Boyer-Moore strategy to efficient approximate string matching

Nadia El-Mabrouk and Maxime Crochemore

IGM, Université Marne la Vallée,
2 rue de la Butte Verte, 93166 Noisy Le Grand Cedex

Abstract. We propose a simple but efficient algorithm for searching all occurrences of a pattern or a class of patterns (length m) in a text (length n) with at most k mismatches.
This algorithm relies on the *Shift-Add* algorithm of Baeza-Yates and Gonnet [6], which involves representing by a bit number the current state of the search and uses the ability of programming languages to handle bit words. State representation should not, therefore, exceeds the word size ω, that is, $m(\lceil \log_2(k+1) \rceil + 1) \leq \omega$. This algorithm consists in a preprocessing step and a searching step. It is linear and performs $3n$ operations during the searching step.
Notions of shift and character skip found in the Boyer-Moore (BM) [9] approach, are introduced in this algorithm. Provided that the considered alphabet is large enough (compared to the Pattern length), the average number of operations performed by our algorithm during the searching step becomes $n(2 + \frac{k+4}{m-k})$.

1 Introduction

Our purpose is approximate matching of a pattern or a class of patterns in a text, all sequences of characters or classes of characters from a finite alphabet Σ. Errors considered here are mismatches. A class of patterns, is a set of patterns with don't care symbols, patterns containing the complementary of a character or any other class of characters. Such a problem has a lot of applications, in particular in molecular biology for predicting potential nuclear gene-coding sequences in genomic DNA sequences. In fact, exact string matching is not sufficient since gene-coding sequences are in general only partially and approximately specified.

Concerning exact string matching, algorithms based on the Boyer-Moore (BM) [9, 13] approach are the fastest in practice. Such algorithms are linear and may even have a sublinear behaviour, in the sense that every character in the text need not be checked. In certain cases, text characters can be "skipped" without missing a pattern occurrence. The larger the alphabet and the longer the pattern, the faster the algorithm works.

Various algorithms have been developed for searching with k mismatches all occurrences of a pattern (length m) in a text (length n), both defined over an alphabet Σ (length c). Running times have ranged from $O(mn)$ for the naive algorithm, to $O(kn)$ [15, 11] or $O(n \log m)$ [12]. The first two algorithms consist in a preprocessing step and a searching step. Grossi and Luccio algorithm [12] uses the suffix tree. Other algorithms have used the BM approach in approximate string matching [4, 18]. Running times are $O(kn)$ for Baeza-Yates and Gonnet [4] and $O(kn(\frac{1}{m-k}+\frac{k}{c}))$ for Tarhio and Ukkonen [18]. The problem of approximate matching of a class of patterns was also studied [2, 1, 5], especially in the case of patterns with don't care symbols [10, 17, 16, 3, 8, 14]. Fisher et Paterson [10] developed an $O(n \log c \log^2 m \log\log m)$ time algorithm based on the linear product. Abrahamson [1] extended this method for generalized string pattern. Pinter [17] has used the Aho and Corasick automaton [2] for searching a set of patterns. Other algorithms have considered the problem of exact matching of patterns with variable length don't cares [16, 8, 14]. As for Akutsu [3], he developed an $O(\sqrt{km}\, n \log c\ \log^2 \frac{m}{k}\ \log\log \frac{m}{k})$ time algorithm for searching a pattern with don't cares in a text with don't cares.

In 1992, several new algorithms for approximate string matching were published [6, 19, 7]. They combine both speed and programming practicality, in contrast with older results, most of which being mainly of theoretical interest. Moreover, they are flexible enough to allow searching for a class of patterns. These algorithms consist in a pattern preprocessing step and a searching step. They are all based on the same approach, consisting in finding, at a given position in the text, all approximate pattern prefixes ending at this position. Speed is increased by representing the state of the search as a bit number [6, 19] or an array [7], and by using the ability of programming languages to handle bit words.

Nevertheless, these algorithms are based on a naive approach and process each character of the text. Our goal is to speed up searching by using a BM strategy and including notions of shift and character skip.

