Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching*

Gonzalo Navarro† Mathieu Raffinot‡

Abstract

The problem of fast exact and approximate searching for a pattern that contains Classes of characters and Bounded size Gaps (CBG) in a text has a wide range of applications, among which a very important one is protein pattern matching (for instance, one PROSITE protein site is associated with the CBG $[RK]-x(2,3)-[DE]-x(2,3)-Y$, where the brackets match any of the letters inside, and $x(2,3)$ a gap of length between 2 and 3). Currently, the only way to search for a CBG in a text is to convert it into a full regular expression (RE). However, a RE is more sophisticated than a CBG, and searching for it with a RE pattern matching algorithm complicates the search and makes it slow. This is the reason why we design in this article two new practical CBG matching algorithms that are much simpler and faster than all the RE search techniques. The first one looks exactly once at each text character. The second one does not need to consider all the text characters and hence it is usually faster than the first one, but in bad cases may have to read the same text character more than once. We then propose a criterion based on the form of the CBG to choose *a-priori* the fastest between both. We also show how to search permitting a few mistakes in the occurrences. We performed many practical experiments using the PROSITE database, and all them show that our algorithms are the fastest in virtually all cases.

1 Introduction

This paper deals with the problem of fast searching of patterns that contain Classes of characters and Bounded size Gaps (CBG) in texts. This problem occurs in various fields, like information retrieval, data mining and computational biology. We are particularly interested in the latter one.

In computational biology, this problem has many applications, among which the most important is protein matching. These last few years, huge protein site pattern databases have been developed, like PROSITE [7, 13]. These databases are collections of protein site descriptions. For each protein site, the database contains diverse information, notably the *pattern*. This is an expression formed with classes of characters and bounded size gaps on the amino acid alphabet (of size 20). This pattern is used to search for a possible occurrence of this protein in a longer one. For example, the protein site number PS00007 has as its pattern

*Partially supported by ECOS-Sud project C99E04.

†Dept. of Computer Science, University of Chile. Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl. Partially supported by Fondecyt grant 1-020831.

‡CNRS - Laboratoire Génome et Informatique, Tour Evry 2, 523, Place des Terrasses de l'Agora, 91034 Evry, France. raffinot@genopole.cnrs.fr

the expression $[RK] - x(2,3) - [DE] - x(2,3) - Y$, where the brackets mean that the position can match any of the letters inside, and $x(2,3)$ means a gap of length between 2 and 3.

Currently, these patterns are considered as full regular expressions (REs) over a fixed alphabet Σ, *i.e* generalized patterns composed of (i) basic characters of the alphabet (adding the empty word ε and also a special symbol x that can match all the letters of Σ), (ii) concatenation (denoted $\cdot$), (ii) union ($|$) and (iii) Kleene closure ($*$). This latter operation $\mathcal{L}^*$ on a set of words $\mathcal{L}$ means that we accept all the words made by a concatenation of words of $\mathcal{L}$. For instance, our previous pattern can be considered as the regular expression $(R|K) \cdot x \cdot x \cdot (x|\varepsilon) \cdot (D|E) \cdot x \cdot x \cdot (x|\varepsilon) \cdot Y$. We note $|RE|$ the length of a RE, that is the number of symbols in it. The search is done with the classical algorithms for RE searching, that are however quite complicated. The RE needs to be converted into an automaton and then searched in the text. It can be converted into a deterministic automaton (DFA) in worst case time $O(2^{|RE|})$, and then the search is linear in the size n of the text, giving a total complexity of $O(2^{|RE|} + n)$. It can also be converted into a nondeterministic automaton (NFA) in linear time $O(|RE|)$ and then searched in the text in $O(n \times |RE|)$ time, giving a total of $O(n \times |RE|)$ time. We give a review of these methods in Section 3. The majority of the PROSITE matching softwares use these techniques [16, 30].

None of the presented techniques are fully adequate for CBGs. First, the algorithms are intrinsically complicated to understand and to implement. Second, all the techniques perform poorly for certain types of REs. The "difficult" REs are in general those whose DFAs are very large, a very common case when translating CBGs to REs. Third, especially with regard to the sizes of the DFAs, the simplicity of CBGs is not translated into their corresponding REs. At the very least, resorting to REs implies solving a simple problem by converting it into a more complicated one. Indeed, the experimental time results when applied to our CBG expressions are far from reasonable in regard of the simplicity of CBGs and compared to the search for expressions that just contain classes of characters [26].

This is the motivation of this paper. We present two new simple algorithms to search for CBGs in a text, that are also experimentally much faster than all the previous ones. These algorithms make plenty use of "bit-parallelism", that consists in using the intrinsic parallelism of the bit manipulations inside computer words to perform many operations in parallel. Competitive algorithms have been obtained using bit parallelism for exact string matching [2, 34], approximate string matching [2, 34, 35, 3, 22], and REs matching [18, 33, 25]. 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.

We performed two different types of experiments, comparing our algorithms against the fastest known ones for RE searching. We use as CBGs the patterns of the PROSITE database. We first compared them as "pure pattern matching", i.e. searching for the CBGs in a compilation of 6 megabytes of protein sequences (from the TIGR Microbial database). We then compared them as "library matching", that is search for a large set of PROSITE patterns in a protein sequence of 300 amino acids. Our algorithms are by far the fastest in both cases. Moreover, in the second case, the search time improvements are dramatic, as our algorithms are about 100 times faster than the best RE matching algorithms when pattern preprocessing times become important.

An extended abstract of this paper has already been published in [27], without all the details and without searching with differences.

We use the following definitions throughout the paper. Σ is the alphabet, a word on Σ is

a finite sequence of characters of Σ. Σ^* means the set of all the words build on Σ. A word $w \in \Sigma^*$ is a *factor* (or substring) of $p \in \Sigma^*$ if p can be written $p = uwv$, $u, v \in \Sigma^*$. A factor w of p is called a *suffix* of p is $p = uw$, $u \in \Sigma^*$, and a *prefix* of p is $p = wu$, $u \in \Sigma^*$.

We note with brackets a subset of elements of Σ: $[ART]$ means the subset $\{A, R, T\}$ (a single letter can be expressed in this way too). We add the special symbol x to denote a subset that corresponds to the whole alphabet. We also add a symbol $x(a, b)$, $a < b$, for a bounded size gap of minimal length a and maximal b, and use $x(a)$ as a short for $x(a, a)$ (so $x = x(1) = x(1, 1)$). A CBG on Σ is formally a finite sequence of symbols that can be (i) brackets, (ii) x and (iii) bounded size gaps $x(a, b)$. We define m as the total number of such symbols in a CBG.

We use the notation $T = t_1 t_2 \dots t_n$ for the text of n characters of Σ in which we are searching for the CBGs. A CBG matches T at position j if there is an alignment of $t_{j-i} \dots t_j$ with the CBG, considering that (i) a bracket matches with any text letter that appears inside brackets; (ii) an x matches any text letter; and (iii) a bounded gap $x(a, b)$ matches at minimum a and at maximum b arbitrary characters of T. We denote by ℓ the minimum size of a possible alignment and L the size of a maximum one. For example, $[RK] - x(2, 3) - [DE] - x(2, 3) - Y$ (where $\ell = 7$ and $L = 9$) matches the text $T = AHLRKDEDATY$ at position 11 by 3 different alignments (see Figure 1).

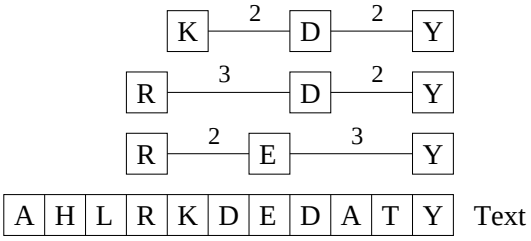


Figure 1: Three different alignments of the CBG $[RK] - x(2, 3) - [DE] - x(2, 3) - Y$ over the text $T = AHLRKDEDATY$ at the same ending position.

DEFINITION **1** *Searching for a CBG in a text $T = t_1 t_2 \dots t_n$ consists in finding all the positions j of T in which there is an alignment of the CBG with a suffix of $t_1 \dots t_j$.*

This paper is organized as follows. We begin in Section 2 by summarizing the two main bit-parallel approaches that lead to fast efficient matching algorithms for simple strings but also for patterns that contain classes of characters. In Section 3, we explain in detail what are the approaches to search for full REs. We then present in Section 4 our new algorithm (which we call a "forward algorithm"), that reads all the characters of the text exactly once. It is based on a new automaton representation and simulation. We present in Section 5 another algorithm (which we call a "backward algorithm" despite that it processes the text basically left to right), that allows us to skip some characters of the text, being generally faster. However, it can not been used for all types of CBGs, and it is sometimes slower than the forward one. Consequently, we give in the next Section 6 a good experimental criterion that enables us to choose *a-priori* the fastest, depending on the form of the CBG. Section 7 is devoted to the experimental results for both algorithms compared to the fastest RE searching algorithms. Section 8 deals with several extensions of the algorithm. Section 9 considers the possibility of permitting a few differences between the occurrences and the patterns specification. Section 10 gives our conclusions.

