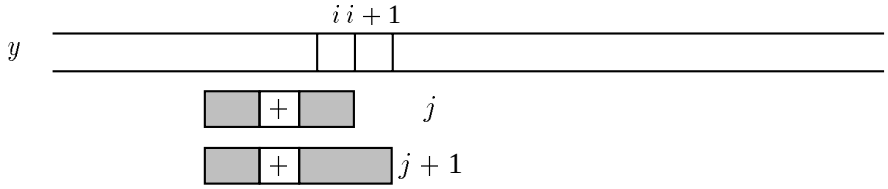


Figure 6.5: $R_{i+1}^1[j+1] = R_i^1[j]$ if $y[i+1] = x[j+1]$.

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 of only one deletion. The method where one insertion, and then where one deletion, is allowed is based on the following elements.

One insertion: R_i^1 indicates all matches with at most one insertion up to text character $y[i]$.
$R_i^1[j] = 0$ if the first $j+1$ characters of x ($x[0 \ldots j]$) match $j+1$ symbols of the last $j+2$ text characters up to $y[i]$.
Array R^0 is maintained as before, and we show how to maintain array R^1. Two cases can arise for a match with at most one insertion on the first $j+2$ characters of x up to $y[i+1]$:

- There is an exact match on the first $j+1$ characters of x ($x[0 \ldots j]$) up to $y[i]$. Then inserting $y[i+1]$ creates a match with one insertion up to $y[i+1]$ (see Figure 6.4). Thus

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

- There is a match with one insertion on the $j+1$ first characters of x up to $y[i]$. Then if $y[i+1] = x[j+1]$ there is a match with one insertion on the first $j+2$ characters of x up to $y[j+1]$ (see Figure 6.5). Thus,

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

Finally, R_{i+1}^1 is updated from R_i^1 with the formula:

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

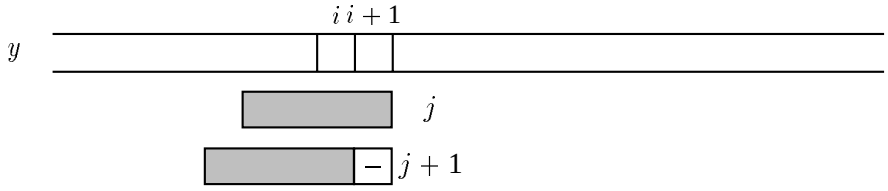


Figure 6.6: If $R^0_{i+1}[j] = 0$ then $R^1_{i+1}[j+1] = 0$.

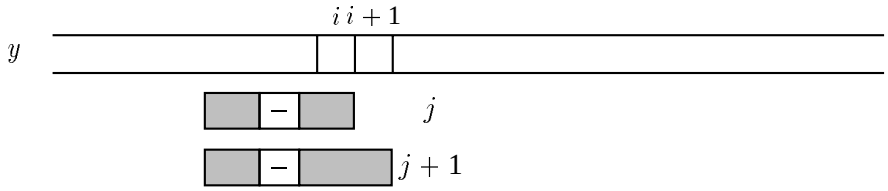


Figure 6.7: $R^1_{i+1}[j+1] = R^1_i[j]$ if $y[i+1] = x[j+1]$.

Example 6.3:
$x =$ `GATAA` and $y =$ `CAGATAAGAGAA`

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

One deletion: We assume now that R^1_i indicates all possible matches with at most one deletion up to $y[i]$. Again two cases arise:

- There is an exact match on the first $j+1$ characters of x ($x[0 \ldots j]$) up to $y[i+1]$ (i.e. $R^0_{i+1}[j] = 0$). Then, deleting $x[j]$ creates a match with one deletion (see Figure 6.6). Thus,

$$R^1_{i+1}[j+1] = R^0_{i+1}[j].$$

- There is a match with one deletion on the first j characters of x up to $y[i]$ and $y[i+1] = x[j]$. Then, there is a match with one deletion on the first $j+1$ characters of x up to $y[i+1]$ (see Figure 6.7). Thus,

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

Finally, R^1_{i+1} is updated from R^1_i with the formula:

$$R^1_{i+1} = (\mathit{SHIFT}(R^1_i) \ \mathit{OR} \ S_{y[i+1]}) \ \mathit{AND} \ \mathit{SHIFT}(R^0_{i+1}).$$

