Fast and Flexible String Matching by Combining Bit-parallelism and Suffix Automata

Gonzalo Navarro
Department of Computer Science, University of Chile
and
Mathieu Raffinot
Equipe génome, cellule et informatique, University of Versailles

The most important features of a string matching algorithm are its efficiency and its flexibility. Efficiency has traditionally received more attention, while flexibility in the search pattern is becoming a more and more important issue. Most classical string matching algorithms are aimed at quickly finding an exact pattern in a text, being Knuth-Morris-Pratt (KMP) and the Boyer-Moore (BM) family the most famous ones. A recent development uses deterministic "suffix automata" to design new optimal string matching algorithms, e.g. BDM and TurboBDM. Flexibility has been addressed quite separately by the use of "bit-parallelism", which simulates automata in their nondeterministic form by using bits and exploiting the intrinsic parallelism inside the computer word, e.g. the Shift-Or algorithm. Those algorithms are extended to handle classes of characters and errors in the pattern and/or in the text, their drawback being their inability to skip text characters. In this paper we merge bit-parallelism and suffix automata, so that a nondeterministic suffix automaton is simulated using bit-parallelism. The resulting algorithm, called BNDM, obtains the best from both worlds. It is much simpler to implement than BDM and nearly as simple as Shift-Or. It inherits from Shift-Or the ability to handle flexible patterns and from BDM the ability to skip characters. BNDM is 30%-40% faster than BDM and up to 7 times faster than Shift-Or. When compared to the fastest existing algorithms on exact patterns (which belong to the BM family), BNDM is from 20% slower to 3 times faster, depending on the alphabet size. With respect to flexible pattern searching, BNDM is by far the fastest technique to deal with classes of characters and is competitive to search allowing errors. In particular, BNDM seems very adequate for computational biology applications, since it is the fastest algorithm to search on DNA sequences and flexible searching is an important problem in that area. As a theoretical development related to flexible pattern matching, we introduce a new automaton to recognize suffixes of patterns with classes of characters. To the best of our knowledge, this automaton has not been studied before.

Partially supported by Chilean Fondecyt Grant 1-990627 (first author) and ECOS/Conicyt action C99E04 (both authors).
Authors' address: Gonzalo Navarro, Department of Computer Science, University of Chile, Blanco Encalada 2120, Santiago, Chile, gnavarro@dcc.uchile.cl. Mathieu Raffinot, Equipe génome, cellule et informatique, Université de Versailles, 45 avenue des Etats-Unis, 78035 Versailles Cedex, raffinot@genetique.uvsq.fr.

Categories and Subject Descriptors: F.2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical algorithms and problems—*Pattern matching, Computations on discrete structures*; H.3.3 [**Information storage and retrieval**]: Information search and retrieval—*Search process*

1. INTRODUCTION

The string-matching problem is to find all the occurrences of a given pattern $p = p_1 p_2 \dots p_m$ in a large text $T = t_1 t_2 \dots t_n$, both being sequences of characters drawn from a finite character set Σ. This problem is fundamental in computer science and is a basic need of many applications, such as text retrieval, symbol manipulation, computational biology, data mining, network security, etc.

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) [Knuth et al. 1977]. The search in KMP is done by scanning the text character by character, and for each text position i remembering the longest prefix of the pattern which is also a suffix of $t_1 \dots t_i$. This approach is $O(n)$ worst-case time but it needs to scan all characters in the text, independently of the pattern. A second algorithm, as famous as KMP and which allows skipping characters, is Boyer-Moore (BM) [Boyer and Moore 1977]. The search in BM is done inside a window of length m, ending at position i in the text. BM searches backwards the longest suffix of $t_1 \dots t_i$ which is also a suffix of the pattern. If the suffix is the whole pattern a match is reported. Then the window is shifted to the next occurrence of the suffix in the pattern. This algorithm leads to several variations, like Horspool [Horspool 1980] and Sunday [Sunday 1990], considered the fastest string-matching algorithms in practice.

A large part of the research in efficient algorithms for string matching can be regarded as the quest 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 too.

Another automaton, called a "suffix automaton", is used in [Crochemore et al. 1993; Crochemore and Rytter 1994; Czumaj et al. 1994; Lecroq 1992; Raffinot 1997b], where the idea is to search a substring instead of a prefix (as KMP), or a suffix (as BM). Optimal sublinear algorithms on average, like BDM or TurboBDM [Crochemore and Rytter 1994; Czumaj et al. 1994], have been obtained with this approach, which has also been extended to multipattern matching [Crochemore et al. 1993; Crochemore and Rytter 1994; Raffinot 1997b] (i.e. looking for the occurrences of any pattern from a given set).

Besides speed, flexibility in the types of patterns that can be searched is becoming a more and more relevant issue in recent years, motivated by text retrieval, computational biology and signal processing applications (see, e.g., [Navarro 2000a]). In these applications the pattern needs not be just a sequence of characters that is to be found verbatim in the text, but it can include

Classes of characters, which are positions that match a set of characters of the alphabet, rather than just on character. This permits in particular searching with *don't care* characters (which match every character) and case insensitive

searching. This models typical needs of text retrieval applications (such as case insensitive searching) and computational biology (where some pattern positions are not completely determined). In computational biology is also possible that the text contains classes of characters.

Bounded length gaps, which are pattern positions that match an arbitrary string whose length is between a minimum and a maximum specified value. This is typically of interest, for example, in protein searching.

Optional and repeatable characters, which are pattern characters or classes that may or may not appear in a text occurrence; or that may appear 0, 1, or more times; respectively. This is of interest in text retrieval.1

Network and regular expressions, which permit building patterns composed from simple letters and the empty string, as well as union and concatenation of other patterns, and (in the case of regular expressions) an arbitrary number of repetitions of another pattern. These patterns are extremely powerful to specify complex searching.

Approximate searching, which permits a limited number of differences between the pattern and its occurrences in the text. Depending on the model, the differences permitted may be character substitutions (Hamming model); character insertions, deletions and substitutions (Levenshtein model); etc. Approximate searching is of great interest when the text is of bad quality (e.g. text obtained by optical character recognition or just poorly written as in the Web, DNA sequences which contain experimental errors, signals that have been corrupted during transmission, etc.), or there is no absolute certainty about the search pattern (e.g. searching for foreign names).

A related line of research in string matching, called "bit-parallelism" [Baeza-Yates 1992], has yielded the best algorithms for flexible searching. The general technique is to use the automata in their nondeterministic form instead of making them deterministic. Usually the nondeterministic versions are very simple and regular. This permits mapping the state of the search onto the bits of a computer word and using the intrinsic parallelism of the bit manipulations of the processor to parallelize the operations necessary to update the state of the search. Competitive algorithms have been obtained for exact string matching (e.g., Shift-Or [Baeza-Yates and Gonnet 1992; Wu and Manber 1992]), as well as for approximate string matching [Baeza-Yates and Gonnet 1992; Baeza-Yates and Navarro 1999; Myers 1999; Wu and Manber 1992; Wu et al. 1996]. Although these algorithms generally work well only on patterns of moderate length, they are simpler, more flexible (e.g. they can easily handle classes of characters), 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 can be seen as a cross between BDM and Shift-Or. BNDM can be extended to search classes of characters, to multipattern search and to approximate search, just like Shift-Or. BNDM uses a nondeterministic suffix automaton that is simulated using bit-parallelism, and it has the advantage of being faster than the previous algorithms that could be extended in such a way (up to 7 times faster than Shift-Or), being faster than BDM (30%-40% faster), and for small alphabets being

up to 3 times faster than the best algorithms of the BM family. Indeed, BNDM is the fastest search algorithm for small alphabets and moderate length patterns. For larger alphabets the BM-Sunday algorithm is up to 20% faster. Additionally BDNM uses few space in comparison with the BDM or TurboBDM algorithms (it does not need to construct the deterministic suffix automaton), and it is very simple to implement (e.g. complex variations of BDM like TurboBDM and BM_BDM are easy to implement).

In particular, the ability to search for classes of characters has never been studied in relation to the BDM family. We give a new definition of an automaton designed to recognize suffixes of patterns with classes of characters) and simulate its nondeterministic version using bit-parallelism.

This paper is organized as follows. In section 2 we present the suffix automaton and the BDM algorithm. In section 3 we present the bit-parallelism approach. In section 4 we present our new algorithms for short and long patterns. We present more complex and improved versions in section 5. The extension to classes of characters is presented in section 6, to multipattern matching in section 7 and to approximate string matching in section 8. We then present experimental results in section 9. Finally, we give our conclusions and future work directions in section 10. Earlier partial versions of this work appeared in [Navarro 1998; Navarro and Raffinot 1998].

We use the following definitions throughout the paper.

A word $x \in \Sigma^*$ is a *factor* (or 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 $p = ux$, $u \in \Sigma^*$. The set of suffixes of p is called Suff(p). When we want to emphasize the inter-letter positions in the pattern, we write $p = {}^0 p_1 {}^1 p_2 {}^2 ... p_{m-1} {}^{m-1} p_m {}^m$.

We denote as $b_\ell ... b_1$ the bits of a computer word of length ℓ. We use exponentiation to denote bit repetition (e.g. $0^3 1 = 0001$). Since the length w of the computer word is fixed, we are hiding the details on where we store the ℓ bits inside it. We give such details when they are relevant. Finally, we use C-like syntax for operations on the bits of computer words: "|" is the bitwise-or, "&" is the bitwise-and, "$\hat{}$" is the bitwise-xor and "$\sim$" complements all the bits. The shift-left operation, "<<", moves the bits to the left and enters zeros from the right, i.e. $b_m b_{m-1} ... b_2 b_1 << r = b_{m-r} ... b_2 b_1 0^r$. The shift-right, ">>" moves the bits in the other direction. Finally, we can perform arithmetic operations on the bits, such as addition and subtraction, which operate the bits as if they formed a number. For instance, $b_\ell ... b_x 10000 - 1 = b_\ell ... b_x 01111$.

2. SEARCHING WITH SUFFIX AUTOMATA

We describe in this section the BDM pattern matching algorithm [Crochemore and Rytter 1994; Czumaj et al. 1994]. This algorithm is based on a suffix automaton. We first describe such automaton and then explain how it is 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 unnecessary 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.

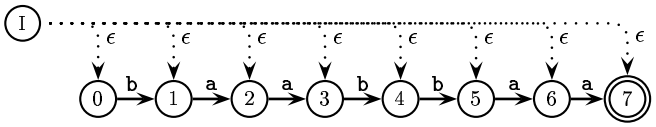

Fig. 1. A nondeterministic suffix automaton for the pattern $p = baabbaa$. Dashed lines represent ϵ-transitions (i.e. they occur without consuming any input). I is the initial state of the automaton.

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 too (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 as a factor of the pattern already recognized, except that we do not distinguish between some factors. Another way to see this 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

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

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 $\{0..m\}$. 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.

The (deterministic) suffix automaton is a well known structure [Blumer et al. 1989; Crochemore 1986; Crochemore and Rytter 1994; Raffinot 1997a], and we do

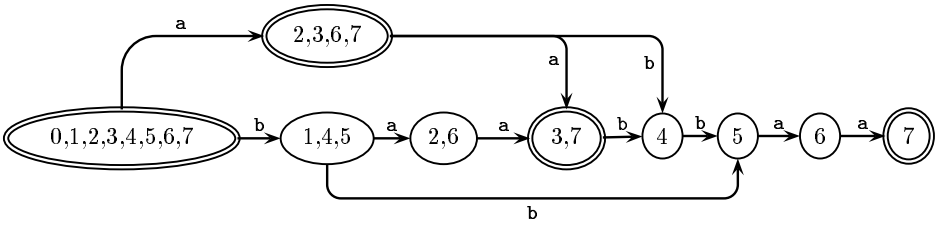

Fig. 2. Deterministic suffix automaton of the word ${}^0b^1a^2a^3b^4b^5a^6a^7$

not prove any of its properties here (nor the correctness of the previous construction). The size of DAWG(p) is linear in m (counting both nodes and edges), and a linear on-line construction algorithm exists [Crochemore 1986]. 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 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 [Crochemore and Rytter 1994; Czumaj et al. 1994] to design a simple pattern matching algorithm called BDM. This algorithm is $O(mn)$ time in the worst case, but optimal on average ($O(n \log m/m)$ time[1]). Other more complex variations such as TurboBDM[Czumaj et al. 1994] and MultiBDM[Crochemore and Rytter 1994; Raffinot 1997b] achieve linear time in the worst 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 recorded (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). Since we remember the last prefix recognized backwards, we have the *longest* prefix of p in the window. This backward search ends in two possible forms:

(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. In this case we shift the window to the right, its starting position corresponding to the position *last* (we cannot miss an occurrence because in that case the suffix automaton would have found its prefix in the window).