2 Bit-Parallelism

In [2], a new approach to text searching was proposed. It is based on *bit-parallelism* [1]. This technique consists 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 significative in practice.

Figure 2 shows a non-deterministic automaton that searches for a pattern in a text. Classical pattern matching algorithms, such as KMP [17], convert this automaton to deterministic form and achieve $O(n)$ search time. The Shift-Or algorithm [2], on the other hand, uses bit-parallelism to simulate the automaton in its nondeterministic form. It achieves $O(mn/w)$ worst-case time, i.e., an optimal speedup over the classical $O(mn)$ simulation. For $m \leq w$, Shift-Or is twice as fast as KMP because of better use of computer registers. Moreover, it is easily extended to handle classes of characters.

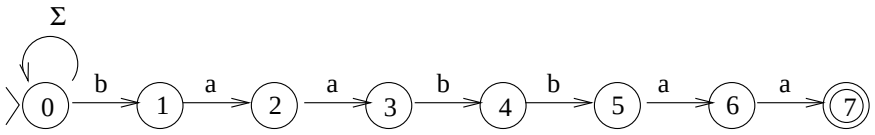


Figure 2: A nondeterministic automaton to search for the pattern $p = baabbaa$ in a text. The initial state is 0.

We use some notation to describe the operations on bits. We use exponentiation to denote bit repetition, e.g. $0^3 1 = 0001$. We denote as $b_\ell \ldots b_1$ the bits of a mask of length ℓ, which is stored somewhere inside the computer word of length w. We use C-like syntax for operations on the bits of computer words, i.e. "|" is the bitwise-or, "&" is the bitwise-and, "$\sim$" complements all the bits, and "<<" moves the bits to the left and enters zeros from the right, e.g. $b_\ell b_{\ell-1} \ldots b_2 b_1 << 3 = b_{\ell-3} \ldots b_2 b_1 000$. We can also perform arithmetic operations on the bits, such as addition and subtraction, which operate the bits as if they formed a number, for instance $b_\ell \ldots b_x 10000 - 1 = b_\ell \ldots b_x 01111$.

We explain now the basic algorithm and then a later improvement over it.

2.1 Forward scanning

We present now the Shift-And algorithm, which is an easier-to-explain (though a little less efficient) variant of Shift-Or. The algorithm builds first a table B which for each character stores a bit mask $b_m \ldots b_1$. The mask in $B[c]$ has the i-th bit set if and only if $p_i = c$. The state of the search is kept in a machine word $D = d_m \ldots d_1$, where d_i is set whenever $p_1 p_2 \ldots p_i$ 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 2 is active). Therefore, we report a match whenever d_m is set.

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

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

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, $T_{j-i+1} \dots T_j = p_1 \dots p_i$ if and only if $T_{j-i+1} \dots T_{j-1} = p_1 \dots p_{i-1}$ 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 "$| \; 0^{m-1}1$" after the shift allows a match to begin at the current text position (this operation is saved in the Shift-Or, where all the bits are complemented). This corresponds to the self-loop at the beginning of the automaton.

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 $O(n)$.

2.2 Classes of characters

The Shift-Or algorithm is not only very simple, but it also has some further advantages. The most immediate one is that it is very easy to extend it to handle classes of characters. That is, each pattern position does not only match a single character but a set of characters. If C_i is the set of characters that match the position i in the pattern, we set the i-th bit of $B[c]$ for all $c \in C_i$. In [2] they show also how to allow a limited number k of mismatches in the occurrences, at $O(nm \log(k)/w)$ cost.

This paradigm was later enhanced [34] to support extended patterns, which allow wild cards, regular expressions, approximate search with nonuniform costs, and combinations. Further development of the bit-parallelism approach for approximate string matching lead to some of the fastest algorithms for short patterns [3, 22]. In most cases, the key idea was to simulate a nondeterministic finite automaton. It is interesting also to mention [11], which searches allowing mismatches by using a combination of bit-parallelism and Boyer-Moore.

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

2.3 Backward scanning

The main disadvantage of Shift-Or is its inability to skip characters, which makes it slower than the algorithms of the Boyer-Moore [5] or the BDM [10, 9] families. We describe in this section the BNDM pattern matching algorithm [26]. This algorithm, a combination of Shift-Or and BDM, has all the advantages of the bit-parallel forward scan algorithm, and in addition it is able to skip some text characters.

BNDM is based on a *suffix automaton*. A *suffix automaton* on a pattern $P = p_1 p_2 \dots p_m$ is an automaton that recognizes all the suffixes of P. The nondeterministic version of this automaton is shown in Figure 3. Note that the automaton will not run out of active states as long as it has read a factor of P. In the original BDM this automaton is made deterministic. BNDM, instead, simulates the automaton using bit-parallelism. Just as for Shift-And, we keep the state of the search using m bits of a computer word $D = d_m \dots d_1$.

A very important fact is that this automaton can not only be used to recognize the suffixes of P, but also factors of P. Note that there is a path labeled by x from the initial state if and only if x is a factor of P. That is, the nondeterministic automaton will not run out of active states as long as it has read a factor of P.

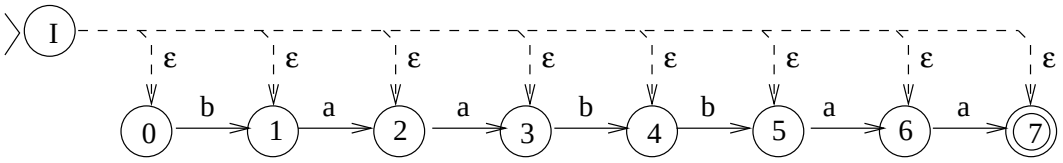


Figure 3: A nondeterministic suffix automaton for the pattern $P = baabbaa$. Dashed lines represent ε-transitions (i.e. they occur without consuming any input).

The suffix automaton is used to design a simple pattern matching algorithm. This algorithm is $O(mn)$ time in the worst case, but optimal on average ($O(n \log_\sigma m/m)$ time). Other more complex variations such as TurboBDM [10] and MultiBDM [9, 29] achieve linear time in the worst case.

To search for 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 backward inside the window for a factor of the pattern P using the suffix automaton, i.e. the suffix automaton of the reverse pattern is fed with the characters in the text window read backward. This backward search ends in two possible forms:

1. We fail to recognize a factor, i.e we reach a window letter σ that makes the automaton run out of active states. This means that the suffix of the window we have read is not anymore a factor of P. Figure 4 illustrates this case. We then shift the window to the right, its starting position corresponding to the position following the letter σ (we cannot miss an occurrence because in that case the suffix automaton would have found a factor of it in the window).

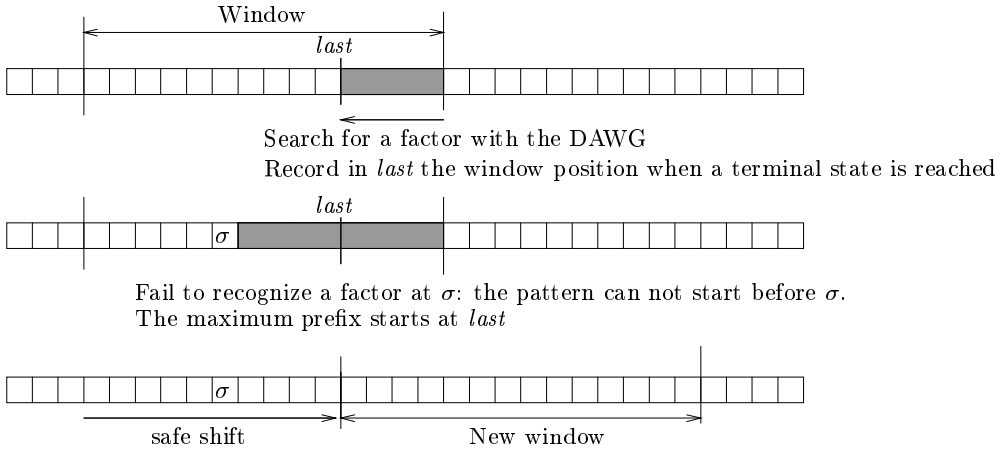


Figure 4: Basic search with the suffix automaton

2. We reach the beginning of the window, therefore recognizing the pattern P since the length-m window is a factor of P (indeed, it is equal to P). We report the occurrence, and shift the window by 1.

The bit-parallel simulation works as follows. Each time we position the window in the

text we initialize $D = 1^m$ and scan the window backward. For each new text character read in the window we update D. If we run out of 1's in D then there cannot be a match and we suspend the scanning and shift the window. If we can perform m iterations then we report the match.

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

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

BNDM is not only faster than Shift-Or and BDM (for $5 \leq m \leq 100$ or so), but it can accommodate all the extensions mentioned. Of particular interest to this work is that it can easily deal with classes of characters by just altering the preprocessing, and it is by far the fastest algorithm to search for this type of patterns [26].

