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

[Extended Abstract]

Gonzalo Navarro [*]
Dept. 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@genome.uvsq.fr

ABSTRACT

The problem of fast searching of 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 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 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 performed many practical experiments using the PROSITE database, and all them show that our algorithms are the fastest in virtually all cases.

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*

General Terms

Algorithms

Keywords

Pattern matching, bit-parallelism, information retrieval, computational biology, PROSITE

[*]Work developed while the author was at postdoctoral stay at the Institut Gaspard-Monge, Univ. de Marne-la-Vallée, France, partially supported by Fundación Andes and ECOS/Conicyt.

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 [6, 9]. 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 a possible occurrence of this protein in a longer one. For example, the protein site number PS00007 has as its pattern 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 an 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 Appendix A. The majority of the PROSITE matching softwares use these techniques [11, 18].

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 of expressions that just contain classes of characters [15].

This is the motivation of this paper. We present two new simple algorithms to search 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, 22], approximate string matching [2, 22, 23, 3, 14], and REs matching [12, 21, 17]. 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 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 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.

The two algorithms we present are patented by the French Centre National de la Recherche Scientifique (CNRS)[1].

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 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 $K - -D - -Y$, $R - - - D - -Y$ and $R - -E - - - Y$.

Definition 1. *Searching 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 $t_1 \dots t_j$.*

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 $b_\ell \dots b_1$ 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, "∼" complements all the bits, and "<<" moves the bits to the left and enters zeros from the right, e.g. $b_\ell b_{\ell-1} \dots b_2 b_1 << 3 = b_{\ell-3} \dots b_2 b_1 000$. 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 $b_\ell \dots b_x 10000 - 1 = b_\ell \dots b_x 01111$.

2. PREVIOUS WORK

Bit-parallelism for simple pattern matching

The *bit-parallelism* technique [1] 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.

We present now the Shift-And algorithm [2, 22]. Figure 1 shows a non-deterministic automaton that searches a pattern in a text. Given a pattern $P = p_1 p_2 \dots p_m \in \Sigma^*$ and a text $T = t_1 t_2 \dots t_n \in \Sigma^*$, the algorithm builds first a table B which for each character stores a bit mask $b_m \dots b_1$. The

[1]French priority patent application n^0 0011093 filed on August 8th , 2000 by the CNRS.

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 $p_1p_2 \ldots p_i$ 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 1 is active). Therefore, we report a match whenever d_m is set.

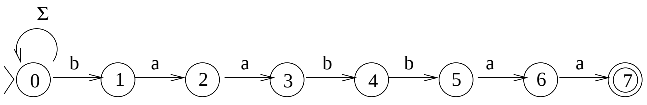


Figure 1: A nondeterministic automaton to search the pattern $P = baabbaa$ in a text.

We set $D = 0^m$ originally, and for each new text character t_j, we update D using the formula $D' \leftarrow ((D << 1) \mid 0^{m-1}1) \ \& \ B[t_j]$ which mimics the movement that occurs in the automaton.

It is very easy to extend to handle classes of characters, where each pattern position may 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 $c \in C_i$. No other change is necessary to the algorithm.

Combining bit-parallelism and suffix automata

The BNDM pattern matching algorithm [15], a combination of Shift-Or and BDM [8, 7], 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 $P = p_1p_2 \ldots p_m$ is an automaton that recognizes all the suffixes of P. The nondeterministic version of this automaton is shown in Figure 2. 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 $D = d_m \ldots d_1$.

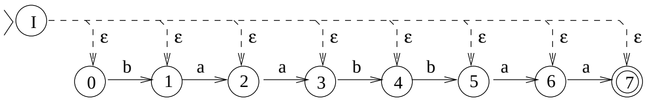


Figure 2: A nondeterministic suffix automaton for the pattern $P = baabbaa$. Dashed lines represent ε-transitions (i.e. they occur without consuming any input).

To search a pattern $P = p_1p_2 \ldots p_m$ in a text $T = t_1t_2 \ldots t_n$, the suffix automaton of $P^r = p_mp_{m-1} \ldots p_1$ (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: *(A)* 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. 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). *(B)* 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.

This algorithm is $O(mn)$ worst case time, but optimal on average ($O(n \log_\sigma m/m)$ time).

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 this type of patterns [15, 16].

Regular expression searching

Many algorithms have been designed to search a regular expression in a text. A survey of the different techniques and automata built is given in the Appendix A.

3. 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 Appendix A), although the functionality is the same.