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

[1]The lower bound of $\Omega(n \log m/m)$ average time for any pattern matching algorithm under a Bernouilli model with uniform character distribution and a RAM complexity model is from A. C. Yao [Yao 1979].

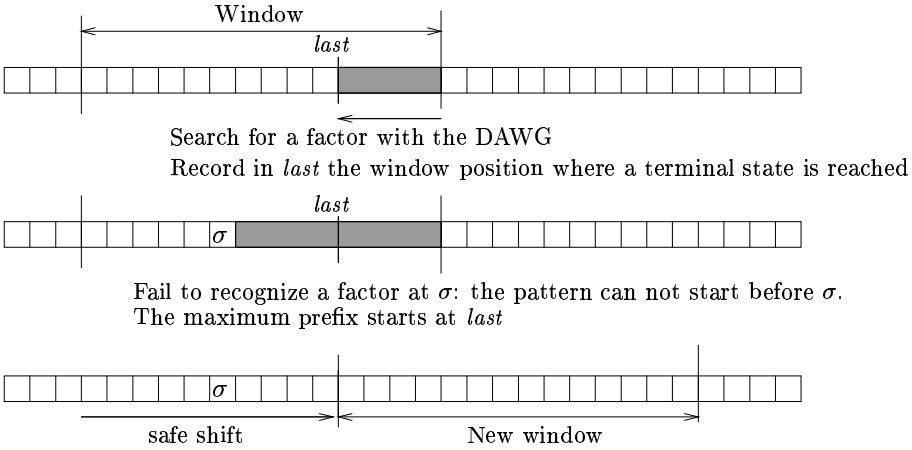

Fig. 3. Basic search with the suffix automaton

The pseudo-code of the BDM algorithm is given in Figure 2.2. We note $\delta_{DAWG}(q,\sigma)$ the transition function of the suffix automaton. $\delta_{DAWG}(q,\sigma)$ is the node that we reach if we move along the edge labeled by σ from the node q. If such an edge does not exist, then $\delta_{DAWG}(q,\sigma)$ is *null*.

```
BDM(p = p_1 p_2 ... p_m, T = t_1 t_2 ... t_n)
1.    Preprocessing
2.        Build DAWG(p^r)
3.    Search
4.        pos ← 0
5.        While pos ≤ n − m do
6.            j ← m, last ← m
7.            state ← initial state of DAWG(p^r)
8.            While state ≠ null do
9.                state ← δ_DAWG(state, t_{pos+j})
10.               j ← j − 1
11.               If state is terminal then
12.                   If j > 0 then last ← j
13.                   Else report an occurrence at pos + 1
14.               End of if
15.           End of while
16.           pos ← pos + last
17.       End of while
```

Fig. 4. Pseudo-code of the BDM algorithm. The variable *pos* points at the character just before the window, j is used to traverse the window backwards and *last* to record the last prefix matched.

2.2.0.1 *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 box. We begin with $T = [\ a\ b\ b\ a\ b\ a\ a\ ]b\ b\ a\ a\ b$, $m = 7$, $last = 7$.

(1) $T = [\ a\ b\ b\ a\ b\ \boxed{a}\ ]\ b\ b\ a\ a\ b$. a is a factor of p^r and a reverse prefix of p. $last = 6$.

(2) $T = [\ a\ b\ b\ a\ \boxed{a\ a}\ ]\ b\ b\ a\ a\ b$. aa is a factor of p^r and a reverse prefix of p. $last = 5$.

(3) $T = [\ a\ b\ b\ \boxed{b\ a\ a}\ ]\ b\ b\ a\ a\ b$. aab is a factor of p^r.

(4) $T = [\ a\ b\ \boxed{a\ b\ a\ a}\ ]\ b\ b\ a\ a\ b$. We fail to recognize the next a. So we shift the window to *last*. We search again in the position: $T = a\ b\ b\ a\ b\ [\ a\ a\ b\ b\ a\ a\ b\ ]$, $last = 7$.

(5) $T = a\ b\ b\ a\ b\ [\ a\ a\ b\ b\ a\ \boxed{b}\ ]$. b is a factor of p^r.

(6) $T = a\ b\ b\ a\ b\ [\ a\ a\ b\ b\ a\ \boxed{a\ b}\ ]$. ba 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 [Baeza-Yates and Gonnet 1992], a new approach to text searching was proposed. It is based on *bit-parallelism* [Baeza-Yates 1992], a technique consisting in taking advantage of the intrinsic parallelism of the bit operations inside a computer word. By using cleverly this fact, the number of operations that an algorithm performs can be cut down by a factor of at most w, where w is the number of bits in the computer word. Since in current architectures w is 32 or 64, the speedup is very significant in practice.

The Shift-Or algorithm uses bit-parallelism to simulate the operation of a nondeterministic automaton that searches the pattern in the text (see Figure 5). As this automaton is simulated in time $O(mn)$, the Shift-Or algorithm achieves $O(mn/w)$ worst-case time (optimal speedup). Notice that if we convert the nondeterministic automaton to a deterministic one so as to have $O(n)$ search time, we get a version of the KMP algorithm [Knuth et al. 1977] (KMP, however, is twice as slow as Shift-Or for $m \leq w$).

We explain now the Shift-And algorithm, which is an easier-to-explain (though a little less efficient) variant of Shift-Or. The algorithm first builds a table B which for each character stores a bit mask $b_m...b_1$. 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 $D = d_m...d_1$, where d_i is set whenever $p_1p_2...p_i$ matches the end of the text read up to now (another way to see it is to consider that d_i tells whether the state numbered i in Figure 5 is active). Therefore, we report a match whenever d_m is set.

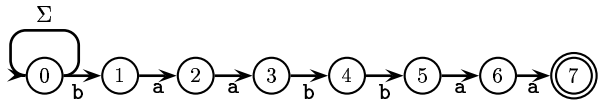

Fig. 5. A nondeterministic automaton to search the pattern $p = baabbaa$ in a text. The initial state is 0.

We set $D = 0^m$ originally and, for each new text character t_j, update D using the formula

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

The formula is correct because the i-th bit is set if and only if the $(i-1)$-th bit was set for the previous text character and the new text character matches the pattern at position i. In other words, $t_{j-i+1}..t_j = p_1..p_i$ if and only if $t_{j-i+1}..t_{j-1} = p_1..p_{i-1}$ and $t_j = p_i$. Again, it is possible to relate this formula to the movement that occurs in the nondeterministic automaton for each new text character: each state gets the value of the previous state, but this happens only if the text character matches the corresponding arrow. Finally, the "$\mid 0^{m-1}1$" after the shift allows a match to begin at the current text position. This corresponds to the self-loop at the beginning of the automaton and is saved in Shift-Or, where all the bits are complemented.

The cost of this algorithm is $O(n)$. Although we consider only masks of length m here, in practice the masks are of length w (as explained earlier) and some provisions may be necessary to handle the unwanted extra bits. 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), with a worst-case cost of $O(mn/w)$ and an average case cost of $O(n)$.

The Shift-And algorithm is very simple, and has some further advantages. The most immediate one is that it is very easy to extend so as to handle classes of characters. That is, each pattern position does not match just a single character but a set of characters. If C_i is the set of characters at position i in the pattern, then we set the i-th bit of $B[c]$ for all $c \in C_i$. In [Baeza-Yates and Gonnet 1992] they show also how to allow a limited number k of mismatches in the occurrences, at $O(nm \log(k)/w)$ cost.

Later [Wu and Manber 1992] enhanced this paradigm to support extended patterns, which allow wild cards (i.e. gaps of unbounded length), regular expressions, approximate search with nonuniform costs, and combinations. Further development of the bit-parallelism approach for approximate string matching led to some of the fastest algorithms for short patterns [Baeza-Yates and Navarro 1999; Myers 1999]. In most cases, the key idea was to simulate a nondeterministic finite automaton. It is interesting also to mention [El-Mabrouk and Crochemore 1996], which searches allowing mismatches by using a combination of bit-parallelism and Boyer-Moore.

Bit-parallelism has become a general way to simulate simple nondeterministic automata instead of converting them to deterministic. This is how we use it in this paper.

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, as shown in the next sections. We first assume that $m \leq w$ and later show how to extend the algorithm for longer patterns.

4.1 The Basic Algorithm

We simulate the automaton of Figure 1 on the reversed pattern. 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 follow any transition. In our case, the bit d_i at iteration k is set if and only if $p_{m-i+1}..._{m-i+k} = t_{pos+1+m-k}..t_{pos+m}$. Some observations follow

—Since we begin at iteration 0, the initial value for D is 1^m (recall that we use exponentiation to denote bit repetition).

—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 (excluding the complete pattern) corresponds to the next window position (variable *last*).

—Since there is no initial self-loop, this automaton eventually runs out of active states. Moreover, states $(m-k) \ldots m$ are inactive at iteration k.

The algorithm works 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 i where $p_i = c$ (just as in Shift-And). Interestingly enough, the formula to update D turns out to be very similar to that of the Shift-Or algorithm:

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

which should not be surprising given the similarity between both automata. The algorithm is summarized in Figure 6. Some optimizations done on the real code, related to improved flow of control and bit manipulation tricks, are not shown for clarity.

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

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

We note the current window between square brackets and the recognized prefix in a box. We begin with

```
BNDM  (p = p1p2...pm,  T = t1t2...tn)
1.   Preprocessing
2.       For c ∈ Σ do B[c] ← 0^m
3.       For i ∈ 1...m do B[p_{m-i+1}] ← B[p_{m-i+1}] | 0^{m-i}10^{i-1}
4.   Search
5.       pos ← 0
6.       While pos ≤ n − m do
7.           j ← m,  last ← m
8.           D = 1^m
9.           While D ≠ 0^m do
10.              D ← D & B[t_{pos+j}]
11.              j ← j − 1
12.              If D & 10^{m-1} ≠ 0^m then
13.                  If j > 0 then last ← j
14.                  Else report an occurrence at pos + 1
15.              End of if
16.              D ← D << 1
17.          End of while
18.          pos ← pos + last
19.      End of while
```

Fig. 6. Bit-parallel code for **BDM**. Some optimizations are not shown for clarity.

$T =$ [*a b b a b a a*] *b b a a b*, $D = 1\,1\,1\,1\,1\,1\,1$, $B =$

a	1 1 0 0 1 1 0
b	0 0 1 1 0 0 1

, $m = 7$, *last* $= 7$, $j = 7$.

(1) $T =$ [*a b b a b a* **a**] *b b a a b*.

	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) $T =$ [*a b b a b* **a a**] *b b a a b*.

	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) $T =$ [*a b b a* **b a a**] *b b a a b*.

	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) $T =$ [*a b b* **a b a a**] *b b a a b*.

	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: $T =$ *a b b a b* [*a a b b a a b*], *last* $= 7$, $j = 7$.

(5) $T =$ *a b b a b* [*a a b b a a* **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) $T =$ *a b b a b* [*a a b b a* **a b**].

	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) $T =$ a b b a b [a a b b a a b].

	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) $T =$ a b b a b [a a b b a a b].

	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) $T =$ a b b a b [a a b b a a 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) $T =$ a b b a b [a a b b a a b].

	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) $T =$ a b b a b [a a b b a a b].

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

$j = 0$
$last = 4$

We 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 of a long computer word (we call this a "multi-word simulation" of the simple algorithm). We propose a different alternative which was experimentally found to be faster.

If $m > w$, we partition the pattern in $M = \lceil m/w \rceil$ consecutive subpatterns s_i, $p = s_1\ s_2\ ...\ s_M$, so that each subpattern s_i is of length $m_i = w$ if $i < M$ and the last one has the remaining characters (i.e. $m_M = m - w(M-1)$). Those subpatterns can therefore be 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 check 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 some subpattern s_i in its window.