Example 6.4:
$x =$ GATAA and $y =$ CAGATAAGAGAA

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

6.4 Wu-Manber algorithm

We present the approximate string matching with at most k differences of the types: insertion, deletion, and substitution. We need to maintain $k+1$ bit arrays $R^0, R^1, \ldots, R^k$. The vector R^0 is maintained similarly as in the exact matching case (Section 6.1). The other vector are computed with the formula ($1 \leq j \leq k$):

$$\begin{aligned} R^j_{i+1} = \quad & (\textit{SHIFT}(R^j_i) \ \textit{OR} \ S_{y[i+1]}) \\ & \textit{AND} \ \textit{SHIFT}(R^{j-1}_{i+1}) \\ & \textit{AND} \ \textit{SHIFT}(R^{j-1}_i) \\ & \textit{AND} \ R^{j-1}_i, \end{aligned}$$

which can be rewritten into:

$$\begin{aligned} R^j_{i+1} = \quad & (\textit{SHIFT}(R^j_i) \ \textit{OR} \ S_{y[i+1]}) \\ & \textit{AND} \ \textit{SHIFT}(R^{j-1}_{i+1} \ \textit{AND} \ R^{j-1}_i) \\ & \textit{AND} \ R^{j-1}_i. \end{aligned}$$

Example 6.5:
$x =$ GATAA and $y =$ CAGATAAGAGAA and $k = 1$

	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 Wu-Manber algorithm (see Figure 6.8) assumes that the pattern length is no more than the memory-word size of the machine, which is often the case in applications. The preprocessing phase takes $O(\sigma m + km)$ memory space, and runs in time $O(\sigma m + k)$. The time complexity of the searching phase is $O(kn)$.

```
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-m+1);
 }
}
```

Figure 6.8: Wu-Manber approximate string-matching algorithm.

Chapter 7

Text compression

In this section we are interested in algorithm that compress texts. This serves both to save storage place and to save transmission time. We shall consider that the uncompressed text is stored in a file. The aim of algorithms is to produce another file containing the compressed version of the same text. Methods of 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 built 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 say it simply, the algorithm, while scanning the text, replaces some already read segments by just a pointer onto their first occurrences.

7.1 Huffman coding

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 occurrences of characters by their new codewords. The method works for any length of blocks (not only 8 bit) but the running time grows exponentially with this length.

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 do not weaken the compression.

A prefix code on the binary alphabet $\{0, 1\}$ can be represented by a trie (see Section 2.6) that is a binary tree. In the present method codes are complete: they correspond to complete trie (internal nodes have exactly two children). The leaves are labelled by the original characters, edges are labelled by 0 or 1, and labels of branches are the words of the code. The condition on the code imply that codewords are identified with leaves only. We adopt the convention that, from a internal node, the edge to its left son is labelled by 0, and the edge to its right son is labelled by 1.

In the model where characters of the text are given new codewords, Huffman algorithm built 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 most frequent characters are given short codewords while the least frequent symbols have longer codewords.

```
int COUNT(FILE *fin, CODE *code)
{
 int c;

 while ((c=getc(fin)) != EOF) code[c].freq++;
 code[END].freq=1;
}
```

Figure 7.1: Counts the character frequencies.

7.1.1 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 in counting the number of occurrences of each character of the original text (see Figure 7.1). 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 root-tree t for each character a with $weight(t) = freq(a)$,
- Repeat
  - Select the two least frequent trees t_1 and t_2,
  - Create a new tree t_3 with left son t_1, right son t_2 and $weight(t_3) = weight(t_1) + weight(t_2)$
- Until it remains only one tree.

The tree is contructed by the algorithm, the only tree remaining at the end of the procedure, is the coding tree.

In the implementation of the above scheme, a priority queue is used to identify the two least frequent trees (see Figures 7.2 and 7.3). After the tree is built, it is possible to recover the codewords associated with characters by a simple depth-first-search of the tree (see Figure 7.4).

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 must be stored with the compressed text. Again this is done via a depth-first-search 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 ASCII code of the corresponding character on 9 bits (so that the end marker can be equal to 256 if all the ASCII characters are used in the original text). This part of the encoding is shown in Figure 7.5.

Encoding of the original text is realized by the algorithm of Figure 7.6. Bits are written 8 by 8 in the compressed file using both a buffer (`buffer`) and a counter (`bits2go`). Each time the counter is equal to 8, the corresponding byte is written (see Figures 7.7 and 7.8). Since the total number of bits is not necessarily a multiple of 8, some bits may have to be produced at the end (see Figure 7.9).