We have chosen to consider such an improvement in the case of the *Shift-Add* algorithm of Baeza-Yates and Gonnet [6]. The main idea of *Shift-Add* is to represent the state of the search as a bit number, and perform a few simple arithmetic and logical operations. Provided that representations don't exceed the word size ω, that is $m(\lceil \log_2(k+1) \rceil + 1) \leq \omega$, each search step does exactly a shift, a test and an addition. Therefore, this algorithm runs in $O(n)$ time and the searching step does $3n$ operations. We developed an algorithm combining the practicality of the *Shift-Add* method and the speed of the BM approach. Provided that the considered alphabet is large enough compared to m, our new algorithm performs on average $n\left(2+\frac{k+4}{m-k}\right)$ operations during the searching step.

The paper is organized as follows. Section 2 summarises the algorithm *Shift-Add*, in the case of exact or approximate matching of a pattern or a class of patterns. Section 3 develops the adaptation of the BM approach to the *Shift-Add* method. An improvement of this last algorithm is given in Section 4. Finally, section 5 gives experimental results obtained with both algorithms.

2 Shift-Add Algorithm

Let $P = p_1 \cdots p_m$ be a pattern and $t = t_1 \cdots t_n$ be a text over a finite alphabet Σ. The problem is to find in t all occurrences of P with at most k mismatches ($0 \leq$

$k \leq m$). In other words, the distance between two patterns of the same length will be defined as the number of their mismatching characters (the Hamming distance). An equivalent problem is then to find in t all substrings of length m such that the Hamming distance between these substrings and P is at most k, that is to find all j positions in the text such that, for $1 \leq i \leq m$, $p_i = t_{j-m+i}$, except for at most k indices.

The main idea is to represent the state of the search as a vector of size m. Thus, S_j denotes the state vector given a current position j in the text. S_j contains individual states of the search between each prefix of P and the corresponding substring of t. Namely, for $1 \leq i \leq m$, $S_j[i]$ is the number of mismatches between $p_1 \cdots p_i$ and $t_{j-i+1} \cdots t_j$.

P matches at j if and only if $S_j[m] < k+1$.

When t_{j+1} is read, the number of mismatches for each prefix of P needs to be completed. Values of boolean expressions $t_{j+1} = p_i$, for $1 \leq i \leq m$, can be computed during a preprocessing step. For each character a in Σ, a vector T_a of size m is constructed such that :

For i, $1 \leq i \leq m$, $\quad T_a[i] = \begin{cases} 0 \text{ if } a = p_i \\ 1 \text{ otherwise} \end{cases} \qquad$ **(1)**

(it is sufficient to construct the T arrays only for characters appearing in the pattern).

Finally, $S_{j+1}[i] = S_j[i-1] \; + \; T_{t_{j+1}}[i]$.

In order to obtain S_{j+1} from S_j by simple arithmetic and logical operations, vectors are considered as numbers and represented in base 2^b, where b is the bit number needed to represent each vector component.

Thus, $S_j = \sum_{i=1}^{m} S_j[i] 2^{(i-1)b}$ and $T_a = \sum_{i=1}^{m} T_a[i] 2^{(i-1)b}$.

Representations should not exceed the word size ω, namely, $mb \leq \omega$.

It is easy now to verify that the transition from S_j to S_{j+1} amounts to no more than a left shift (denoted by $<<$) of b bits and an addition :

$$S_{j+1} = (S_j \;<<\; b) \; + \; T_{t_{j+1}} \qquad \textbf{(2)}$$

Initial state is $S_0 = 0$. P matches at j if and only if $S_j < (k+1)2^{(m-1)b}$.

Possible values of the vector state components are $1, \cdots, m$. Thus, to represent each component, $b = \lceil \log_2(m+1) \rceil$ bits are required. However, since we need only to compare the number of mismatches with k, it is enough to represent values from 1 to k. In this case, one more bit is needed for carrying over additions. The improved algorithm uses $b = \lceil \log_2(k+1) \rceil + 1$ bits. At each position j in the text, the overflow bits are recorded in an overflow state R_j and the overflow bits of S_j are reset.

The *Shift-Add* algorithm works in $O(n)$ time, and the searching step (disregarding the overflow state) performs $3n$ operations. In fact, at each step, that is for each position j, three operations are performed: one shift, one addition and one test to determine whether P matches at position j.