Note that this type of search is called "backward" scanning because the text characters inside the window are read backwards. However, the search progresses from left to right in the text as the window is shifted.

3 Regular expression searching

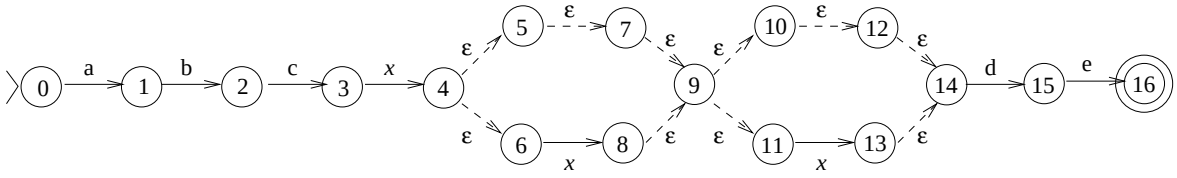
The usual way of dealing with an expression with character classes and bounded gaps is actually to search for it as a full regular expression (RE) [16, 30]. A gap of the form $x(a, b)$ is converted into a letters x followed by $b - a$ subexpressions of the form $(x|\varepsilon)$.

The traditional technique [31] to search for a RE of length $O(m)$ in a text of length n is to convert the expression into a nondeterministic finite automaton (NFA) with $O(m)$ nodes. Then, it is possible to search the text using the automaton at $O(mn)$ worst case time, or to convert the NFA into a deterministic finite automaton (DFA) in worst case time $O(2^m)$ and then scan the text in $O(n)$ time.

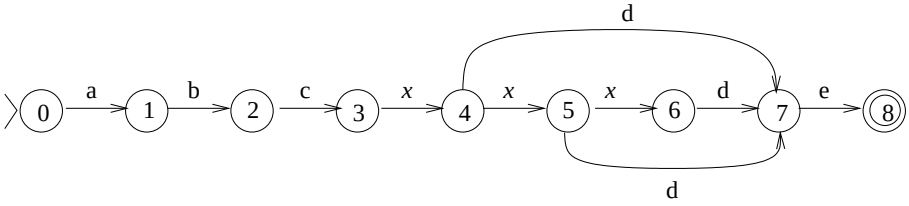
Some techniques have been proposed to obtain a good tradeoff between both extremes. In 1992, Myers [18] presented a four-russians approach which obtains $O(mn/\log n)$ worst-case time and extra space. Other simulation techniques that aim at good tradeoffs based on combinations of DFAs and bit-parallel simulation of NFAs are given in [34, 25].

There exist currently many different techniques to build an NFA from a regular expression R. The most classical one is Thompson's construction [31], which builds an NFA with at most $2m$ states (where m is counted as the number of letters and ε's in the RE). A second one is Glushkov's construction, popularized by Berry and Sethi in [4]. The NFA resulting of this construction has the advantage of having just $m+1$ states (where m is counted as the number of letters in the RE).

A lot of research on Glushkov's construction has been pursued, like [6], where it is shown that the resulting NFA is quadratic in the number of edges in the worst case. In [14], a long time open question about the minimal number of edges of an NFA (without ϵ-transition) with linear number of states was answered, showing a $O(m^2)$ construction with $O(m)$ states and $O(m(\log m)^2)$ edges, as well as a lower bound of $O(m \log m)$ edges. In [12], the construction time was improved to $O(m(\log m)^2)$. Hence, Glushkov construction is not space-optimal. An improvement has been proposed in [15], building a quotient of Glushkov's automaton. Some research has been done also to try to construct directly a DFA from a regular expression, without constructing an NFA, such as [8].



(a) Thompson construction



(b) Glushkov construction

Figure 5: The two classical NFA constructions on our example $a \cdot b \cdot c \cdot x \cdot (x|\varepsilon) \cdot (x|\varepsilon) \cdot d \cdot e$. We recall that x matches the whole alphabet Σ. The Glushkov automaton is ε free, but both present some difficulties to perform an efficient bit-parallelism on them.

We show in Figure 5 the Thompson and Glushkov automata for an example CBG $a - b - c - x(1,3) - d - e$, which we translate into the regular expression $a \cdot b \cdot c \cdot x \cdot (x|\varepsilon) \cdot (x|\varepsilon) \cdot d \cdot e$.

Both Thompson and Glushkov automata present some particular properties. Some algorithms like [18, 34] make use of Thompson's automaton properties and some others, like [25], make use of Glushkov's ones.

Finally, some work has been pursued in skipping characters when searching for a RE. A simple heuristic that has very variable success is implemented in *Gnu Grep*, where they try to find a plain substring inside the RE, so as to use the search for that substring as a filter for the search of the complete RE. In [32] they propose to reduce the search of a RE to a multipattern search for all the possible strings of some length that can match the RE (using a multipattern Boyer-Moore like algorithm). In [25] they propose the use of an automaton that recognizes reversed factors of strings accepted by the RE (in fact a manipulation of the original automaton) using a BNDM-like scheme to search for those factors (see Section 2).

However, none of the presented techniques seems fully adequate for CBGs. First, the algorithms are intrinsically complicated to understand and to implement. Second, all the techniques perform poorly for a certain type of REs. The "difficult" REs are in general those whose DFAs are very large, a very common case when translating CBGs to REs. Third, especially with regard to the sizes of the DFAs, the simplicity of CBGs is not translated into their corresponding REs. For example, the CBG "$[RK] - x(2,3) - [DE] - x(2,3) - Y$" considered in the Introduction yields a DFA which needs about 600 pointers to be represented.

At the very least, resorting to REs implies solving a simple problem by converting it into a more complicated one. Indeed, the experimental time results when applied to our CBG expressions are far from reasonable in regard of the simplicity of CBGs, as seen in Section 7. As we show in that section, CBGs can be searched for much faster by designing specific

algorithms for them. This is what we do in the next sections.

4 A forward search algorithm for CBG patterns

We express the search problem of a pattern with classes of characters and gaps using a non-deterministic automaton. Compared to the automaton for simple patterns (Section 2), this one permits the existence of gaps between consecutive positions, so that each gap has a minimum and a maximum length. The automaton we use does not correspond to any of those obtained with the regular expression simulations (see Section 3), although the functionality is the same.

Figure 6 shows an example for the pattern $a - b - c - x(1,3) - d - e$. Between the letters c and d we have inserted three transitions that can be followed by any letter, which corresponds to the maximum length of the gap. Two ε-transitions leave the state where abc has been recognized and skip one and two subsequent edges, respectively. This allows skipping one to three text characters before finding the cd at the end of the pattern. The initial self-loop allows the match to begin at any text position.

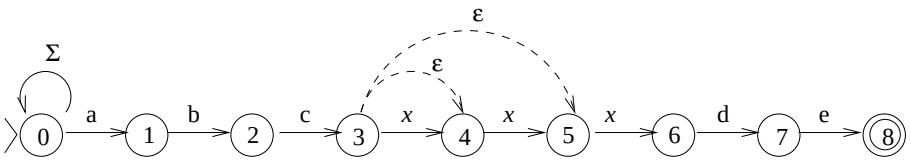


Figure 6: Our non-deterministic automaton for the pattern $a - b - c - x(1,3) - d - e$.

To build the NFA, we start with the initial state S_0 and read the pattern symbol by symbol (a symbol being a class of characters or a gap[1]). We add new automaton edges and states for each new symbol read. If after creating state S_i the next pattern symbol is a class of characters C we create a state S_{i+1} and add an edge labeled C from state S_i to state S_{i+1}. On the other hand, if the new pattern symbol is a gap of the form $x(a,b)$, we create b states $S_{i+1} \ldots S_{i+b}$ and edges labeled Σ linking state S_j to S_{j+1} for $j \in i \ldots i+b-1$. Additionally, we create $b-a$ ε-transitions from state S_i to states $S_{i+1} \ldots S_{i+b-a}$. The last state created in the whole process is the final state.

We are now interested in an efficient simulation of the above automaton. Despite that this is a particular case of a regular expression, its simplicity permits a more efficient simulation. In particular, a fast bit-parallel simulation is possible.

We represent each automaton state by a bit in a computer word. The initial state is not represented because it is always active. As with the normal Shift-And, we shift all the bits to the left and use a table of masks B indexed by the current text character. This accounts for all the arrows that go from states S_j to S_{j+1}.

The remaining problem is how to represent the ε-transitions. For this sake, we chose[2] to represent active states by 1 and inactive states by 0. We call "gap-initial" states those states S_i from where an ε-transition leaves. For each gap-initial state S_i corresponding to a gap $x(a,b)$, we define its "gap-final" state to be $S_{i+b-a+1}$, i.e. the one following the last state

[1] Note that x and single letters can also be seen as classes of characters.

[2] It is possible to devise a formula for the opposite case, but unlike Shift-Or, it is not faster.

reached by an ε-transition leaving S_i. In the example of Figure 6, we have one gap-initial state (S_3) and one gap-final state (S_6).