All the preceding steps are composed to give a complete encoding program, which is given in Figure 7.11.

```
int BUILD_HEAP(CODE *code, PRIORITY_QUEUE queue)
{
 int i, size;
 NODE node;

 size=0;
 for (i=0; i <= END; i++)
    if (code[i].freq != 0) {
      node=ALLOCATE_NODE();
      PUT_WEIGHT(node, code[i].freq);
      PUT_LABEL(node, i);
      PUT_LEFT(node, NULL);
      PUT_RIGHT(node, NULL);
      HEAP_INSERT(queue, node);
      size++;
    }
 return(size);
}
```

Figure 7.2: Builds the priority queue of trees.

```
NODE BUILD_TREE(PRIORITY_QUEUE queue, int size)
{
 NODE root, leftnode, rightnode;

 while (size > 1) {
   leftnode=HEAP_EXTRACT_MIN(queue);
   rightnode=HEAP_EXTRACT_MIN(queue);
   root=ALLOCATE_NODE();
   PUT_WEIGHT(root, GET_WEIGHT(leftnode)+GET_WEIGHT(rightnode));
   PUT_LEFT(root, leftnode);
   PUT_RIGHT(root, rightnode);
   HEAP_INSERT(queue, root);
   size--;
 }
 return(root);
}
```

Figure 7.3: Builds the coding tree.

```
void BUILD_CODE(NODE root, CODE *code, int length)
{
 static char temp[ASIZE+1];
 int c;

 if (GET_LEFT(root) != NULL) {
   temp[length]=0;
   BUILD_CODE(GET_LEFT(root), code, length+1);
   temp[length]=1;
   BUILD_CODE(GET_RIGHT(root), code, length+1);
 }
 else {
   c=GET_LABEL(root);
   code[c].codeword=(char *)malloc(length);
   code[c].lg=length;
   strncpy(code[c].codeword, temp, length);
 }
}
```

Figure 7.4: Builds the character codes by a depth-first-search of the coding tree.

```
void CODE_TREE(FILE *fout, NODE root)
{
 if (GET_LEFT(root) != NULL) {
   SEND_BIT(fout, 0);
   CODE_TREE(fout, GET_LEFT(root));
   CODE_TREE(fout, GET_RIGHT(root));
 }
 else {
   SEND_BIT(fout, 1);
   ASCII2BITS(fout, GET_LABEL(root));
 }
}
```

Figure 7.5: Memorizes the coding tree in the compressed file.

```
void CODE_TEXT(FILE *fin, FILE *fout)
{
 int c, i;

 rewind(fin);
 while ((c=getc(fin)) != EOF)
    for (i=0; i < code[c].lg; i++)
       SEND_BIT(fout, code[c].codeword[i]);
 for (i=0; i < code[END].lg; i++)
    SEND_BIT(fout, code[END].codeword[i]);
}
```

Figure 7.6: Encodes the characters in the compressed file.

```
void SEND_BIT(FILE *fout, int bit)
{
 buffer>>=1;
 if (bit) buffer|=0x80;
 bits2go++;
 if (bits2go == 8) {
   putc(buffer, fout);
   bits2go=0;
 }
}
```

Figure 7.7: Sends one bit in the compressed file.

```
int ASCII2BITS(FILE *fout, int n)
{
 int i;

 for (i=8; i>= 0; i--) SEND_BIT(fout, (n>>i)&1);
}
```

Figure 7.8: Encodes n on 9 bits.

```
void SEND_LAST_BITS(FILE *fout)
{
 if (bits2go) putc(buffer>>(8-bits2go), fout);
}
```

Figure 7.9: Outputs a final byte if necessary.

```
void INIT(CODE *code)
{
 int i;

 for (i=0; i < ASIZE; i++) {
    code[i].freq=code[i].lg=0;
    code[i].codeword=NULL;
 }
}
```

Figure 7.10: Initializes the array code.

```
#define ASIZE 255
#define END   (ASIZE+1) /* code of EOF */

typedef struct {
           int freq, lg;
           char *codeword;
        } CODE;

int buffer;
int bits2go;

void CODING(char *fichin, char *fichout)
{
 FILE *fin, *fout;
 int size;
 PRIORITY_QUEUE queue;
 NODE root;
 CODE code[ASIZE+1];

 if ((fin=fopen(fichin, "r")) == NULL) exit(0);
 if ((fout=fopen(fichout, "w")) == NULL) exit(0);
 INIT(code);
 COUNT(fin, code);
 size=BUILD_HEAP(code, queue);
 root=BUILD_TREE(queue, size);
 BUILD_CODE(root, code, 0);
 buffer=bits2go=0;
 CODE_TREE(fout, root);
 CODE_TEXT(fin, fout);
 SEND_LAST_BITS(fout);
 fclose(fin);
 fclose(fout);
}
```

