# New Techniques for Regular Expression Searching *

Gonzalo Navarro[†] Mathieu Raffinot[‡]

**Abstract**

We present two new techniques for regular expression searching and use them to derive faster practical algorithms.

Based on the specific properties of Glushkov's nondeterministic finite automaton construction algorithm, we show how to encode a deterministic finite automaton (DFA) using $O(m2^m)$ bits, where $m$ is the number of characters, excluding operator symbols, in the regular expression. This compares favorably against the worst case of $O(m2^m|\Sigma|)$ bits needed by a classical DFA representation (where $\Sigma$ is the alphabet) and $O(m2^{2m})$ bits needed by the Wu and Manber approach implemented in *Agrep*.

We also present a new way to search for regular expressions, which is able to skip text characters. The idea is to determine the minimum length $\ell$ of a string matching the regular expression, manipulate the original automaton so that it recognizes all the reverse prefixes of length up to $\ell$ of the strings originally accepted, and use it to skip text characters as done for exact string matching in previous work.

We combine these techniques into two algorithms, one able and one unable to skip text characters. The algorithms are simple to implement, and our experiments show that they permit fast searching for regular expressions, normally faster than any existing algorithm.

## 1 Introduction

The need to search for regular expressions arises in many text-based applications, such as text retrieval, text editing and computational biology, to name a few. A *regular expression* is a generalized pattern composed of (i) basic strings, (ii) union, concatenation and Kleene closure of other regular expressions. Readers unfamiliar with the concept and terminology related to regular expressions are referred to a classical book such as [1]. We call $m$ the length of our regular expression, not counting operator symbols. The alphabet is denoted $\Sigma$, and $n$ is the length of the text.

The traditional techniques to search for a regular expression achieve $O(n)$ search time. Their main problem has been always their space requirement, which can be as high as $O(m2^{2m}|\Sigma|)$ bits to code the deterministic automaton (DFA) [26, 1]. An alternative is $O(mn)$ search time and $O(m)$ space [26], which is slow in practice.

*Supported in part by ECOS-Sud project C99E04 and, for the first author, Fondecyt grant 1-020831.

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

[‡]Equipe Génome et Informatique, Tour Evry 2, 523, place des terrasses de l'Agora, 91034 Evry, France. raffinot@genopole.cnrs.fr.

Newer techniques based on the Four Russians approach have obtained $O(mn/\log s)$ search time using $O(s)$ space [19]. The same result with a simpler implementation was obtained using bit-parallelism [29]: they use a table of $O(m2^{2m})$ bits that can be split in $k$ tables of $O(m2^{2m/k})$ bits each, at a search cost of $O(kn)$ table inspections.

These approaches are based on Thompson's nondeterministic finite automaton (NFA) construction [26], which produces an NFA of $m+1$ to $2m$ states. Several regularities of Thompson's NFA have been essential in the design of the newer algorithms [19, 29]. Another NFA construction is Glushkov's [12, 4]. Although it does not provide the same regularities of Thompson's, this construction always produces an NFA of minimal number of states, $m+1$. Hence its corresponding DFA needs only $O(m2^m|\Sigma|)$ bits, which is significantly less than the worst case using Thompson's NFA.

Another more recent trend in regular expression searching is to avoid inspecting every text character. Adapting well known techniques in simple string matching, these techniques find a set of strings such that some string in the set appears inside every occurrence of the regular expression. Hence the problem is reduced to multi string searching plus verification of candidate positions ([27, Chapter 5] and *Gnu Grep v2.0*). The verification has to be done with a classical algorithm.

This paper presents new contributions to the problem of regular expression searching. A first one is a bit-parallel representation of the DFA based on Glushkov's NFA and its specific properties, which needs $O(m2^m)$ bits and can be split in $k$ tables as the existing one [29]. This is the most compact representation we are aware of. A second contribution is an algorithm able to skip text characters based on a completely new concept that borrows from the BDM and BNDM string matching algorithms [11, 22].

The net result is a couple of algorithms, one unable and one able of skipping text characters. The former is experimentally shown to be at least 10% faster than any previous nonskipping algorithm to search for regular expressions of moderate size, which include most cases of interest. The latter is interesting when the regular expression does not match too short or too frequent strings, in which case it is faster than the former algorithm and generally faster than every character skipping algorithm.

We organize the paper as follows. In the rest of this section we introduce the notation used. Section 2 reviews related work and puts our contribution in context in more detail. Section 3 presents basic concepts on Glushkov's construction and Section 4 uses them to present our compact DFA and our first search algorithm. Section 5 presents the general character skipping approach, and Section 6 our extension to regular expressions. Section 7 gives an empirical evaluation of our new algorithms compared to the best we are aware of. Finally, Section 8 gives our conclusions.

Earlier versions of this work appeared in [21, 23]. The techniques presented here have been used in the recent pattern matching software *Nrgrep* [20].

### 1.1 Notation

Some definitions that are used in this paper follow. A *word* is a string or sequence of characters over a finite alphabet $\Sigma$. The empty word is denoted $\varepsilon$ and the set of all words built on $\Sigma$ ($\varepsilon$ included) is $\Sigma^*$. A word $x \in \Sigma^*$ is a *factor* (or substring) of $p$ if $p$ can be written $p = uxv$, $u, v \in \Sigma^*$. A factor $x$ of $p$ is called a *suffix* (resp. *prefix*) of $p$ is $p = ux$ (resp. $p = xv$), $u \in \Sigma^*$.

We define also the language to denote regular expressions. Union is denoted with the infix

sign "|", Kleene closure with the postfix sign "$*$", and concatenation simply by putting the subexpressions one after the other. Parentheses are used to change the precedence, which is normally "$*$", ".", "|". We adopt some widely used extensions: $[c_1...c_k]$ (where $c_i$ are characters) is a shorthand for $(c_1|...|c_k)$. Instead of a character $c$, a range $c^1$-$c^2$ can be specified to avoid enumerating all the characters between (and including) $c^1$ and $c^2$. Finally, the period (.) represents any character. We call $RE$ our regular expression pattern, which is of length $m$. We note $L(RE)$ the set of words generated by $RE$.

We also define some terminology for bit-parallel algorithms. A *bit mask* is a sequence of bits. Typical bit operations are infix "|" (bitwise *or*), infix "&" (bitwise *and*), prefix "$\sim$" (bit complementation), and infix "<<" (">>"), which moves the bits of the first argument (a bit mask) to higher (lower) positions in an amount given by the argument on the right. Additionally, one can treat the bit masks as numbers and obtain specific effects using the arithmetic operations $+$, $-$, etc. Exponentiation is used to denote bit repetition, e.g. $0^3 1 = 0001$.

## 2 Our Work in Context

The traditional technique [26] to search for a regular expression of length $m$ in a text of length $n$ is to convert the expression into a nondeterministic finite automaton (NFA) with $m+1$ to $2m$ nodes. Then, it is possible to search the text using the automaton at $O(mn)$ worst case time. The cost comes from the fact that more than one state of the NFA may be active at each step, and therefore all may need to be updated. A more efficient choice [1] is to convert the NFA into a deterministic finite automaton (DFA), which has only one active state at a time and therefore allows searching the text at $O(n)$ cost, which is worst-case optimal. The problem with this approach is that the DFA may have $O(2^{2m})$ states, which implies a preprocessing cost and extra space exponential in $m$. Several heuristics have been proposed to alleviate this problem, from the well-known lazy DFA that builds only the states reached while scanning the text (implemented for example in *Gnu Grep*) to attempts to represent the DFA more compactly [18]. Yet, the space usage of the DFA is still the main drawback of this approach.

Some techniques have been proposed to obtain a good tradeoff between both extremes. In 1992, Myers [19] presented a Four-Russians approach which obtains $O(mn/\log s)$ worst-case time and $O(s)$ extra space. The idea is to divide the syntax tree of the regular expression into "modules", which are subtrees of a reasonable size. These subtrees are implemented as DFAs and are thereafter considered as leaf nodes in the syntax tree. The process continues with this reduced tree until a single final module is obtained.

The DFA simulation of modules is done using *bit-parallelism*, which is a technique to code many elements in the bits of a single computer word (which is called a "bit mask") and manage to update all them in a single operation. In our case, the vector of active and inactive states is stored as bits of a computer word. Instead of (ala Thompson [26]) examining the active states one by one, the whole computer word is used to index a table which, together with the current text character, provides the new bit mask of active states. This can be considered either as a bit-parallel simulation of an NFA, or as an implementation of a DFA (where the identifier of each deterministic state is the bit mask as a whole).

Pushing even more in this direction, we may resort to pure bit-parallelism and forget about the

modules. This was done in [29] by Wu and Manber, and included in their software *Agrep* [28]. A computer word is used to represent the active (1) and inactive (0) states of the NFA. If the states are properly arranged and the Thompson construction [26] is used, all the arrows carry 1's from bit positions $i$ to $i+1$, except for the $\varepsilon$-transitions. Then, a generalization of Shift-Or [3] (the canonical bit-parallel algorithm for exact string matching) is presented, where for each text character two steps are performed. First, a forward step moves all the 1's that can move from a state to the next one. This is achieved by precomputing a table $B$ of bit masks, such that the $i$-th bit of $B[c]$ is set if and only if the character $c$ matches at the $i$-th position of the regular expression. Second, the $\varepsilon$-transitions are carried out. As $\varepsilon$-transitions follow arbitrary paths, a table $E: \; 2^{O(m)} \rightarrow 2^{O(m)}$ is precomputed, where $E[D]$ is the $\varepsilon$-closure of $D$. To move from the state set $D$ to the new $D'$ after reading text character $c$, the action is

$$D' \;\; \leftarrow \;\; E[(D << 1) \;\&\; B[c]]$$

Possible space problems are solved by splitting this table "horizontally" (i.e. less bits per entry) in as many subtables as needed, using the fact that $E[D_1 D_2] = E[D_1 0^{|D_2|}] \mid E[0^{|D_1|} D_2]$. This can be thought of as an alternative decomposition scheme, instead of Myers' modules.