We create a bit mask I which has 1 in the gap-initial states, and another mask F that has 1 in the gap-final states. Then, if we keep the state of the search in a bit mask D, then after performing the normal Shift-And step, we simulate all the ε-moves with the operation

$$D' \leftarrow D \mid ((F - (D \,\&\, I)) \,\&\, \sim F)$$

The rationale is as follows. First, $D \,\&\, I$ isolates the *active* gap-initial states. Subtracting this from F has two possible results for each gap-initial state S_i. First, if it is active the result will have 1 in all the states from S_i to S_{i+b-a}, successfully propagating the active state S_i to the desired target states. Second, if S_i is inactive the result will have 1 only in $S_{i+b-a+1}$. This undesired 1 is removed by operating the result with "$\& \sim F$". Once the propagation has been done, we *or* the result with the already active states in D. Note that the propagations of different gaps do not interfere with each other, since all the subtractions have local effect.

Let us consider again our example of Figure 6. The corresponding I and F masks are 00000100 and 00100000, respectively (recall that the bit masks are read right-to-left). Let us also consider that we have read the text *abc*, and hence our D mask is 00000100. At this point the ε-transitions should take effect. Indeed, $((F - (D \,\&\, I)) \,\&\, \sim F)$ yields $((00100000 - 00000100) \,\&\, 11011111) = 00011100$, where states S_3, S_4 and S_5 have been activated. If, on the other hand, $D = 00000010$, the propagation formula yields $((00100000 - 00000000) \,\&\, 11011111) = 00000000$ and nothing changes.

Figure 7 shows the complete algorithm. For simplicity the code assumes that there cannot be gaps at the beginning or at the end of the pattern (which are meaningless anyway). The value L (maximum length of a match) is obtained in $O(m)$ time by a simple pass over the pattern P, summing up the maximum gap lengths and individual classes (recall that m is the number of symbols in P). The preprocessing takes $O(L|\Sigma|)$ time, while the scanning needs $O(n)$ time. If $L > w$, however, we need several machine words for the simulation, which thus takes $O(n\lceil L/w \rceil)$ time.

5 A backward search algorithm for CBG patterns

When the searched patterns contain just classes of characters, the backward bit-parallel approach (see Section 2) leads to the fastest algorithm BNDM [26]. The search is done by sliding over the text (in forward direction) a window that has the size of the minimum possible alignment (ℓ). We read the window backwards trying to recognize a factor of the pattern. If we reach the beginning of the window, then we found an alignment. Else, we shift the window to the beginning of the longest factor found.

We extend now BNDM to deal with CBGs. To recognize all the reverse factors of a CBG, we use quite the same automaton built in Section 4 on the reversed pattern, but without the initial self-loop, and considering that all the states are active at the beginning. We create an initial state I and ε-transitions from I to each state of the automaton. Figure 8 shows the automaton for the pattern $a - b - c - x(1,3) - d - e$. A word read by this automaton is a factor of the CBG as long as there exists at least one active state.

The bit-parallel simulation of this automaton is quite the same as that of the forward automaton (see Section 4). The only modifications are (a) that we build it on P^r, the reversed pattern; (b) that the the bit mask D that registers the state of the search has to be

```
Search (P_{1...m},T_{1...n})

                    /* Preprocessing */
   L ← maximum length of a match
   for c ∈ Σ do B[c] ← 0^L
   I ← 0^L, F ← 0^L
   i ← 0
   for j ∈ 1...m
      if P_j is of the form x(a,b) then /* a gap */
         I ← I | (1 << (i − 1))
         F ← F | (1 << (i + b − a))
         for c ∈ Σ, k ∈ i...i + b − 1 do B[c] ← B[c] | (1 << k)
         i ← i + b
      else /* P_j is a class of characters */
         for c ∈ P_j do B[c] ← B[c] | (1 << i)          i ← i + 1
   nF ← ∼F
   M ← 1 << (L − 1)     /* final state */

                    /* Scanning */
   D ← 0^L
   for j ∈ 1...n
      if D & M ≠ 0^L then report a match ending at j − 1
      D ← ((D << 1) | 0^{L−1}1) & B[t_j]
      D ← D | ((F − (D & I)) & nF)
```

Figure 7: The forward scanning algorithm.

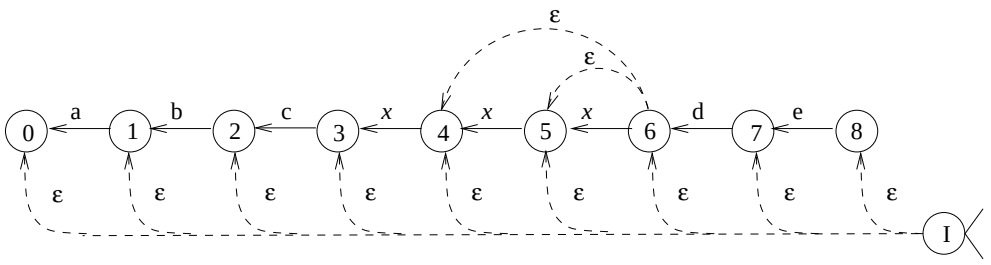


Figure 8: The non-deterministic automaton built in the backward algorithm to recognize all the reversed factors of the CBG $a - b - c - x(1,3) - d - e$.

initialized with $D = 1^L$ to perform the initial ε-transitions; and (c) that we do not *or* D with $0^{L-1}1$ when we shift it, for there is no more initial self-loop.

The backward CBG matching algorithm shifts a window of size ℓ along the text. Inside each window, it traverses backward the text trying to recognize a factor of the CBG (this is why the automaton that recognizes all the factors has to be built on the reverse pattern P^r).

If the backward search inside the window fails (i.e. there are no more active states in the backward automaton) before reaching the beginning of the window, then the search window is shifted to the beginning of the longest factor recognized, exactly like in the first case of the classic BNDM (see Section 2).

If the begining of the window is reached with the automaton still holding active states, then some factor of length ℓ of the CBG is recognized in the window. Unlike the case of exact string matching, where all the occurrences have the same length of the pattern, this does not automatically imply that we have recognized the whole pattern. We need a way to verify a possible alignment (that can be longer than ℓ) starting at the beginning of the window. So we read the characters again from the beginning of the window with the forward automaton of Section 4, but without the initial self-loop. This forward verification ends when (1) the automaton reaches its final state, in which case we found the pattern; (2) there are no more active states in the automaton, in which case there is no pattern occurrence starting at the window. As there is no initial loop, the forward verification surely finishes after reading at most L characters of the text. We then shift the search window one character to the right and resume the search.

Figure 9 shows the complete algorithm. Some optimizations are not shown for clarity, for example many tests can be avoided by breaking loops from inside, some variables can be reused, etc.

The worst case complexity of the backward scanning algorithm is $O(nL)$, which is quite bad in theory. However, on the average, the backward algorithm is expected to be faster than the forward one. The next section gives a good experimental criterion to know in which cases the backward algorithm is faster than the forward one. The experimental search results (see Section 7) on the PROSITE database show that the backward algorithm is almost always the fastest.

6 Which algorithm to use ?

We have now two different algorithms, a forward and a backward one, so a natural question is which one should be chosen for a particular problem. We seek for a simple criterion that

```
Backward search (P_{1...m},T_{1...n})

   L ← maximum length of a match                          /* Preprocessing */
   ℓ ← minimum length of a match
   for c ∈ Σ do B_f[c] ← 0^L, B_b[c] ← 0^L
   I_f ← 0^L, F_f ← 0^L, I_b ← 0^L, F_b ← 0^L
   i ← 0
   for j ∈ 1...m
      if P_j is of the form x(a,b) then /* a gap */
         I_f ← I_f | (1 << (i−1))        ,  I_b ← I_b | (1 << (L−(i+b)−1))
         F_f ← F_f | (1 << (i+b−a))      ,  F_b ← F_b | (1 << (L−i−a))
         for c ∈ Σ, k ∈ i...i+b−1 do
            B_f[c] ← B_f[c] | (1 << k), B_b[c] ← B_b[c] | (1 << (L−k−1))
         i ← i+b
      else /* P_j is a class of characters */
         for c ∈ P_j do
           B_f[c] ← B_f[c] | (1 << i), B_b[c] ← B_b[c] | (1 << (L−i−1))
         i ← i+1
   nF_f ← ∼F_f, nF_b ← ∼F_b
   M ← 1 << (L−1)     /* final state for the forward scan* /

   pos ← 0                                                /* Scanning */
   while pos ≤ n−ℓ do
      j ← ℓ, D_b ← 1^L
      while D_b ≠ 0^L and j > 0
         D_b ← D_b & B_b[t_{pos+j}]
         D_b ← D_b | ((F_b − (D_b & I_b)) & nF_b)
         j ← j−1
         if D_b ≠ 0^L and j = 0          /* f orward scan */
            D_f ← 0^{L−1}1, v ← 1
            while D_f ≠ 0^L and pos+v ≤ n
               D_f ← D_f & B_f[t_{pos+v}]
               D_f ← D_f | ((F_f − (D_f & I_f)) & nF_f)
               if D_f & M ≠ 0^L then
                  report a match beginning at pos+1
                  D_f ← 0^L
               D_f ← (D_f << 1)
               v ← v+1
         D_b ← (D_b << 1)
      pos ← pos+j+1
```

Figure 9: The backward scanning algorithm.

enables us to choose the best algorithm.

