A Bit-Parallel Approach to Suffix Automata: Fast Extended String Matching

Gonzalo Navarro[1][3] Mathieu Raffinot[2]

[1] Dept. of Computer Science, University of Chile. Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl.

[2] Institut Gaspard Monge, Cité Descartes, Champs-sur-Marne, 77454 Marne-la-Vallée Cedex 2, France. raffinot@monge.univ-mlv.fr.

[3] Partially supported by Chilean Fondecyt grant 1-950622.

Abstract. We present a new algorithm for string matching. The algorithm, called BNDM, is the bit-parallel simulation of a known (but recent) algorithm called BDM. BDM skips characters using a "suffix automaton" which is made deterministic in the preprocessing. BNDM, instead, simulates the nondeterministic version using bit-parallelism. This algorithm is 20%-25% faster than BDM, 2-3 times faster than other bit-parallel algorithms, and 10%-40% faster than all the Boyer-Moore family. This makes it *the fastest algorithm in all cases* except for very short or very long patterns (e.g. on English text it is the fastest between 5 and 110 characters). Moreover, the algorithm is very simple, allowing to easily implement other variants of BDM which are extremely complex in their original formulation. We show that, as other bit-parallel algorithms, BNDM can be extended to handle classes of characters in the pattern and in the text, multiple patterns and to allow errors in the pattern or in the text, combining simplicity, efficiency and flexibility. We also generalize the suffix automaton definition to handle classes of characters. To the best of our knowledge, this extension has not been studied before.

1 Introduction

The string-matching problem is to find all the occurrences of a given pattern $p = p_1 p_2 \ldots p_m$ in a large text $T = t_1 t_2 \ldots t_n$, both sequences of characters from a finite character set Σ.

Several algorithms exist to solve this problem. One of the most famous, and the first having linear worst-case behavior, is Knuth-Morris-Pratt (KMP) [14]. A second algorithm, as famous as KMP, which allows to skip characters, is Boyer-Moore (BM) [6]. This algorithm leads to several variations, like Hoorspool [12] and Sunday [20], forming the fastest known string-matching algorithms.

A large part of the research in efficient algorithms for string matching can be regarded as looking for automata which are efficient in some sense. For instance, KMP is simply a deterministic automaton that searches the pattern, being its main merit that it is $O(m)$ in space and construction time. Many variations of the BM family are supported by an automaton as well.

Another automaton, called "suffix automaton" is used in [9, 10, 11, 15, 19], where the idea is to search a substring of the pattern instead of a prefix (as KMP), or a suffix (as BM). Optimal sublinear algorithms on average, like "Backward DAWG Match" (BDM) or Turbo_BDM [10, 11], have been obtained with this approach, which has also been extended to multipattern matching [9, 11, 19] (i.e. looking for the occurrences of a set of patterns).

Another related line of research is to take those automata in their nondeterministic form instead of making them deterministic. Usually the nondeterministic versions are very simple and regular and can be simulated using "bit-parallelism" [1]. This technique uses the intrinsic parallelism of the bit manipulations inside computer words to perform many operations in parallel. Competitive algorithms have been obtained for exact string matching [2, 22], as well as approximate string matching [22, 23, 3]. Although these algorithms work well only on relatively short patterns, they are simpler, more flexible, and have very low memory requirements.

In this paper we merge some aspects of the two approaches in order to obtain a fast string matching algorithm, called Backward Nondeterministic Dawg Matching (BNDM), which we extend to handle classes of characters, to search multiple patterns, and to allow errors in the pattern and/or in the text, like Shift-Or [2]. BNDM uses a nondeterministic suffix automaton that is simulated using bit-parallelism. This new algorithm has the advantage of being faster than previous ones which could be extended in such a way (typically 2-3 times faster than Shift-Or), faster than its deterministic-automaton counterpart BDM (20%-25% faster), using little space in comparison with the BDM or Turbo_BDM algorithms, and being very simple to implement. It becomes the **fastest** string matching algorithm, beating all the Boyer-Moore family (Sunday included) by 10% to 40%. Only for very short (up to 2-6 letters) or very long patterns (past 90-150 letters), depending on $|\Sigma|$ and the architecture, other algorithms become faster than BNDM (Sunday and BDM, respectively). Moreover, we define a new suffix automaton which handles classes of characters and we simulate its nondeterministic version using bit-parallelism. This extension has not been considered for the BDM or Turbo_BDM algorithms before.

We introduce some notation now. A word $x \in \Sigma^*$ is a *factor* (i.e. substring) of p if p can be written $p = uxv$, $u, v \in \Sigma^*$. We denote Fact(p) the set of factors of p. A factor x of p is called a *suffix* of p is $p = ux$. The set of suffixes of p is called Suff(p).

We denote as $b_\ell ... b_1$ the bits of a mask of length ℓ. We use exponentiation to denote bit repetition (e.g. $0^3 1 = 0001$). We use C-like syntax for operations on the bits of computer words: "|" is the bitwise-or, "&" is the bitwise-and, "^" is the bitwise-xor and "~" complements all the bits. The shift-left operation, "<<", moves the bits to the left and enters zeros from the right, i.e. $b_m b_{m-1} ... b_2 b_1 << r = b_{m-r} ... b_2 b_1 0^r$. We can interpret bit masks as integers also to perform arithmetic operations on them.

An expanded version of this work can be found in [17].

2 Searching with Suffix Automata

We describe in this section the BDM pattern matching algorithm [10, 11]. This algorithm is based on a suffix automaton. We first describe such automaton and then explain how is it used in the search algorithm

2.1 Suffix Automata

A *suffix automaton* on a pattern $p = p_1 p_2 \ldots p_m$ (frequently called DAWG(p) - for Deterministic Acyclic Word Graph) is the minimal (incomplete) deterministic finite automaton that recognizes all the suffixes of this pattern. By "incomplete" we mean that some transitions are not present.

The nondeterministic version of this automaton has a very regular structure and is shown in Figure 1. We show now how the corresponding deterministic automaton is built.

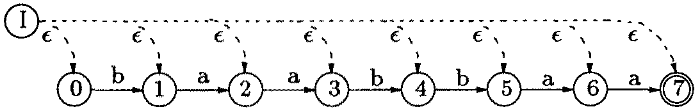

Fig. 1. A nondeterministic suffix automaton for the pattern $p = baabbaa$. Dashed lines represent epsilon transitions (i.e. they occur without consuming any input). I is the initial state of the automaton.

Given a factor x of the pattern p, *endpos(x)* is the set of all the pattern positions where an occurrence of x ends (there is at least one, since x is a factor of the pattern, and there are as many as repetitions of x inside p). Formally, given $x \in \text{Fact}(p)$, we define *endpos(x)* $= \{i \;/\; \exists u,\; p_1 p_2 ... p_i = ux\}$. We call each such integer a *position*. For example, $endpos(baa) = \{3, 7\}$ in the word *baabbaa*. Notice that $endpos(\epsilon)$ is the complete set of possible positions (recall that ϵ is the empty string). Notice that for any u, v, $endpos(u)$ and $endpos(v)$ are either disjoint or one contained in the other.