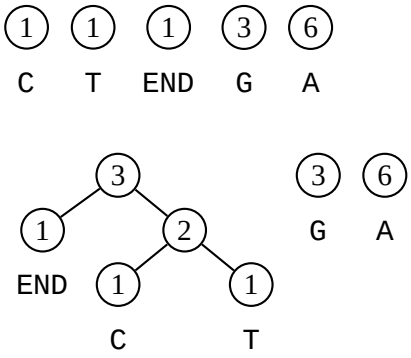
Figure 7.11: Complete function for Huffman coding.

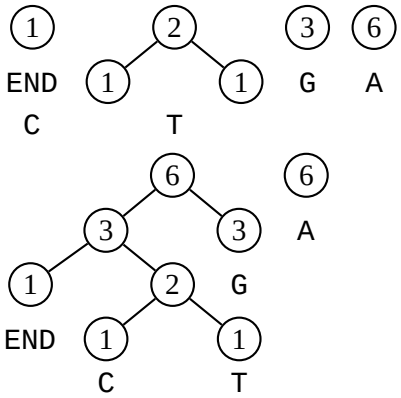
Example 7.1:
$y =$ CAGATAAGAGAA
Length: 12*8=96 bits (if an 8-bit code is assumed)

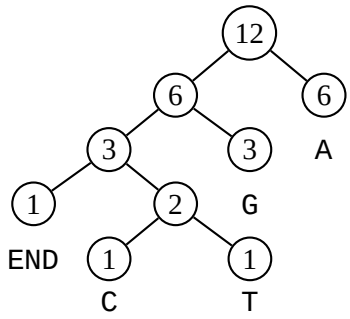
character frequencies:

A	C	G	T	END
6	1	3	1	1

The different steps during the construction of the coding tree:







character codewords:

A	C	G	T	END
1	0010	01	0011	000

Tree code: 0001binary(END,9)01binary(C,9)1 binary(T,9)1binary(G,9)1binary(A,9)
thus: 0001 100000000 01 001000011 1 001010100 1 001000111 1 001000001
Length: 54

Text code: 0010 1 01 1 0011 1 1 01 1 01 1 1 000
Length: 24

Total length: 78

The construction of the tree takes $O(\sigma \log \sigma)$ if the priority queue 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.

```
void REBUILD_TREE(FILE *fin, NODE root)
{
 NODE leftnode, rightnode;

 if (READ_BIT(fin) == 1) { /* leaf */
   PUT_LEFT(root, NULL);
   PUT_RIGHT(root, NULL);
   PUT_LABEL(root, BITS2ASCII(fin));
 }
 else {
   leftnode=ALLOCATE_NODE();
   PUT_LEFT(root, leftnode);
   REBUILD_TREE(fin, leftnode);
   rightnode=ALLOCATE_NODE();
   PUT_RIGHT(root, rightnode);
   REBUILD_TREE(fin, rightnode);
 }
}
```

Figure 7.12: Rebuilds the tree read from the compressed file.

```
int BITS2ASCII(FILE *fin)
{
 int i, value;

 value=0;
 for (i=8; i >= 0; i--) value=value<<1+READ_BIT(fin);
 return(value);
}
```

Figure 7.13: Reads the next 9 bits in the compressed file and returns the corresponding value.

```
void DECODE_TEXT(FILE *fin, FILE *fout, NODE root)
{
 NODE node;

 node=root;
 while (1) {
    if (GET_LEFT(node) == NULL)
      if (GET_LABEL(node) != END) {
        putc(GET_LABEL(node), fout);
        node=root;
      }
      else break;
    else if (READ_BIT(fin) == 1) node=GET_RIGHT(node);
    else node=GET_LEFT(node);
 }
}
```

Figure 7.14: Reads the compressed text and produces the uncompressed text.

7.1.2 Decoding

Decoding a file containing a text compressed by Huffman algorithm is a mere programming exercise. First the coding tree is rebuild by the algorithm of Figure 7.12. It uses a function to decode a integer written on 9 bits (see Figure 7.13). 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 branch when a 0 is read or a right branch 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. The decoding of the text is presented in Figure 7.14. Again in this algorithm the bits are read with the use of a buffer (`buffer`) and a counter (`bits_in_stock`). A byte is read in the compressed file only if the counter is equal to zero (see Figure 7.15).