Figure 3 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 de at the end of the pattern. The initial self-loop allows the match to begin at any text position.

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.

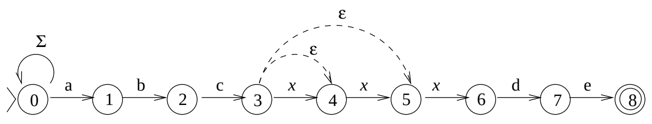


Figure 3: Our non-deterministic automaton for the pattern $a - b - c - x(1,3) - d - e$.

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 reached by an ε-transition leaving S_i. In the example of Figure 3, 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 interfer with each other, since all the subtractions have local effect.

The complete algorithm is given in Appendix B. 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.

4. 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 [15, 16]. 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.

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

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 3 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 4 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.

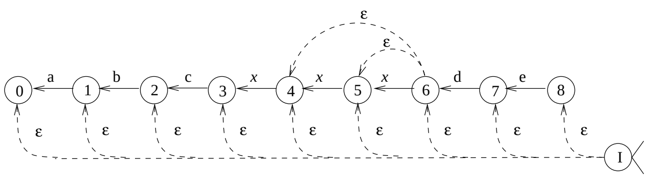


Figure 4: The non-deterministic automaton built in the backward algorithm to recognize all the reversed factors of the CBG $a - b - c - x(1,3) - d - e$.

The bit-parallel simulation of this automaton is quite the same as that of the forward automaton (see Section 3). 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 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 3, 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.

The complete algorithm is given in Appendix C. 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 6) on the PROSITE database shows that the backward algorithm is almost always the fastest.

5. 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 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.

6. EXPERIMENTAL RESULTS

We have tested our algorithms over an example of 1,168 PROSITE patterns [11, 9] 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 (we had to resort to forward scanning for them). 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 3, *Bwd* is the backward-scan algorithm of Section 4 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 5 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 5 Mb/sec, the backward scan can be as fast as 20 Mb/sec.

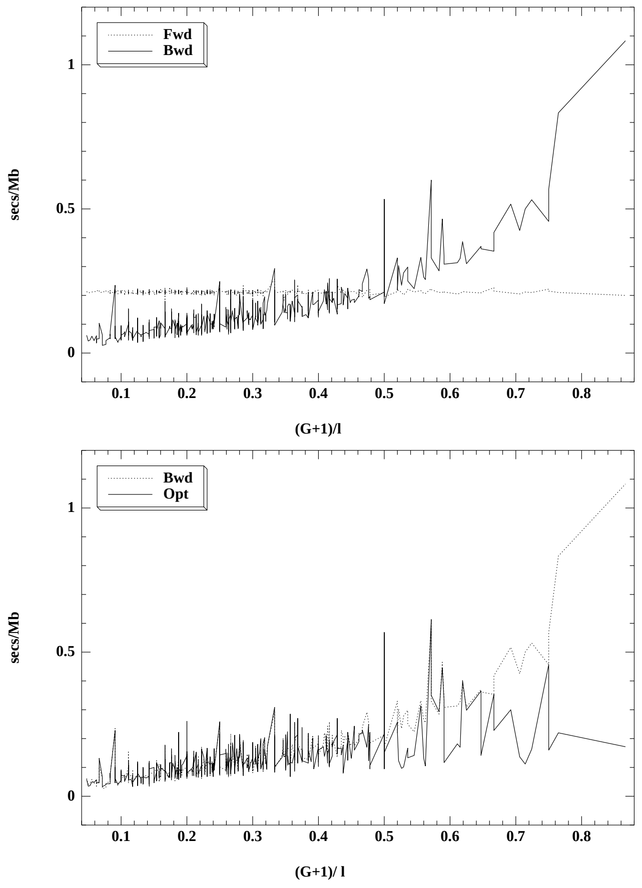


Figure 5: Search times (in seconds per Mb) for all the patterns classified by their $(G+1)/\ell$ value.

What Figure 5 fails to show is that in fact most PROSITE patterns have a very low $(G+1)/\ell$ value. Figure 6 plots the number of patterns achieving a given search time, after removing a few outliers (the 12 that took more than 0.4 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 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*.