In particular, let us consider the maximum gap length G in the CBG. If $G \geq \ell$, then every text window of length ℓ *is* a factor of the CBG, so we will surely traverse all the window during the backward scan and always shift in 1, for a complexity of $\Omega(n\ell)$ at least. Consequently, the backward approach we have presented must be restricted at least to CBGs in which $G < \ell$.

This can be carried on further. Each time we position a window in the text, we know that at least $G+1$ characters in the window will be inspected before shifting. Moreover, the window will not be shifted by more than $\ell - G$ positions. Hence the total number of character inspections across the search is at least $(G+1)n/(\ell - G)$, which is larger than n (the number of characters inspected by a forward scan) whenever $\ell < 2G + 1$.

Hence, we define $(G+1)/\ell$ as a simple parameter governing most of the performance of the backward scan algorithm, and predict that 0.5 is the point above which the backward scanning is worse than forward scanning. Of course this measure is not perfect, as it disregards the effect of other gaps, classes of characters and the cost of forward checking in the backward scan, but a full analysis is extremely complicated and, as we see in the next section, this simple criterion gives good results.

According to this criterion, we can design an optimized version of our backward scanning algorithm. The idea is that we can choose the "best" prefix of the pattern, i.e. the prefix that minimizes $(G+1)/\ell$. The backward scanning can be done using this prefix, while the forward verification of potential matches is done with the full pattern. This could be extended to selecting the best factor of the pattern, but the code would be more complicated (as the verification phase would have to scan in both directions, buffering would be complicated, and, as we see in the next section, the difference is not so large.

7 Experimental results

We have tested our algorithms over an example of 1,168 PROSITE patterns [16, 13] and a 6 megabytes (MB) text containing a concatenation of protein sequences taken from the TIGR Microbial database. The set had originally 1,316 patterns from which we selected the 1,230 whose L (maximum length of a match) does not exceed w, the number of bits in the computer word of our machine. This leaves us with 93% of the patterns. From them, we excluded the 62 (5%) for which $G \geq \ell$, which as explained cannot be reasonably searched with backward scanning. This leaves us with the 1,168 patterns.

We have used an Intel Pentium III machine of 500 MHz running Linux. We show user times averaged over 10 trials. Three different algorithms are tested: *Fwd* is the forward-scan algorithm described in Section 4, *Bwd* is the backward-scan algorithm of Section 5 and *Opt* is the same *Bwd* where we select for the backward searching the best prefix of the pattern, according to the criterion of the previous section.

A first experiment aims at measuring the efficiency of the algorithms with respect to the criterion of the previous section. Figure 10 shows the results, where the patterns have been classified along the x axis by their $(G+1)/\ell$ value. As predicted, 0.5 is the value from which *Bwd* starts to be worse than *Fwd* except for a few exceptions (where the difference is not so big anyway). It is also clear that *Opt* avoids many of the worst cases of *Bwd*. Finally, the plot shows that the time of *Fwd* is very stable. While the forward scan runs always at around 50 MB/sec, the backward scan can be as fast as 200 MB/sec.

What Figure 10 fails to show is that in fact most PROSITE patterns have a very low

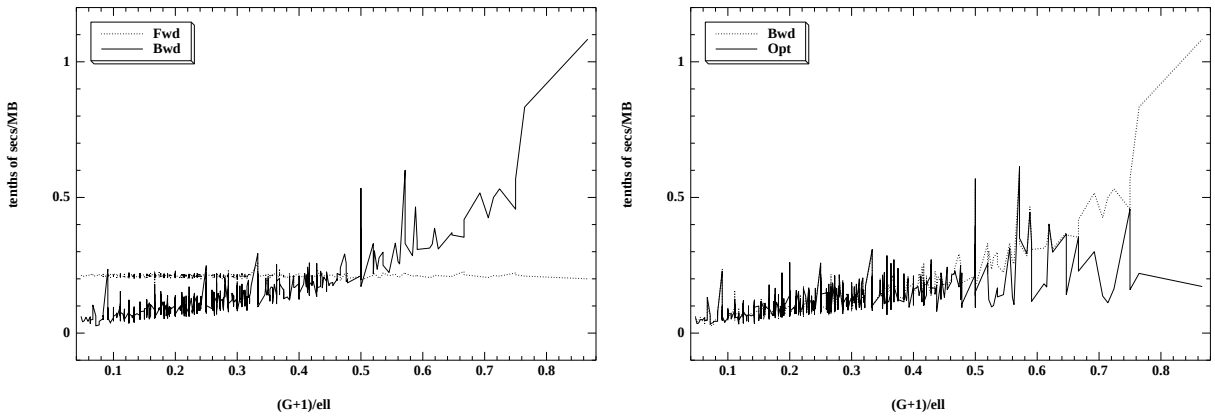


Figure 10: Search times (in tenths of seconds per MB) for all the patterns classified by their $(G+1)/\ell$ value.

$(G+1)/\ell$ value. Figure 11 plots the number of patterns achieving a given search time, after removing a few outliers (the 12 that took more than 0.04 seconds for *Bwd*). *Fwd* has a large peak because of its stable time, while the backward scanning algorithms have a wider histogram whose main body is well before the peak of *Fwd*. Indeed, 95.6% of the patterns are searched for faster by *Bwd* than by *Fwd*, and the percentage raises to 97.6% if we consider *Opt*. The plot also shows that there is little statistical difference between *Bwd* and *Opt*. Rather, *Opt* is useful to remove some very bad cases of *Bwd*.

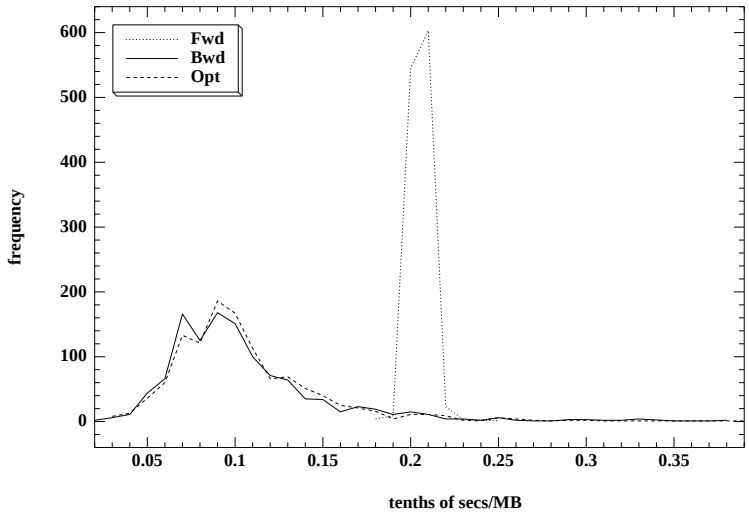


Figure 11: Histogram of search times for our different algorithms.

Our third experiment aims at comparing our search method against converting the pattern to a regular expression and resorting to general regular expression searching. From the existing algorithms to search for regular expressions we have selected the following.

Dfa: Builds a deterministic finite automaton and uses it to search the text.

Nfa: Builds a non-deterministic finite automaton and uses it to search the text, updating all

the states at each text position.

Myers: Is an intermediate between Dfa and Nfa [18], a non-deterministic automaton formed by a few blocks (up to 4 in our experiments) where each block is a deterministic automaton over a subset of the states. "$(x|\varepsilon)$" was expressed as "`.?`" in the syntax of this software.

Agrep: Is an existing software [34, 33] that implements another intermediate between Dfa and Nfa, where most of the transitions are handled using bit-parallelism and the ε-transitions with a deterministic table. "$(x|\varepsilon)$" was expressed as "`(.|"")`" in the syntax of this software.

Grep: Is *Gnu Grep* with the option `"-E"` to make it accept regular expressions. This software uses a heuristic that, in addition to (lazy) deterministic automaton searching, looks for long enough literal pattern substrings and uses them as a fast filter for the search. The gaps "$x(a,b)$" were converted to "`.{a,b}`" to permit specialized treatment by *Grep*.

BNDM: Uses the backward approach we have extended to CBGs, but adapted to general REs instead [25]. It needs to build to deterministic automata, one for backward search and another for forward verification.

Multipattern: Reduces the problem to multipattern Boyer-Moore searching of all the strings of length ℓ that match the RE [32]. We have used "`agrep -f`" as the multipattern search algorithm.

To these, we have added our *Fwd* and *Opt* algorithms. Figure 12 shows the results. From the forward scanning algorithms (i.e. *Fwd*, *Dfa*, *Nfa* and *Myers*, unable to skip text characters), the fastest is our *Fwd* algorithm thanks to its simplicity. *Agrep* has about the same mean but much more variance. *Dfa* suffers from high preprocessing times and large generated automata. *Nfa* needs to update many states one by one for each text character read. *Myers* suffers from a combination of both and shows two peaks that come from its specialized code to deal with small automata.