We define an equivalence relation $\equiv$ between factors of the pattern. For $u, v \in \text{Fact}(p)$, we define

$$u \equiv v \text{ if and only if } endpos(u) = endpos(v)$$

(notice that one of the factors must be a suffix of the other for this equivalence to hold, although the converse is not true). For instance, in our example pattern $p = baabbaa$, we have that $baa \equiv aa$ because in all the places where aa ends in the pattern, baa ends also (and vice-versa).

The nodes of the DAWG correspond to the equivalence classes of $\equiv$, i.e. to sets of positions. A state, therefore, can be thought of a factor of the pattern already recognized, except because we do not distinguish between some factors. Another way to see it is that the set of positions is in fact the set of active states in the nondeterministic automaton.

There is an edge labeled σ from the set of positions $\{i_1, i_2, \ldots i_k\}$ to $\gamma_p(i_1 + 1, \sigma) \cup \gamma_p(i_2 + 1, \sigma) \cup \ldots \cup \gamma_p(i_k, \sigma)$, where

$$\gamma_p(i, \sigma) = \begin{cases} \{i\} \text{ if } i \le m \text{ and } p_i = \sigma \\ \emptyset \text{ otherwise} \end{cases}$$

which is the same to say that we try to extend the factor that we recognized with the next text character σ, and keep the positions that still match. If we are left with no matching positions, we do not build the transition. The initial state corresponds to the set $\{0..m\}$. Finally, a state is *terminal* if its corresponding subset of positions contains the last position m (i.e. we matched a suffix of the pattern). As an example, the deterministic suffix automaton of the word ***baabbaa*** is given in Figure 2.

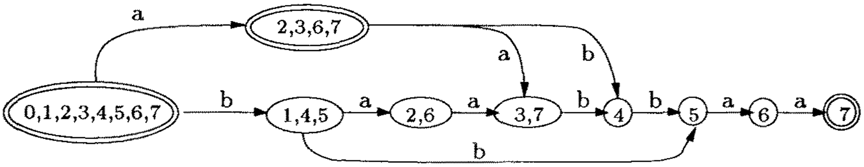

Fig. 2. Deterministic suffix automaton of the word *baabbaa*. The largest node is the initial state.

The (deterministic) suffix automaton is a well known structure [8, 5, 11, 18], and we do not prove any of its properties here (neither the correctness of the previous construction). The size of DAWG(p) is linear in m (counting both nodes and edges), and can be built in linear time [8]. A very important fact for our algorithm is that this automaton can not only be used to recognize the suffixes of p, but also factors of p. By the suffix automaton definition, there is a path labeled by x form the initial node of DAWG(p) if and only if x is a factor of p.

2.2 Search Algorithm

The suffix automaton structure is used in [10, 11] to design a simple pattern matching algorithm called BDM. This algorithm is $O(mn)$ time in the worst case, but optimal in average ($O(n \log m/m)$ time)[4]. Other more complex variations such as Turbo_BDM[10] and MultiBDM[11, 19] achieve linear time in the worst

[4] The lower bound of $O(n \log m/m)$ in average for any pattern matching algorithm under a Berbouilli model is from A. C. Yao in [24].

case. To search a pattern $p = p_1 p_2 \dots p_m$ in a text $T = t_1 t_2 \dots t_n$, the suffix automaton of $p^r = p_m p_{m-1} \dots p_1$ (i.e the pattern read backwards) is built. A window of length m is slid along the text, from left to right. The algorithm searches backwards inside the window for a factor of the pattern p using the suffix automaton. During this search, if a terminal state is reached which does not correspond to the entire pattern p, the window position is remembered (in a variable *last*). This corresponds to finding a *prefix* of the pattern starting at position *last* inside the window and ending at the end of the window (since the suffixes of p^r are the reverse prefixes of p). The last recognized prefix is the *longest* one. The backward search ends because of two possible reasons:

1. We fail to recognize a factor, i.e we reach a letter σ that does not correspond to a transition in DAWG(p^r). Figure 3 illustrates this case. We then shift the window to the right in *last* characters (we cannot miss an occurrence because in that case the suffix automaton would have found its prefix in the window).

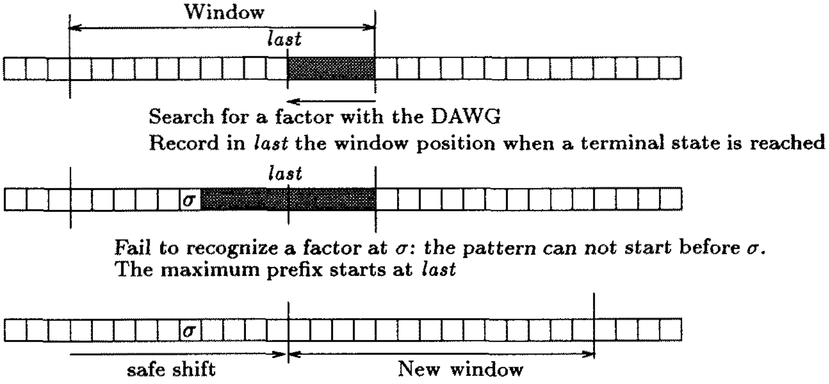
Window
last
Search for a factor with the DAWG
Record in last the window position when a terminal state is reached
last
σ
Fail to recognize a factor at σ: the pattern can not start before σ.
The maximum prefix starts at last
σ
safe shift
New window

Fig. 3. Basic search with the suffix automaton

2. We reach the beginning of the window, therefore recognizing the pattern p. We report the occurrence, and we shift the window exactly as in the previous case (notice that we have the previous *last* value).

Search example: we search the pattern *aabbaab* in the text

$$T = a\ b\ b\ a\ b\ a\ a\ b\ b\ a\ a\ b.$$

We first build DAWG(p^r =*baabbaa*), which is given in Figure 2. We note the current window between square brackets and the recognized prefix in a rectangle. We begin with

$T =\ [a\ b\ b\ a\ b\ a\ a\,]b\ b\ a\ a\ b$, $m = 7$, $last = 7$.

1. $T = [a\,b\,b\,a\,b\,a\,\boxed{a}]\,b\,b\,a\,a\,b.$
   a is a factor of p^r and a reverse prefix of p. *last* $= 6$.
2. $T = [a\,b\,b\,a\,b\,\boxed{a\,a}]\,b\,b\,a\,a\,b.$
   aa is a factor of p^r and a reverse prefix of p. *last* $= 5$.
3. $T = [a\,b\,b\,a\,\boxed{b\,a\,a}]\,b\,b\,a\,a\,b.$
   aab is a factor of p^r.
   We fail to recognize the next a. So we shift the window to *last*. We search again in the position: $T = a\,b\,b\,a\,b\,[a\,a\,b\,b\,a\,a\,b]$, *last* $= 7$.
4. $T = a\,b\,b\,a\,b\,[a\,a\,b\,b\,a\,a\,\boxed{b}].$
   b is a factor of p^r.
5. $T = a\,b\,b\,a\,b\,[a\,a\,b\,b\,a\,\boxed{a\,b}].$
   ba is a factor of p^r.
6. $T = a\,b\,b\,a\,b\,[a\,a\,b\,b\,\boxed{a\,a\,b}].$
   baa is a factor of p^r.
7. $T = a\,b\,b\,a\,b\,[a\,a\,b\,b\,\boxed{a\,a\,b}].$
   baa is a factor of p^r, and a reverse prefix of p. *last* $= 4$.
8. $T = a\,b\,b\,a\,b\,[a\,a\,b\,\boxed{b\,a\,a\,b}].$
   $baab$ is a factor of p^r.
9. $T = a\,b\,b\,a\,b\,[a\,a\,\boxed{b\,b\,a\,a\,b}].$
   $baabb$ is a factor of p^r.
10. $T = a\,b\,b\,a\,b\,[a\,\boxed{a\,b\,b\,a\,a\,b}].$
    $baabba$ is a factor of p^r.
11. $T = a\,b\,b\,a\,b\,[\boxed{a\,a\,b\,b\,a\,a\,b}].$
    We recognize the word *aabbaab* and report an occurrence.