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

Although this algorithm searches on a shorter window (i.e. of length $w < m$) and therefore it performs shifts shorter than the multi-word simulation, this multi-word simulation has to work on M computer words to traverse the window, in general cancelling any possible benefit from performing a longer shift. Finally, the multi-word simulation switches very fast the D_t word it operates on, while our algorithm operates a long time over a single D_t word, therefore making it profitable to put D_t in a computer register for faster operation.

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 of the search time is the same as for the BDM algorithm. That is, $O(mn)$ in the worst case (e.g. $T = a^n$, $p = a^{m-1}b$),

$O(n/m)$ in the best case (e.g. $T = a^n$, $p = b^m$), and $O(n \log_{|\Sigma|} m/m)$ on average. 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 $O(nm^2/w)$ time 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 leads to $O(n/w)$ time. 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 search for s_2 in the window costs $O(w)$ at most. With probability $1/|\Sigma|^w$ we find s_2 and search for s_3, and so on. The total cost incurred by the existence of $s_2 ... s_M$ is at most

$$\sum_{i=1}^{M} \frac{w}{|\Sigma|^{wi}} \quad \leq \quad \varepsilon \quad = \quad \frac{w}{|\Sigma|^w} = O(1)$$

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 weighted with their probability of participating, we get into the same sum above, which is $\varepsilon = O(1)$. Therefore, the average shift is $last_1 + \varepsilon = last_1 + O(1)$, and hence the cost is that of searching s_1 plus lower order terms.

Notice that, on the other hand, the multi-word simulation has worse complexity, namely $O(n \log_{|\Sigma|}(m)/w)$, since it performs the same number of operations as BDM (i.e. $O(n \log_{|\Sigma|}(m)/m)$) but for each operation it has to update $O(m/w)$ machine words.

5. FURTHER IMPROVEMENTS

5.1 A Linear Time Algorithm

Although our algorithm has an 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 by one character (e.g. $T = a^n$, $p = a^{m-1}b$). In the worst case, the algorithm is $O(nm^2/w)$. Our aim now is to reduce its worst case to $O(nm/w)$, i.e. $O(n)$ when $m = O(w)$.

In the last few years, studies have been undertaken to obtain, using DAWGs, algorithms which are linear in the worst case and still sublinear on average, for instance TurboRF[2] in [Czumaj et al. 1994], TurboBDM in [Czumaj et al. 1994; Lecroq 1992]. The main idea is to avoid retraversing the same characters in the backward window

[2]TurboRF uses a suffix tree, but it can be adapted to DAWGs.

verification. When we determine that the window must be advanced in *last* positions, for $last < m$, we already know that $t_{j+last}..t_{j+m-1}$ is a prefix of the pattern, and therefore it is possible to use this knowledge to avoid traversing backwards the complete window $t_{j+last}..t_{j+last+m-1}$. The ending position $(j + last + m - 1)$ of the prefix in the window is usually called the *critical position*. Therefore, we want to avoid that the backward window verification continues after reaching the critical position.

The main problem is how to determine the next shift if we are not going to traverse again the area $t_{j+last}..t_{j+m-1}$. Recall that we have not stored information about the next possible shifts following *last* (we only remembered the shortest shift).

Two main strategies exist. The first one is to use a KMP algorithm to read again the characters we read with the DAWG once we reach the critical position. We keep in memory the longest prefix of the pattern that is also a suffix of the text we read. We stop using the KMP algorithm when the maximal prefix we found is less than half the size of the pattern. This strategy is used in [Crochemore and Rytter 1994; Lecroq 1992; Raffinot 1997b]. The algorithm obtained is linear in the worst case, but the DAWG is used just to "help" KMP to skip some characters.

The second strategy makes a better use of the power of DAWGs by adding a kind of BM machine to the BDM algorithm. To explain the algorithm we need the definition of a *border*: the border of a string u is the set of prefixes of u which are also suffixes.

The algorithm works as follows: if we reach the critical position after reading 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, then we have recognized the whole pattern p, and the next shift corresponds to the longest prefix of p that is also a suffix of p, i.e the longest border of p, which can be computed in advance.

—If z^r is not a suffix, then it appears in the pattern in a set of positions which is given by the state we reached in the suffix automaton. If we shift to the rightmost occurrence of z^r in the pattern, like in the BM algorithm, then the shift is safe.

It is not difficult to simulate this idea in our BNDM algorithm. To know whether the factor z we read with the DAWG is a suffix, we just have to test whether there is a 1 at the $|z|$-th bit in D, i.e. $d_{|z|}$. To get the rightmost occurrence, we seek the rightmost 1 in D, which we can get (if it exists) in constant time with $\log_2(D \,\&\, \sim (D-1))$ [3]. We implemented this algorithm under the name BM_BNDM in the experimental part of this paper, but the plain BNDM is faster in practice.

Still this algorithm remains quadratic, because we do not keep a prefix of the pattern after the BM shift. To make it linear time, we must keep this prefix. This situation is shown in Figure 7.

Let u be the prefix finishing at the critical position. The TurboRF algorithm (second variation) [Czumaj et al. 1994] uses a complicated preprocessing phase to

[3] In practice, 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 $D \,\hat{}\, (D-1)$ and get the same result. However, the $\log_2$ expression is important in theory because it can be computed in constant time.

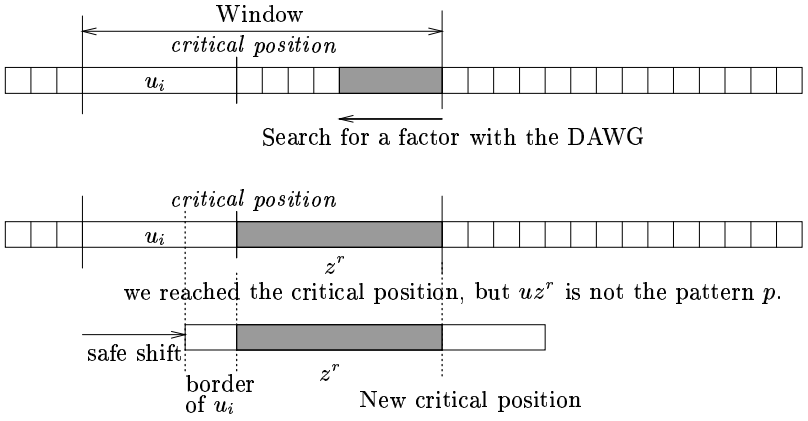

Fig. 7. Skeleton of the BM shift if we reach the critical position.

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 TurboRF uses a suffix tree, and it is quite difficult (though not impossible) 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 can be precomputed in $O(m)$ time. Now, to join one occurrence of z^r with 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 TurboBNDM 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 $\Sigma_2 = \{0,1\}$. 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 discarded later). This involves an average search time of

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

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

We present now some extensions applicable to our basic scheme, which form a successful combination of efficiency and flexibility. The general concept is that all

the extensions devised for the Shift-Or algorithm can be enriched with our approach in order to speed them up.

6. HANDLING 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. Some solutions for the case of don't care characters (i.e. pattern positions that match any character) have been presented in [Abrahamson 1987; Fischer and Paterson 1974; Pinter 1985], but these have been shown to be only of theoretical interest in [Baeza-Yates and Gonnet 1992]. Simple attempts to extend classical algorithms such as KMP or BM do not work well. To the best of our knowledge, the fastest algorithm for this problem is Shift-Or.

This type of patterns is called "limited expressions" in [Wu et al. 1996], and it is a subset of the wealth of alternatives for "extended patterns" presented in [Baeza-Yates and Gonnet 1992; Wu and Manber 1992]. Although formally it is enough to say that each pattern position can match a set of characters, it is useful to give an intuitive idea of the power allowed. The following patterns are examples of limited expressions:

—`word` in case insensitive, i.e. $\{\mathtt{w},\mathtt{W}\}\{\mathtt{o},\mathtt{O}\}\{\mathtt{r},\mathtt{R}\}\{\mathtt{d},\mathtt{D}\}$.

—`wo.d`, where the `'.'` means any character, i.e. $\{\mathtt{w}\}\{\mathtt{o}\}\Sigma\{\mathtt{d}\}$.

—`wor[a-z]`, where `[a-z]` means any character in the range from `'a'` to `'z'`, i.e. $\{\mathtt{w}\}\{\mathtt{o}\}\{\mathtt{r}\}\{\mathtt{a..z}\}$.

—`wo[abx]d`, where `[abx]` means `'a'`, `'b'` or `'x'`, i.e. $\{\mathtt{w}\}\{\mathtt{o}\}\{\mathtt{a},\mathtt{b},\mathtt{x}\}\{\mathtt{d}\}$.

—`w[~ou]rd`, where `[~o]` means any character except `'o'` and `'u'`, i.e. $\{\mathtt{w}\}(\Sigma - \{\mathtt{o},\mathtt{u}\})\{\mathtt{r}\}\{\mathtt{d}\}$.

We denote a limited expression $p = C_1C_2\ldots C_m$. A word $x = x_1x_2\ldots x_r$ in Σ^* is a factor of a limited expression $p = C_1C_2\ldots C_m$ 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 $x = x_1x_2\ldots x_r$ of $p = C_1C_2\ldots C_m$ 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 the suffixes of a limited expression $p = C_1C_2\ldots C_m$. 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, and then prove its correctness.

6.1 Construction

The construction we use is quite similar to the one given for the DAWG, but with 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 limited expression *b[a,b]abbaa*, and *L-endpos*$(bba) = \{3, 6\}$ (notice that, unlike before, the sets of positions may be non-disjoint and no one a subset of the other). We define the equivalence relation $\equiv_E$ for u, v factors of p by

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

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 a limited expression and $\equiv_E$ the equivalence relation on its factors (as previously defined). The equivalence relation $\equiv_E$ is compatible with the concatenation of words.*

PROOF. Let u and v be two different factors of p that belong to the same equivalence class q, and let $\sigma \in \Sigma$. $S = \{i_1, i_2, \ldots, i_k\}$ is the set of positions corresponding to q. Two cases appear:

—if $u\sigma$ (resp. $v\sigma$) is not a factor of p, neither is $v\sigma$ (resp. $u\sigma$). Suppose $u\sigma$ is not a factor, but $v\sigma$ is. Then there exists a position $2 \leq i \leq m$ where $v\sigma$ ends in p. Hence v ends at $i-1$. But, as u and v are at the same positions, u appears also at position $i-1$ in p, and $u\sigma$ appears in i. A contradiction.

—if $u\sigma$ (resp. $v\sigma$) is a factor of p, $v\sigma$ (resp. $u\sigma$) is also a factor of p and $u\sigma \equiv_E v\sigma$. Assume that $u\sigma$ is a factor, then $u\sigma$ ends in p at positions $S_\sigma = \gamma(i_1,\sigma) \cup \ldots \cup \gamma(i_k,\sigma)$. As v ends at the same set of positions S as u, $v\sigma$ ends at S_σ too. Therefore $u\sigma$ and $v\sigma$ belong to the same equivalence class.

Hence, the equivalence $\equiv_E$ is compatible with the concatenation. □

This lemma allows us to define an automaton from our equivalence class. The 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 this is not empty. The initial node of the automaton is the set that contains all the positions. Terminal nodes of the automaton are the sets of positions that contain m. As an example, the suffix automaton of the word *[a,b]aa[a,b]baa* is given in Figure 8.

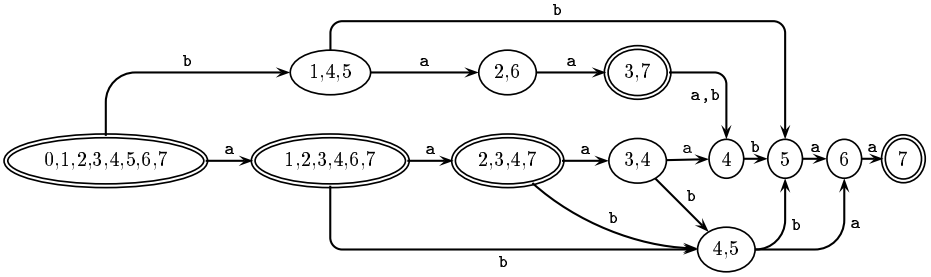

Fig. 8. Extended_DAWG of the limited expression ${}^0[a,b]^1a^2a^3[a,b]^4b^5a^6a^7$.