2.1 Exact string matching

In the case of exact string matching, it is only necessary to know whether a given prefix of P matches exactly the considered substring of t. We define S_j as follows:

$$\text{For } 1 \leq i \leq m, \quad S_j[i] = \begin{cases} 0 \text{ if } p_1 \cdots p_i = t_{j-i+1} \cdots t_j \\ 1 \text{ otherwise} \end{cases}$$

When t_{j+1} is read, we need to determine whether t_{j+1} can extend any of the partial matches. Thus, in order to have a match of $p_1 \cdots p_i$ at position $j+1$, both $S_j[i-1] = 0$ and $t_{j+1} = p_i$ should be satisfied. Here, $b = 1$, and in formula **(2)** the + symbol should be replaced by an OR operation. The algorithm based on this new formula is called *Shift-Or*.

2.2 Extensions

Flexibility is one of the principal advantages of the *Shift-Add* method. It can be easily adapted to a class of patterns. A class of patterns is a set of patterns, defined by a string in which each position is a set of characters. A set of characters is for example a subset of Σ or the complementary of a subset of Σ. A pattern class defined by a string in which each position is either a single character or the whole alphabet is called pattern with don't care symbols.

To take into account such classes, only the definition of the T array needs changing: for a position i in P and a character a in Σ, $T_a[i]$ will contain 0 if a belongs to the set of characters corresponding to that position in P, and 1 otherwise. Thus, the T array computed during the preprocessing step contains all needed information about the pattern. Then, the searching step is not modified.

3 Boyer-Moore Approach to Shift-Add method

Here, we consider the problem of searching all occurrences of a pattern string $P = p_1 \cdots p_m$ in a text string $t = t_1 \cdots t_n$ with at most k mismatches, $1 \leq k \leq m$. The problem of exact string matching can be solved by substituting the OR operation to the *add* operation. In the case of string matching with classes, the T array is modified as in section (2.2).

For some position j, the state S_j is a bit number (represented in base 2^b) defined as previously: each individual state $S_j[i]$, for $1 \leq i \leq m$, contains the number of mismatches between $p_1 \cdots p_i$ and $t_{j-m+1} \cdots t_j$. Here, the introduction of the overflow state is ignored.

3.1 Shift

Our goal is to avoid processing each character of the text, in other words, avoid computing S_j for every position j. It is easy to see that if for some prefix of length i of P, $S_j[i] > k$, then since $S_{j+(m-i)}[m] \geq S_j[i]$, P will not occur at position $j + (m - i)$. The following proposition can be deduced:

Proposition 1. *For some position j in t, let l be the largest index i, $1 \leq i \leq m-1$, such that $S_j[i] \leq k$, if such an index exists and 0 otherwise. Let $d = m - l$.*

Then, the next position after j where P is likely to occur is $j_{next} = j + d$. In other words, $S_{j'}[m] > k$, for every j' such that $j < j' < j_{next}$.

d is the next shift and $1 \leq d \leq m - k$.

Consider now the transition between S_j and S_{j+d}.

The number of mismatches between the prefix $p_1 \cdots p_i$ of P, for $d+1 \leq i \leq m$, and the substring $t_{j+d-i+1} \cdots t_{j+d}$ of t, i.e. $S_{j+d}[i]$, is the sum of the number of mismatches between $p_1 \cdots p_{i-d}$ and $t_{j+d-i+1} \cdots t_j$, i.e. $S_j[i-d]$, and the number of mismatches between $p_{i-d+1} \cdots p_i$ and $t_{j+1} \cdots t_{j+d}$.

The T array defined in **(1)** contains the information about the occurrence of a given character a at a given position in the pattern.