3 Bit-Parallelism

In [2], a new approach to text searching was proposed. It is based on *bit-parallelism* [1], which consists in taking advantage of the intrinsic parallelism of the bit operations inside a computer word to cut down the number of operations by a factor of at most w, where w is the number of bits in the computer word.

The Shift-Or algorithm uses bit-parallelism to simulate the operation of a nondeterministic automaton that searches the pattern in the text (see Figure 4). As this automaton is simulated in time $O(mn)$, the Shift-Or algorithm achieves $O(mn/w)$ worst-case time (optimal speedup). If we convert the automaton to deterministic we get a version of KMP [14], which is $O(n)$ search time, although twice as slow in practice for $m \leq w$.

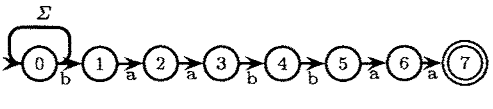

Fig. 4. A nondeterministic automaton to search the pattern p = *baabbaa* in a text. The initial state is 0.

We explain now a variant of the Shift-Or algorithm (called Shift-And). The algorithm builds first a table B which for each character stores a bit mask $b_m \ldots b_1$. The mask in $B[c]$ has the i-th bit set if and only if $p_i = c$. The state of the search is kept in a machine word $D = d_m \ldots d_1$, where d_i is set whenever the state numbered i in Figure 4 is active. Therefore, we report a match whenever d_m is set.

We set $D = 0$ originally, and for each new text character T_j, we update D using the formula

$$D' \leftarrow ((D << 1) \mid 0^{m-1}1) \ \& \ B[T_j]$$

which mimics what occurs inside the nondeterministic automaton for each new text character: each state gets the value of the previous one, provided the text character matches the corresponding arrow. The "$\mid 0^{m-1}1$" corresponds to the initial self-loop. For patterns longer than the computer word (i.e. $m > w$), the algorithm uses $\lceil m/w \rceil$ computer words for the simulation (not all them are active all the time).

This algorithm is very simple and can be extended to handle classes of characters (i.e. each pattern position matches a set of characters), and to allow mismatches. This paradigm was later enhanced to support wild cards, regular expressions, approximate search, etc. yielding the fastest algorithms for those problems [22, 3]. Bit-parallelism became a general way to simulate simple nondeterministic automata instead of converting them to deterministic. This is how we use it in our algorithm.

4 Bit-Parallelism on Suffix Automata

We simulate the BDM algorithm using bit-parallelism. The result is an algorithm which is simpler, uses less memory, has more locality of reference, and is easily extended to handle more complex patterns. We first assume that $m \leq w$ and show later how to extend the algorithm for longer patterns.

4.1 The Basic Algorithm

We simulate the reverse version of the automaton of Figure 1. Just as for Shift-And, we keep the state of the search using m bits of a computer word $D = d_m ... d_1$.

The BDM algorithm moves a window over the text. Each time the window is positioned at a new text position just after pos, it searches backwards the window $T_{pos+1}..T_{pos+m}$ using the DAWG automaton, until either m iterations are performed (which implies a match in the current window) or the automaton cannot perform any transition.

In our case, the bit d_i at iteration k is set if and only if $p_{m-i+1..m-i+k} = T_{pos+1+m-k}..T_{pos+m}$. Since we begin at iteration 0, the initial value for D is 1^m. There is a match if and only if after iteration m it holds $d_m = 1$. Whenever $d_m = 1$, we have matched a prefix of the pattern in the current window. The longest prefix matched corresponds to the next window position.

The algorithm is as follows. Each time we position the window in the text we initialize D and scan the window backwards. For each new text character we update D. Each time we find a prefix of the pattern ($d_m = 1$) we remember the position in the window. If we run out of 1's in D then there cannot be a match and we suspend the scanning (this corresponds to not having any transition to follow in the automaton). If we can perform m iterations then we report a match.

We use a mask B which for each character c stores a bit mask. This mask sets the bits corresponding to the positions where the pattern has the character c (just as in Shift-And). The formula to update D follows

$$D' \leftarrow (D \ \& \ B[T_j]) \ << \ 1$$

The algorithm is summarized in Figure 5. Some optimizations done on the real code, related to improved flow of control and bit manipulation tricks, are not shown for clarity.

```
BNDM ($p = p_1p_2...p_m$, $T = t_1t_2...t_n$)
    Preprocessing
        For $c \in \Sigma$ do $B[c] \leftarrow 0^m$
        For $i \in 1..m$ do $B[p_{m-i+1}] \leftarrow B[p_{m-i+1}] \mid 0^{m-i}10^{i-1}$
    Search
        $pos \leftarrow 0$
        While $pos <= n - m$ do
            $j \leftarrow m$, $last \leftarrow m$
            $D = 1^m$
            While $D \mathrel{!=} 0^m$ do
                $D \leftarrow D \ \& \ B[T_{pos+j}]$
                $j \leftarrow j - 1$
                if $D \ \& \ 10^{m-1} \mathrel{!=} 0^m$ then
                    if $j > 0$ then $last \leftarrow j$
                              else report an occurrence at $pos + 1$
                $D \leftarrow D << 1$
            End of while
            $pos \leftarrow pos + last$
        End of while
```

Fig. 5. Bit-parallel code for **BDM**. Some optimizations are not shown for clarity.

Search example: we search the pattern *aabbaab* in the text $T = a\ b\ b\ a\ b\ a\ a\ b\ b\ a\ a\ b$. Immediately after each step number (1 to 11) we show the text and note the current window between square brackets, as well as the recognized prefix in a rectangle. We begin with

$T = [\, a\ b\ b\ a\ b\ a\ a \,]\ b\ b\ a\ a\ b$, $D = 1\ 1\ 1\ 1\ 1\ 1\ 1$, $B = \begin{array}{|c|c|}\hline a & 1\,1\,0\,0\,1\,1\,0 \\ \hline b & 0\,0\,1\,1\,0\,0\,1 \\ \hline \end{array}$, $m = 7$, $last = 7$, $j = 7$.

1. [*abbaba* $\boxed{a}$] *bbaab*.