All the approaches mentioned are based on the Thompson construction of the NFA, whose properties have been exploited in different ways. An alternative and much less known NFA construction algorithm is Glushkov's [12, 4]. A good point of this construction is that, for a regular expression of $m$ characters, the NFA obtained has exactly $m+1$ states and is free of $\varepsilon$-transitions. Thompson's construction, instead, produces between $m+1$ and $2m$ states. This means that Wu and Manber's table may need a table of size $2^{2m}$ entries of $2m$ bits each, for a total space requirement of $2m(2^{2m+1} + |\Sigma|)$ bits ($E$ plus $B$ tables).

Unfortunately, the structural property that arrows are either forward or $\varepsilon$-transitions does not hold on Glushkov's NFA. As a result, we need a table $M: \; 2^{m+1} \times \Sigma \rightarrow 2^{m+1}$ indexed by the current state and text character, for a total space requirement of $(m+1)2^{m+1}|\Sigma|$ bits. The transition action is simply $D' \;\leftarrow\; M[D, c]$, just as for a classical DFA implementation.

In this paper, we use specific properties of the Glushkov construction (namely, that all the arrows arriving to a state are labeled by the same letter) to eliminate the need of a separate table per text character. As a result, we obtain the best of both worlds: we can have tables whose arguments have just $m+1$ bits and we can have just one table instead of one per character. Thus we can represent the DFA using $(m+1)(2^{m+1} + |\Sigma|)$ bits, which is not only better than both previous bit parallel implementations but also better than the classical DFA representation, which needs in the worst case $(m+1)2^{m+1}|\Sigma|$ bits using Glushkov's construction.

The net result is a simple algorithm for regular expression searching which uses normally less space and has faster preprocessing and search time. Although all are $O(n)$ search time, a smaller DFA representation implies more locality of reference.

The ideas presented up to now aim at a good implementation of the automaton, but they must inspect all the text characters. In many cases, however, the regular expression involves sets of relatively long substrings that must appear inside any occurrence of the regular expression. In [27, chapter 5], a multipattern search algorithm is generalized to regular expression searching, in order to take advantage of this fact. The resulting algorithm finds all suffixes (of a predetermined length) of words of the language denoted by the regular expression and uses the Commentz-Walter

algorithm [9] to search for them. Another technique of this kind is used in *Gnu Grep v2.0*, which extracts a single string (the longest) that must appear in any match. This string is searched for and the neighborhoods of its occurrences are checked for complete matches using a lazy deterministic automaton. Note that it is possible that there is no such single string, in which case the scheme cannot be applied.

In this paper, we present a new regular expression search algorithm able to skip text characters. It is based on extending BDM and BNDM [11, 22]. These are simple string search algorithms whose main idea is to build an automaton able to recognize the reverse prefixes of the pattern, and to examine backwards a window of length $m$ on the text. This automaton helps us to determine (i) when it is possible to shift the window because no pattern substring has been seen, and (ii) the next position where the window can be placed, i.e. the last time that a pattern prefix was seen. BNDM is a bit-parallel implementation of this automaton, faster and much simpler than the traditional version, BDM, which makes the automaton deterministic.

Our algorithm for regular expression searching is an extension where, by manipulating the original automaton, we search for any reverse prefix of a possible occurrence of the regular expression. Hence, this transformed automaton is a compact device to achieve the same multipattern searching, at much less space.

# 3 Glushkov Automaton

There exist currently many different techniques to build an NFA from a regular expression $RE$ of $m$ characters (without counting the special symbols). The most classical one is the Thompson construction [26], which builds an NFA with at most $2m$ states (and at least $m+1$). This NFA has some particular properties (e.g. $O(1)$ transitions leaving each node) that have been extensively exploited in several regular expression search algorithm such as that of Thompson [26], Myers [19] and Wu and Manber [29, 28].

Another particularly interesting NFA construction algorithm is by Glushkov [12], popularized by Berry and Sethi in [4]. The NFA resulting from this construction has the advantage of having just $m+1$ states (one per position in the regular expression). Its number of transitions is worst case quadratic, but this is unimportant under bit-parallel representations (it just means denser bit masks). We present this construction in depth.

## 3.1 Glushkov Construction

The construction begins by marking the positions of the characters of $\Sigma$ in $RE$, counting only characters. For instance, `(AT|GA)((AG|AAA)*)` is marked $(A_1T_2|G_3A_4)((A_5G_6|A_7A_8A_9)*)$. A *marked expression* from a regular expression $RE$ is denoted $\overline{RE}$ and its language (including the indices on each character) $L(\overline{RE})$. On our example, $L((A_1T_2|G_3A_4)((A_5G_6|A_7A_8A_9)^*)) = \{A_1T_2,\ G_3A_4,\ A_1T_2A_5G_6,\ G_3A_4A_5G_6,\ A_1T_2A_7A_8A_9,\ G_3A_4A_7A_8A_9,\ A_1T_2A_5G_6A_5G_6, \ldots\}$. Let $Pos(\overline{RE})$ be the set of positions in $\overline{RE}$ (i.e., $Pos = \{1 \ldots m\}$) and $\overline{\Sigma}$ the marked character alphabet.

The Glushkov automaton is built first on the marked expression $\overline{RE}$ and it recognizes $L(\overline{RE})$. We then derive from it the Glushkov automaton that recognizes $L(RE)$ by erasing the position indices of all the characters (see below).

The idea of Glushkov is the following. The set of positions is taken as a reference, becoming the set of states of the resulting automaton (adding an initial state 0). So we build $m+1$ states labeled from 0 to $m$. Each state $j$ represents the fact that we have read in the text a string that ends at NFA position $j$. Now if we read a new character $\sigma$, we need to know which positions $\{j_1 \ldots j_k\}$ we can reach from $j$ by $\sigma$. Glushkov computes from a position (state) $j$ all the other accessible positions $\{j_1 \ldots j_k\}$.

We need four new definitions to explain in depth the algorithm. We denote below by $\sigma_y$ the indexed character of $\overline{RE}$ that is at position $y$.

**Definition** $First(\overline{RE}) = \{x \in Pos(\overline{RE}),\ \exists u \in \overline{\Sigma}^*,\ \sigma_x u \in L(\overline{RE})\}$, *i.e. the set of initial positions of* $L(\overline{RE})$, *that is, the set of positions at which the reading can start. In our example,* $First((A_1T_2|G_3A_4)((A_5G_6|A_7A_8A_9)*)) = \{1,3\}$).

**Definition** $Last(\overline{RE}) = \{x \in Pos(\overline{RE}),\ \exists u \in \overline{\Sigma}^*,\ u\sigma_x \in L(\overline{RE})\}$, *i.e. the set of final positions of* $L(\overline{RE})$, *that is, the set of positions at which a string read can be recognized. In our example,* $Last((A_1T_2|G_3A_4)((A_5G_6|A_7A_8A_9)*)) = \{2,4,6,9\}$).

**Definition** $Follow(\overline{RE}, x) = \{y \in Pos(\overline{RE}),\ \exists u, v \in \overline{\Sigma}^*,\ u\sigma_x\sigma_y v \in L(\overline{RE})\}$, *i.e. all the positions in* $Pos(\overline{RE})$ *accessible from* $x$. *For instance, in our example, if we consider position 6, the set of accessible positions* $Follow((A_1T_2|G_3A_4)\ ((A_5G_6|\ A_7A_8A_9)*), 6) = \{7,5\}$.

**Definition** $Empty_{RE}$ *is* TRUE *if* $\varepsilon$ *belongs to* $L(RE)$ *and* FALSE *otherwise.*

The Glushkov automaton $\overline{GL} = (S, \Sigma, I, F, \overline{\delta})$ that recognizes the language $L(\overline{RE})$ is built from these three sets in the following way (Figure 1 shows our example NFA).

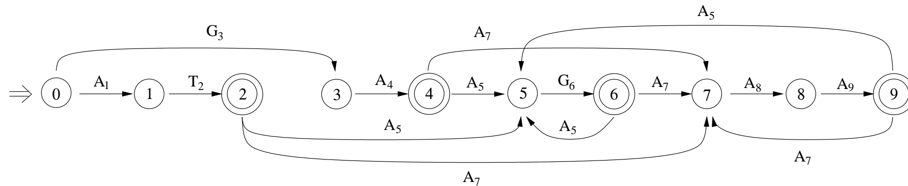

Figure 1: Marked Glushkov automaton built on the marked regular expression $(A_1T_2|G_3A_4)$ $((A_5G_6|A_7A_8A_9)*)$. The state 0 is initial. Double-circled states are final.

1. $S$ is the set of states, $S = \{0, 1, \ldots, m\}$, i.e., the set of positions $Pos(\overline{RE})$ and the initial state is $I = 0$.

2. $F$ is the set of final states, $F = Last(\overline{RE})$ if $Empty_{RE} =$ FALSE and $F = Last(\overline{RE}) \cup \{0\}$ otherwise. Informally, a state (position) $i$ is final if it is in $Last(\overline{RE})$ (in which case when reaching such a position we know that we recognized a string in $L(\overline{RE})$). The initial state 0 is also final if the empty word $\varepsilon$ belongs to $L(\overline{RE})$.