The backward scanning algorithms *Opt* and *Grep* (able to skip text characters) are faster than the previous ones in almost all cases. Among them, *Opt* is faster on average and has less variance, while the times of *Grep* extend over a range that surpasses the time of our *Fwd* algorithm for a non-negligible portion of the patterns. This is because *Grep* cannot always find a suitable filtering substring and in that case it resorts to forward scanning. Note that *BNDM* and *Multipattern* have been excluded from the plots due to their poor performance on this set of patterns.

Apart from the faster text scanning, our algorithms also benefit from lower preprocessing times when compared to the algorithms that resort to regular expression searching. This is barely noticeable in our previous experiment, but it is important in a common scenario of the protein searching problem: all the patterns from a set are searched for inside a new short protein. In this case the preprocessing time for all the patterns is much more important than the scanning time over the (normally rather short) protein.

We have simulated this scenario by selecting 100 random substrings of length 300 from our text and running the previous algorithms on all the 1,168 patterns. Table 1 shows the time averaged over the 100 substrings and accumulated over the 1,168 patterns. The difference in

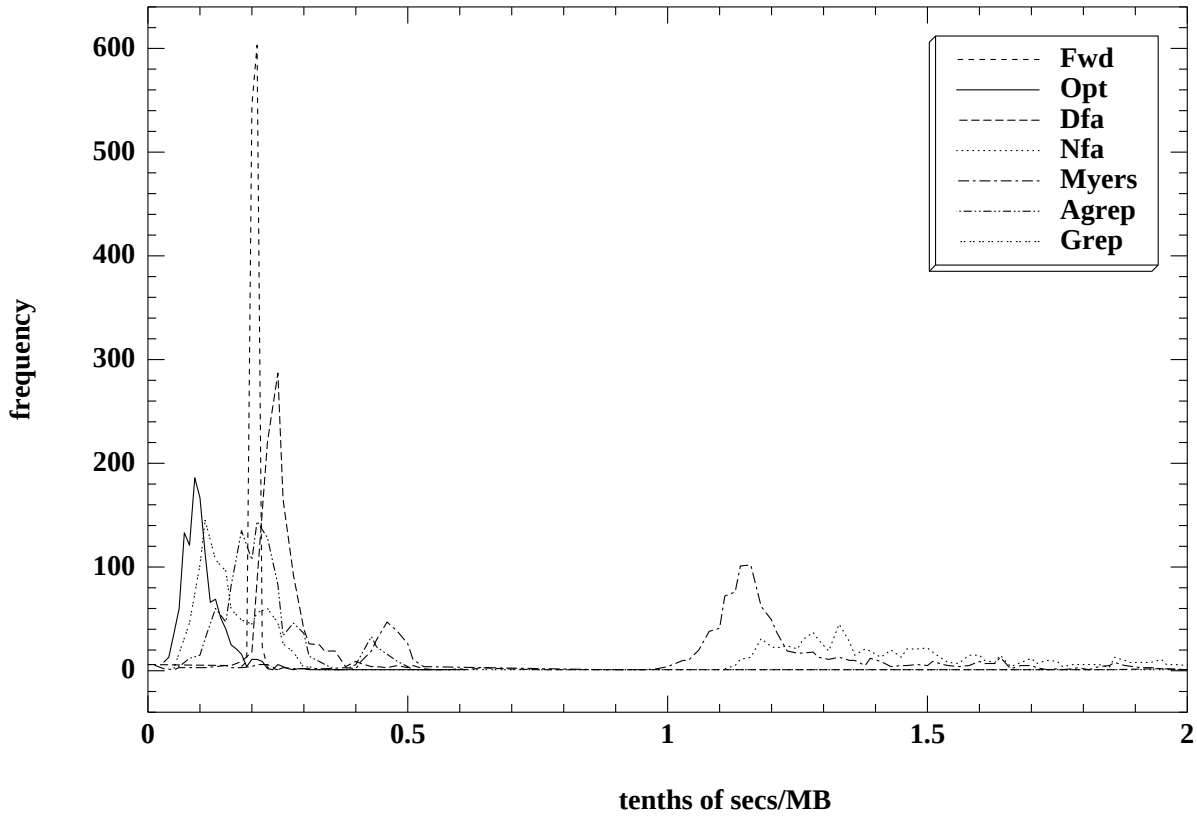


Figure 12: Histogram of search times for our best algorithms and for regular expression searching algorithms.

favor of our new algorithms is drastic. Note also that this problem is an interesting field of research for multipattern CBG search algorithms.

Algorithm	Fwd	Bwd	Opt	Dfa	Nfa	Myers	Agrep	Grep
Time	0.058	0.056	**0.050**	125.91	4.43	7.84	10.22	9.42

Table 1: Search time in seconds for all the 1,168 patterns over a random protein of length 300.

8 Extensions

In this section we include several extensions to the basic algorithms depicted in previous sections.

8.1 Obtaining all the text occurrences

Our forward scanning algorithm reports all the final positions of the text occurrences, while our backward scanning algorithm reports all the initial text positions. In several applications

```
ShiftLeft (X_t ... X_1, i)
    prevX ← 0^w
    for i ∈ 1 ... t
        Z_i  ←  (X_i << i) | prevX
        prevX  ←  X_i >> (w − i)
    return Z_t ... Z_1

Subtract (X_t ... X_1, Y_t ... Y_1)
    carry  ←  0
    for i ∈ 1 ... t
        Z_i  ←  X_i − Y_i − carry
        if Z_i > X_i or (Y_i = 1^w and carry = 1)
            carry  ←  1
        else carry  ←  0
    return Z_t ... Z_1
```

Figure 13: Multiword algorithms for $X << i$ and $X - Y$.

it is necessary to report all the occurrences, not only their initial or final positions. That is, if several occurrences of a CBG begin/end at the same position, we want to report all the corresponding final/initial positions.

If we use backward scanning, we are forced to use a forward verification, which is done with an automaton similar to the forward search one, but without the initial self-loop. In Section 5 we only wanted to report the initial positions, so we stopped the forward verification as soon as (i) we reached a final state, (ii) we ran out of final states, (iii) the text finished. If we want to report all the final positions corresponding to a given initial position, then, instead of stopping as soon as condition (i) is met, we will report a new final position every time (i) holds, and will keep reporting occurrences until (ii) or (iii) hold.

In case of forward scanning we have to do the symmetric job. Each time the forward scanning finds a final occurrence position, we should perform a backward verification, reading characters backward from the final position, and using a verification automaton built on the reversed pattern.

However, in some applications, one can be interested in reporting other information, especially when the matches intersect. Many studies have been done on that subject for regular expressions and these might be applied for CBG [21].

8.2 Handling longer patterns

If the number L of bits needed to represent the pattern is larger than w, the number of bits in the computer word, then several computer words have to be employed for the simulation. The easiest way to handle this is to implement all the operations on a new data type formed by an array of computer words. This is rather simple for operations that operate the bits locally, e.g. "&, "|", "$\sim$", "=", and so on, but it becomes a bit trickier for others, such as "$<<$" and "$-$". We give pseudocode in Figure 13 for these two operations.

Since operating with multiple words is usually much slower than with a single word, it

may be advisable to choose a small enough subpattern to scan, and verify the existence of the complete pattern with the multiword algorithm only when the shorter subpattern is found. This was done in Section 6 with other purposes, and it can be used to ensure that the search is done for a short subpattern, even if we opt for forward scanning.

8.3 Selecting the best subpattern to search for

In Section 6 we have shown that a clever idea is to choose a prefix of the search pattern that minimizes $(G+1)/\ell$, where G is its longest gap and ℓ the minimum length of a string matching the prefix. In the previous subsection we have also shown that it may be better to choose a small enough prefix to perform a fast scanning. Each occurrence of the prefix has to be verified for the occurrence of the whole pattern.

An interesting topic is that we are not forced to choose a prefix, but any factor of the pattern would do. An inconvenient is that verification is more complex, since we have to verify for full occurrences in both directions. That is, if we choose $P_{i\ldots i'}$ for backward scanning, then for each initial position j of an occurrence of $P_{i\ldots i'}$, we have to verify $t_{j-1}t_{j-2}\ldots$ for the reverse of $P_{1\ldots i-1}$ and $t_jt_{j+1}\ldots$ for $P_{i\ldots m}$. However, the reward for a more complex code can be significant if we have to search for a pattern with a long gap near the beginning.

8.4 Backward scanning with linear worst case time

In certain cases, although the average speed of the backward scan may be desirable, the risk of a quadratic search time for a given pattern may be unacceptable. We show in this section that it is possible to skip characters while still guaranteeing the linear worst case search time of the forward algorithm. In practice the resulting algorithm is slower than a pure backward scanning, but it is better than no skipping characters at all if linear worst case time has to be guaranteed.