	1 1 1 1 1 1 1
&	1 1 0 0 1 1 0
$D =$	1 1 0 0 1 1 0

$j = 6$, $last = 6$

2. [*abbab* $\boxed{aa}$] *bbaab*.

	1 0 0 1 1 0 0
&	1 1 0 0 1 1 0
$D =$	1 0 0 0 1 0 0

$j = 5$, $last = 5$

3. [*abba* $\boxed{baa}$] *bbaab*.

	0 0 0 1 0 0 0
&	0 0 1 1 0 0 1
$D =$	0 0 0 1 0 0 0

$j = 4$, $last = 5$

4. [*abb* $\boxed{abaa}$] *bbaab*.

	0 0 1 0 0 0 0
&	1 1 0 0 1 1 0
$D =$	0 0 0 0 0 0 0

$j = 3$, $last = 5$

We fail to recognize the next *a*. So we shift the window to *last*. We search again in the position: *abbab* [*aabbaab*], *last* $= 7$, $j = 7$.

5. *abbab* [*aabbaa* $\boxed{b}$].

	1 1 1 1 1 1 1
&	0 0 1 1 0 0 1
$D =$	0 0 1 1 0 0 1

$j = 6$, $last = 7$

6. *abbab* [*aabba* $\boxed{ab}$].

	0 1 1 0 0 1 0
&	1 1 0 0 1 1 0
$D =$	0 1 0 0 0 1 0

$j = 5$, $last = 7$

7. *abbab* [*aabb* $\boxed{aab}$].

	1 0 0 0 1 0 0
&	1 1 0 0 1 1 0
$D =$	1 0 0 0 1 0 0

$j = 4$, $last = 4$

8. *abbab* [*aab* $\boxed{baab}$].

	0 0 0 1 0 0 0
&	0 0 1 1 0 0 1
$D =$	0 0 0 1 0 0 0

$j = 3$, $last = 4$

9. *abbab* [*aa* $\boxed{bbaa\ b}$].

	0 0 1 0 0 0 0
&	0 0 1 1 0 0 1
$D =$	0 0 1 0 0 0 0

$j = 2$, $last = 4$

10. *abbab* [*a* $\boxed{abbaab}$].

	0 1 0 0 0 0 0
&	1 1 0 0 1 1 0
$D =$	0 1 0 0 0 0 0

$j = 2$, $last = 4$

11. *abbab* [$\boxed{aabbaab}$].

	1 0 0 0 0 0 0
&	1 1 0 0 1 1 0
$D =$	1 0 0 0 0 0 0

$j = 0$, $last = 4$

Report an occurrence at 6.

4.2 Handling Longer Patterns

We can cope with longer patterns by setting up an array of words D_t and simulating the work on a long computer word. We propose a different alternative which was experimentally found to be faster.

If $m > w$, we partition the pattern in $M = \lceil m/w \rceil$ subpatterns s_i, such that $p = s_1\ s_2\ ...\ s_M$ and s_i is of length $m_i = w$ if $i < M$ and $m_M = m - w(M-1)$. Those subpatterns are searched with the basic algorithm.

We now search s_1 in the text with the basic algorithm. If s_1 is found at a text position j, we verify whether s_2 follows it. That is, we position a window at $T_{j+m_1}..T_{j+m_1+m_2-1}$ and use the basic algorithm for s_2 in that window. If s_2 is in the window, we continue similarly with s_3 and so on. This process ends either because we find the complete pattern and report it, or because we fail to find a subpattern s_i.

We have to move the window now. An easy alternative is to use the shift $last_1$ that corresponds to the search of s_1. However, if we tested the subpatterns s_1 to s_i, each one has a possible shift $last_i$, and we use the maximum of all shifts.

4.3 Analysis

The preprocessing time for our algorithm is $O(m+|\Sigma|)$ if $m \leq w$, and $O(m(1+|\Sigma|/w))$ otherwise.

In the simple case $m \leq w$, the analysis is the same as for the BDM algorithm. That is, $O(mn)$ in the worst case (e.g. $T = a^n$, $p = a^m$), $O(n/m)$ in the best case (e.g. $T = a^n$, $p = a^{m-1}b$), and $O(n \log_{|\Sigma|} m/m)$ on average (which is optimal). Our algorithm, however, benefits from more locality of reference, since we do not access an automaton but only a few variables which can be put in registers (with the exception of the B table). As we show in the experiments, this difference makes our algorithm the fastest one.

When $m > w$, our algorithm is $O(nm^2/w)$ in the worst case (since each of the $O(mn)$ steps of the BDM algorithm forces to work on $\lceil m/w \rceil$ computer words). The best case occurs when the text traversal using s_1 always performs its maximum shift after looking one character, which is $O(n/w)$. We show, finally, that the average case is $O(n \log_{|\Sigma|} w/w)$. Clearly these complexities are worse than those of the simple BDM algorithm for long enough patterns. We show in the experiments up to which length our version is faster in practice.

The search cost for s_1 is $O(n \log_{|\Sigma|} w/w)$. With probability $1/|\Sigma|^w$, we find s_1 and check for the rest of the pattern. The check for s_2 in the window costs $O(w)$ at most. With probability $1/|\Sigma|^w$ we find s_2 and check s_3, and so on. The total cost incurred by the existence of $s_2 ... s_M$ is at most

$$\sum_{i=1}^{M-1} \frac{w}{|\Sigma|^{wi}} \leq \varepsilon = \frac{w}{|\Sigma|^w}\left(1+O(w/|\Sigma|^w)\right) = O(1)$$

which therefore does not affect the main cost to search s_1 (neither in theory since the extra cost is $O(1)$ nor in practice since ε is very small). We consider the shifts now. The search of each subpattern s_i provides a shift $last_i$, and we take the maximum shift. Now, the shift $last_i$ participates in this maximum with probability $1/|\Sigma|^{wi}$. The longest possible shift is w. Hence, if we *sum* (instead of taking the maximum) the *longest possible* shifts w with their probabilities, we get into the same sum above, which is $\varepsilon = O(1)$. Therefore, the average shift is longer than $last_1$ and shorter than $last_1 + \varepsilon = last_1 + O(1)$, and hence the cost is that of searching s_1 plus lower order terms.

5 Further Improvements

5.1 A Linear Algorithm

Although our algorithm has optimal average case, it is not linear in the worst case even for $m \leq w$, since we can traverse the complete window backwards and advance it in one character. Our aim now is to reduce its worst case from $O(nm^2/w)$ to $O(nm/w)$, i.e. $O(n)$ when $m = O(w)$.

Improved variations on BDM already exist, such as Turbo_BDM and Turbo_RF [10, 15], the last one being linear in the worst case and still sublinear on average. The main idea is to avoid retraversing the same characters in the backward window verification using the fact that when we advance the window in *last* positions, we already know that $T_{i+last}..T_{i+m-1}$ is a prefix of the pattern (recall Figure 3). The ending position of the prefix in the window is usually called the *critical position*. The main problem if this area is not retraversed is how to determine the next shift, since among all possible shifts in $T_{i+last}..T_{i+m-1}$ we remember only the first one.