3. $\overline{\delta}$ is the transition function of the automaton, defined by

$$\forall x \in Pos(\overline{RE}),\ \forall y \in Follow(\overline{RE}, x),\ \ \overline{\delta}(x, \sigma_y) = y. \tag{1}$$

Informally, there is a transition from state $x$ to $y$ by $\sigma_y$ if $y$ follows $x$.

The Glushkov automaton of the original $RE$ is now simply obtained by erasing the position indices in the marked automaton. The new automaton recognizes the language $L(RE)$. The Glushkov automaton of our example `(AT|GA)((AG|AAA)*)` is shown in Figure 2.

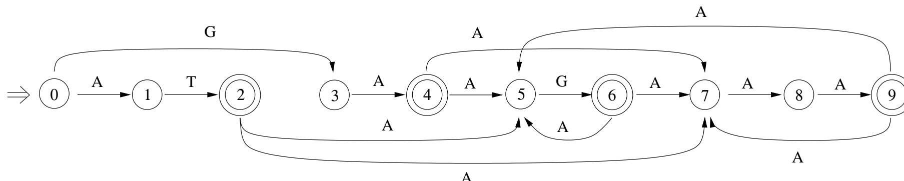


Figure 2: Glushkov automaton built on the regular expression `(AT|GA)((AG|AAA)*)`. The state 0 is initial. Double-circled states are final.

The complexity of this construction is $O(m^3)$, which can be reduced to $O(m^2)$ in different ways by using distinct properties of the $First$ and $Follow$ sets [5, 8]. However, when using bit parallelism, the complexity is directly reduced to $O(m^2)$ by manipulating all the states in a register (see Section 4).

### 3.2 A Bit-parallel Implementation

Let us consider how to compute the variables $First$, $Last$, $Follow$ and $Empty$. All along the Glushkov algorithm we manipulate sets of NFA states. As it is useful for the search algorithms presented later, we will use bit-parallelism to represent these sets of states, that is, we will represent sets using bit masks of $m+1$ bits, where the $i$-th bit is 1 if and only if state number $i$ belongs to the set. We will blur the distinction between sets of states and bit masks in the rest of the paper.

An immediate advantage of using a bit-parallel implementation is that we can easily handle *classes of characters*. This means that at each position of the regular expression there is not just a character of $\Sigma$ but a set of characters, any of which is good to traverse the corresponding arrow. Rather than just converting the set $\{a_1, a_2, \ldots a_k\}$ into $(a_1|a_2|\ldots|a_k)$ (and creating $k$ positions instead of one), we can consider the class as a single letter.

The algorithm of Glushkov is based on the parse tree of the regular expression. Each node $v$ of this tree represents a sub-expression $RE_v$ of $RE$. For each node, its variables $First(v)$, $Last(v)$, and $Empty_v$ are computed in postfix order. At the same time we will fill the values of a global table $Follow$ that corresponds to the whole $RE$. We will consider that regular expressions contain classes of characters rather than single characters at the leaves of their syntax trees.

Figure 3 shows this preprocessing. Together with the above mentioned variables, we fill a table of bit masks $B:\ \Sigma \rightarrow 2^{m+1}$, such that the $i$-th bit of $B[c]$ is set if and only if $c$ belongs to the class at the $i$-th position of the regular expression. We assume that the table is initialized with zeros.

We do not complete the Glushkov algorithm because we do not really use its NFA. Rather, we build directly from its $First$, $Last$ and $Follow$ variables.

**Glushkov_variables**($v_{RE}$, $lpos$)

1. **If** $v = [\,|\,]\,(v_l, v_r)$ OR $v = \boxed{\cdot}\,(v_l, v_r)$ **Then**
2.     $lpos \leftarrow$ **Glushkov_variables**$(v_l, lpos)$
3.     $lpos \leftarrow$ **Glushkov_variables**$(v_r, lpos)$
4. **Else If** $v = \boxed{*}\,(v_*)$ **Then** $lpos \leftarrow$ **Glushkov_variables**$(v_*, lpos)$
5. **End of if**
6. **If** $v = (\varepsilon)$ **Then**
7.     $First(v) \leftarrow 0^{m+1}$, $Last(v) \leftarrow 0^{m+1}$, $Empty_v \leftarrow$ TRUE
8. **Else If** $v = (C)$ , $C \subseteq \Sigma$ **Then**
9.     $lpos \leftarrow lpos + 1$
10.     **For** $\sigma \in C$ **Do** $B[\sigma] \leftarrow B[\sigma] \mid 0^{m-lpos}10^{lpos}$
11.     $First(v) \leftarrow 0^{m-lpos}10^{lpos}$, $Last(v) \leftarrow 0^{m-lpos}10^{lpos}$
12.     $Empty_v \leftarrow$ FALSE , $Follow(lpos) \leftarrow 0^{m+1}$
13. **Else If** $v = [\,|\,]\,(v_l, v_r)$ **Then**
14.     $First(v) \leftarrow First(v_l) \mid First(v_r)$, $Last(v) \leftarrow Last(v_l) \mid Last(v_r)$
15.     $Empty_v \leftarrow Empty_{v_l}$ OR $Empty_{v_r}$
16. **Else If** $v = \boxed{\cdot}\,(v_l, v_r)$ **Then**
17.     $First(v) \leftarrow First(v_l)$, $Last(v) \leftarrow Last(v_r)$
18.     **If** $Empty_{v_l} =$ TRUE **Then** $First(v) \leftarrow First(v) \mid First(v_r)$
19.     **If** $Empty_{v_r} =$ TRUE **Then** $Last(v) \leftarrow Last(v_l) \mid Last(v)$
20.     $Empty_v \leftarrow Empty_{v_l}$ AND $Empty_{v_r}$
21.     **For** $x \in Last(v_l)$ **Do** $Follow(x) \leftarrow Follow(x) \mid First(v_r)$
22. **Else If** $v = \boxed{*}\,(v_*)$ **Then**
23.     $First(v) \leftarrow First(v_*)$, $Last(v) \leftarrow Last(v_*)$, $Empty_v \leftarrow$ TRUE
24.     **For** $x \in Last(v_*)$ **Do** $Follow(x) \leftarrow Follow(x) \mid First(v_*)$
25. **End of if**
26. **Return** $lpos$

Figure 3: Computing the variables for the Glushkov algorithm. The syntax tree can be a union node ($[\,|\,]\,(v_l, v_r)$) or a concatenation node ($\boxed{\cdot}\,(v_l, v_r)$) of subtrees $v_l$ and $v_r$; a Kleen star node ($\boxed{*}\,(v_*)$) with subtree $v_*$, or a leaf node corresponding to the empty string ($\varepsilon$) or a class of characters ($C$).

## 4 A Compact DFA Representation

The classical algorithm to produce a DFA from an NFA consists in making each DFA state represent a set of NFA states which may be active at that point. A possible way to represent the states of a DFA (i.e. the sets of states of an NFA) is to use a bit mask of $O(m)$ bits, as already explained. Previous bit-parallel implementations [19, 29] are built on this idea. We present in this section a new bit-parallel DFA representation based on Glushkov's construction (recall Section 3.2).

### 4.1 Properties of Glushkov's Construction

We study now some properties of the Glushkov construction which are necessary for our compact DFA representation. All them are very easy to prove [6]. The proofs are include here for self-containment.

First, since we do not build any $\varepsilon$-transitions (see Formula (1)), we have that Glushkov's NFA is $\varepsilon$-free. That is, in the approach of Wu and Manber [29], the $\varepsilon$-transitions are the complicated part, because all the others move forward. We do not have these transitions in the Glushkov automaton, but on the other hand the normal transitions do not follow such a simple pattern. However, there are still important structural properties in the arrows. One of these is captured in the following Lemma.

**Lemma 1** *All the arrows leading to a given state in Glushkov's NFA are labeled by the same character. Moreover, if classes of characters are permitted at the positions of the regular expression, then all the arrows leading to a given state in Glushkov's NFA are labeled by the same class.*

*Proof.* This is easily seen in Formula (1). The character labeling every arrow that arrives at state $y$ is precisely $\sigma_y$. This also holds if we consider that $\sigma_y$ is in fact a subset of $\Sigma$. $\square$

These properties can be combined with the $B$ table to yield our most important property.

**Lemma 2** *Let $B(\sigma)$ be the set of positions of the regular expression that contain character $\sigma$. Let $Follow(x)$ be the set of states that can follow state $x$ in one transition, by Glushkov's construction. Let $\delta : S \times \Sigma \rightarrow \mathcal{P}(S)$ be the transition function of Glushkov's NFA, i.e. $y \in \delta(x, \sigma)$ if and only if from state $x$ we can move to state $y$ by character $\sigma$. Then, it holds*

$$\delta(x, \sigma) \quad = \quad Follow(x) \ \cap \ B(\sigma)$$

*Proof.* The lemma follows from Lemma 1. Let state $y \in \delta(x, \sigma)$. This means that $y$ can be reached from $x$ by $\sigma$ and therefore $y \in Follow(x) \cap B(\sigma)$. Conversely, let $y \in Follow(x) \cap B(\sigma)$. Then $y$ can be reached by letter $\sigma$ and it can be reached from $x$. But Lemma 1 implies that every arrow leading to $y$ is labeled by $\sigma$, including the one departing from $x$, and hence $y \in \delta(x, \sigma)$. $\square$

Finally, a last property is necessary for technical reasons made clear shortly.