Consequently:

$$S_{j+d}[i] = \begin{cases} S_j[i-d] + \sum_{r=0}^{d-1} T_{t_{j+d-r}}[i-r] \text{ if } d < i \leq m \\ \sum_{r=0}^{i-1} T_{t_{j+d-r}}[i-r] \text{ otherwise} \end{cases} \quad \textbf{(3)}$$

In order to obtain S_{j+d} as a sum of numbers in base 2^b, the next definition is needed:

Definition 2. D denotes the $|\Sigma| \times m$ matrix such that, element $D[a][m-r]$ for each $a \in \Sigma$ and $0 \leq r \leq m-1$, is denoted by $D_{a,m-r}$ and defined as follows:

$$D_{a,m-r} = \sum_{i=r+1}^{m} T_a[i-r] 2^{(i-1)b} \quad .$$

Intuitively, $D_{a,m-r}$ denotes positions in $p_1 \cdots p_{m-r}$ containing character a. For a fixed r, $D_{a,m-r}$ is obtained by a left shift of T_a of rb positions.

S_{j+d} can then be represented as follows:

$$S_{j+d} = (S_j \ << \ bd) \ + \sum_{r=0}^{d-1} D_{t_{j+d-r},m-r} \quad .$$

Initial values are $S_0 = 0$ and $d = m$.

Practically, in order to determine the shift d, S_j is shifted b bits at a time, until the obtained number is below $k2^{(m-1)b}$. S_j will have finally been shifted d times to obtain $S_j << bd$. Therefore, shifts are not grouped.

Example 1: Let $\Sigma = \{a, b, c, d\}$, $P = abbac$ and $k = 1$.

The D matrix is :

	a	b	c	d
1	00000	10000	10000	10000
2	10000	01000	11000	11000
3	11000	00100	11100	11100
4	01100	10010	11110	11110
5	10110	11001	01111	11111

Successive states and shifts when searching P in t with at most k mismatches:

Positions	0	1	2	3	4	5	6	7	8	9	10	11
t		a	b	d	a	b	b	a	b	b	a	c
S_j	00000					24201			13201			03311
d	5					3			3			3

Remark: Introduction of shifts does not improve the complexity of the *Shift-Add* algorithm . It only has the effect of grouping additions and tests. However, shifts are essential to introduce the notion of characters skip which will finally speed up the algorithm.

3.2 Character skip

The Boyer-Moore (BM) algorithm is an efficient exact string matching algorithm. It is fast since it is possible, in certain conditions, to skip substrings of the text, that is not process them, without loss of information. At each step, characters of the text are processed from right to left.

In this section, we try to find conditions in which parts of the text can be avoided without missing occurrences of the pattern.

Assume j is the last position scanned in the text and d is the next shift. The substring of the text still to be scanned at this step of the search is then $t_{j+1} \cdots t_{j+d}$. This substring is processed from right to left, that is beginning with t_{j+d}, and the processing stops when t_{j+1} is reached, or when the information for all prefixes of P ending at position $j+d$ is obtained.

Practically, in order to compute state S_{j+d}, S_j should first be shifted on the left of bd bits. Let $S_{j+d,0} = S_j << bd$ be the obtained number. Then, each of the d characters $t_{j+d-r+1}$, with $1 \leq r \leq d$, should be processed. Let $S_{j+d,r}$ be the partial state obtained after processing characters $t_{j+d}, \cdots, t_{j+d-r+1}$ of t. Then, $S_{j+d,r} = S_{j+d,r-1} + D_{t_{j+d-r+1},m-r+1}$ and we have $S_{j+d} = S_{j+d,d}$.

For given indexes r, $1 \leq r \leq d$, and i, $1 \leq i \leq m$:

(a) If $S_{j+d,r}[i] > k$, then without processing the remaining characters $t_{j+d-r}, \cdots, t_{j+1}$, we know that the prefix of length i of P does not occur at position $j+d$.

(b) If $S_{j+d,r}[i] \leq k$ and no more comparisons have to be performed for the prefix of length i of P, then this prefix matches at position $j+d$.

Therefore, instead of computing the number of mismatches with the corresponding substring of t, for each prefix of P, i.e. the terminal state, the computation stops at the first partial state giving enough information for further processing. Let S'_j be this partial state. Differences between S_j and S'_j are located only in individual states exceeding $k+1$.

Algorithm We suppose that the D matrix has been computed during a preprocessing step. For a given position j in the text, an index r and a prefix of length i of P, *State* denotes the bit number consisting in individual states of partial state $S_{j,r}$ for prefixes with length from 1 to i. More precisely, State $= S_{j,r}[i] \cdots S_{j,r}[1]0 \cdots 0$.