LEMMA 2. *The Extended_DAWG of a limited expression $p = C_1C_2\ldots C_m$ recognizes the set of suffixes of p.*

PROOF. (1) Let $u = u_1u_2\ldots u_r$ be a suffix of p. We show that u is recognized by Extended_DAWG(p). We call $E_r = \{i_1, i_2, \ldots i_k\}$ the set of ending positions of u in p, which is not empty since it at least contains m. We denote:

$$E_0 = \{0,1,2,\ldots,m\} \text{ and } E_j = \{i_1 - r + j, i_2 - r + j, \ldots, i_k - r + j\}.$$

E_0 is the initial set of Extended_DAWG(p). There is a path from E_0 to a state $E'_1 \supseteq E_1$ labeled u_1, because $E'_1 = \gamma_p(1, u_1) \cup \gamma_p(2, u_1) \cup \ldots \cup \gamma_p(m, u_1)$, and there is at least one u_1 in the positions E_1 (set of beginning positions of u in p). Assume now there is a path from the initial state labeled $u_1 u_2 \ldots u_j$ arriving at the set of nodes E'_j, $j < r$ and $E'_j \supseteq E_j$. Let E'_{j+1} the state we reached by using the edge labeled u_{j+1} from E'_j. This state exists, because $E_j \subseteq E'_j$, E_j is not empty and u_{j+1} appears at least at position E_{j+1}. More than that, for the same reason, $E_{j+1} \subseteq E'_{j+1}$. By induction, we proved that there is a path from the initial node labeled u arriving at the set of nodes E'_r, which contains E_r. As E_r contains m, E'_r also does. Therefore, E'_r is marked as a terminal state in Extended_DAWG(p) and the suffix u is recognized.

(2) If there is a path from the initial state to a final state labeled by the word u in Extended_DAWG(p), then we show that u is a suffix of p. Let now E_j be the state we reach with $u_1 \ldots u_j$. E_r contains m. To arrive at this state by reading u_r, u_r must at least belong to C_m, and the previous state, E_{r-1}, contains $m-1$. By induction, it is clear that $u_r \in C_m, u_{r-1} \in C_{m-1}, \ldots u_1 \in C_{m-r+1}$, and hence u is a suffix of p.

Therefore, Extended_DAWG(p) recognizes the set of suffixes of p. □

We can use this new automaton to recognize the set of suffixes of a limited expression 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.

6.2 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 6 to

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

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

We combine the flexibility of limited expressions 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, as we show later in the experiments, BNDM 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.

7. SEARCHING FOR MULTIPLE PATTERNS

Suppose we are interested in searching a set of patterns $P^1...P^r$ (where $P^i = p^i_1..p^i_{m_i}$), i.e. reporting the occurrences of all P^i's. Assume that they are all of the same length m, otherwise truncate them to the length of the shortest one. This may be ineffective for patterns of very different lengths but it is a common practice in all the algorithms of the Boyer-Moore family as well.

If the total length of the patterns does not exceed the size of a computer word, i.e. $r \times m \leq w$, we can very efficiently search all the patterns in parallel, exploiting again the intrinsic parallelism inside computer words. This technique, based on an arrangement described in [Baeza-Yates and Gonnet 1992], concatenates the r patterns $P^1...P^r$ as follows

$$P = p^1_1\ p^2_1\ ...p^r_1\ p^1_2\ p^2_2\ ...p^r_2\ .....\ p^1_m\ p^2_m\ ...p^r_m$$

(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 6 is that the shift is not by one bit but by r bits in line 16 (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 $1^r0^{r(m-1)}$ in line 12.

This method will automatically search for 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 as follows:

$$P = P^1\ P^2\ ...\ P^r$$

(i.e. just concatenate the patterns). In this case the shift in line 16 is by 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 16 by $D = (D << 1)\ \&\ (1^{m-1}0)^r$. This involves an extra operation. Finally, this arrangement allows us to have patterns of different lengths for the algorithm of [Baeza-Yates and Gonnet 1992] which is not possible in their current proposal.

Clearly this technique cannot be applied to the case $m > w$. However, if $2m \leq w$ 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. Therefore the cost to search r patterns of length m can be made $O(rm^2n/w)$ in the worst case, and $O(rn/w)$ in the best case. This is respectively better than $O(rmn)$ and $O(rn/m)$ (which corresponds to sequentially searching the r patterns with BDM).

8. APPROXIMATE STRING MATCHING

Approximate string matching is the problem of finding all text factors which are at a "distance" of at most k to the pattern. This has a number of applications in text retrieval, computational biology, pattern recognition, signal processing, etc. Of course, the nature of the problem depends directly on the distance function we use. Many distances exist, and among them two are commonly used: the Hamming

and the Levenshtein (or edit) distance. We explain now how to use our algorithm for approximate matching with these two distances.

8.1 Hamming Distance

The *Hamming distance* between two words is the minimal number of substitutions of characters that have to be performed to make them equal. For example, $d(\texttt{"test"},\texttt{"text"}) = 1$. A number of algorithms exist to solve this problem [Baeza-Yates and Gonnet 1994; El-Mabrouk and Crochemore 1996; Tarhio and Ukkonen 1993].

To adapt our algorithm to this problem, we still move a window of length m on the text, and search backward a suffix u of the window that matches a pattern factor after at most k substitutions. Instead of just storing one bit to know whether u^r ends at each position i in the reverse pattern p^r, we use $L = \lfloor \log_2(k) \rfloor + 1$ bits to encode the distance between u^r and the factor of length $|u|$ that ends at position i in p^r. If this distance is larger than k we just encode $k+1$. We record in the variable *last* the longest suffix of the window that is at distance at most k to a pattern prefix.

When reading a new character of the window, we update the state of the search by *adding* a properly spaced B mask to the current set of distances, so that each mismatch adds 1 to the distances. Some provisions are needed to prevent the distances to grow over $k+1$ (basically clearing overflow bits). We can know in $O(1)$ time whether or not to update *last* by examining the L highest bits of the computer word and determining whether the number is larger than k or not. If all the distances in the computer word are greater than k, then we can shift the window to *last* since no pattern factor matches the window with k errors or less. This fact can be tested in constant time by storing the distances plus $2^L - k - 1$, so when the distance reaches $k+1$ the highest bit is set. Hence when all the highest bits are set we know that we can shift the window.

Figure 9 illustrates this algorithm. We note that most of the bit manipulation part comes from [Baeza-Yates and Gonnet 1994].

8.2 Edit Distance

The *Levenshtein distance* (or just *edit distance*) between two words is the minimal number of substitutions, insertions and deletions of characters needed to make them equal. For instance, $d(\texttt{"survey"},\texttt{"surgery"}) = 2$. A number of solutions to this problem exist [Navarro 2000a], being [Baeza-Yates and Navarro 1999; Navarro and Baeza-Yates 1999; Jokinen et al. 1996; Myers 1999; Navarro 1997; Wu et al. 1996] the fastest in practice.

We present two extensions of our algorithm for approximate string matching. Just like the original proposals they are based on, our solutions can be extended to handle extensions of the edit distance, e.g. permitting each operation to have a different cost.

8.2.1 *Partitioning into Exact Searching.* In [Wu and Manber 1992], a simple but very effective filter is proposed for approximate string matching. It is based on the observation that if a pattern of length m appears with at most k errors in a text position, and we divide the pattern in $k+1$ pieces, then at least one of the

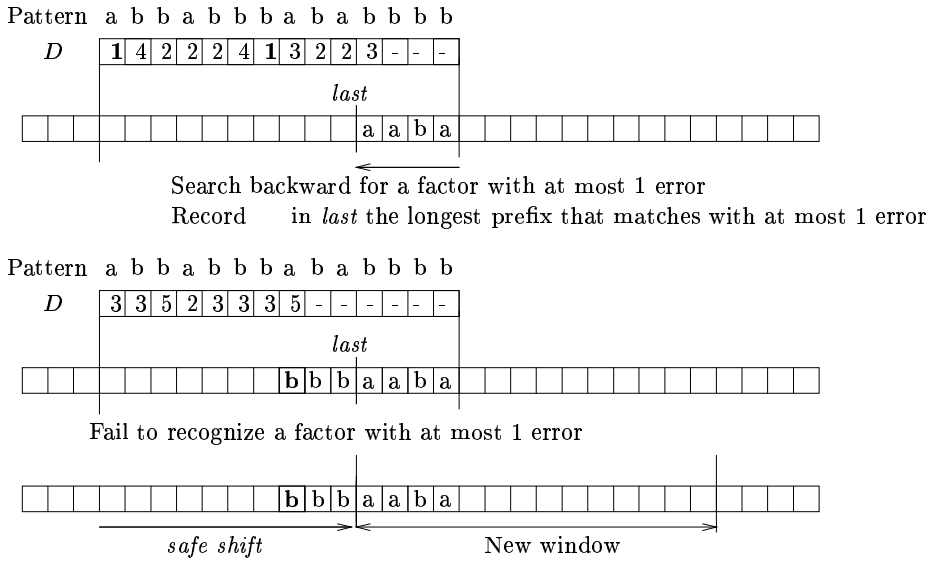

Fig. 9. Basic search for approximate pattern matching with the Hamming distance.

pieces will appear with no errors in the occurrence (since k errors cannot alter $k+1$ pieces). Therefore, they propose to split the pattern into $k+1$ pieces of equal length $\lfloor m/(k+1) \rfloor$ (discarding some characters at the end if necessary) and searching all the pieces in parallel. A classical algorithm is run on the text areas surrounding the occurrences of pattern pieces, therefore filtering out all the rest of the text.

The multipattern search mechanism they propose is very similar to our setup of Section 7 (although the bit arrangement is different). However, they use the Shift-Or algorithm to search and therefore their efficiency is limited. On the other hand, they keep their ability to handle classes of characters and other extensions.

Later, [Navarro and Baeza-Yates 1999] used a multipattern Boyer-Moore strategy to perform the above search, which at the cost of not allowing limited expressions gives a much more efficient algorithm. This algorithm was shown to be the fastest in practice when the number of errors is low enough (this is, $k/m \leq 1/(3 \log_{|\Sigma|} m)$ on random text and $k/m \leq 1/4$ on natural language).

Our multipattern search technique presented in Section 7 combines the best of both worlds: our performance is comparable to that of the algorithms of the Boyer-Moore family, and we keep the flexibility of the Shift-Or approach to handle classes of characters. In this case the Sunday extension to multipattern search used in [Navarro and Baeza-Yates 1999] is slightly faster in general because the search patterns are rather short. We show later their relative performance.

8.2.2 *A New Bit-Parallel Algorithm.* Another algorithm for approximate string matching is presented in [Wu and Manber 1992]. It is based on the bit-parallel simulation of an NFA built from the pattern, which recognizes its approximate occurrences in the text. In [Baeza-Yates and Navarro 1999] this automaton is simulated using a different technique.

Our approach is based on the same automaton. We modify the NFA so that it

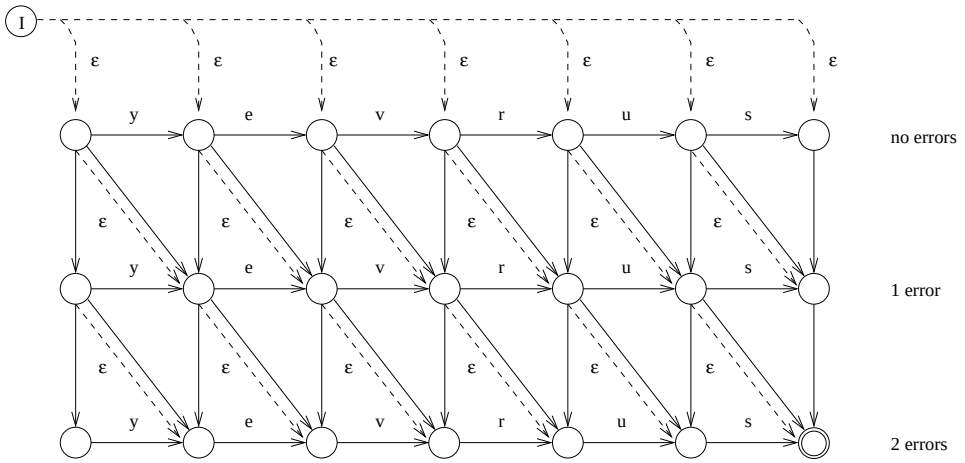

Fig. 10. Our NFA to recognize suffixes of the pattern "survey" reversed. Unlabeled arrows match any character.

recognizes not only the whole pattern but also any suffix of the pattern, allowing up to k errors.