**Lemma 3** *The initial state 0 in Glushkov's NFA does not receive any arrow.*

*Proof.* This is clear since all the arrows are built in Formula (1), and the initial state is not in the $Follow$ set of any other state (see the definition of $Follow$). $\square$

### 4.2 A Compact Representation

We now use Lemma 2 to obtain a compact representation of the DFA. The idea is to compute the transitions by using two tables: the first one is simply $B[\sigma]$, which is built in algorithm **Glushkov_variables** and gives a bit mask of the states reachable by each letter (no matter from where). The second is a deterministic version of $Follow$, i.e. a table $T$ from sets of states to sets of states (in bit mask form) which tells which states can be reached from an active state in $D$, no matter by which character:

$$T[D] \;=\; \bigcup_{i \in D} Follow(i) \tag{2}$$

By Lemma 2, it holds that

$$\delta(D,\sigma) \;=\; \bigcup_{i \in D} Follow(i) \;\cap\; B(\sigma) \;=\; T[D] \;\&\; B[\sigma]$$

(we are now using the bit mask representation for sets). Hence instead of the complete transition table $\delta :\; 2^{m+1} \times \Sigma \rightarrow 2^{m+1}$ we build and store only $T :\; 2^{m+1} \rightarrow 2^{m+1}$ and $B :\; \Sigma \rightarrow 2^{m+1}$. The number of bits required by this representation is $(m+1)(2^{m+1}+|\Sigma|)$. Figure 4 shows the algorithm to build $T$ from $Follow$ at optimal cost $O(2^m)$.

```
BuildT (Follow, m)
1.    T[0] ← 0
2.    For i ∈ 0...m Do
3.        For j ∈ 0...2^i − 1 Do
4.            T[2^i + j] ← Follow(i) | T[j]
5.        End of for
6.    End of for
7.    Return T
```

Figure 4: Construction of table $T$ from Glushkov's variables. We use a numeric notation for the argument of $T$ and use $Follow$ in bit mask form.

### 4.3 A Search Algorithm

We present now the search algorithm based on the previous construction. Let us call $First$ and $Last$ the variables corresponding to the whole regular expression.

Our first step will be to set $Follow(0) = First$ for technical convenience. Second, we will add a self loop at state 0 which can be traversed by any $\sigma \in \Sigma$. This is because, for searching purposes, the NFA that *recognizes* a regular expression must be converted into one that *searches for* the regular expression. This is achieved by appending $\Sigma^*$ at its beginning, or which is the same, adding a self-loop as described. As, by Lemma 3, no arrow goes to state 0, it still holds that all the arrows leading to a state are labeled the same way (Lemma 1). Figure 5 shows the search algorithm.

```
Search(RE, T = t_1 t_2 ... t_n)
1.    Preprocessing
2.        (v_RE, m) ← Parse(RE) /* parse the regular expression */
3.        Glushkov_variables(v_RE,0) /* build the variables on the tree */
4.        Follow(0) ← 0^m 1 | First /* add initial self-loop */
5.        For σ ∈ Σ Do B[σ] ← B[σ] | 0^m 1
6.        T ← BuildT(Follow, m) /* build T table */
7.    Searching
8.        D ← 0^m 1 /* the initial state */
9.        For j ∈ 1 ... n Do
10.           If D & Last ≠ 0^{m+1} Then report an occurrence ending at j − 1
11.           D ← T[D] & B[t_j] /* simulate transition */
12.       End of for
```

Figure 5: Glushkov-based bit-parallel search algorithm. We assume that **Parse** gives the syntax tree $v_{RE}$ and the number of positions $m$ in $RE$, and that **Glushkov_variables** builds $B$, $First$, $Last$ and $Follow$.

### 4.4 Horizontal Table Splitting

As mentioned, we can split the $T$ table if it turns out to be too large. The splitting is based on the following property. Let $T : 2^{m+1} \rightarrow 2^{m+1}$ be the table obtained applying Eq. (2). Let $D = D_1 : D_2$ be a splitting of mask $D$ into two submasks, a left and a right submask. In our set representation of bit masks, $D_1 = D \cap \{0 \ldots \ell - 1\}$ and $D_2 = D \cap \{\ell \ldots m\}$. If we define $T_1 : 2^\ell \rightarrow 2^{m+1}$ and $T_2 : 2^{m-\ell+1} \rightarrow 2^{m+1}$ using for $T_1[D_1]$ and $T_2[D_2]$ the same formula of Eq. (2), then it is clear that $T[D] = T[D_1] \cup T[D_2]$. In the bit mask view, we have a table $T_1$ that receives the first $\ell$ bits of $D$ and a table $T_2$ that receives the last $m - \ell + 1$ bits of $D$. Each table gives the bit mask of the states reachable from states in their submask. The total set of states is no more than the union of both sets: those reachable from states in $\{0 \ldots \ell - 1\}$ and those reachable from states in $\{\ell \ldots m\}$. Note that the results of each subtable still has $m + 1$ bits.

In general we can split $T$ in $k$ tables $T_1 \ldots T_k$, such that $T_i$ addresses the bits roughly from $(i-1)(m+1)/k$ to $i(m+1)/k - 1$, that is, $(m+1)/k$ bits. Each such table needs $2^{(m+1)/k}(m+1)$ bits to be represented, for a total space requirement of $O(2^{m/k}mk)$ bits. The cost is that, in order to perform each transition, we need to pay for $k$ table accesses so as to compute

$$T[D_1 D_2 \ldots D_k] \quad = \quad T_1[D_1] \mid T_2[D_2] \mid \ \ldots \ \mid T_k[D_k]$$

which makes the search time $O(kn)$ in terms of table accesses. If we have $O(s)$ bits of space, then we solve for $s = 2^{m/k}mk$, to obtain a search time of $O(kn) = O(mn/\log s)$.

### 4.5 Comparison

Compared to Wu and Manber's algorithm [29], ours has the advantage of needing $(m+1)(2^{m+1}+|\Sigma|)$ bits of space instead of their $m(2^{2m+1} + |\Sigma|)$ bits in the worst case (their best case is equal to our

complexity). Just as they propose, we can split $T$ horizontally to reduce space, so as to obtain $O(mn/\log s)$ time with $O(s)$ space. Compared to our previous algorithm [21], the new one compares favorably against its $(m+1)2^{m+1}|\Sigma|$ bits of space. Therefore, our new algorithm should be always preferred over previous bit parallel algorithms.

With respect to a classical DFA implementation, its worst case is $2^{m+1}$ states, and it stores a table which for each state and each character stores the new state. This requires $(m+1)2^{m+1}|\Sigma|$ bits in the worst case. However, in the classical algorithm it is customary to build only the states that can actually be reached, which can be much less than all the $2^{m+1}$ possibilities.

We can do something similar, in the sense of filling only the reachable cells of $T$ (yet, we cannot pack them consecutively in memory as a classical DFA). Figure 6 shows the recursive construction of this table, which is invoked with $D = 0^m1$, the initial state, and assumes that $T$ is initialized with zeros and that $B$, $Follow$ and $m$ are already computed.

```
BuildTrec (D)
1.    For i ∈ 0...m Do /* first build T[D] */
2.        If D & 0^(m−i)10^i ≠ 0^(m+1) Then T[D] ← T[D] | Follow(i)
3.    End of for
4.    N ← T[D]
5.    For σ ∈ Σ Do
6.        If T[N & B[σ]] = 0^(m+1) Then /* not built yet */
7.            BuildTrec (N & B[σ])
8.    End of for
```

Figure 6: Recursive construction of table $T$. We fill only the reachable cells.

Finally, we notice that we do not need to represent state 0, as it is always active. This reduces our space requirement to $m(2^m + |\Sigma|)$ bits of space. This improvement, however, cannot be used in the techniques presented later in this paper.

## 5 The Reverse Factor Search Approach

In this section we describe the general reverse factor search approach currently used to search for single patterns [17, 11, 22] or multiple patterns [10, 25].

The search is done using a window which has the length of the minimum word that we search for (if we search for a single word, we just take its length). We note this minimum length $\ell$.

We shift the window along the text, and for each position of the window, we search backwards (i.e from right to left, see Figure 7) for any factor of any length-$\ell$ prefix of our set of patterns (if we search for a single word, this means any factor of the word). Also, each time we recognize a factor which is indeed a prefix of some of the patterns, we store our position in the window in a variable *last* (which is overwritten, so we know the last time that this happened). Now, two possibilities appear:

(i) We do not reach the beginning of the window. This case is shown in Figure 7. The search

for a factor fails on a letter $\sigma$, i.e $\sigma u$ is not a factor of a length-$\ell$ prefix of any pattern. We can directly shift the window to start at position *last*, since no pattern can start before, and begin the search again.

(ii) We reach the beginning of the window. If we search for just one pattern, we have recognized it and we report the occurrence. Otherwise, we just recognized a length-$\ell$ prefix of one or more patterns. We verify directly in the text if there is a match of a pattern that starts at the initial window position, with a forward (i.e left to right) scan. This can be done with a trie of the patterns. Next, in both cases, we shift the window to position *last*.