Algorithm1: BM approach to approximate string matching

$j := m$; $d := m$; State $:= 0$;
lim $:= (k+1) << (m-1)$;
While $j \leq n$ do
    (0) $i := m$; $r := 0$;
    (1) *If* State $\geq$ lim then
            State := State $<< b$;
            $i := i-1$; Go to (1);
    (2) *Else* :
    (2.1) *If* MIN$(d, i) > r$ then
            $r := r+1$;
            State := State + $D_{t_{j-r}, m-r}$; Go to (1).
    (2.2) *Else* :
        (2.2.1) *If* $i = m$ then
            "Occurrence of P at position j";
            State := State $<< b$;
            $i := i-1$; Go to (1);
        (2.2.2) *Else* :
            $d := m - i$;
    $j := j + d$;
End of While.

Proposition 3. *Algorithm1 finds all occurrences of the pattern P in the text t with at most k mismatches.*

PROOF :

Step **(1)** of the algorithm corresponds to situation **(a)**, that is when the partial number $S_{j,r}[i]$ of mismatches found at this step of the search for the prefix i of P, exceeds $k+1$. In this case, this prefix is ignored and the next prefix $i-1$ of P is considered.

Step **(2.1)** corresponds to the situation where there is not enough information to stop comparing. In fact, for the prefix i of P, the number $S_{j,r}[i]$ of mismatches obtained at this step of the search is less than $k+1$, but a number of comparisons remain to be done for this prefix.

Step **(2.2)** corresponds to situation **(b)**, that is when $S_{j,r}[i] < k+1$ for the prefix i, no more comparisons have to be performed for this prefix. In this case, if $i = m$, then position j matches and the search for the next shift goes on. If $i < m$, then the next shift is equal to $m - i$. In fact, i corresponds to the length of the longest prefix of P matching the corresponding substring of t •

Example 2: Let Σ, P, t and k be those defined in example 1.
Shifts are the same as for example1 and only state S_5 is not the terminal state.

Positions	0	1	2	3	4	5	6	7	8	9	10	11
t		a	b	d	a	b	b	a	b	b	a	c
S_j	00000					23201			13201			03311
d	5					3			3			3

Complexity Our goal is to evaluate the average number of operations performed by *Algorithm1*. Operations are of three kinds: shifts, additions and tests.

Recall that the scanning of t by the *Shift-Add* algorithm needs $3n$ operations, since each search step does exactly a shift, an addition and a test. It is not difficult to see that our algorithm does the same number of shifts (n) and less additions. In fact, one addition is performed for each character processed in the text, and not all characters are examined. However, the number of tests increases, since in addition to those considered by the *Shift-Add* algorithm, those which make transitions between partial states should be considered.

Let t be a random text and $|\Sigma| = c$. The probability of a given character to occur at a given position in the text is then $\frac{1}{c}$.

Let X be a random variable denoting the length of the shift in *Algorithm1* when searching pattern P in the random text t with at most k mismatches. The following lemma gives the average shift d_m, that is the expected value $< X >$ of the random variable X.

Lemma 1 *Provided c is large enough compared to m, the average shift d_m obtained by Algorithm1 exceeds d'_m, with:*

$$d'_m \sim \left(m - k - 1 + \left(1 - \frac{1}{c}\right)^{k+1}\right) \left(1 - \frac{1}{c}\right)$$

Now, we analyze the average number M_k of characters processed at a given position $j + d_m$ of the text, where j is the last position scanned in the text. This number of characters is the length of the smallest substring of t ending at position $j + d_m$ and mismatching all substrings of P which are not prefixes. The maximum number of characters to be processed at this step is d_m.

From the last remarks, we can deduce the following lemma:

Lemma 2 *Provided c is large enough compared to m, $M_k \sim k+2$.*

We are able now to evaluate the complexity of *Algorithm1*.

Proposition 4. *The average number OP_k of operations performed by Algorithm1 is $n\left(2+\frac{3M_k+2}{d_m}\right)$. When the considered alphabet is large enough, this number becomes $OP_k \sim n\left(2+\frac{3k+8}{m-k}\right)$.*

PROOF :

Let $OP_{d_m,k}$ be the average number of operations performed by Algorithm1 at each step of the search. Thus, $OP_k = \frac{n}{d_m} OP_{d_m,k}$.