One strategy adds a kind of BM machine to the BDM algorithm. It works as follows: let u be the pattern prefix before the critical position. If we reach the critical position after reading (backwards) a factor z with the DAWG, it is possible to know whether z^r is a suffix of the pattern p: if z^r is a suffix, (i.e. $p = uz^r$) we recognize the whole pattern p, and the next shift corresponds to the longest *border* of p (i.e. the longest proper prefix that is also a suffix), which can be computed in advance. If z^r is not a suffix, it appears in the pattern in a set of positions which is given by the state we reached in the suffix automaton. We shift to the rightmost occurrence of z^r in the pattern.

It is not difficult to simulate this idea in our BNDM algorithm. To know if the factor z we read with the DAWG is a suffix, we test whether $d_{|z|} = 1$. To get the rightmost occurrence, we seek the rightmost 1 in D, which we can get (if it exists) in constant time with $\log_2(D \,\&\sim (D-1))$ [5]. We implemented this algorithm under the name BM_BNDM in the experimental part of this paper.

This algorithm remains quadratic, because we do not keep a prefix of the pattern after the BM shift. We do that now. Recall that u is the prefix before the critical position. The Turbo_RF (second variation) [10] uses a complicated preprocessing phase to associate in linear time an occurrence of z^r in the pattern to a border b_u of u, in order to obtain the maximal prefix of the pattern that is a suffix of uz^r. Moreover, the Turbo_RF uses a suffix tree, and it is quite difficult to use this preprocessing phase on DAWGs. With our simulation, this preprocessing phase becomes simple. To each prefix u_i of the pattern p, we associate a mask *Bord*[i] that registers the starting positions of the borders of u_i (ϵ included). This table is precomputed in linear time. Now, to join one occurrence of z^r to a border of u, we want the positions which start a border of u *and* continue with an occurrence of z^r. The first set of positions is *Bord*[i], and the second one is precisely the current D value (i.e. positions in the pattern where the recognized factor z ends). Hence, the bits of $X = Bord[i] \,\&\, D$ are the positions satisfying both criteria. As we want the rightmost such occurrence (i.e. the maximal prefix), we take again $\log_2(X \& \sim (X-1))$. We implemented this algorithm under the name Turbo_BNDM in the experimental part of this paper.

5.2 A Constant-Space Algorithm

It is also interesting to notice that, although the algorithm needs $O(|\Sigma| m/w)$ extra space, we can make it constant space on a binary alphabet $\Sigma_2 = \{0, 1\}$.

[5] It is faster and cleaner to implement this $\log_2$ by shifting the mask to the right until it becomes zero. Using this technique we can use the simpler expression D ^ $(D-1)$ and get the same result.

The trick is that in this case, $B[1] = p$ and $B[0] = \sim B[1]$. Therefore, we need no extra storage apart from the pattern itself to perform all the operations. In theory, any text over a finite alphabet Σ could be searched in constant space by representing the symbols of Σ with bits and working on the bits (the misaligned matches have to be later discarded). This involves an average search time of

$$\frac{n \log_2 |\Sigma|}{m \log_2 |\Sigma|} \log_2(m \log_2 |\Sigma|) \;=\; \text{Normal time} \times \log_2 |\Sigma| \left(1 + \frac{\log_2 \log_2 |\Sigma|}{\log_2 m}\right)$$

which if the alphabet is considered of constant size is of the same order of the normal search time.

6 Extensions

We analyze now some extensions applicable to our basic scheme, which form a successful combination of efficiency and flexibility.

6.1 Classes of Characters

As in the Shift-Or algorithm, we allow that each position in the pattern matches not only a single character but an arbitrary set of characters. We call "extended patterns" those that are more complex than a simple string to be searched. In this work the only extended patterns we deal with are those allowing a class of characters at each position.

We denote $p = C_1 C_2 \ldots C_m$ such extended patterns. A word $x = x_1 x_2 \ldots x_r$ in Σ^* is a factor of an extended pattern $p = C_1 C_2 \ldots C_m$ if there exists an i such that $x_1 \in C_{i-r+1}, x_2 \in C_{i-r+2}, \ldots, x_r \in C_i$. Such an i is called a *position* of x in p. A factor $x = x_1 x_2 \ldots x_r$ of $p = C_1 C_2 \ldots C_m$ is a *suffix* if $x_1 \in C_{m-r+1}, x_2 \in C_{i-r+2}, \ldots, x_r \in C_m$.

Similarly to the first part of this work, we design an automaton which recognizes all suffixes of an extended pattern $p = C_1 C_2 \ldots C_m$. This automaton is not anymore a DAWG. We call it Extended_DAWG. To our knowledge, this kind of automaton has never been studied. We first give a formal construction of the Extended_DAWG (proving its correctness) and later present a bit-parallel implementation.

Construction The construction we use is quite similar to the one we give for the DAWG, except for the new definition of suffixes. For any x factor of p, we denote *L-endpos(x)* the set of positions of x in p. For example, *L-endpos*$(baa) = \{3, 7\}$ in the extended pattern *b[a,b]abbaa*, and *L-endpos*$(bba) = \{3, 6\}$ (notice that, unlike before, the sets of positions can be not disjoint and no one a subset of the other). We define the equivalence relation $\equiv_E$ for u, v factors of p by

$$u \equiv_E v \text{ if and only if } L\text{-}endpos(u) = L\text{-}endpos(v).$$

We define $\gamma_p(i,\sigma)$ with $i \in \{0,1,\ldots,m,m+1\}, \sigma \in \Sigma$ by

$$\gamma_p(i,\sigma) = \begin{cases} \{i\} \text{ if } i \leq m \text{ and } \sigma \in C_i \\ \emptyset \text{ otherwise} \end{cases}$$

LEMMA 1 *Let p be an extended pattern and $\equiv_E$ the equivalence relation on its factors (as previously defined). The equivalence relation $\equiv_E$ is compatible with the concatenation on words.*

This lemma allows us to define an automaton from this equivalence class. States of the automaton are the equivalence classes of $\equiv_E$. There is an edge labeled by σ from the set of positions $\{i_1, i_2, \ldots i_k\}$ to $\gamma_p(i_1+1,\sigma) \cup \gamma_p(i_2+1,\sigma) \cup \ldots \cup \gamma_p(i_k+1,\sigma)$, if it is not empty. The initial node of the automaton is the set that contains all the positions. Terminals nodes of the automaton are the set of positions that contain m. As an example, the suffix automaton of the word *[a,b]aa[a,b]baa* is given in Figure 6.