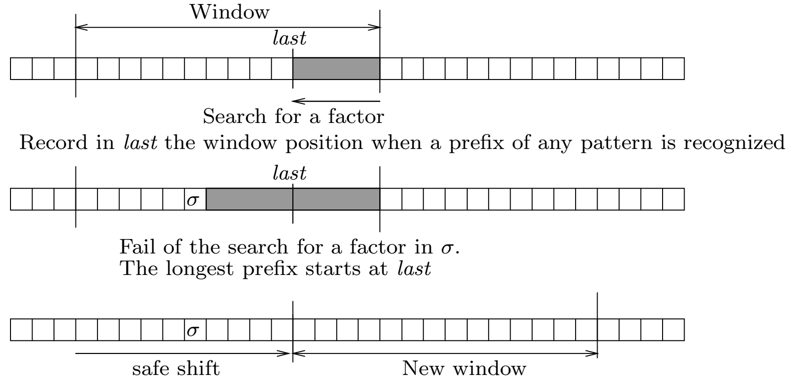


Figure 7: The reverse factor search approach.

This simple approach leads to very fast algorithms in practice, such as BDM [11] and BNDM [22]. For a single pattern, this is optimal on average, matching Yao's bound [30] of $O(n\log(\ell)/\ell)$, where $\ell$ the pattern length. In the worst case, this scheme is quadratic ($O(n\ell)$ complexity). There exists however a general technique to keep the algorithms sub-linear on average and linear in the worst case.

## 5.1 A Linear Worst Case Algorithm

The main idea used in [11, 22, 10, 25] is to avoid retraversing the same characters in the backward window verification. We divide the work done on the text in 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. This is easily achieved with a KMP automaton [16] (for one pattern) or an Aho-Corasick automaton [2] (for multiple patterns). All the matches are found using the forward scan.

However, we need to use also backward searching in order to skip characters. The idea is that the window 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 $\ell$. Usually, a backward scan is attempted when the prefix is less than $\lfloor \ell/\alpha \rfloor$, where $0 < \alpha < \ell$ is fixed arbitrary (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. This case is shown in Figure 8. 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.

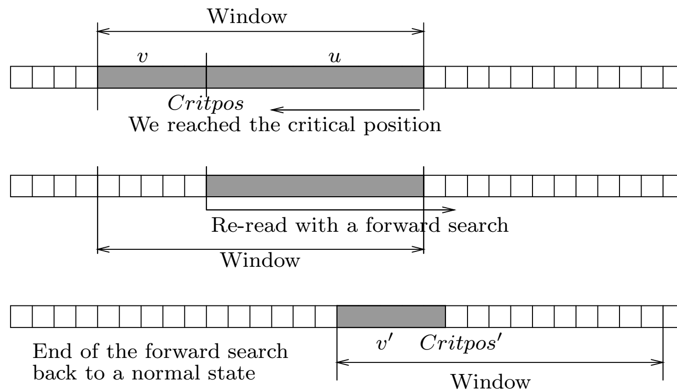


Figure 8: The critical position is reached, in the linear-time algorithm.

(ii) We do not reach the critical position. This case is shown in Figure 9. This means that there cannot be a match in the current window. We start a forward scan from scratch at position *last*, totally retraverse the window, and continue until a new backward scan seems promising.

# 6 A Character Skipping Algorithm

In this section we explain how to adapt the general approach of Section 5 to regular expression searching. We first explain a simple extension of the basic approach and later show how to keep the worst case linear. Recall that we search for a regular expression called $RE$ of size $m$, which generates the language $L(RE)$.

## 6.1 Basic Approach

The search in the general approach needs a window of length $\ell$ (shortest pattern we search for). In regular expression searching this corresponds to the length of the shortest word of $L(RE)$. Of course, if this word is $\varepsilon$, the problem of searching is trivial since every text position matches. We consider in the following that $\ell > 0$.

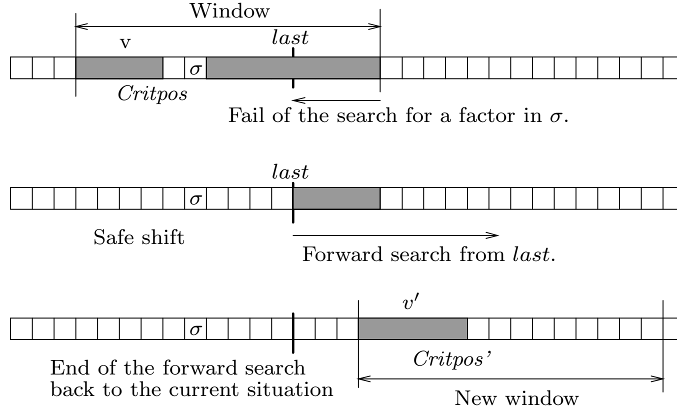


Figure 9: The critical position is not reached, in the linear-time algorithm.

We use the general approach of Section 5, consisting of a backward and, when we reach the beginning of the window, a forward scan. To adapt this scheme to regular expression searching, we need two modifications:

(i) In the backward search step we recognize any factor of the reverse prefixes of length $\ell$ of $L(RE)$. Moreover, we mark in a variable *last* the longest prefix of $L(RE)$ recognized (of course this prefix will not be longer $\ell$).

(ii) The forward search, applied when we reach the beginning of the window, verifies whether there is a match of the regular expression starting at the beginning of the window. However, the occurrence can be much longer than $\ell$).

We detail now the steps of the preprocessing and searching phases.

**Preprocessing** The preprocessing consists of 3 steps:

1. Build the automaton that recognizes $RE$. We note it $F(RE)$, and it is represented by the $B$ and $T$ tables as in Section 4.3. The difference with respect to that automaton is that this time we do not have an initial self-loop.

2. Determine $\ell$ and compute the set $P_i(RE)$ of all the nodes of $F(RE)$ reachable in $i$ steps or less from the initial state, for each $0 \leq i \leq \ell$ (so $P_i(RE) \subseteq P_{i+1}(RE)$). Both values are easily computed with a breadth-first search from the initial state until a final node is reached (being then $\ell$ the current depth at that point).

3. Build the automaton $B(RE)$ that recognizes any factor of the reverse prefixes of length $\ell$ of $L(RE)$. This is achieved by restricting the original automaton $F(RE)$ to the nodes of $P_\ell(RE)$, reversing the arrows, taking as (the only) terminal state the initial state of $F(RE)$, and having all the states as initial states.

```
Compute_P (T, Last)
 1.   P_0 ← 0^m 1 /* the initial state */
 2.   ℓ ← 0
 3.   While P_ℓ & Last = 0^{m+1} Do
 4.       P_{ℓ+1} ← P_ℓ | T[P_ℓ]
 5.       ℓ ← ℓ + 1
 6.   End of while
 7.   Return (ℓ, P)

Reverse_Arrows (Follow)
 8.   For i ∈ 0...m Do
 9.       Precede(i) ← 0^{m+1}
10.       For j ∈ 0...m Do
11.           If Follow(j) & 0^{m-i} 1 0^i ≠ 0^{m+1} Then
12.               Precede(i) ← Precede(i) | 0^{m-j} 1 0^j
13.           End of if
14.       End of for
15.   End of for
16.   Return Precede
```

Figure 10: Functions used in the preprocessing of the backward search.

Figure 10 gives some auxiliary functions for this preprocessing.

Let us focus on $B(RE)$, our device to recognize the reverse factors of prefixes of length $\ell$ of $L(RE)$. It is not hard to see that any such factor corresponds to a path in $F(RE)$ that touches only nodes in $P_\ell(RE)$. In $B(RE)$ there exists the same path with the arrows reversed, and since all the states of $B(RE)$ are initial, there exists a path from an initial state that spells out the reversed factor. Moreover, if the factor is a prefix, then the corresponding path in $B(RE)$ leads to its final state (0).

Note, however, that $B(RE)$ can recognize more words than desired. For instance, if there are loops in $B(RE)$, then it can recognize words longer than $\ell$. However, we can restrict more the set of words recognized by $B(RE)$. The idea is that, if a state of $B(RE)$ is active but it is farther than $i$ positions to the final state of $B(RE)$, and only $i$ window characters remain to be read, then this state cannot lead to a match. Hence, if we have to read $i$ more characters of the window, we intersect the current active states of $B(RE)$ with the set $P_i(RE)$.

It is easy to see that, with this modification, the automaton recognizes exactly the desired prefixes, since if a state has not been "killed" by intersecting it with $P_i(RE)$ it is because it is still possible to obtain a useful prefix from it. Hence, only the desired (reverse) factors can survive all the process until they arrive to the final state and become (reverse) prefixes.

In fact, an alternative method in this sense would be a classical multi-pattern algorithm to recognize the reverse factors of the set of prefixes of length $\ell$ of $L(RE)$. However, this set may be large and the resulting scheme may need much more memory. The automaton $B(RE)$ is a more compact device to obtain the same result.

How to represent $B(RE)$ deserves further consideration. After reversing the arrows of our automaton, it is not anymore the result of Glushkov's construction over a regular expression, and in particular the property that all the arrows arriving at a state being labeled by the same character does not hold anymore. Hence, our bit-parallel simulation of Section 4 cannot be applied.

Fortunately, a dual property holds: all the arrows *leaving* a state are labeled by the same character or class. Hence, if we read a text character $\sigma$, we can first kill the automaton states whose leaving transitions are not labeled by $\sigma$, and then take all the transitions from them. Let $T_b$ be the $T$ mask corresponding to $B(RE)$, and $B$ the mask of $F(RE)$. Then, a transition can be carried out by

$$\delta_b(D, \sigma) \quad = \quad T_b[D \ \& \ B[\sigma]]$$

**Searching** The search follows the general approach of Section 5. For each window position, we activate all the states of $B(RE)$ and traverse the window backwards updating *last* each time the final state of $B(RE)$ is reached (recall that after each step, we "kill" some states of $B(RE)$ using $P_i(RE)$). If $B(RE)$ runs out of active states we shift the window to position *last*. Otherwise, if we reach the beginning of the window, we start a forward scan using $F(RE)$ from the beginning of the window until either a match is found[1], we reached the end of the text, or $F(RE)$ runs out of active states. After the forward scan, we shift the window to position *last*.