The main classical idea [10, 28] to build such a linear worst case algorithm is to avoid retraversing the same characters in the backward window verification. Assume we search for a simple string P of length p. The search is done through a window of length p. We divide the work done on the sequence into two parts: forward and backward scanning. To be linear in the worst case, none of these two parts must retraverse characters. In the forward scan, it is enough to keep track of the longest pattern prefix v that matches the current text suffix.

However, we need to use also backward searching in order to skip characters. The idea is that the window of length p is placed so that the current longest prefix matched v is aligned with the beginning of the window. The position of the current text character inside the window (i.e. $|v|$) is called the *critical position*. At any point in the forward scan we can place the window (shifted $|v|$ characters from the current text position) and try a backward search. Clearly, this is only promising when v is not very long compared to p. Usually, a backward scan is attempted when the prefix is less than $\lfloor p/\alpha \rfloor$, where $0 < \alpha < p$ is a fixed arbitrary constant (usually $\alpha = 2$).

The backward search proceeds almost as before, but it finishes as soon as the critical position is reached. The two possibilities are:

(i) We reach the critical position. In this case we are not able to skip characters. The forward search is resumed in the place where it was left (i.e. from the critical position), totally retraverses the window, and continues until the condition to try a new backward scan holds again.

(ii) We do not reach the critical position, as we fail reading backwards on a character σ. This means that there cannot be a match in the current window. We start a forward scan from scratch just after σ, we then totally retraverse the window, and continue until a new backward scan seems promising.

This simple approach for a simple word must be adapted to the case of CBG. First, to avoid missing any match, we fix the size of the window to ℓ, the length of a smallest possible match of the CBG. All matches of the CBG are found through the forward scan. A prefix of the window corresponds in the CBG to paths beginning at the origin marked by active states in the current bit mask D (see Section 4). We replace the criterion of $\lfloor p/\alpha \rfloor$ by that of testing if no active bits remain on the right half of D, which can be tested in contant time.

In the case of a general regular expression, the linear worst case framework can be applied, but with more involved modifications [25].

9 Searching allowing differences

Apart from the flexibility in permitting classes of characters and gaps, it is useful to that the occurrences differ by a few characters from the pattern specification. This is modeled as follows. Let $d : \Sigma^* \times \Sigma^* \longrightarrow \mathbb{R}^+$ be a distance function between strings. A *threshold* k is given together with the search pattern. Then, we are interested in reporting text substrings w such that $d(w, v) \leq k$ for some v that matches the search pattern.

DEFINITION **2** *Searching for a CBG in a text $T = t_1 t_2 \ldots t_n$ with threshold k under a distance $d()$ consists in finding all the positions j of T such that there is a suffix w of $t_1 \ldots t_j$ where $d(w, v) \leq k$ for some alignment v of the CBG.*

The distance between the two strings is usually regarded as the cost to convert one string into the other via a sequence of operations over one or the other. The distance, called in general an *edit distance*, is the sum of the costs incurred across all the operations. Different applications permit different operations and assign them different costs. In computational biology, the usual operations are (a) substitute a given character by another, at a real-valued cost that depends on the characters substituted; (b) insert/delete characters into/from either string, at a cost that depends on the characters inserted/deleted and on the number of consecutive insertions/deletions made (usually inserting/deleting a group costs less than the sum of the individual operations).

There exist good algorithms to deal with regular expression searching allowing k differences [20, 19]. In particular, the latter focuses on the so-called "network expressions", which are regular expressions without cycles. Although the worst-case search cost is $O(mn)$ [20], somewhat improved average time search algorithms are possible [19]. These algorithms are rather slow in comparison to those for exact searching, but they permit using the complex cost functions that are of interest in computational biology. Since CBG patterns can be translated into network expressions, these algorithms give a solution.

It is interesting, however, that much faster searching is possible if we fix the cost of character insertions, deletions and substitutions at 1. This simpler distance, called Levenshtein distance, can be rephrased as the number of insertions, deletions and substitutions necessary to make both strings equal. The search algorithms for Levenshtein distance are so fast in comparison to those for a general edit distance, that it becomes interesting to use them as a

pre-filter for more refined searches into the proteins that happen to be interesting candidates for the search.

There exist some algorithms to handle regular expression searching allowing differences under Levenshtein distance [34, 36, 24] (the latter indeed permits arbitrary integer weights). Albeit much faster than the algorithms for general edit distances, they are considerably slower than those for exact regular expression searching. It is natural to ask whether we could design specific algorithms for CBG patterns, which could be simpler and faster.

A general technique introduced in [34] permits adapting any bit-parallel exact search algorithm to one permitting k differences. Recall our automaton of Figure 6 for the pattern $a - b - c - x(1,3) - d - e$. The automaton of Figure 14 searches for it permitting at most 2 differences. The vertical arrows are traversed by Σ and the diagonal arrows by $\Sigma \cup \{\varepsilon\}$.

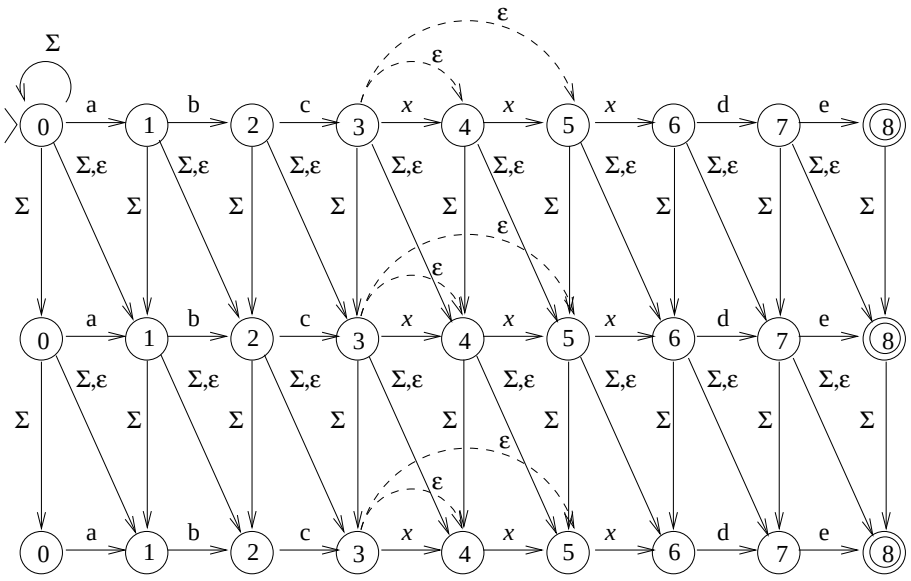


Figure 14: A non-deterministic automaton to search for the pattern $a - b - c - x(1,3) - d - e$ with 2 differences.

The rationale of the automaton is as follows. The original automaton has been replicated $k = 2$ times apart from its original version, which stays on top. Let us call "copy i" the i-th copy of the automaton, where copy zero is the original one. Then, copy i will reach its final state whenever we find the pattern in the text with i errors. This should be immediate for copy zero. Once we accept it for copy i, we notice that a vertical arrow permits us skipping a text character without changing the automaton state we are at, except that we activate the same state in the next copy. This is equivalent to permitting a deletion in the text (or an insertion in the pattern) and incrementing the number of differences seen. A diagonal arrow traversed by ε permits us skipping a character of the pattern without actually seeing it in the text, so it corresponds to an insertion in the text (or a deletion in the pattern). Finally, a diagonal arrow traversed by a character Σ permits us advancing in the pattern and in the text without having a match, so we are indeed substituting a text character by a pattern character, or vice versa. Hence we will reach the final state in copy $i + 1$ whenever we need a single further difference to extend an occurrence found by copy i.

In the original formulation [34, 28], more arrows would have been necessary, as in general we have to insert a "diagonal" arrow between rows i and $i + 1$ for every arrow in the original

```
Search (P_{1...m},T_{1...n},k)

  /* Preprocessing identical to forward searching, Figure 7 */
            /* Scanning */
   D_0  ←  0^L
   for i ∈ 1...k
      D_i  ←  D_{i-1} | (D_{i-1} << 1) | 0^{L-1}1
      D_i  ←  ((F - (D_i & I)) & nF)
   for j ∈ 1...n
      if D_k & M ≠ 0^L then report a match ending at j - 1
      oldD  ←  D_0
      D_0  ←  ((D_0 << 1) | 0^{L-1}1) & B[t_j]
      D_0  ←  D_0 | ((F - (D_0 & I)) & nF)
      for i ∈ 1...k
         newD  ←  (D_i << 1) & B[t_j]
         newD  ←  newD | oldD | ((oldD | D_{i-1}) << 1) | 0^{L-1}1
         newD  ←  newD | ((F - (newD & I)) & nF)
         oldD  ←  D_i
         D_i  ←  newD
```

Figure 15: The forward scanning algorithm allowing k differences.

automaton. In our case, this turns out to be unnecessary (and is reflected in simpler code). The only arrows not considered are the ε-transitions of Figure 6. Keeping in mind that, whenever state j is active at copy i, it has to be active at copy $i+1$ (since this represents matching a pattern prefix with i and $i+1$ differences), it is not hard to see that these extra arrows are unnecessary.