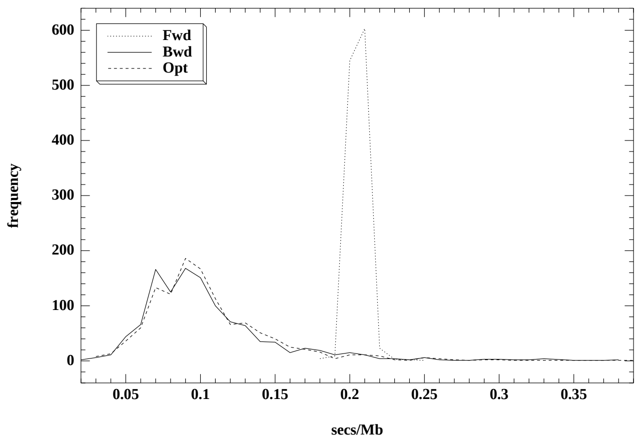


Figure 6: Histogram of search times for our different algorithms.

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 [12], 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.
- **Agrep:** Is an existing software [22, 21] 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.
- **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.
- **BNDM:** Uses the backward approach we have extended to CBGs, but adapted to general REs instead [17]. 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 [20]. We have used "`agrep -f`" as the multipattern search algorithm.

To these, we have added our *Fwd* and *Opt* algorithms. Figure 7 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 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 favor of our new algorithms is drastic. Note also that this problem is an interesting field of research for multipattern CBG search algorithms.

7. 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. Our algorithms are specifically designed for CBGs and are based on BNDM, a combination of bit-parallelism and backward searching with suffix automata.

We have presented experiments showing that our new algorithms are much faster and more predictable than all the

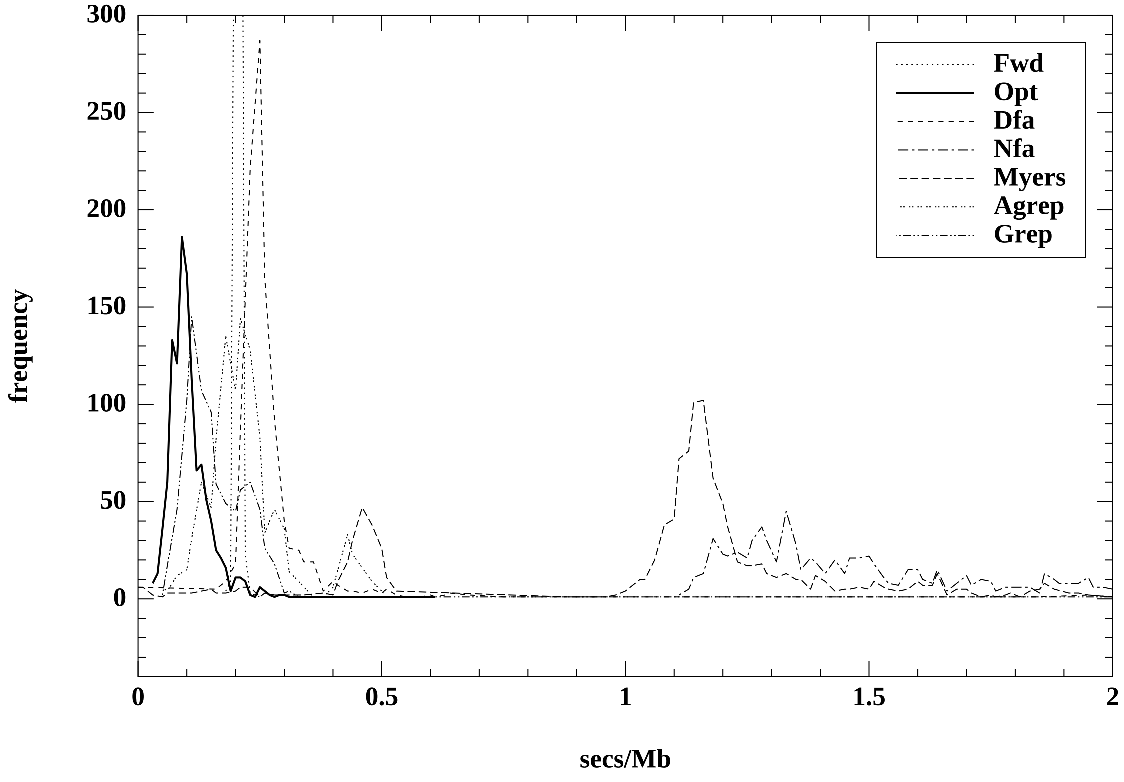


Figure 7: Histogram of search times for our best algorithms and for regular expression searching algorithms. Fwd reaches 600.

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

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

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.