Figure 11 shows the search algorithm. As before, large $T$ tables can be split.

### 6.2 Linear Worst Case Extension

We also extended the general linear worst case approach (Section 5.1) to the case of regular expression searching.

We transform the forward scan automaton $F(RE)$ of the previous algorithm by adding a self-loop at its initial state, for each letter of $\Sigma$ (so now it recognizes $\Sigma^* L(RE)$). This is our forward scanning automaton of Section 4.3.

The main difficulty to extend the general linear approach is determining where to place the window in order to not lose a match. The general approach considers the longest prefix of the pattern already recognized. However, this information cannot be inferred only from the active states of the automaton (for instance, it is not known how many times we have traversed a loop). We use an alternative concept: instead of considering the longest prefix already matched, we consider the shortest path to reach a final state. This value can be determined from the current set of states. We devise two different alternatives that differ on the use of this information.

Prior to explaining both alternatives, we introduce some notation. In general, the window is placed so that it finishes $\ell'$ characters ahead of the current text position (for $0 \leq \ell' \leq \ell$). To simplify the exposition, we call this $\ell'$ the "forward-length" of the window.

In the first alternative the forward-length of the window is the shortest path from an active state of $F(RE)$ to a final state (this same idea has been used for multipattern matching in [10]). In this case, we need to recognize any reverse factor of $L(RE)$ in the backward scan (not only

[1]Since we report the beginning of matches, we stop the forward search as soon as we find a match.

```
Backward-Search(RE, T = t_1 t_2 ... t_n)
1.    Preprocessing
2.        (v_RE, m) ← Parse(RE) /* parse the regular expression */
3.        Glushkov_variables(v_RE, 0) /* build the variables on the tree */
4.        Follow(0) ← First
5.        T ← BuildT(Follow, m) /* build T table */
6.        (ℓ, P) ← Compute_P(T, Last)
7.        Precede ← Reverse_Arrows(Follow)
8.        T_b ← BuildT(Precede, m)
9.    Searching
10.       pos ← 0
11.       While pos ≤ n − ℓ Do
12.           j ← ℓ, last ← ℓ
13.           D ← P_ℓ
14.           While D ≠ 0^{m+1} AND j > 0 Do
15.               D ← Tb[D & B[t_{pos+j}]] & P_{j−1}
16.               j ← j − 1
17.               If D & 0^m 1 ≠ 0^{m+1} Then /* prefix recognized */
18.                   If j > 0 Then last ← j
19.                   Else /* check a possible occurrence starting at pos + 1 */
20.                       D ← 0^m 1, i ← pos + 1
21.                       While i ≤ n AND D & Last = 0^{m+1} AND D ≠ 0^{m+1} Do
22.                           D ← T[D] & B[t_i]
23.                       End of while
24.                       If D & Last ≠ 0^{m+1} Then
25.                           Report an occurrence beginning at pos + 1
26.                       End of if
27.                   End of if
28.               End of if
29.           End of while
30.           pos ← pos + last
31.       End of while
```

Figure 11: Backward search algorithm for regular expressions. Do not confuse the bit-mask *Last* of final states with the window position *last* of the last prefix recognized.

the factors of prefixes of length $\ell$)[2]. Each time $\ell'$ is large enough to be promising ($\ell' \geq \alpha\ell$, for some heuristically fixed $\alpha$), we stop the forward scan and start a backward scan on a window of forward-length $\ell'$ (the *critical position* being $\ell - \ell'$). If the backward automaton runs out of active states before reaching the critical position, we shift the window as in the general scheme (using the *last* prefix found) and restart a fresh forward scan. Otherwise, we continue the previous forward scan from the critical position, totally traversing the window and continuing until the condition to start a backward scan holds again.

The previous approach is linear in the worst case (since each text position is scanned at most once forward and at most once backwards), and it is able to skip characters. However, a problem is that *all* the reverse factors of $L(RE)$ have to be recognized, which makes the backward scans longer and the shifts shorter. Also, the window forward-length $\ell'$ is never larger than our previous $\ell$, since the initial state of $F(RE)$ is always active.

The second alternative solves some of these problems. The idea now is that we continue the forward scan until all the active states belong to $P_i(RE)$, for some fixed $i < \ell$ (say, $i = \ell/2$). In this case, the forward-length of the window is $\ell' = \ell - i$, since it is not possible to have a match before reading that number of characters. Again, we select heuristically a minimum $\ell' = \alpha\ell$ value. In this case, we do not need to recognize all the factors. Instead, we can use the already known $B(RE)$ automaton. Note that the previous approach applied to this case (with all active states belonging to $P_i(RE)$) yields different results. In this case we limit the set of factors to recognize, which allows to shift the window sooner. On the other hand, its forward-length is shorter.

# 7 Experimental Results

A major problem when presenting experiments on regular expressions is that there is no concept of a "random" regular expression, so it is not possible to search for, say, 1,000 random patterns. Lacking such good choice, we present two types of experiments. A fist type is a kind of "qualitative test", where we manually generate regular expressions of different characteristics in order to show how these characteristics affect the performance of our and other algorithms. These tests permit one to understand the main factors that influence the performance, but they say little about how the algorithms perform in real life because we focus in showing all the interesting cases rather than on more or less "typical cases" in practice. A second type is a real-life test, where we choose a set of regular expression patterns from a real application and perform a more massive comparison. Another feature of this massive test is that we handle in general larger regular expressions.

We have used three different texts: **DNA**, the DNA sequence of `h.influenzae` with lines cut every 70 characters; **English**, consisting of writings of B. Franklin, filtered to lower-case; and **Proteins**, proteic sequences from the TIGR Database (TDB, `http://www.tigr.org/tdb`), with lines cut every 70 characters. All the texts were replicated so as to obtain exactly 10 Mb.

Our machine is a Sun UltraSparc-1 of 167 MHz, with 64 Mb of RAM, running Solaris 2.5.1. It is a 32 bit machine, so when we use 64 bits masks we are simulating it with two 32-bit native words. We measured CPU times in seconds, averaging 100 runs over the 10 Mb (the variance was

[2] A stricter choice is to recognize any reverse factor of any word of length $\ell'$ that starts at an active state in $F(RE)$, but this needs much more space and preprocessing time.

very low). We include the time for preprocessing in the search figures, except where we show them separately. We show the results in tenths of second per megabyte.

The conclusion of the experiments is that our forward scanning algorithm is better than any competing technique on patterns that do not permit enough character skipping. On the patterns that do, our backward scanning algorithm is in most cases the best choice, although depending on the particularities of the pattern other character skipping techniques may work better. We also show that our algorithms adapt better than the others to long patterns.

A good rule of thumb to determine whether a regular expression $RE$ permits character skipping is to consider the length $\ell$ of the shortest string in $L(RE)$, and then count how many different prefixes of length $\leq \ell$ exist in $L(RE)$. As this number approaches $\sigma^\ell$, character skipping becomes more difficult and a forward scanning becomes a better choice.

### 7.1 Qualitative Tests

We divide this comparison in three parts. First, we compare different existing algorithms to implement an automaton. These algorithms process all the text characters, one by one, and they only differ in the way they keep track of the state of the search. Second, we compare, using our automaton simulation, a simple forward-scan algorithm against the different variants of backward search proposed, to show that backward searching can be faster depending on the pattern. Finally, we compare our backward search algorithm against other algorithms that are also able to skip characters.

For this comparison we have fixed a set of 10 patterns for English and 10 for DNA, which were selected to illustrate different interesting cases, as explained. The patterns are given in Tables 1 and 2. We show their number of letters, which is closely related to the size of the automata recognizing them, the minimum length $\ell$ of a match for each pattern, and a their empirical matching probability (number of matches divided by $n$). The period (.) in the patterns matches any character except the end of line (lines have approximately 70 characters).

<table>
<tr><th>No.</th><th>Pattern</th><th>Size (# letters)</th><th>Minimum length $\ell$</th><th>Prob. match (empirical)</th></tr>
<tr><td>1</td><td>AC((A|G)T)*A</td><td>6</td><td>3</td><td>.015710000</td></tr>
<tr><td>2</td><td>AGT(TGACAG)*A</td><td>10</td><td>4</td><td>.002904000</td></tr>
<tr><td>3</td><td>(A(T|C)G)|((CG)*A)</td><td>7</td><td>1</td><td>.331600000</td></tr>
<tr><td>4</td><td>GTT|T|AG*</td><td>6</td><td>1</td><td>.665100000</td></tr>
<tr><td>5</td><td>A(G|CT)*</td><td>4</td><td>1</td><td>.382500000</td></tr>
<tr><td>6</td><td>((A|CG)*|(AC(T|G))*)AG</td><td>9</td><td>2</td><td>.046700000</td></tr>
<tr><td>7</td><td>AG(TC|G)*TA</td><td>7</td><td>4</td><td>.003886000</td></tr>
<tr><td>8</td><td>[ACG][ACG][ACG][ACG][ACG][ACG]T</td><td>7</td><td>7</td><td>.033150000</td></tr>
<tr><td>9</td><td>TTTTTTTTTT[AG]</td><td>11</td><td>11</td><td>.000001049</td></tr>
<tr><td>10</td><td>AGT.*AGT</td><td>7</td><td>6</td><td>.003109000</td></tr>
</table>

Table 1: The patterns used on DNA.