Figure 10 illustrates the modified NFA. First disregard the state labeled "I" and the ϵ-transitions leaving it. Each row denotes the number of errors seen. The first one 0, the second one 1, and so on. Every column represents matching the pattern up to a given position. Horizontal arrows represent matching a character, vertical arrows represent insertions in the pattern, solid diagonal arrows represent substitutions, and dashed diagonal arrows represent deletions in the pattern (they are ϵ-transitions). The automaton accepts a text position as the end of a match with k errors whenever the rightmost state of the last row is active.

Consider now the initial state "I" we added. The ϵ-transitions leaving from the initial state allow the automaton to recognize, allowing k errors, not only the whole pattern but also any suffix of it. Our second modification on the original automaton of [Baeza-Yates and Navarro 1999; Wu and Manber 1992] is the removal of a self-loop at the top-left state, which allowed it to start a match at any text position.

In the case of edit distance, the size of a text occurrence ranges from $m-k$ to $m+k$. We move a window of length $m-k$ on the text, and search backward a suffix u of the window which matches the pattern with at most k errors. This search is done using the NFA explained above, which is built on the reversed pattern. We record in the variable *last* the longest suffix of the window that is at distance at most k to a pattern prefix. We can know in constant time when to update *last* by checking whether the rightmost bottom state of the NFA is active. On the other hand, if the NFA runs out of active states we know that a match is not possible in the window and we can shift to the *last* position where we found a prefix, as in the exact matching algorithm.

Each time we move the window to a new position we restart the automaton with all its states active, which represents setting the initial state to active and letting the ϵ-transitions flush this activation to all the automaton (the states in the lower-left triangle are also activated to allow initial insertions). If after reading the whole

window the automaton still has active states, then it is possible that the current window starts an occurrence, so we use the traditional automaton to compute the edit distance from the initial window position in the text. After reading at most $m + k$ characters we have either found a match starting at the window position or left the automaton without active states.

The rationale for this algorithm is as follows. We are interested only in occurrences that start at the current window position. Any occurrence has a length between $m - k$ and $m + k$. If there is an occurrence of the pattern p starting at the window position with k errors, then a prefix of p must match the first $m - k$ characters with k errors or less. Hence, we cannot miss an occurrence if we keep count of the matches of all the pattern prefixes in a window of length $m - k$. If the automaton runs out of active states, then we cannot miss the start of an occurrence and we shift the window to the next candidate. Finally, if the automaton has active states after reading the complete window, then a match starting at the window is possible and we have to check it explicitly since we can only ensure that a factor of the pattern matches in the window.

The automaton can be simulated in a number of ways. Wu and Manber [Wu and Manber 1992] do it row-wise (each row of the automaton is packed in a computer word), while Baeza-Yates and Navarro [Baeza-Yates and Navarro 1999] do it diagonal-wise. In this case we prefer the technique of Wu and Manber, since in the other the initial diagonals of length $\leq k$ are discarded and they are needed here.

9. EXPERIMENTAL RESULTS

We ran extensive experiments on random and real-world texts in order 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 and a machine word of 32 bits, under Solaris 2.5.1. We measured CPU times and repeated the experiments many times so that the relative error of the results is $\pm 2\%$ with 95% confidence (this involved thousands of repetitions).

All the algorithms were implemented by ourselves with a uniform I/O interface. The text is read in chunks of 64 Kb, which gives the best tradeoff between locality of reference and disk accesses in our machine. We use `open` instead of `fopen` because it is much faster. The pattern is placed at the end of the text buffer to avoid checking for the end point all the times. We made our best coding effort to implement all the algorithms, carefully optimizing the register usage and turning on the compiler optimizations.

We used texts of 10 Mb of size over which we searched many patterns. We ran experiments on random text with uniformly distributed alphabets of sizes from 2 to 64, as well as non-random text, such as English text (from the TREC Wall Street Journal collection) and DNA sequences (from "h.influenzae"). For random text the patterns were randomly generated on the same alphabet, while for non-random texts the patterns were selected randomly from the same text (at word beginnings in the case of natural language).

9.1 Structural Measures

Before measuring real CPU times, we will study the number of operations of different kinds executed by our algorithms in comparison to the rest. These measures

help explain why the simple BNDM version is better than the other algorithms in a wide range of cases. We have included BDM, the simple BNDM, our two variants BM_BNDM and TurboBNDM, the classical Boyer-Moore, its Horspool and Sunday versions and the linear time algorithms that do not improve with the pattern length: the naive algorithm, KMP and Shift-Or.

Figure 11 shows the number of characters inspected on random texts of different alphabet sizes. In these plots BDM and BNDM are indistinguishable. As can be seen, the theoretical improvements of BM_BNDM and TurboBNDM have a practical effect only for binary alphabets and short patterns ($m \leq 10$). Only Boyer-Moore comes close to BDM/BNDM for small alphabet sizes, but it gets farther and farther as the pattern length grows. From the linear time algorithms, Shift-Or performs exactly one access to each text character, followed by KMP and Naive.

As the alphabet size grows, all the Boyer-Moore algorithms become closer to BDM/BNDMs. The classical Boyer-Moore becomes indistinguishable from BDM for an alphabet of size 16 (which is similar to natural language), while Horspool and Sunday never get close enough. The linear time algorithms also get closer to Shift-Or, but they never reach it.

This shows that our new algorithms are the best in terms of number of text characters inspected, but they are not better than BDM and (sometimes) than Boyer-Moore in this respect.

To get more insight on the reasons behind the different behavior of the algorithms, Figure 12 shows the number of table accesses performed by the algorithms. By a "table access" we mean any access to an indexed array (including the pattern itself). We do not pay attention to the sizes of the tables, since all are of size $O(m + |\Sigma|)$ and very small in practice. In all cases the tables fit even in very small caches, so their sizes should not affect the relative performance of the algorithms.

As can be seen, BDM and Boyer-Moore pay their few accesses to the text with a high number of table accesses. In this respect, the BNDM algorithm and its variants are by far superior. The relative performances between our three algorithms remain unchanged when considering table accesses. About the linear time algorithms, we see that KMP also pays a high price for its guaranteed linear time, being worse than the naive algorithm in terms of table accesses. For non binary alphabets we do not show KMP anymore. It is stabilized around 3 table accesses per text character.

For higher alphabet sizes, BDM gets closer to the BNDM family, while Boyer-Moore stays far away, even farther than Horspool and Sunday.

The third part of the cost is given by the number of register accesses. We understand that every non-indexed variable is stored in a register, which is realistic in modern architectures with many registers and for our algorithms that normally have a few important variables. We observe that, despite that the register accesses are much cheaper than the previous operations considered, the number of accesses is an order of magnitude higher, so they should have an effect on the performance.

Only here we see the price paid by BM_BNDM and TurboBNDM. These more complex algorithms inspect (slightly) less text characters and table cells than the simple BNDM, but they pay this with much more accesses to register variables. BDM and Boyer-Moore also stay far away from BNDM. On the other hand, KMP and the naive algorithm are much more expensive than Shift-Or, KMP stabilizing at a higher cost for larger alphabets (they are not shown in all the plots, but they

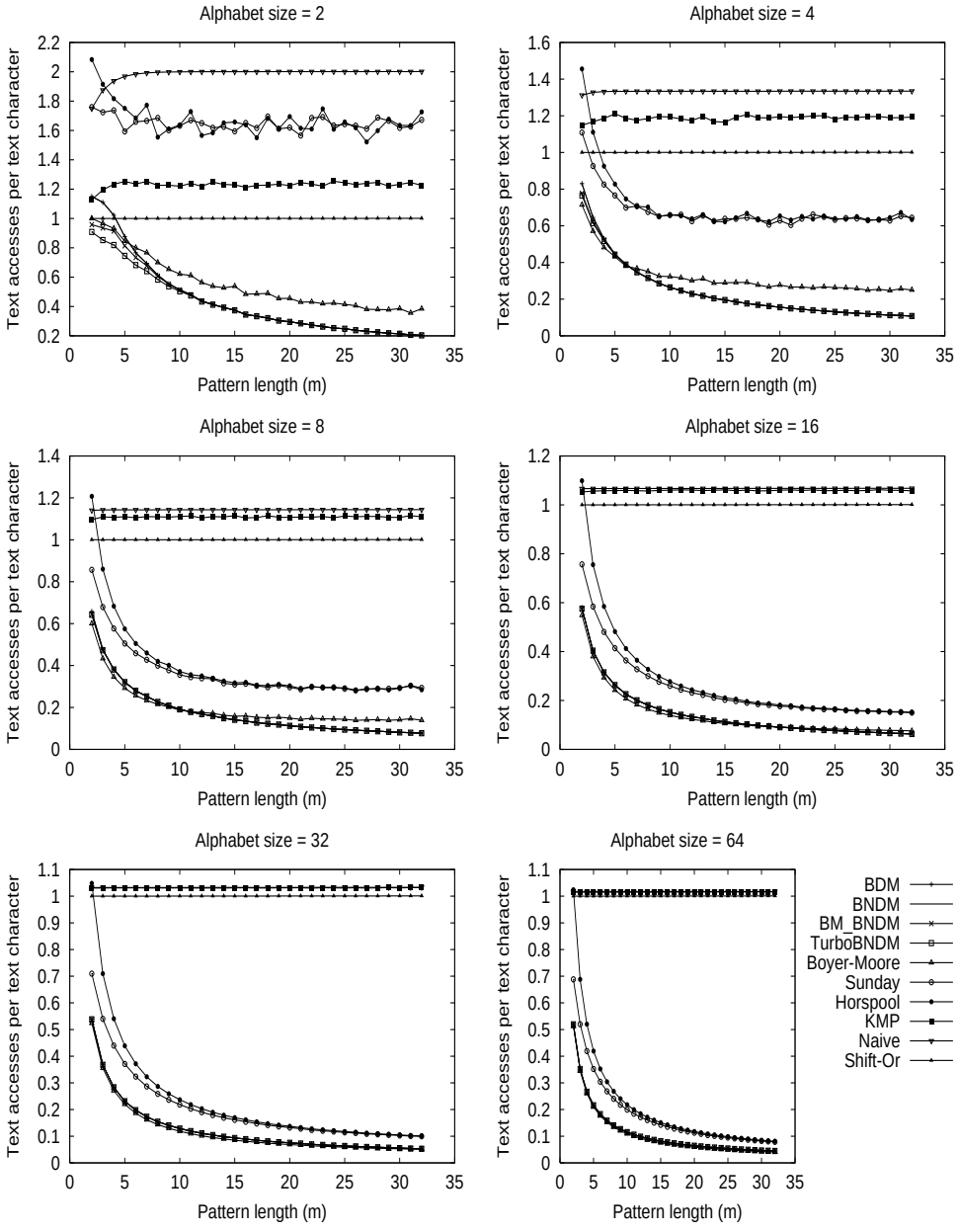

Fig. 11. Fraction of text characters inspected, for random text and increasing pattern length, on different alphabet sizes.