The complete decoding program is given in Figure 7.16. 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.

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 probality distribution of symbols).

The dictionary is the central point of the algorithm. It has the property to be 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 Lempel-Ziv-Welsh method after several improvement introduced by Welsh. The algorithm is implemented by the `compress` command existing under the UNIX operating system.

```
int READ_BIT(FILE *fin)
{
 int bit;

 if (bits_in_stock == 0) {
   buffer=getc(fin);
   bits_in_stock=8;
 }
 bit=buffer&1;
 buffer>>=1;
 bits_in_stock--;
 return(bit);
}
```

Figure 7.15: Reads the next bit from the compressed file.

```
int buffer;
int bits_in_stock;

void DECODING(char *fichin, char *fichout)
{
 FILE *fin, *fout;
 NODE root;

 if ((fin=fopen(fichin, "r")) == NULL) exit(0);
 if ((fout=fopen(fichout, "w")) == NULL) exit(0);
 INIT(code);
 buffer=bits_in_stock=0;
 root=ALLOCATE_NODE();
 REBUILD_TREE(fin, root);
 UNCOMPRESS(fin, fout, root);
 fclose(fin);
 fclose(fout);
}
```

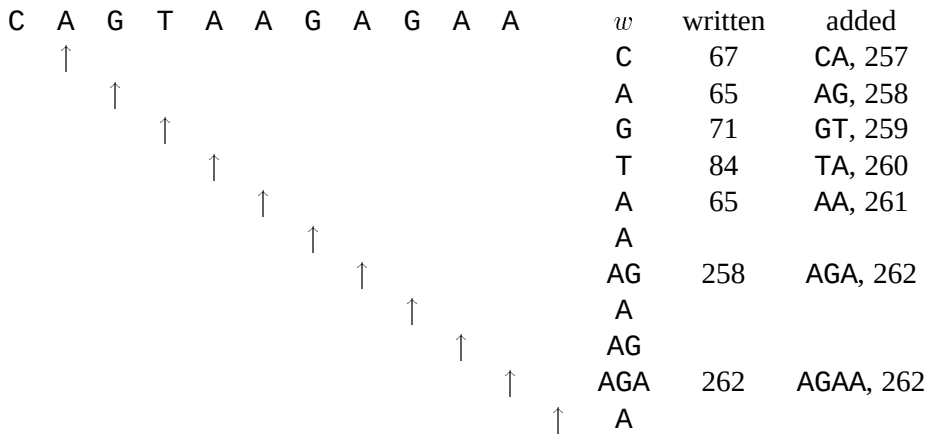
Figure 7.16: Complete function for decoding.

7.2.1 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 of the text. Let a be the next symbol (just following w). Then we proceed that way:

- 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 w is the first letter of the source text.
Example 7.2:
$y =$ CAGTAAGAGAA



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

7.2.2 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 on 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.

```
unsigned int GET_SON(int father, int son)
{
 int index, offset;

 index=(son<<(BITS-8))^father;
 if (index == 0) offset=1;
 else offset=TABLE_SIZE-index;
 while (1) {
    if (dict[index].code== UNDEFINED) return(index);
    if (dict[index].father == father &&
        dict[index].character == (char)son) return(index);
    index-=offset;
    if (index < 0) index+=TABLE_SIZE;
 }
}
```

Figure 7.17: Hashing function to access the dictionary.

Example 7.3:
decoding: 67, 65, 71, 84, 65, 258, 262, 65

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

7.2.3 Implementation

For the compression algorithm shown in Figure 7.18, the dictionary is stored in a table declared as follows:

```
struct dictionary {
   int code, father;
   char character;
} dict[TABLE_SIZE];
```

The table is accessed with a hashing function (see Figure 7.17) in order to have a fast access to the son of a node.