At each step of the search, operations performed by *Algorithm1* are: M_k additions (one addition per character), d_m shifts (lines 5. and 14. of the algorithm) and at most d_m+2+2M_k tests. In fact, note first that condition 4 ($State \geq \lim$, step (1)) is true at most d_m times and in that case we do not proceed to step (2). Thus exactly d_m tests are performed in these cases. Moreover, in order to know the next shift, we should go once through step (2.2.2) and then do tests 4. and 8. (test 12. could be avoided by changing the algorithm such that case $i=m$ is examined at a previous step). Finally, since there are exactly M_k additions, we should go through line 10. exactly M_k times and at each time do tests 4. and 8. The average number of tests is therefore $T_{d_m,k} = d_m + 2 + 2M_k$.

So, $OP_{d_m,k} = 2d_m + 2 + 3M_k$ and $OP_k = n\left(2+\frac{3M_k+2}{d_m}\right)$.

The case of a large alphabet is deduced from lemmas 1 and 2 •

4 Improvement

In order to speed up the algorithm, it is obvious that a way should be found to perform less tests. Our idea is to process a certain number of characters at each step of the search, that is, do a certain number of additions before beginning tests.

Algorithm2

Let j be the current position in the text and d be the last shift obtained. We denote by C_d the following number: $C_d = \min\,(d, k+2)$.

C_d is the average number M_k (lemma 2) of characters processed at each step by *Algorithm1*, provided this number does not exceed the maximum number of characters to be processed at this step, that is d.

Thus, before going through steps (1)-(2), our improved algorithm (*Algorithm2*) will process first the C_d characters $t_{j-C_d+1}\cdots t_j$ and compute the partial state S_{j,C_d}, that is do C_d additions.

Algorithm2 is hence obtained by adding a preliminary step (0') before step (0) in *Algorithm1*.

$$\textbf{(0')}\quad \text{State} := \text{State} + \sum_{l=0}^{C_d-1} D_{t_{j-l},m-l}\ ;$$
$$r := r + C_d;$$

Obviously, *Algorithm2* finds the same results, the same shifts and so the same average shift d_m that *Algorithm1*.

Proposition 5. *Provided the alphabet is large enough, the average number of operations performed by Algorithm2 is* $OP_k \sim n\left(2+\frac{k+4}{m-k}\right)$.

PROOF :

Our goal is to evaluate the average number $OP_{d_m,k}$ of operations performed by *Algorithm2* at any search step.

First, M_k characters are processed and M_k additions are performed. Two cases are then encountered:

1. The partial state holds enough information, so no more characters are processed at this step. In this case, d_m+2 tests and d_m shifts are performed.
2. The partial state does not hold enough information. In this case, the maximum number of characters still to be performed is $d_m - M_k$. The number of shifts is the same as that of the previous state and there are $2(d_m - M_k)$ more tests.

Let P_k be the probability of the second case. Then, $OP_{d_m,k} = M_k + 2d_m + 2 + P_k\left(3\left(d_m - M_k\right)\right)$

and

$$OP_k = n\left(2+\frac{M_k+2+3P_k\left(d_m-M_k\right)}{d_m}\right)$$

When the considered alphabet is large enough, $d_m \sim m-k$ (Lemma1), $M_k \sim k+2$ (Lemma2) and we can prove that $P_k \sim 0$. Thus, $OP_k \sim n\left(2+\frac{k+4}{m-k}\right)$ •

5 Experiments

Our goal is to find out under which conditions *Algorithm2* is fastest than algorithm *Shift-Add*.