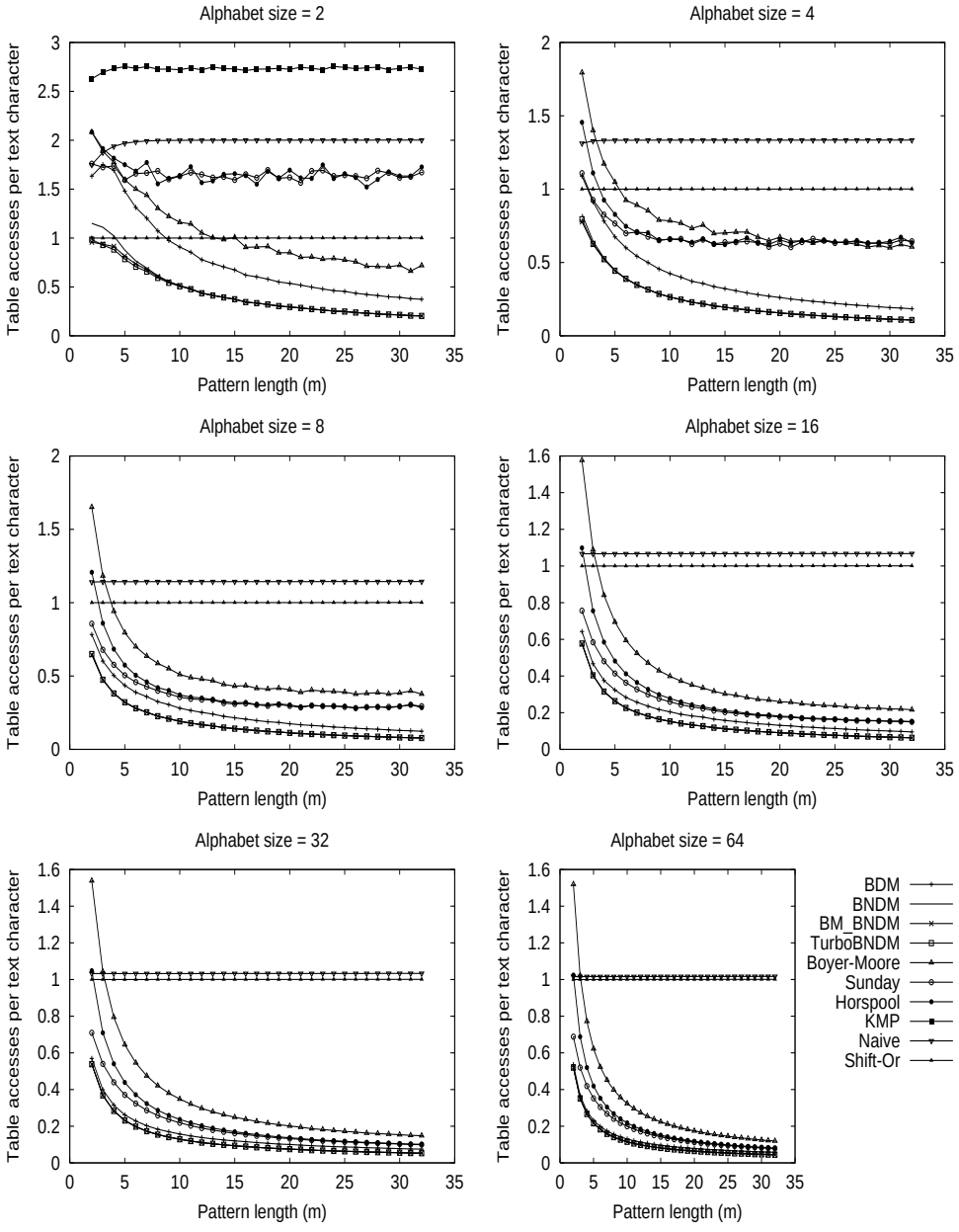

Fig. 12. Number of table accesses per text character, for random text and increasing pattern length, on different alphabet sizes.

stay about the same as for $|\Sigma| = 4$).

For higher alphabets, we see that BDM gets closer to BNDM (but it is always worse than BM_BNDM and TurboBNDM), while Boyer-Moore stays definitely more expensive. A much more interesting effect is achieved by Sunday, which progressively gets better than BNDM as the alphabet size grows.

As we have seen, BNDM has important algorithmic advantages over its competitors. It inspects far less text characters than Horspool and Sunday, it pays far less table accesses than Boyer-Moore and BDM, and it pays far less register accesses than BM_BNDM and TurboBNDM. Hence, BNDM provides the best combination when all the costs are considered together.

In which follows we see see how these algorithmic advantages map into real improvements in the CPU time. There are many reasons that make this mapping nontrivial to predict. For example, different machines will have different access costs in their memory hierarchy. But more important, the pipelining mechanism of the processor may permit performing some operations in parallel (e.g. a memory fetch can be done in parallel with some register accesses), so it is not just a matter of counting accesses multiplied by their relative costs.

9.2 Exact Matching

We consider real CPU times from now on. We included in this comparison all the algorithms of the previous experiments. To make the plots more readable, we removed the least interesting algorithms: Horspool is discarded because it is strictly worse than Sunday and also slower in practice, the naive algorithm is always slower than Shift-Or (4–10 milliseconds per megabyte), KMP is even slower (11–14 milliseconds per megabyte) and, in some cases, Shift-Or (always around 4 milliseconds per megabyte) is outside the range of interesting values.

Figure 14 shows the results for random text. For small alphabet sizes (up to 4) BNDM is the fastest algorithm, provided the pattern is not too short. In particular, simple BNDM is slightly faster than the BM_BNDM and TurboBNDM variations because of the number of register accesses.

BNDM is especially good for small alphabets since it uses more information than others on the matched text, and pays less table and register accesses to do it. BDM and Boyer-Moore also use enough information on the matched text, but they pay more processing time. Sunday pays little processing but it accesses more text characters.

As the alphabet size grows, the differences in terms of text accesses with the Sunday algorithm start to blur, and Sunday starts to dominate for short patterns. The area where BNDM is the fastest starts to shrink, totally disappearing for $|\Sigma| \geq 32$. Only Sunday beats BNDM, never by more than 20%.

We do not show the results for longer patterns but comment the main result: BNDM ceases to improve for patterns longer than $w = 32$ letters, so BDM eventually becomes faster. This cut point is close to $m = 50$.

Figure 15 shows the results on non-random text: English and DNA. The results are very similar to random text for $|\Sigma| = 16$ and $|\Sigma| = 4$, respectively. That is, BNDM is reasonably competitive on English and the fastest for DNA. On French and Spanish texts we obtained results similar as on English.

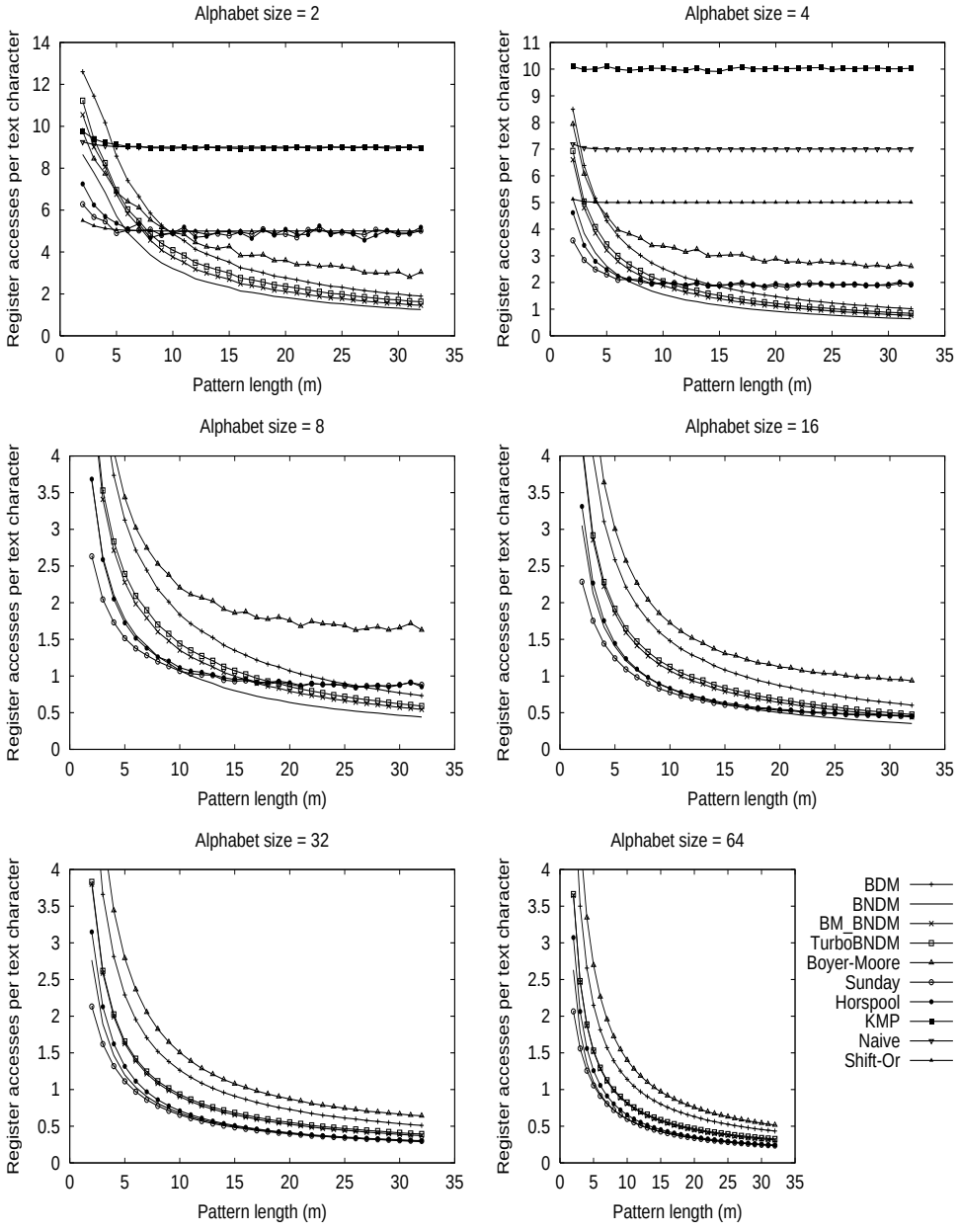

Fig. 13. Number of register accesses per text character, for random text and increasing pattern length, on different alphabet sizes.

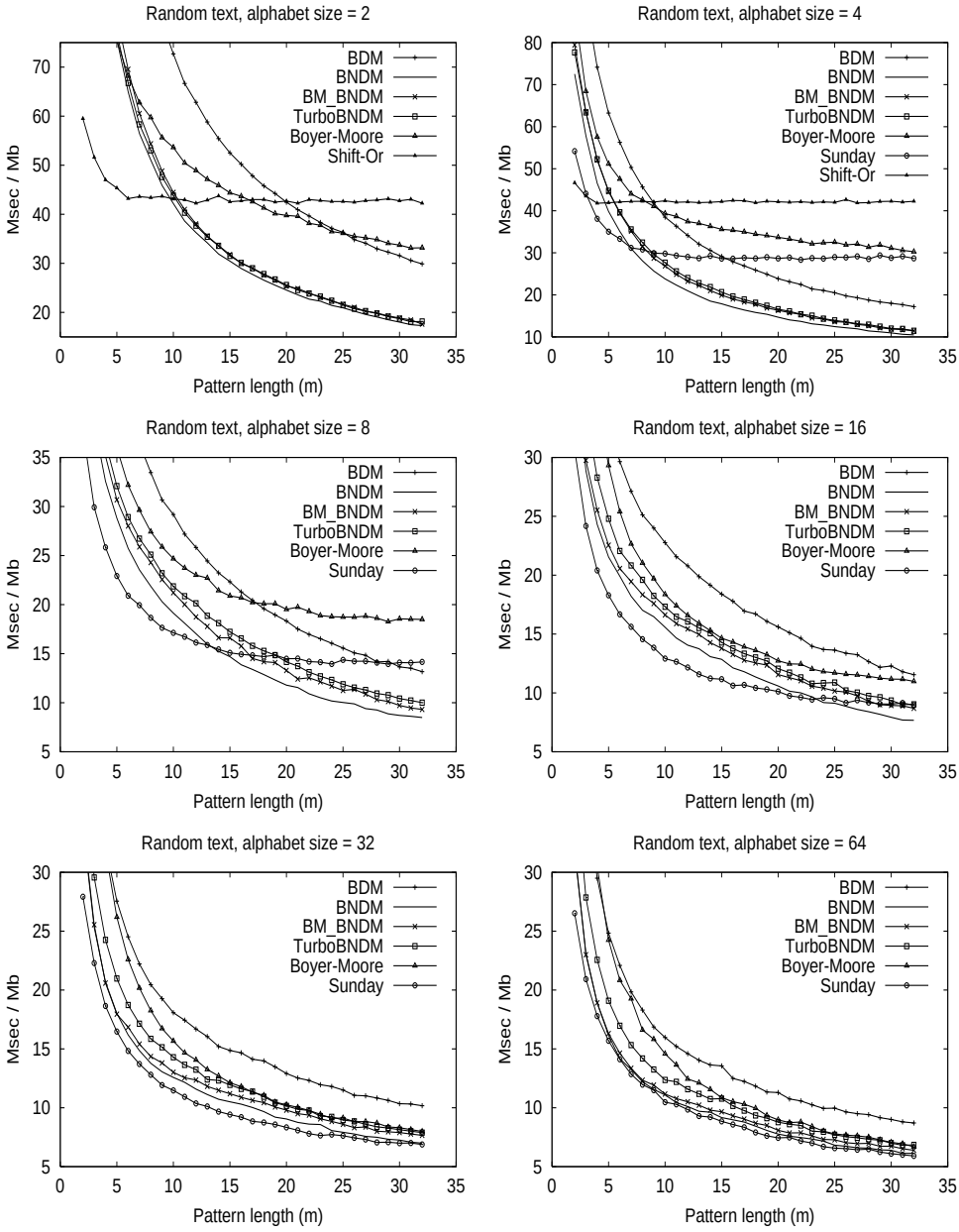

Fig. 14. Times in milliseconds per megabyte, for random text and increasing pattern length, on different alphabet sizes.