For the decompression algorithm, no hashing technique is necessary. Having the index of the next segment, a bottom-up walk in the trie underlying the dictionary produces the mirror image of the segment. A stack is then used to reverse it (see Figure 7.19). The bottom-up walk follows the parent links of the data structure. The decompression algorithm is given Figure 7.20.

```
void COMPRESS(fin, fout)
   FILE *fin, *fout;
{
 int next_code, character, string_code;
 unsigned int index, i;

 PREPARE_WRITE();
 next_code=FIRST_CODE;
 for (i=0; i < TABLE_SIZE; i++) dict[i].code=UNDEFINED;
 if ((string_code=getc(fin)) == EOF) string_code=END;
 while ((character=getc(fin)) != EOF) {
    index=GET_SON(string_code, character);
    if (dict[index].code != UNDEFINED)
      string_code=dict[index].code;
    else {
      if (next_code <= MAX_CODE) {
        dict[index].code=next_code++;
        dict[index].father=string_code;
        dict[index].character=(char)character;
      }
      OUTPUT_BITS(fout, string_code);
      string_code=character;
    }
 }
 OUTPUT_BITS(fout, string_code);
 OUTPUT_BITS(fout, END);
 SEND_LAST_BITS(fout);
}
```

Figure 7.18: LZW compression algorithm.

```
unsigned int DECODE_STRING(count, code)
   unsigned int count, code;
{
 while (code > 255) {
    decode_stack[count++]=dict[code].character;
    code=dict[code].father;
 }
 decode_stack[count++]=(char)code;
 return(count);
}
```

Figure 7.19: Bottom-up search in the coding tree.

```
void UNCOMPRESS(fin, fout)
   FILE *fin, *fout;
{
 unsigned int next_code, new_code, old_code, count;
 int character;

 PREPARE_READ();
 next_code=FIRST_CODE;
 old_code=INPUT_BITS(fin, BITS);
 if (old_code == END) return;
 character=old_code;
 putc(old_code, fout);
 while ((new_code=INPUT_BITS(fin, BITS)) != END) {
    if (new_code >= next_code) {
      decode_stack[0]=(char)character;
      count=DECODE_STRING(1, old_code);
    }
    else count=DECODE_STRING(0, new_code);
    character=decode_stack[count-1];
    while (count > 0) putc(decode_stack[--count], fout);
    if (next_code <= MAX_CODE) {
      dict[next_code].father=old_code;
      dict[next_code].character=(char)character;
      next_code++;
    }
    old_code=new_code;
 }
}
```

Figure 7.20: LZW decompression algorithm.

Source text size in bytes	French 62816	C sources 684497	Alphabet 530000	Random 70000
Huffman	53.27%	62.10%	72.65%	**55.58**%
Ziv-Lempel	**41.46**%	34.16%	2.13%	63.60%
Factor	47.43%	**31.86**%	**0.09**%	73.74%

Figure 7.21: Sizes of texts compressed with three algorithms.

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

7.3 Experimental results

The table of Figure 7.21 contains a sample of experimental results showing the behaviour 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 containg 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: Huffman algorithm of Section 7.1, Ziv-Lempel algorithm of Section 7.2, and a third algorithm called Factor. This latter algorithm encodes segments 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 7.21 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%.

Chapter 8

Research Issues and Summary

The string searching algorithm by hashing has been introduced by Harrison (1971), and later fully analysed by Karp and Rabin (1987).

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

Boyer-Moore algorithm has been 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 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 Boyer-Moore algorithm avoid the quadratic behaviour when searching for all occurrences of the pattern. Among the more efficient in term of number of symbol comparisons are: the algorithm of Apostolico and Giancarlo (1986), Turbo-BM algorithm by Crochemore et alii (1992) (the two algorithms are analysed 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 BM algorithm similar to the Quick Search algorithm of Sunday (1990) described in the report have been 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 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 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 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 Chapter 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).

All these data structures implement full indexes while applications sometimes need only uncomplete indexes. The design of compact indexes is still unsolved.

Hirchsberg (1975) presents the computation of the LCS in linear space. This is an important result because the algorithm is used on large sequences. The quadratic time complexity of the algorithm to compute the Levenstein distance is a bottleneck in practical string comparison for the same reason.

The 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 book related to compiling techniques. The algorithms of Chapter 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 coding tree is used to encode the next character and simultaneously 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) present practical implementations of various compression algorithms.

Chapter 9

Defining Terms

Border: A word $u \in \Sigma^*$ is a segment 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 form w by deleting zero or more symbols that need not be consecutive. (u is also 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 which edges are labelled by letters or words.

Chapter 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. 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.

Hirchsberg, 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. To appear in *Software - Practice & Experience*.

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.

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.

Chapter 11

Further Information

Problems and algorithms presented in the report are just a sample of questions related to pattern matching. They share in common 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.
- 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*, 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).
- *Data Compression Conference*, which is regularly held at Snowbird.

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