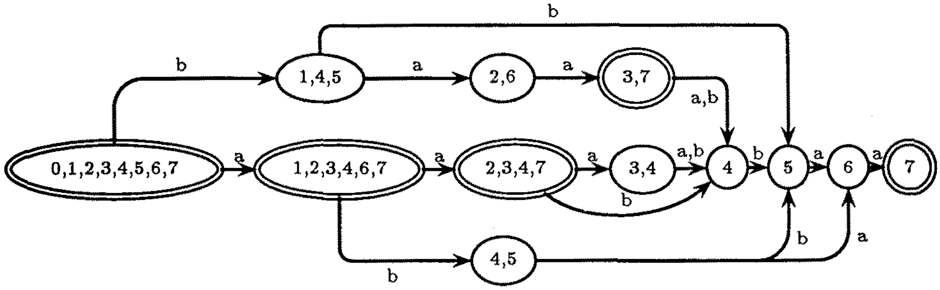

Fig. 6. Extended_DAWG of the extended pattern $^0[a,b]^1a^2a^3[a,b]^4b^5a^6a^7$

LEMMA 2 *The Extended_DAWG of an extended pattern $p = C_1C_2\ldots C_m$ recognizes the set of suffixes of p.*

We can use this new automaton to recognize the set of suffixes of an extended pattern p. We do not give an algorithm to build this Extended_DAWG in its deterministic form, but we simulate the deterministic automaton using bit-parallelism.

A bit-parallel implementation: from the above construction, the only modification that our algorithm needs is that the B table has the i-th bit set for all characters belonging to the set of the i-th position of the pattern. Therefore we simply change line 3 (part of the preprocessing) in the algorithm of Figure 5 to

For $i \in 1..m, c \in \Sigma$ **do if** $c \in p_i$ **then** $B[c] \leftarrow B[c] \mid 0^{m-i}10^{i-1}$

such that now the preprocessing takes $O(|\Sigma|m)$ time but the search algorithm does not change.

We combine the flexibility of extended patterns with the efficiency of a Boyer-Moore-like algorithm. It should be clear, however, that the efficiency of the shifts can be degraded if the classes of characters are significantly large and prevent long shifts. However, this is much more resistant than some simple variations of Boyer-Moore since it uses more knowledge about the matched characters.

We point out now another extension related to classes of characters: the text itself may have basic characters as well as other symbols denoting sets of basic characters. This is common, for instance, in DNA databases. We can easily handle such texts. Assume that the symbol C represents the set $\{c_1, ..., c_r\}$. Then we set $B[C] = B[c_1] \mid ... \mid B[c_r]$. This is much more difficult to achieve with algorithms not based on bit-parallelism.

6.2 Multiple Patterns

To search a set of patterns $P1...Pr$ (i.e. reporting the occurrences of all them) of length m in parallel, we can use an arrangement proposed in [22], which concatenates the patterns as follows: $P = P1_1\ P2_1\ ...Pr_1\ P1_2\ P2_2\ ...Pr_2\ .....\ P1_m\ P2_m\ ...Pr_m$ (i.e. all the first letters, then all the second letters, etc.) and searches P just as a single pattern. The only difference in the algorithm of Figure 5 is that the shift is not in one bit but in r bits in line 15 (since we have r bits per multipattern position) and that instead of looking for the highest bit d_m of the computer word we consider all the r bits corresponding to the highest position. That is, we replace the old 10^{m-1} test mask by $1^r 0^{r(m-1)}$ in line 12.

This will automatically search for r words of length m and keep all the bits needed for each word. Moreover, it will report the matches of any of the patterns and will not allow shifting more than what all patterns allow to shift.

An alternative arrangement is: $P = P1\ P2\ ...\ Pr$ (i.e. just concatenate the patterns). In this case the shift in line 15 is for one bit, and the mask for line 12 is $(10^{m-1})^r$. On some processors a shift in one position is faster than a shift in $r > 1$ positions, which could be an advantage for this arrangement. On the other hand, in this case we must clear the bits that are carried from the highest position of a pattern to the next one, replacing line 15 for $D = (D << 1)\ \&\ (1^{m-1}0)^r$. This involves an extra operation. Finally, this arrangement allows to have patterns of different lengths for the algorithm of Wu and Manber [22] which is not possible in their current proposal.

Clearly these techniques cannot be applied to the case $m > \lfloor w/2 \rfloor$. However, if $m \leq \lfloor w/2 \rfloor$ and $r \times m > w$ we divide the set of patterns into $\lceil r/\lfloor w/m \rfloor \rceil$ groups, so that the patterns in each group fit in w bits. Since this skips characters, it is better on average than [22]. As we show in the experiments, this is also better than sequentially searching each pattern in turn, even given that the shifts are the most conservative among all the r patterns.

6.3 Approximate String Matching

Approximate string matching is the problem of finding all the occurrences of a pattern in a text allowing at most k "errors". The errors are insertions, deletions and replacements to perform in the pattern so that it matches the text. In [22], an efficient filter is proposed to determine that large text areas cannot contain an occurrence. It is based on dividing the pattern in $k + 1$ pieces and searching all the pieces in parallel. Since k errors cannot destroy the $k+_1$ pieces, some of the pieces must appear with no errors close to each occurrence. They use the multipattern search algorithm mentioned in the previous paragraph. In [4, 3], a multipattern Boyer-Moore strategy is preferred, which is faster but does not handle classes of characters and other extensions. This algorithm is the fastest one for low error levels.

Our multipattern search technique presented in the previous section combines the best of both worlds: our performance is comparable to Boyer-Moore algorithms and we keep the flexibility of bit-parallelism handle classes of characters. We show in the experiments how our algorithm performs in this setup.

7 Experimental Results

We ran extensive experiments on random and natural language text to show how efficient are our algorithms in practice. The experiments were run on a Sun UltraSparc-1 of 167 MHz, with 64 Mb of RAM, running SunOS 5.5.1. We measure CPU times, which are within ±2% with 95% confidence. We used random texts and patterns with $\sigma = 2$ to 64, as well as natural language text and DNA sequences.

We show in Figure 7 some of the results for short ($m \leq w$) and long ($m > w$) patterns. The comparison includes the best known algorithms: BM, BM-Sunday, KMP (very slow to appear in the plots, close to 0.14 sec/Mb), Shift-Or (not always shown, close to 0.07 sec/Mb), classical BDM, and our three bit-parallel variants: BNDM, BM_BNDM and Turbo_BNDM.

Our bit-parallel algorithms are always the fastest for short patterns, except for $m \leq$ 2-6. The fastest algorithm is BM_BNDM, though it is very close to simple BNDM. Classical BDM, on the other hand, is sometimes slower than the BM family. Turbo_BNDM is competitive with simple BNDM and has linear worst case. Our algorithms are especially good for small alphabets since they use more information on the matched pattern than others. The only good competitor for small alphabets is Boyer-Moore, which however is slower because the code is more complex (notice that Boyer-Moore is faster than BDM, but slower than BNDM). For larger alphabets, on the other hand, another very simple algorithm gets very close: BM-Sunday. However, we are always at least 10% faster.

On longer patterns[6] our algorithm ceases to improve because it basically

[6] We did not include the more complex variations of our algorithm because they have already been shown very similar to the simple one. We did not include also the algorithms which are known not to improve, such as Shift-Or and KMP.

searches for the first w letters of the pattern, while classical BDM keeps improving. Hence, our algorithm ceases to be the best one (beaten by BDM) for $m \geq$ 90-150. This value would at least duplicate in a 64-bit architecture.