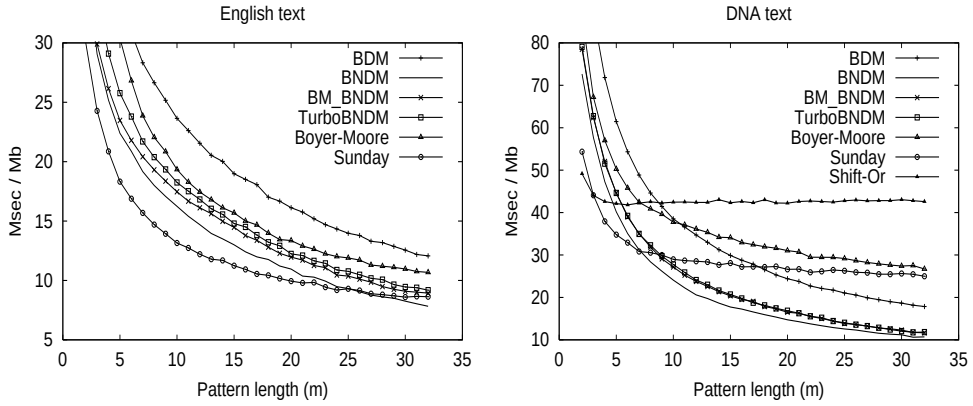

Fig. 15. Times in milliseconds per megabyte, for non-random text and increasing pattern length.

9.3 Classes of Characters

We show some illustrative results using classes of characters, which were generated as follows: we generated random texts of alphabet sizes $|\Sigma| = 4, 16$ and 64, inside which we searched random patterns of length 15 (resp. 30). In those patterns we introduced from 1 to 7 (resp. 1 to 15) *don't cares* randomly placed. By a *don't care* we mean a class of characters that matches all the alphabet. The results are shown in Figure 16. Our algorithm is the fastest in all cases, far below Shift-Or (which stays almost constant whatever the number of *don't cares* is), Sunday and Boyer-Moore extended to classes of characters[4]. As the length of the patterns grows, the difference between our algorithm and the others increases sharply.

9.4 Multipattern Search

We present in Figure 17 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 separately. We take random groups of five patterns of length 6 and show how our multipattern algorithm (Multi-BNDM, in its first and second versions) performs against five sequential searches with our sequential algorithm (BNDM), and against the parallel version proposed in [Baeza-Yates and Gonnet 1992] (Multi-Shift-Or).

As it can be seen, our second arrangement is slightly more efficient than the first one, both 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 both more efficient than the proposal of [Baeza-Yates and Gonnet 1992] provided $|\Sigma| \geq 8$.

9.5 Searching Allowing Substitutions

We show now the performance of our approximate string matching algorithm for Hamming distance. Figure 18 shows the results for $m = 10$. We show random

[4]These extensions consist simply in redefining the equality among characters when a *don't care* is involved.

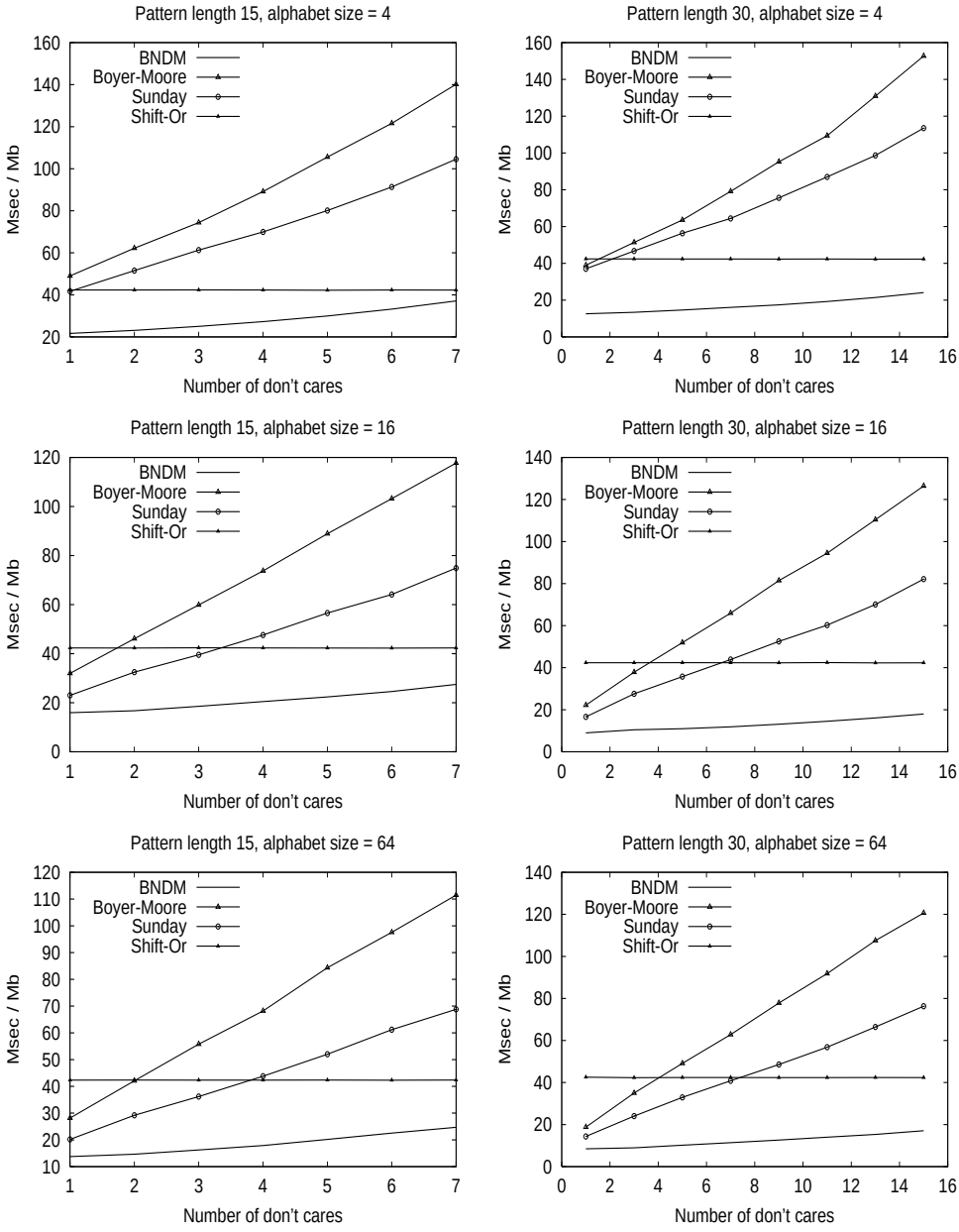

Fig. 16. Times to search with classes of characters, in milliseconds per megabyte, for random text and increasing number of *don't cares*, on different alphabet sizes.

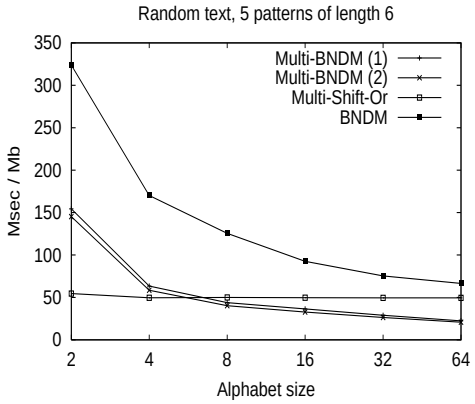

Fig. 17. Times in milliseconds per megabyte, for multipattern search on random text of different alphabet sizes.

text with $|\Sigma| = 8$ as well as English text. We included in the comparison all the algorithms we are aware of: Shift-Add [Baeza-Yates and Gonnet 1994; Wu and Manber 1992], BY-filter [Baeza-Yates and Gonnet 1994], EMC-filter [El-Mabrouk and Crochemore 1996], TU-filter [Tarhio and Ukkonen 1993], and Counting [Baeza-Yates and Gonnet 1994]. We also included some algorithms that were designed for edit distance and that we adapted for this simpler case: NFA [Baeza-Yates and Navarro 1999], Part.Ex. [Navarro and Baeza-Yates 1999] and DFA [Navarro 1997]. Our algorithm is called simply BNDM in the plots, and we include the Naive algorithm as well (the trivial extension of the naive exact string matching algorithm).

In this case our algorithm is the fastest for moderate error levels (i.e. $k \leq 3$). The same happens for $4 \leq |\Sigma| \leq 16$ and pattern lengths between 10 and 16. It is interesting to notice that for Hamming distance our algorithm beats exact partitioning [Navarro and Baeza-Yates 1999], which is the fastest known algorithm for edit distance.

In the areas where exact partitioning is faster, our algorithm is still reasonably competitive. Moreover, we can efficiently handle classes of characters, while exact partitioning quickly degrades if it uses the Sunday search algorithm. On the other hand, exact partitioning can be made more resistant to errors by using our extension of BNDM to multipattern search.

9.6 Searching Allowing Errors

We show now the performance of our extensions to deal with errors. We first show how our multipattern algorithm performs when used for approximate string matching. This algorithm is called Ex.Part./BNDM. We include also the fastest known algorithms in the comparison: Ex.Part./Sunday is the same algorithm except that Sunday is used for the multipattern search [Navarro and Baeza-Yates 1999] (this is the fastest known algorithm for low error levels); Ex.Part./Shift-Or is the same using Shift-Or for the multipattern search [Wu and Manber 1992]; Bit.Par.NFA [Baeza-Yates and Navarro 1999] and Bit.Par.Matrix [Myers 1999] are bit-parallel

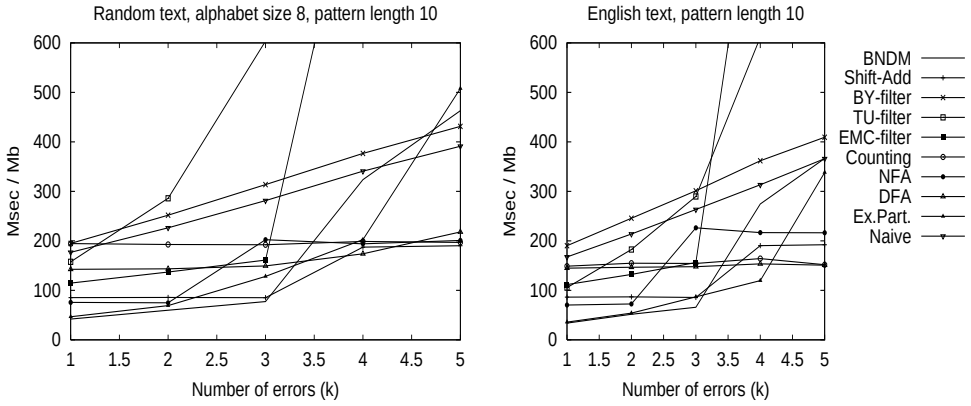

Fig. 18. Times in milliseconds per megabyte, for approximate search under Hamming distance on random and English text. We use $m = 10$ and the x axis is the number of errors allowed.

algorithms; and finally we include algorithms based on Counting [Jokinen et al. 1996], DFA [Navarro 1997] and 4-Russians [Wu et al. 1996].