<table>
<tr><th>No.</th><th>Pattern</th><th>Size<br>(# letters)</th><th>Minimum<br>length ℓ</th><th>Prob. match<br>(empirical)</th></tr>
<tr><td>1</td><td><code>benjamin|franklin</code></td><td>16</td><td>8</td><td>.000035860</td></tr>
<tr><td>2</td><td><code>benjamin|franklin|writing</code></td><td>23</td><td>7</td><td>.000101400</td></tr>
<tr><td>3</td><td><code>[a-z][a-z0-9]*[a-z]</code></td><td>3</td><td>2</td><td>.609200000</td></tr>
<tr><td>4</td><td><code>benj.*min</code></td><td>8</td><td>7</td><td>.000007915</td></tr>
<tr><td>5</td><td><code>[a-z][a-z][a-z][a-z][a-z]</code></td><td>5</td><td>5</td><td>.202400000</td></tr>
<tr><td>6</td><td><code>(benj.*min)|(fra.*lin)</code></td><td>15</td><td>6</td><td>.000035860</td></tr>
<tr><td>7</td><td><code>ben(a|(j|a)*)min</code></td><td>9</td><td>6</td><td>.009491000</td></tr>
<tr><td>8</td><td><code>be.*ja.*in</code></td><td>8</td><td>6</td><td>.000012110</td></tr>
<tr><td>9</td><td><code>ben[jl]amin</code></td><td>8</td><td>8</td><td>.000007915</td></tr>
<tr><td>10</td><td><code>(be|fr)(nj|an)(am|kl)in</code></td><td>14</td><td>8</td><td>.000035860</td></tr>
</table>

Table 2: The patterns used on English text.

### 7.1.1 Forward Scan Algorithms

In principle, any forward scan algorithm can be enriched with backward searching to skip characters. Some are easier to adapt than others, however. In this experiment we only consider the performance of the forward scan methods. The purpose of this test is to evaluate our new forward search algorithm of Section 4.3. We have tested the following algorithms for the forward scanning (see Section 2 for detailed descriptions of previous work). We left aside some algorithms which proved not competitive, at least for the sizes of the regular expressions we are considering: Thompson's [26] and Myers' [19]. This last one is more competitive for larger patterns, as we show in Section 7.2. We have also left aside lazy deterministic automata implementations. However, as we show in Section 7.1.3, these also tend to be slower than ours.

**DFA:** builds the classical deterministic automaton and runs it over the text. We have implemented the scheme using Glushkov's construction, not minimizing the final automaton.

**Agrep:** builds over Thompson's NFA and uses a bit mask to handle the active states [29]. The software [28] is from S. Wu and U. Manber. We forced the best choice of number of subtables.

**Ours-naive:** our new algorithm of Section 4.3, building the whole table $T$ with **BuildT**. We always use one table.

**Ours-optim:** our new algorithm where we build only the $T$ mask for the reachable states, using **BuildTrec**. We always use one table.

The goal of showing two versions of our algorithm is as follows. **Ours-naive** builds the complete $T_d$ table for all the $2^{m+1}$ possible combinations (reachable or not) of active and inactive states. It permits comparing directly against **Agrep** and showing that our technique is superior. **Ours-optim** builds only the reachable states and it permits comparing against **DFA**, the classical algorithm. The disadvantage of **Ours-optim** is that it does not permit splitting the tables (neither does **DFA**), while **Ours-naive** and **Agrep** do.

Tables 3 and 4 show the results on the different patterns, where we have separated preprocessing and search times. As it can be seen, **Ours-naive** compares favorably in search time against **Agrep**, scanning (averaging over the 20 patterns) 16.0 Mb/sec versus about 13.2 Mb/sec of **Agrep**. In some patterns they are very close and **Agrep** wins sometimes by a few milliseconds (except English pattern # 3), but in the rest our approach is much faster. **Ours-naive** works quite well except on large patterns such as the English pattern #2. **Ours-optim** behaves well in those situations too, and always compares favorably against the classical **DFA**, which scans the text at 14.4 Mb/sec. This means that our new algorithm is at least 10% faster than any alternative approach.

In all cases, searching for larger expressions costs more, both in preprocessing and in search time because of locality of reference. Note that **Ours-optim** is sometimes worse than **Ours-naive**. This occurs when most states are reachable, in which case **Ours-naive** fills them all without the overhead of the recursion. But this only happens when the preprocessing time is negligible.

| Pattern | DFA | Agrep | Ours-naive | Ours-optim |
|---|---|---|---|---|
| 1 | $0.034 + 0.643n$ | $0.104 + 0.756n$ | $0.009 + 0.584n$ | $0.005 + \mathbf{0.575n}$ |
| 2 | $0.006 + 0.624n$ | $0.133 + 0.754n$ | $0.049 + \mathbf{0.566n}$ | $0.000 + 0.576n$ |
| 3 | $0.028 + 0.796n$ | $0.095 + \mathbf{0.758}n$ | $0.007 + 0.759n$ | $0.087 + 0.775n$ |
| 4 | $0.025 + 0.883n$ | $0.101 + \mathbf{0.760}n$ | $0.012 + 0.788n$ | $0.029 + 0.807n$ |
| 5 | $0.018 + 0.831n$ | $0.089 + 0.757n$ | $0.005 + \mathbf{0.755}n$ | $0.008 + 0.777n$ |
| 6 | $0.007 + 0.658n$ | $0.126 + 0.762n$ | $0.014 + \mathbf{0.584}n$ | $0.004 + 0.592n$ |
| 7 | $0.004 + 0.634n$ | $0.104 + 0.750n$ | $0.015 + 0.571n$ | $0.040 + \mathbf{0.567}n$ |
| 8 | $0.004 + 0.646n$ | $0.101 + 0.831n$ | $0.008 + 0.583n$ | $0.071 + \mathbf{0.582}n$ |
| 9 | $0.006 + 0.621n$ | $0.096 + 0.694n$ | $0.005 + 0.568n$ | $0.007 + \mathbf{0.565}n$ |
| 10 | $0.038 + 0.639n$ | $0.108 + 0.748n$ | $0.011 + \mathbf{0.560}n$ | $0.036 + 0.562n$ |

Table 3: Forward search times on DNA. The times have the form $a+bn$, where $a$ is the preprocessing time and $b$ is the search time per megabyte, all in tenths of seconds.

| Pattern | DFA | Agrep | Ours-naive | Ours-optim |
|---|---|---|---|---|
| 1 | $0.010 + 0.633n$ | $0.114 + 0.779n$ | $0.074 + 0.569n$ | $0.009 + \mathbf{0.563}n$ |
| 2 | $0.022 + 0.629n$ | $0.112 + 1.583n$ | $20.61 + 0.575n$ | $0.019 + \mathbf{0.569}n$ |
| 3 | $0.003 + 0.932n$ | $0.106 + \mathbf{0.769}n$ | $0.007 + 0.856n$ | $0.022 + 0.898n$ |
| 4 | $0.068 + 0.639n$ | $0.100 + 0.755n$ | $0.008 + \mathbf{0.567}n$ | $0.000 + 0.578n$ |
| 5 | $0.009 + 0.879n$ | $0.095 + 0.871n$ | $0.050 + \mathbf{0.664}n$ | $0.000 + 0.684n$ |
| 6 | $0.242 + 0.645n$ | $1.494 + 0.775n$ | $0.043 + 0.569n$ | $0.013 + \mathbf{0.567}n$ |
| 7 | $0.007 + 0.631n$ | $0.122 + 0.755n$ | $0.006 + \mathbf{0.572}n$ | $0.001 + 0.578n$ |
| 8 | $0.081 + 0.628n$ | $0.103 + 0.755n$ | $0.048 + 0.562n$ | $0.027 + \mathbf{0.556}n$ |
| 9 | $0.000 + 0.627n$ | $0.102 + 0.704n$ | $0.011 + 0.567n$ | $0.002 + \mathbf{0.565}n$ |
| 10 | $0.012 + 0.632n$ | $0.774 + 0.789n$ | $0.018 + 0.567n$ | $0.005 + \mathbf{0.561}n$ |

Table 4: Forward search times on English. The times have the form $a + bn$, where $a$ is the preprocessing time and $b$ is the search time per megabyte, all in tenths of seconds.

#### 7.1.2 Forward Versus Backward Scanning

We compare now our new forward scan algorithm (called **Fwd** in this section and **Ours-optim** in Section 7.1.1) against backward scanning. There are three backward scanning algorithms. The simplest one, presented in Section 6.1, is called **Bwd**. The two linear variations presented in Section 6.2 are called **LBwd-All** (that recognizes all the reverse factors) and **LBwd-Pref** (that recognizes reverse factors of length-$\ell$ prefixes). The linear variations depend on an $\alpha$ parameter, which is between 0 and 1. We have tested the values 0.25, 0.50 and 0.75 for $\alpha$. We have built only the reachable entries of their $T_b$ tables.