We plan to extend the present work by allowing negative gaps and errors in the matches (see, e.g. [13]). Our algorithms are especially easy to extend to permit errors and we are pursuing in that direction.

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

[6] 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.

[7] Maxime Crochemore and W. Rytter. *Text algorithms.* Oxford University Press, 1994.

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

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

[10] Juraj Hromkovič, Sebastian Seibert, and Thomas Wilke. Translating regular expression into small ε-free nondeterministic automata. In *STACS 97*, Lecture Notes in Computer Science, pages 55–66. Springer-Verlag, 1997.

[11] 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.

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

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

[14] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, 1999.

[15] G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *Proc. CPM'98*, LNCS v. 1448, pages 14–33. Springer-Verlag, 1998.

[16] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. Technical Report TR/DCC-98-4, Dept. of Computer Science, Univ. of Chile, August 1998. To appear in *ACM Journal of Experimental Algorithmics (JEA)*.

[17] G. Navarro and M. Raffinot. Fast regular expression search. In *Proc. WAE'99*, LNCS 1668, pages 198–212, 1999.

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

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

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

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

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

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

APPENDIX

A. REGULAR EXPRESSION SEARCHING

The usual way of dealing with an expression with character classes and bounded gaps is actually to search it as a full regular expression (RE) [11, 18]. 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 [19] to search an 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 [12] 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 [22, 17].

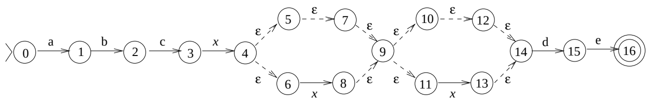
There exist currently many different techniques to build an NFA from a regular expression R. The most classical one is Thompson's construction [19], which builds an NFA with at most $2m$ states and $4m$ transitions (where m is counted as the number of letters and ε's in the RE). A second one is Glushkov's 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 Gluskov's construction has been pursued, like [5], where it is shown that the resulting NFA is quadratic in the number of edges in the worst case. In [10], 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 construction with $O(m)$ states and $O(m(\log m)^2)$ edges, as well as a lower bound of $O(m \log m)$ edges. Hence, Glushkov construction is not space-optimal.

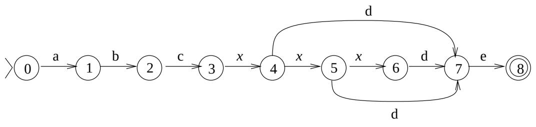
We show in Figure 8 the Thompson and Gluskov automata for an example CBG $a-b-c-x(1,3)-d-e$, which we translate into the regular expression $a\cdot b\cdot c\cdot x\cdot(x|\varepsilon)\cdot(x|\varepsilon)\cdot d\cdot e$.

Both Thompson and Gluskov automata present some particular properties. Some algorithms like [12, 22] make use of Thompson's automaton properties and some others, like [17], make use of Gluskov's ones.

Finally, some work has been pursued in skipping characters when searching for an 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 [20] 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 [17] 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) Thompson construction



(b) Gluskov construction

Figure 8: 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 Gluskov automaton is ε free, but both present some difficulties to perform an efficient bit-parallelism on them.

a BNDM-like scheme to search those factors (see Section 2).

However, none of the presented techniques seems fully adequate for CBGs. First, the algorithms are intrinsequely 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 6. As we show in that section, CBGs can be searched much faster by designing specific algorithms for them. This is what we do in the next sections.

B. FORWARD SEARCH PSEUDOCODE

Figure 9 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).

C. BACKWARD SEARCH PSEUDOCODE

Figure 10 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.

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

          /* Preprocessing */

  $L \leftarrow$ maximum length of a match
  for $c \in \Sigma$ do $B[c] \leftarrow 0^L$
  $I \leftarrow 0^L$, $F \leftarrow 0^L$
  $i \leftarrow 0$
  for $j \in 1 \ldots m$
     if $P_j$ is of the form $x(a,b)$ then /* a gap */
        $I \leftarrow I \mid (1 << (i-1))$
        $F \leftarrow F \mid (1 << (i+b-a))$
        for $c \in \Sigma$, $k \in i \ldots i+b-1$ do
             $B[c] \leftarrow B[c] \mid (1 << k)$
        $i \leftarrow i+b$
     else /* $P_j$ is a class of characters */
        for $c \in P_j$ do $B[c] \leftarrow B[c] \mid (1 << i)$
        $i \leftarrow i+1$
  $nF \leftarrow \sim F$
  $M \leftarrow 1 << (L-1)$     /* final state */