Algorithm3: Exact string matching

```
j := m; d := m; State := 0;
lim:= (k + 1) << (m − 1); initial:=1_{mb}1_{mb−1} ··· 1;
While j ≤ n do
    i := m; r := 0 ;
    While State ≠ initial and r < d do
        State := State OR D_{t_{j−r},m−r};
        r:=r+1;
    End of While.
    If State = initial then
        State := 0;
        d := m;
    Else
        If State < lim then
            "Occurrence of P at position j";
        State := State << b;
        While State ≥ lim
            State := State << b;
            i := i − 1;
        End of While.
        d := m − i;
j := j + d;
End of While.
```

5.1 Exact string matching

In this case, the considered algorithm is *Shift-Or* (2.1). Baeza-Yates and Gonnet have introduced the following improvement: if at a given position j in t, $S_j = 1_{mb}1_{mb-1}\cdots 1$, that is all prefixes of P mismatch at position j, then the next character processed in the text is p_1 (if such a character exists). In fact, the state remains the same for all other characters.

We improve *Algorithm2* as well: at a given step of the search, characters are processed until the partial state is equal to $1_{mb}1_{mb-1}\cdots 1$ (*Algorithm3*).

We have experimented algorithms *Shift-Or* and *Algorithm3* on a 4,000,000 character text (the french version of the Bible). Figure1 shows the execution time while searching 100 random patterns from the Bible. The first column of the table shows the lengths of the considered patterns.

We can see that the longest the pattern, the fastest *Algorithm3* works. Moreover, for patterns of lengths up to 3, *Algorithm3* is faster than *Shift-Or*.

m	*Shift-Or*	*Algorithm3*
3	**76.97**	79.32
4	77.95	**71.85**
6	79.27	**63.07**
8	78.15	**56.98**
10	78.72	**50.30**
12	78.85	**44.48**
14	77.52	**39.32**
16	78.35	**37.02**
20	77.43	**31.42**
30	77.84	**25.44**

Figure 1: Experimental results (in seconds) for exact searching 100 random patterns in the Bible. First column gives the lengths of the considered patterns.

5.2 Approximate string matching

Figure2 shows experimental results for *Algorithm2* and *Shift-Add*, while searching 100 random patterns in the Bible (5MO) with at most 1 or 2 mismatches. When m is large enough compared to k, *Algorithm2* is faster than *Shift-Add*: for $k = 1$, m should be larger than 7 and for $k = 2$ larger than 9. Since mb should not exceed the word size ω, large values of m cannot be considered. For $\omega = 32$, the efficiency of *Algorithm2* is then limited to $k \leq 2$.

For longer patterns, we need to use more than a word per number. It is not difficult to extend the algorithm for this case. Baeza-Yates and Gonnet have noticed that *Shift-Add* is still a good practical algorithm for string matching with mismatches and classes, provided the number of words per number is small.

Figure3 shows results in the case of two bit words per number. Notice that they extend the results in Figure2.

6 Conclusion

We have developed an algorithm combining both the programming practicality of the *Shift-Add* method and the speed of the BM approach. Flexibility is another advantage of this algorithm . In fact, it can be easily adapted to classes of patterns.

Nevertheless, as for the BM algorithm, the larger the alphabet and the longer the pattern, the faster our algorithm works. For a large alphabet (ASCII code), the searching step does on average $n(2 + \frac{k+4}{m-k})$ operations.

In some cases, it is necessary to consider small alphabets. In particular, in molecular biology when detecting potential gene-coding sequences in genomic DNA sequences. The considered alphabet consists in the four nucleotides

	$k=1$		$k=2$	
m	*Shift-Add*	*Algorithm2*	*Shift-Add*	*Algorithm2*
6	**157.56**	178.05	**157.24**	215.77
7	**155.83**	161.88	**156.25**	195.68
8	156.01	**154.05**	**155.64**	175.67
9	155.57	**148.13**	**156.33**	161.02
10	155.10	**145.85**	155.63	**150.72**
11	155.76	**141.67**		
12	154.66	**135.80**		
13	155.02	**129.40**		
14	155.45	**125.28**		
15	154.94	**121.08**		
16	155.73	**117.21**		

Figure 2: Experimental results (in seconds) for searching 100 random patterns in the Bible with at most 1 or 2 mismatches. For $k = 2$ and $m > 10$, more than one word per number is needed.