We show also some illustrative results using classes of characters, which were generated manually as follows: we select from an English text an infrequent word, namely "`responsible`" (close to 10 matches per megabyte). Then we replace its first or last characters by the class $\{a..z\}$. This will adversely affect the shifts of the BNDM algorithm. We compare the efficiency against Shift-Or. The result is presented in Table 1, which shows that even in the case of three initial or final letters allowing a large class of characters the shifts are significant and we double the performance of Shift-Or. Hence, our goals of handling classes of characters with improved search times are achieved.

Pattern	Shift-Or	BNDM
`responsible`	6.58	2.71
`responsibl?`	6.51	2.96
`responsib??`	6.52	3.23
`responsi???`	6.49	3.40
`?esponsible`	6.46	2.93
`??sponsible`	6.55	3.42
`???ponsible`	6.51	3.78

Table 1. Search times with classes of characters, in 1/100-th of seconds per megabyte on English text. The question mark '`?`' represents the class $\{a..z\}$.

We present in Figure 8 some results on our multipattern algorithm, to show that although we take the minimum shift among all the patterns, we can still do better than searching each pattern in turn. We take random groups of five patterns of length 6 and compare our multipattern algorithm (in its two versions, called Multi-BNDM (1) and (2) attending to their presentation order), against five sequential searches with BNDM (called BNDM in the legend), and against the parallel version proposed in [22] (called Multi-WM). As it can be seen, our first arrangement is slightly more efficient than the second one, they are always more efficient than a sequential search (although the improvement is not five-fold but two- or three-fold because of shorter shifts), and are more efficient than the proposal of [22] provided $\sigma \geq 8$.

Finally, we show the performance of our multipattern algorithm when used for approximate string matching. We include the fastest known algorithms in the comparison [4, 3, 7, 13, 16, 23, 22]. We compare those algorithms against our version of [4] (where the Sunday algorithm is replaced by our BNDM), while we consider [22] not as the bit-parallel algorithm presented there but their other proposal, namely reduction to exact searching using their algorithm Multi-WM for multipattern search (shown in the previous experiment). Figure 9 shows the results for different alphabet sizes and $m = 20$.

Since BNDM is not very good for very short patterns, the approximate search algorithm ceases to be competitive short before the original version [4]. This is because the length of the patterns to search for is $O(m/k)$. Despite this drawback, our algorithm is quite close to [4] (sometimes even faster) which makes it a reasonably competitive yet more flexible alternative, while being faster than the other flexible candidate [22].

8 Conclusions

We present a new algorithm (called BNDM) based on the bit-parallel simulation of a nondeterministic suffix automaton. This automaton has been previously used in deterministic form in an algorithm called BDM. Our new algorithm is experimentally shown to be very fast on average. It is the fastest algorithm in all cases for patterns from length 5 to 110 (on English; the bounds vary depending on the alphabet size and the architecture). We present also some variations called Turbo_BNDM and BM_BNDM which are derived from the corresponding variants of BDM. These variants are much more simply implemented using bit-parallelism and become practical algorithms. Turbo_BNDM has average performance very close to BNDM, though $O(n)$ worst case behavior, while BM_BNDM is slightly faster than BNDM. The BNDM algorithm can be extended simply and efficiently to handle classes of characters, multiple pattern matching and approximate pattern matching, among others.

The new suffix automaton we introduce and simulate for classes of characters has never been studied. Its study should permit to extend the BDM and Turbo_RF to handle classes of characters.

The Agrep software [21] is in many cases faster than BNDM. However, Agrep is just a BM algorithm which uses pairs of characters instead of single ones. This is an orthogonal technique that can be incorporated in all algorithms, and a general study of this technique would permit to improve the speed of pattern matching softwares. We plan to work on this idea too.

References

1. R. Baeza-Yates. Text retrieval: Theory and practice. In *12th IFIP World Computer Congress*, volume I, pages 465–476. Elsevier Science, September 1992.
2. R. Baeza-Yates and G. Gonnet. A new approach to text searching. *CACM*, 35(10):74–82, October 1992.
3. R. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In *Proc. of CPM'96*, pages 1–23, 1996.
4. R. Baeza-Yates and C. Perleberg. Fast and practical approximate pattern matching. In *Proc. CPM'92*, pages 185–192. Springer-Verlag, 1992. LNCS 644.
5. A. Blumer, A. Ehrenfeucht, and D. Haussler. Average sizes of suffix trees and dawgs. *Discrete Applied Mathematics*, 24(1):37–45, 1989.
6. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

7. W. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proc. of CPM'92*, pages 172–181, 1992. LNCS 644.
8. M. Crochemore. Transducers and repetitions. *Theor. Comput. Sci.*, 45(1):63–86, 1986.
9. M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Fast practical multi-pattern matching. Rapport 93-3, Institut Gaspard Monge, Université de Marne la Vallée, 1993.
10. M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithms. *Algorithmica*, (12):247–267, 1994.
11. M. Crochemore and W. Rytter. *Text algorithms.* Oxford University Press, 1994.
12. R. N. Horspool. Practical fast searching in strings. *Softw. Pract. Exp.*, 10:501–506, 1980.
13. P. Jokinen, J. Tarhio, and E. Ukkonen. A comparison of approximate string matching algorithms. *Software Practice and Experience*, 26(12):1439–1458, 1996.
14. D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):323–350, 1977.
15. T. Lecroq. *Recherches de mot.* Thèse de doctorat, Université d'Orléans, France, 1992.
16. G. Navarro. A partial deterministic automaton for approximate string matching. In *Proc. of WSP'97*, pages 112–124. Carleton University Press, 1997.
17. G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. Technical Report TR/DCC-98-1, Dept. of Computer Science, Univ. of Chile, Jan 1998. `ftp://ftp.dcc.uchile.cl/pub/users/-gnavarro/bndm.ps.gz`.
18. M. Raffinot. Asymptotic estimation of the average number of terminal states in dawgs. In R. Baeza-Yates, editor, *Proc. of WSP'97*, pages 140–148, Valparaiso, Chile, November 12-13, 1997. Carleton University Press.
19. M. Raffinot. On the multi backward dawg matching algorithm (MultiBDM). In R. Baeza-Yates, editor, *Proceedings of the 4rd South American Workshop on String Processing*, pages 149–165, Valparaiso, Chile, November 12-13, 1997. Carleton University Press.
20. D. Sunday. A very fast substring search algorithm. *CACM*, 33(8):132–142, August 1990.
21. S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. of USENIX Technical Conference*, pages 153–162, 1992.
22. S. Wu and U. Manber. Fast text searching allowing errors. *CACM*, 35(10):83–91, October 1992.
23. S. Wu, U. Manber, and E. Myers. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.
24. A. C. Yao. The complexity of pattern matching for a random string. *SIAM Journal on Computing*, 8(3):368–387, 1979.