          /* Scanning */

  $D \leftarrow 0^L$
  for $j \in 1 \ldots n$
     if $D$ & $M \neq 0^L$ then
          report a match ending at $j-1$
     $D \leftarrow ((D << 1) \mid 0^{L-1}1)$ & $B[t_j]$
     $D \leftarrow D \mid ((F - (D$ & $I))$ & $nF)$
```

Figure 9: Forward search pseudocode

D. MULTIPLE WORD EXTENSION

The two previous algorithms can be used for longest word by simulating the computer words operations on table of words. All the commands used to update the state of the search in one computer word D are trivially extended to a sequence of words $D_1 \ldots D_d$ (where the lowest bits are in D_1). The only exception is the subtraction operation, where the operation on D_i can affect D_{i+1}. Let us say that we have two computer multi-words $A = A_1 \ldots A_d$ and $B = B_1 \ldots B_d$, and we want to compute $C = A - B = C_1 \ldots C_d$. The algorithm is as follows (we assume that the numbers are unsigned)

```
$carry \leftarrow 0$
for $i \in 1 \ldots d$
   $C_i \leftarrow A_i - B_i - carry$
   if $A_i < B_i + carry$ or $B_i + carry < B_i$
      then $carry \leftarrow 1$
      else $carry \leftarrow 0$
```

where the fourth line has two checks: a first one covers the normal cases and the second one covers the special case $B_i = 1^w$.

```
Backward search ($P_{1\dots m}$,$T_{1\dots n}$)

          /* Preprocessing */

 $L \leftarrow$ maximum length of a match
 $\ell \leftarrow$ minimum length of a match
   for $c \in \Sigma$ do $B_f[c] \leftarrow 0^L$, $B_b[c] \leftarrow 0^L$
   $I_f \leftarrow 0^L$, $F_f \leftarrow 0^L$, $I_b \leftarrow 0^L$, $F_b \leftarrow 0^L$
   $i \leftarrow 0$
   for $j \in 1 \dots m$
      if $P_j$ is of the form $x(a,b)$ then /* a gap */
        $I_f \leftarrow I_f \mid (1 << (i-1))$
        $I_b \leftarrow I_b \mid (1 << (L-(i+b)-1))$
        $F_f \leftarrow F_f \mid (1 << (i+b-a))$
        $F_b \leftarrow F_b \mid (1 << (L-i-a))$
        for $c \in \Sigma$, $k \in i \dots i+b-1$ do
            $B_f[c] \leftarrow B_f[c] \mid (1 << k)$
            $B_b[c] \leftarrow B_b[c] \mid (1 << (L-k-1))$
         $i \leftarrow i+b$
      else /* $P_j$ is a class of characters */
         for $c \in P_j$ do
            $B_f[c] \leftarrow B_f[c] \mid (1 << i)$
            $B_b[c] \leftarrow B_b[c] \mid (1 << (L-i-1))$
         $i \leftarrow i+1$
   $nF_f \leftarrow \sim F_f$, $nF_b \leftarrow \sim F_b$
   $M \leftarrow 1 << (L-1)$

          /* Scanning */

   pos $\leftarrow 0$
   while $pos \leq n-\ell$ do
      $j \leftarrow \ell$, $D_b \leftarrow 1^L$
      while $D_b \neq 0^L$ and $j > 0$
         $D_b \leftarrow D_b$ & $B_b[t_{pos+j}]$
         $D_b \leftarrow D_b \mid ((F_b - (D_b \,\&\, I_b))$ & $nF_b)$
         $j \leftarrow j-1$
         if $D_b \neq 0^L$ and $j = 0$     /* forward scan */
            $D_f \leftarrow 0^{L-1}1$, $v \leftarrow 1$
            while $D_f \neq 0^L$ and $pos+v \leq n$
               $D_f \leftarrow D_f$ & $B_f[t_{pos+v}]$
               $D_f \leftarrow D_f \mid ((F_f - (D_f \,\&\, I_f))$ & $nF_f)$
               if $D_f$ & $M \neq 0^L$ then
                  report a match beginning at $pos+1$
                  $D_f \leftarrow 0^L$
               $D_f \leftarrow (D_f << 1)$
               $v \leftarrow v+1$
         $D_b \leftarrow (D_b << 1)$
      $pos \leftarrow pos+j+1$
```

Figure 10: The backward scanning algorithm.