Figure 19 shows the results for two alphabet sizes and $m = 20$ (we obtained similar results for $m = 10$ and 30). As it can be seen, our implementation of exact partitioning is quite close to Ex.Part./Sunday (sometimes even faster) and therefore our algorithm is a competitive yet more flexible replacement, while it is faster than the other flexible candidate Ex.Part./Shift-Or [Wu and Manber 1992].

Since BNDM is not very good for very short patterns, our algorithm works better for $m = 20$ and 30. Moreover, it ceases to be competitive for higher error levels since the length of the patterns to search for is $O(m/k)$.

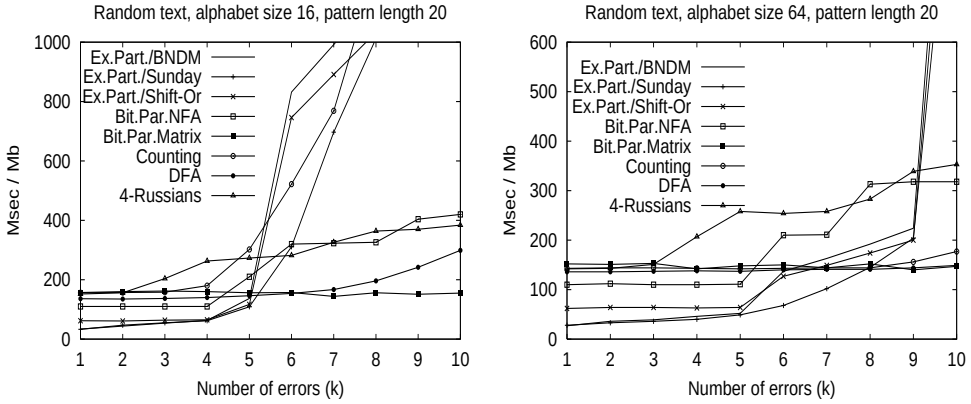

Fig. 19. Times in milliseconds per megabyte, for random text on patterns of length 20, and $|\Sigma| = 16$ and 64, using edit distance. The x axis is the number of errors allowed.

Finally, we show the performance of our new algorithm for approximate string matching based on the NFA simulation (NFA/BNDM). Figure 20 shows the results.

As the algorithm works well for very low error levels, we show only the case $k = 1$, for random ($|\Sigma| = 4$) and English text. In the first case (very similar to DNA) our algorithm outperforms all the others (this happens also for $k = 2$ and $k = 3$). For English text, it can be seen that for very low error levels and intermediate pattern lengths, our algorithm becomes very close to [Navarro and Baeza-Yates 1999], which is the fastest known algorithm for low error levels, beating all the other algorithms (we have shown only those that are the fastest for these cases).

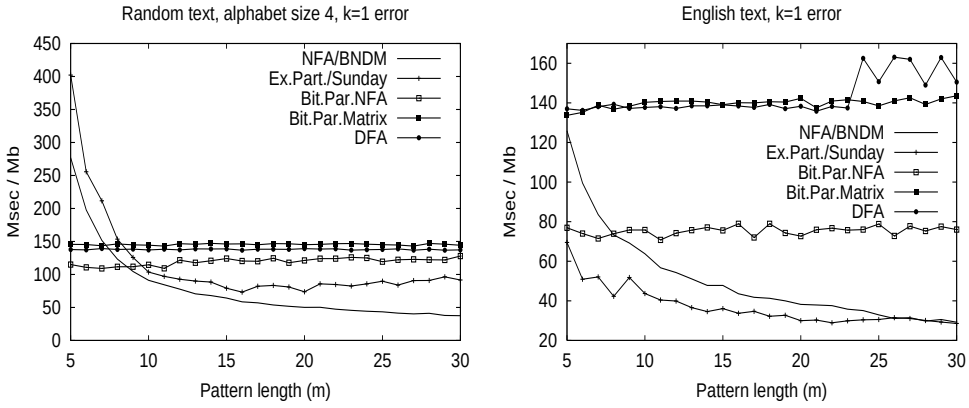

Fig. 20. Times in milliseconds per megabyte, for random and English and $k = 1$ error under edit distance.

10. CONCLUSIONS AND FUTURE WORK

We have presented a new text searching algorithm called BNDM, which is 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. Bit-parallelism is a general way to simulate nondeterministic automata using the bits of the computer word, which has up to now led to flexible but slow algorithms for exact searching and to competitive algorithms for approximate searching. Hence, BNDM obtains the best of both worlds: the speed of BDM and the flexibility of bit-parallelism.

We present also some variations called TurboBNDM 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. TurboBNDM has an average performance very close to BNDM and $O(n)$ worst case. We have also extended BNDM in simple ways to solve a large set of extensions over the basic string matching problem, such as matching classes of characters, multiple pattern matching and approximate pattern matching.

Our new algorithm is experimentally shown to be very fast in practice. For exact patterns it is the fastest on small alphabets and remains competitive for larger ones. It is also competitive when dealing with extended patterns, being in particular the fastest to handle classes of characters and some cases of approximate pattern

matching. In particular, BNDM seems to be a very attractive choice in computational biology applications, since it is the fastest to search DNA text for patterns of lengths between 7 and 50, and its flexibility is necessary in many applications of that field.

Recently, BNDM has been successfully extended to deal with other extensions of the search problem, such as regular expression searching [Navarro and Raffinot 1999] and PROSITE patterns for protein searching (i.e. permitting classes of characters and bounded length gaps) [Navarro and Raffinot 2000]. A software called *nrgrep* [Navarro 2000b] able of exact and approximate searching of patterns containing classes of characters, optional and repeatable characters and regular expressions, has been built based on the ideas presented in this paper.

The new suffix automaton we introduce and simulate for classes of characters has never been studied. A study of this new automaton (maximal number of nodes and edges, minimality, algorithms to build it, average number of nodes and edges) would be interesting by itself and should permit us to extend the BDM and TurboRF to handle classes of characters.

REFERENCES

Abrahamson, K. 1987. Generalized string matching. *SIAM Journal on Computing 16*, 6, 1039–1051.

Baeza-Yates, R. 1992. Text retrieval: Theory and practice. In *12th IFIP World Computer Congress*, Volume I (Sept. 1992), pp. 465–476. Elsevier Science.

Baeza-Yates, R. and Gonnet, G. 1992. A new approach to text searching. *CACM 35*, 10 (Oct.), 74–82.

Baeza-Yates, R. and Gonnet, G. 1994. Fast string matching with mismatches. *Information and Computation 108*, 2, 187–199.

Baeza-Yates, R. and Navarro, G. 1999. Faster approximate string matching. *Algorithmica 23*, 2, 127–158.

Baeza-Yates, R. and Perleberg, C. 1992. Fast and practical approximate pattern matching. In *Proc. CPM'92* (1992), pp. 185–192. Springer-Verlag. LNCS 644.

Blumer, A., Ehrenfeucht, A., and Haussler, D. 1989. Average sizes of suffix trees and dawgs. *Discrete Applied Mathematics 24*, 1, 37–45.

Boyer, R. S. and Moore, J. S. 1977. A fast string searching algorithm. *Commun. ACM 20*, 10, 762–772.

Crochemore, M. 1986. Transducers and repetitions. *Theoretical Computer Science 45*, 1, 63–86.

Crochemore, M., Czumaj, A., Gasieniec, L., Jarominek, S., Lecroq, T., Plandowski, W., and Rytter, W. 1993. Fast practical multi-pattern matching. Rapport 93–3, Institut Gaspard Monge, Université de Marne la Vallée.

Crochemore, M. and Rytter, W. 1994. *Text algorithms*. Oxford University Press.

Czumaj, A., Crochemore, M., Gasieniec, L., Jarominek, S., Lecroq, T., Plandowski, W., and Rytter, W. 1994. Speeding up two string-matching algorithms. *Algorithmica 12*, 247–267.

El-Mabrouk, N. and Crochemore, M. 1996. Boyer-Moore strategy to efficient approximate string matching. In *Proc. of CPM'96*, Number 1075 in Lecture Notes in Computer Science (1996), pp. 24–38. Springer-Verlag, Berlin.

Fischer, M. J. and Paterson, M. 1974. String matching and other products. In R. M. Karp Ed., *Proceedings SIAM-AMS Complexity of Computation* (Providence, RI, 1974), pp. 113–125.

Horspool, R. N. 1980. Practical fast searching in strings. *Software Practice and Experience 10*, 501–506.

JOKINEN, P., TARHIO, J., AND UKKONEN, E. 1996. A comparison of approximate string matching algorithms. *Software Practice and Experience 26*, 12, 1439–1458.

KNUTH, D. E., MORRIS, J. H., AND PRATT, V. R. 1977. Fast pattern matching in strings. *SIAM Journal on Computing 6*, 1, 323–350.

LECROQ, T. 1992. *Recherches de mot.* Ph. D. thesis, Université d'Orléans, France.

MYERS, G. 1999. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM 46*, 3, 395–415.

NAVARRO, G. 1997. A partial deterministic automaton for approximate string matching. In *Proc. of WSP'97* (1997), pp. 112–124. Carleton University Press.

NAVARRO, G. 1998. *Approximate Text Searching.* Ph. D. thesis, Department of Computer Science, University of Chile. Tech. Report TR/DCC-98-14. `ftp://ftp.dcc.uchile.cl/-pub/users/gnavarro/thesis98.ps.gz`.

NAVARRO, G. 2000a. A guided tour to approximate string matching. *ACM Computing Surveys*. To appear.

NAVARRO, G. 2000b. Nrgrep: a fast and flexible pattern matching tool. Technical Report TR/DCC-2000-3 (Aug.), Dept. of Computer Science, Univ. of Chile. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/nrgrep.ps.gz`.

NAVARRO, G. AND BAEZA-YATES, R. 1999. Very fast and simple approximate string matching. *Information Processing Letters 72*, 65–70.

NAVARRO, G. AND RAFFINOT, M. 1998. A bit-parallel approach to suffix automata: Fast extended string matching. In *Proc. CPM'98*, LNCS v. 1448 (1998), pp. 14–33. Springer-Verlag.

NAVARRO, G. AND RAFFINOT, M. 1999. Fast regular expression search. In J. VITTER AND C. ZAROLIAGIS Eds., *Proceedings of the 3rd Workshop on Algorithm Engineering*, Number 1668 in Lecture Notes in Computer Science (1999), pp. 199–213. Springer-Verlag, Berlin.

NAVARRO, G. AND RAFFINOT, M. 2000. Fast and simple character classes and bounded gaps pattern matching, with application to protein searching. In *ACM RECOMB 2001* (2000). To appear.

PINTER, R. Y. 1985. Efficient string matching with don't care pattern. In A. APOSTOLICO AND Z. GALIL Eds., *Combinatorial Algorithms on Words*, pp. 11–29. Springer-Verlag, Berlin.

RAFFINOT, M. 1997a. Asymptotic estimation of the average number of terminal states in dawgs. In *Proc. WSP'97* (1997), pp. 140–148. Carleton University Press.

RAFFINOT, M. 1997b. On the multi backward dawg matching algorithm (MultiBDM). In *Proc. WSP'97* (1997), pp. 149–165. Carleton University Press.

SUNDAY, D. 1990. A very fast substring search algorithm. *CACM 33*, 8 (Aug.), 132–142.

TARHIO, J. AND UKKONEN, E. 1993. Approximate Boyer-Moore string matching. *SIAM J. on Computing 22*, 2, 243–260.

WU, S. AND MANBER, U. 1992. Fast text searching allowing errors. *CACM 35*, 10 (Oct.), 83–91.

WU, S., MANBER, U., AND MYERS, E. 1996. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica 15*, 1, 50–67.

YAO, A. 1979. The complexity of pattern matching for a random string. *SIAM J. on Computing 8*, 368–387.