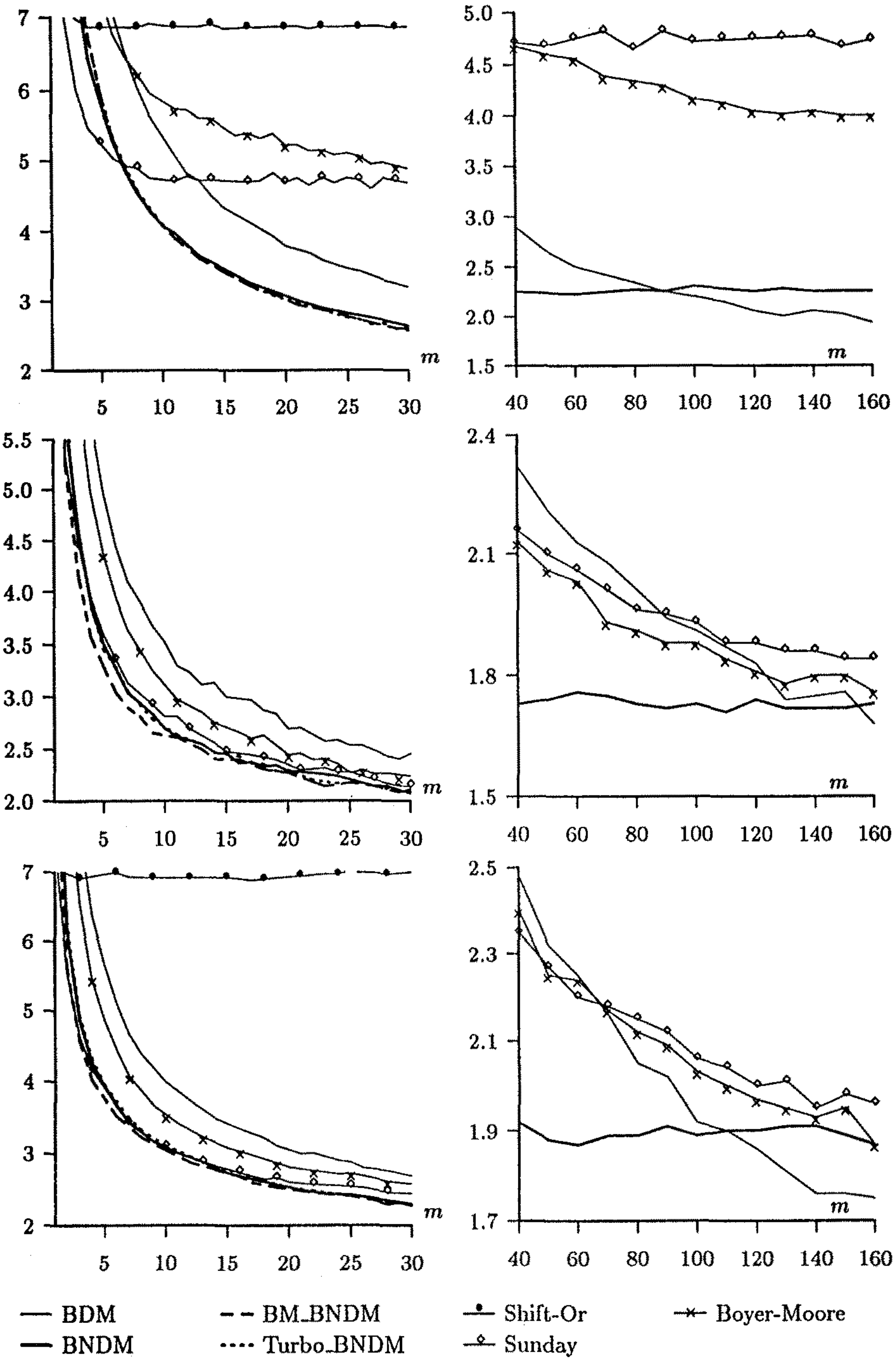

Fig. 7. Times in 1/100-th of seconds per megabyte. For first to third row, random text with $\sigma = 4$, random text with $\sigma = 64$ and English text. Left column shows short patterns, right column shows long patterns.

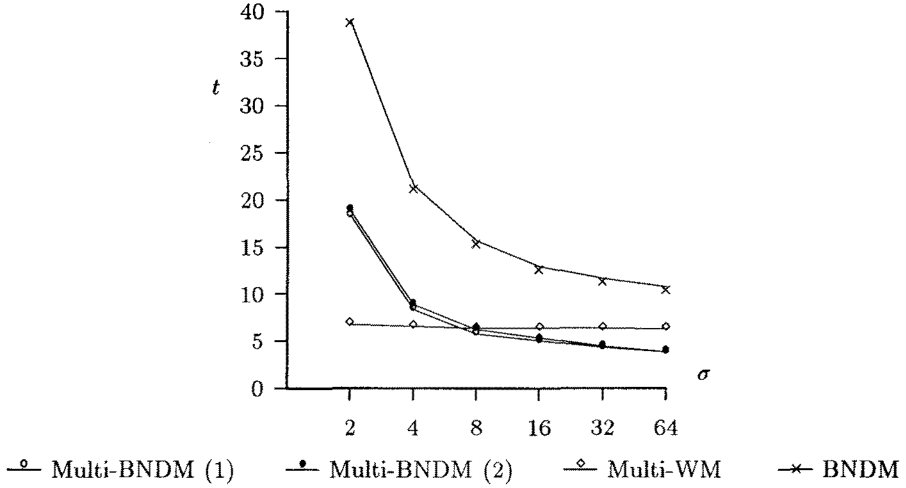

Fig. 8. Times in 1/100-th of seconds per megabyte, for multipattern search on random text of different alphabet sizes (x axis).

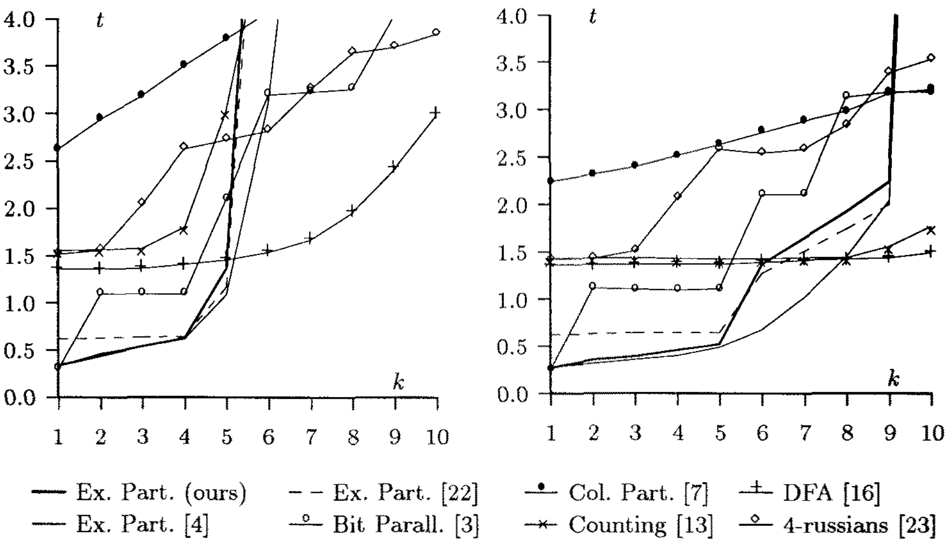

Fig. 9. Times in seconds per megabyte, for random text on patterns of length 20, and σ = 16 and 64 (first and second column, respectively). The x axis is the number of errors allowed.