Simulating the behavior of the automaton is easy once we know how to simulate each row. Say that D_i is the bit mask that contains the state of the search at copy i. Then, in order to update the current values $D_0 \dots D_k$ to the new values $D'_0 \dots D'_k$, we first compute D'_0 from D_0 using the usual formula for exact searching. Then, for each $i \in 1 \dots k$, we compute D'_i from D_i, D_{i-1} and D'_{i-1}. The right order to consider the arrows is to consider horizontal, vertical and diagonal first, and then leaving the treatment of gaps for the second stage, as follows (note than only D_0 has a self-loop)

$$\begin{aligned} D'_i &\leftarrow (D_i << 1) \ \&\ B[t_j] \\ D'_i &\leftarrow D'_i \mid D_{i-1} \mid ((D_{i-1} << 1) \mid 0^{L-1}1) \mid (D'_{i-1} << 1) \\ D'_i &\leftarrow D'_i \mid ((F - (D'_i \ \&\ I)) \ \&\ nF) \end{aligned}$$

where, in the middle line, D_{i-1} accounts for the vertical arrows, $(D_{i-1} << 1) \mid 0^{L-1}1$ for the diagonal arrows via Σ, and $D'_{i-1} << 1$ (that is, the *new* value of D_i) accounts for diagonal arrows via ε. The complete scanning algorithm is given in Figure 15.

It would be possible to adapt this technique to backward scanning as well, but in practice the shifts are too short when we allow differences, so this is never better than forward scanning.

k	Ours	Nrgrep
0	0.221	0.546
1	0.361	1.224
2	0.583	1.985
3	0.797	2.805

Table 2: Comparison between algorithms for searching allowing k differences. Times are expressed in tenths of seconds, averaged over all the patterns.

Table 2 shows a comparison between this algorithm and the fastest algorithm for regular expression searching with unitary cost errors [34]. As shown in [24], the technique of [34] (from where we have adapted our CBG search algorithm) is by far the fastest for this case. Although that technique is already implemented in *Agrep* [33], we have adapted the implementation of *Nrgrep* [23], which poses less limits on pattern lengths and k values. In both cases, specific code is written and optimized for each k value, and we just count the number of matches. For a description of the machine, the text and the patterns used, see Section 7.

As it can be seen, our method is 3.4 to 3.5 times faster, thanks to simpler code and more locality of reference. In addition, it is much simpler to program. As for exact forward scanning, variance is very low, so considering average values is enough. We have included the times for the corresponding exact search algorithms to show the price of permitting differences.

10 Conclusions

We have presented two new search algorithms for CBGs, i.e. expressions formed by a sequence of classes of characters and bounded gaps. CBGs are of special interest to computational biology applications. All the current approaches rely on converting the CBG into a regular expression (RE), which is much more complex. Therefore the search cost is much higher than necessary for a CBG.

Our algorithms are specifically designed for CBGs and are based on BNDM, a combination of bit-parallelism and backward searching with suffix automata. This combination has been recently proved to be very effective for patterns formed by simple letters and classes of characters [26]. We have extended BNDM to allow for limited gaps.

We have presented experiments showing that our new algorithms are much faster and more predictable than all the other algorithms based on regular expression searching. In addition, we have presented a criterion to select the best among the two that has experimentally shown to be very reliable. This makes the algorithms of special interest for practical applications, such as protein searching.

Finally, we have shown how to handle several extensions, such as skipping characters while at the same time ensuring a linear worst case time, permitting a few differences between the pattern and the text, handling large patterns, recovering initial and final positions of occurrences, and finding the optimal search subpattern to optimize the search time.

A more challenging type of search permits negative gaps in the matches (combined with differences). This is solved in [19], but whether a faster bit-parallel algorithm can be designed remains an open question. Another relevant question is whether we can extend the search allowing differences to the case where insertions, deletions and substitutions have different

weights, as done recently for regular expressions in [24]. With the approach we have presented it is not hard to give a given integer cost I to all insertions, D to all deletions, and S to all substitutions, but it is not possible that the cost depends on the characters involved.

References

[1] R. Baeza-Yates. Text retrieval: Theory and practice. In *12th IFIP World Computer Congress*, volume I, pages 465–476. Elsevier Science, September 1992.

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

[3] R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

[4] G. Berry and R. Sethi. From regular expression to deterministic automata. *Theor. Comput. Sci.*, 48(1):117–126, 1986.

[5] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[6] A. Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213, November 1993.

[7] P. Bucher and A. Bairoch. A generalized profile syntax for biomolecular sequences motifs and its function in automatic sequence interpretation. In *Proceedings 2nd International Conference on Intelligent Systems for Molecular Biology*, pages 53–61, AAAIPress, Menlo Park,, 1994.

[8] C.-H. Chang and R. Paige. From regular expression to DFA's using NFA's. In *Proceedings of the CPM'92*, LNCS 664, pages 90–110. Springer-Verlag, Berlin, 1992.

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

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

[11] N. El-Mabrouk and M. Crochemore. Boyer-Moore strategy to efficient approximate string matching. In *Proc. CPM'96*, LNCS 1075, pages 24–38. Springer-Verlag, Berlin, 1996.

[12] C. Hagenah and A. Muscholl. Computing epsilon-free NFA from regular expressions in $O(n \log(n))$ time. In *Proc. MFCS'98*, LNCS 1450, pages 277–285, 1998.

[13] K. Hofmann, P. Bucher, L. Falquet, and A. Bairoch. The PROSITE database, its status in 1999. *Nucleic Acids Res.*, 27:215–219, 1999.

[14] J. Hromkovič, S. Seibert, and T. Wilke. Translating regular expression into small ε-free nondeterministic automata. In *Proc. STACS 97*, LNCS 1200, pages 55–66. Springer-Verlag, 1997.

[15] L. Ilie and S. Yu. Constructing NFAs by optimal use of positions in regular expressions. In *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 2373, pages 115–132. Springer-Verlag, Berlin, 2002.

[16] L.F. Kolakowski Jr., J.A.M. Leunissen, and J.E. Smith. ProSearch: fast searching of protein sequences with regular expression patterns related to protein structure and function. *Biotechniques*, 13:919–921, 1992.

[17] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):323–350, 1977.

[18] E. Myers. A four-russian algorithm for regular expression pattern matching. *J. of the ACM*, 39(2):430–448, 1992.

[19] E. Myers. Approximate matching of network expressions with spacers. *Journal of Computational Biology*, 3(1):33–51, 1996.

[20] E. W. Myers and W. Miller. Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, 51:7–37, 1989.

[21] E. W. Myers, P. Oliva, and K. Guimãraes. Reporting exact and approximate regular expression matches. In *Proc. CPM'98*, LNCS 1448, pages 91–103. Springer-Verlag, Berlin, 1998.

[22] G. Myers. A fast bit-vector algorithm for approximate pattern matching based on dynamic progamming. In *Proc. CPM'98*, LNCS v. 1448, pages 1–13. Springer-Verlag, 1998.

[23] G. Navarro. NR-grep: a fast and flexible pattern matching tool. *Software Practice and Experience (SPE)*, 31:1265–1312, 2001.

[24] G. Navarro. Approximate regular expression searching with arbitrary integer weights. Technical Report TR/DCC-2002-6, Dept. of Computer Science, University of Chile, July 2002.

[25] G. Navarro and M. Raffinot. Fast regular expression matching. In *Proc. WAE'99*, LNCS 1668, pages 198–212. Springer-Verlag, Berlin, 1999.

[26] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics (JEA)*, 5(4), 2000. `http://www.jea.acm.org/2000/NavarroString`.

[27] G. Navarro and M. Raffinot. Fast and simple character classes and bounded gaps pattern matching, with application to protein searching. In *Proceedings of the 5th Annual International Conference on Computational Molecular Biology (RECOMB)*, pages 231–240. ACM Press, 2001.

[28] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, 2002. ISBN 0-521-81307-7. 280 pages.

[29] M. Raffinot. On the multi backward dawg matching algorithm (MultiBDM). In *Proc. WSP'97*, pages 149–165. Carleton University Press, 1997.

[30] R. Staden. Screening protein and nucleic acid sequences against libraries of patterns. *DNA Sequence*, 1:369–374, 1991.

[31] K. Thompson. Regular expression search algorithm. *CACM*, 11(6):419–422, 1968.

[32] B. Watson. *Taxonomies and toolkits of regular language algorithms.* PhD thesis, Eindhoven Univ. of Technology, The Netherlands, 1995.

[33] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. USENIX Technical Conference*, pages 153–162, 1992.

[34] S. Wu and U. Manber. Fast text searching allowing errors. *CACM*, 35(10):83–91, October 1992.

[35] S. Wu, U. Manber, and E. Myers. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.

[36] S. Wu, U. Manber, and E. W. Myers. A subquadratic algorithm for approximate regular expression matching. *Journal of Algorithms*, 19(3):346–360, 1995.