	$k=1$		$k=2$		$k=3$	
m	*Shift-Add*	*Algorithme2*	*Shift-Add*	*Algorithme2*	*Shift-Add*	*Algorithme2*
12			281.56	**275.96**	**282.38**	308.53
14			281.27	**266.48**	281.75	**278.75**
16			281.17	**250.82**	281.25	**265.46**
18	280.54	**219.24**	280.78	**238.18**	281.12	**256.98**
20	282.25	**211.78**	281.85	**227.57**	281.65	**246.60**
22	280.48	**205.70**				
26	281.38	**198.13**				
28	280.85	**194.20**				
32	280.93	**188.52**				

Figure 3: Complementary results when $mb > 32$. Two bit words per number are used.

$\{A, C, G, T\}$. For such alphabets, our algorithm does $n(2+\epsilon)$ operations, with $\epsilon < 1$, provided that the length m of the pattern is very large compared to k.

In order to consider large patterns, one solution is to use more than a bit word per number. Moreover, Baeza-Yates and Perleberg (BYP) [7] have developed an algorithm for approximate string matching, based on the same naive method than for the *Shift-Add* algorithm, but using arrays instead of numbers. In this case, there is no condition on the length of the searched pattern, however the algorithm is slower. The main difference is that BYP considers the number of matches instead of the number of mismatches. The BYP algorithm can be adapted from BM in the same way the *Shift-Add* was and it is then possible to consider long patterns.

References

1. K. Abrahamson. Generalized string matching. *SIAM J. Comput.*, 16(6):1039–1051, December 1987.
2. A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18:333–340, 1975.
3. T. Akutsu. Approximate string matching with don't care characters. In M. Crochemore and D. Gusfield, editors, *Lecture Notes in Computer Science*, volume 807 of *Combinatorial Pattern Matching (*5^{th} *Annual Symposium, CPM94)*, pages 229–242. Springer-Verlag, 1994.
4. R. Baeza-Yates and G.H.Gonnet. Fast string matching with k mismatches. Technical Report CS-88-36, Data Structuring Group, September 1988.
5. R. Baeza-Yates and G. Gonnet. Efficient text searching of regular expressions. 16th International colloquium on Automata, Languages and Programming. Stresa, Italy, July 1989.
6. R. Baeza-Yates and G. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, October 1992.
7. R. Baeza-Yates and C. Perleberg. Fast and practical approximate string matching. In *Lecture Notes in Computer Science*, volume 644 of *Combinatorial Pattern Matching (*3^{th} *Annual Symposium, CPM92)*, pages 185–191. Springer-Verlag, 1992.
8. A. Bertossiand and F. Logi. Parallel string matching with variable length don't cares. *Journal of parallel and distributed computing*, 22:229–234, 1994.
9. R. Boyer and J. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, October 1977.
10. M. Fischer and M. Paterson. String-matching and other products. In R. Karp, editor, *Complexity of Computation (SIAM-AMS Proceedings 7)*, volume 7, pages 113–125. American Mathematical Society, Providence, R.I., 1974.
11. Z. Galil and R. Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17:52–54, 1986.
12. R. Grossi and F. Luccio. Simple and efficient string matching with k mismatches. *Inf. Proc. Letters*, 3(33):113–120, November 1989.
13. D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6:323–350, June 1977.
14. G. Kucherov and M. Rusinowitch. Matching a set of strings with variable length don't cares. In Z. Galil and E. Ukkonen, editors, *Lecture Notes in Computer Science*, volume 937 of *6th annual symposium,CPM95*, pages 230–247. Espoo,Finland, Springer-Verlag, July 1995.
15. G. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoret. Comput. Sci.*, (43):239–249, 1986.
16. U. Manber and R. Baeza-Yates. An algorithm for string matching with a sequence of don't cares. *Information Proceeding Letters*, 37:133–136, 1991.
17. R. Pinter. Efficient string matching whith don't-care patterns. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume F12, pages 11–29. Springer-Verlag, 1985.
18. J. Tarhio and E. Ukkonen. Boyer-moore approach to approximate string matching. In J. R. Gilbert and R. G. Karlsson, editors, *Lecture Notes in Computer Science*, volume 447 of *2nd Scandinavian Workshop in Algorithmic Theory, SWAT'90*, pages 348–359. Bergen, Norway, Springer-Verlag, July 1990.
19. S. Wu and U. Manber. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, October 1992.