Tables 5 and 6 show the results. The improvements are modest on the DNA patterns we selected. This is a consequence of their shortness and high probability of occurrence (in particular, the method makes little sense if the minimum length is 1, and therefore patterns 3 to 5 were removed). However, from minimum length 4 or more the backward search becomes competitive (with the exception of pattern 8, because it matches with very high probability).

| Pattern | Fwd | Bwd | LBwd-All | | | LBwd-Pref | | |
|---|---|---|---|---|---|---|---|---|
| | | | $\alpha = 0.25$ | $\alpha = 0.50$ | $\alpha = 0.75$ | $\alpha = 0.25$ | $\alpha = 0.50$ | $\alpha = 0.75$ |
| 1 | **0.576** | 0.823 | 1.931 | 2.058 | 2.112 | 1.764 | 1.854 | 1.911 |
| 2 | **0.576** | 0.658 | 1.653 | 1.648 | 1.640 | 1.478 | 1.491 | 1.534 |
| 6 | **0.592** | 2.066 | 2.572 | 3.164 | 3.147 | 2.512 | 3.043 | 3.013 |
| 7 | **0.571** | 0.710 | 1.534 | 1.530 | 1.568 | 1.564 | 1.572 | 1.601 |
| 8 | **0.589** | 1.416 | 1.669 | 1.702 | 1.707 | 1.804 | 1.818 | 1.870 |
| 9 | 0.566 | **0.265** | 0.591 | 0.573 | 0.559 | 0.671 | 0.669 | 0.668 |
| 10 | **0.566** | 1.086 | 1.927 | 1.949 | 2.240 | 1.593 | 1.591 | 1.909 |

Table 5: Backward search times on DNA, in seconds for 10 Mb. Preprocessing times are included.

The results are much better on our English patterns, where we obtained improvements in half of the patterns. In general, the linear versions are quite bad in comparison with the simple one, although in some cases they are faster than a forward search. It is difficult to determine which of the two versions is better in which cases, and which is the best value for $\alpha$.

#### 7.1.3 Character Skipping Algorithms

Finally, we consider other algorithms able to skip characters. Basically, the other algorithms are based in extracting one or more strings from the regular expression, so that some of those strings must appear in any match. A single- or multi-pattern exact search algorithm is then used as a filter, and only where some string in the set is found, its neighborhood is checked for an occurrence of the whole regular expression. Two approaches exist:

**Single pattern:** one string is extracted from the regular expression, so that the string must appear inside every match. If this is not possible the scheme cannot be applied. We use *Gnu Grep v2.4*, which implements this idea. Where the filter cannot be applied, *Grep* uses a forward scanning algorithm based on a lazy deterministic automaton (i.e., built on the fly as the text

| Pattern | Fwd | Bwd | LBwd-All | | | LBwd-Pref | | |
|---|---|---|---|---|---|---|---|---|
| | | | $\alpha = 0.25$ | $\alpha = 0.50$ | $\alpha = 0.75$ | $\alpha = 0.25$ | $\alpha = 0.50$ | $\alpha = 0.75$ |
| 1 | 0.564 | **0.210** | 0.442 | 0.432 | 0.463 | 0.468 | 0.489 | 0.499 |
| 2 | 0.571 | **0.294** | 1.173 | 1.001 | 1.092 | 0.932 | 0.981 | 0.952 |
| 3 | **0.900** | 2.501 | 3.303 | 2.591 | 3.012 | 2.559 | 2.562 | 2.561 |
| 4 | **0.578** | 0.589 | 1.702 | 1.684 | 1.712 | 0.941 | 0.928 | 0.917 |
| 5 | **0.684** | 1.951 | 2.051 | 2.404 | 2.094 | 2.128 | 2.147 | 2.177 |
| 6 | **0.568** | 0.725 | 1.821 | 1.846 | 1.842 | 1.009 | 1.122 | 1.092 |
| 7 | 0.578 | **0.250** | 0.459 | 0.447 | 0.471 | 0.511 | 0.519 | 0.510 |
| 8 | **0.559** | 0.934 | 1.748 | 1.855 | 1.871 | 1.333 | 1.447 | 1.470 |
| 9 | 0.565 | **0.184** | 0.373 | 0.368 | 0.391 | 0.413 | 0.393 | 0.411 |
| 10 | 0.562 | **0.216** | 0.421 | 0.454 | 0.438 | 0.468 | 0.461 | 0.483 |

Table 6: Backward search times on English, in seconds for 10 Mb.

is read). Hence, we plot this value only where the idea can be applied. We point out that *Grep* also abandons a line when it finds the first match in it.

**Multiple patterns:** this idea was presented in [27]. A length $\ell' < \ell$ is selected, and all the possible suffixes of length $\ell'$ of $L(R)$ are generated and searched for. The choice of $\ell'$ is not obvious, since longer strings make the search faster, but there are more of them. Unfortunately, the code of [27] is not public, so we have used the following procedure: first, we extract by hand the suffixes of length $\ell'$ for each regular expression; then we use the multipattern search of *Agrep* [28], which is very fast, to search for those suffixes; and finally the matching lines are sent to *Grep*, which checks the occurrence of the regular expression in the matching lines. We find by hand the best $\ell'$ value for each regular expression. The resulting algorithm is quite similar to the idea of [27].

Tables 7 and 8 show the results. The single pattern filter is a very effective trick, but it can be applied only in a restricted set of cases. It improves over **Bwd** in a couple of cases, only one really significant. The multipattern filter, on the other hand, is more general, but its times are higher than ours in general, especially where backward searching is better than forward searching. Note that, in general, **Bwd** is the best whenever it is better than **Fwd**.

### 7.2 A Real-Life Test

It is not hard to find dozens of examples of regular expression patterns used in real life. A quick overview on the Web shows patterns to recognize URLs, email addresses, email fields, IP addresses, phone numbers, zip codes, HTML tags, dates, hours, floating point numbers, programming language variables, comments, and so on. In general, these patterns are rather small and we have obtained the same performance as for our short frequently-matching patterns of the previous sections. This shows that many real-life patterns would behave as our examples on DNA.

We would like to consider other applications where the above conditions do not hold. We have focused on a computational biology application related to peptidic site searching, which permits us

| Pattern | Fwd | Bwd | Single pattern filter | Multipattern filter |
|---|---|---|---|---|
| 1 | **0.576** | 0.823 | 1.231 | 2.282 |
| 2 | **0.576** | 0.658 | 0.919 | 1.203 |
| 6 | **0.592** | 2.066 | 1.060 | 2.248 |
| 7 | **0.571** | 0.710 | 1.331 | 1.650 |
| 8 | **0.589** | 1.416 | 1.162 | 2.104 |
| 9 | 0.566 | 0.265 | **0.202** | 0.310 |
| 10 | **0.566** | 1.022 | 0.932 | 1.833 |

Table 7: Algorithm comparison on DNA, in seconds for 10 Mb.

| Pattern | Fwd | Bwd | Single pattern filter | Multipattern filter |
|---|---|---|---|---|
| 1 | 0.564 | **0.210** | — | 0.309 |
| 2 | 0.571 | **0.294** | — | 0.371 |
| 3 | **0.900** | 2.501 | — | 1.648 |
| 4 | 0.578 | 0.589 | **0.171** | 0.873 |
| 5 | **0.684** | 1.951 | — | 2.024 |
| 6 | **0.568** | 0.725 | — | 1.003 |
| 7 | 0.578 | **0.250** | 0.260 | 0.442 |
| 8 | **0.559** | 0.934 | 0.632 | 0.661 |
| 9 | 0.565 | **0.184** | 0.193 | 0.307 |
| 10 | 0.562 | **0.216** | 0.983 | 0.348 |

Table 8: Algorithm comparison on English, in seconds for 10 Mb.

experimenting with larger patterns.

PROSITE is a well known set of patterns used for protein searching [13]. Proteins are regarded as texts over a 20-letter upper case alphabet. PROSITE patterns are formed by the following items: $(i)$ classes of characters, denoted by the set of characters they match, for example $[AKDE]$ is equivalent to $(A|K|D|E)$; $(ii)$ bounded length gaps, denoted by $x(a,b)$, which matches any string of length $a$ to $b$, for example $x(1,3)$ is equivalent to $\Sigma(\Sigma|\varepsilon)(\Sigma|\varepsilon)$. Except for very recent developments [24], PROSITE patterns are searched for as regular expressions, so this is quite a real application in computational biology.

Not every PROSITE pattern is interesting for our purposes. In particular, many of them have no gaps and hence are in fact linear patterns, very easy to search with much simpler algorithms [3, 22]. We also removed 8 patterns having more than 64 states because they were too few to be representative of these large NFAs. The result is 329 PROSITE patterns. Those patterns occur too frequently (classes are large and gaps are long) to permit any backward search approach, so we consider only forward searching in this experiment.

Our forward scanning algorithm is that of Section 4.3. To accommodate up to 64 states, we have used a couple of integers (or a long integer) to hold the bit mask of 64 bits. This time we

have horizontally split the tables into subtables, so the preprocessing is done using **BuildT**. The width of the subtables is limited to 16 bits, so the number of subtables is $\lceil m/16 \rceil$, ranging from 1 to 4. For 1 and 2 subtables we use specialized code, while the cases of 3 and 4 tables are handled with a loop sentence. On the other hand, using bit masks of 32 bits makes all the operations much faster for the shorter patterns that can be accommodated in 32 bits. So, when comparing against algorithms that cannot handle more than 32 NFA states, we use our 32-bit-masks version; and when comparing against other algorithms we use our slower 64-bit-masks version.

Figure 12 shows the search time of our algorithm as a function of Glushkov's NFA size (that is, $m+1$), both for 32-bit and 64-bit masks. Clear jumps can be seen when moving to more subtables. The difference between specialized code for up to 2 tables is also clear. Inside homogeneous areas, the increase in time corresponding to having larger subtables is also clear.

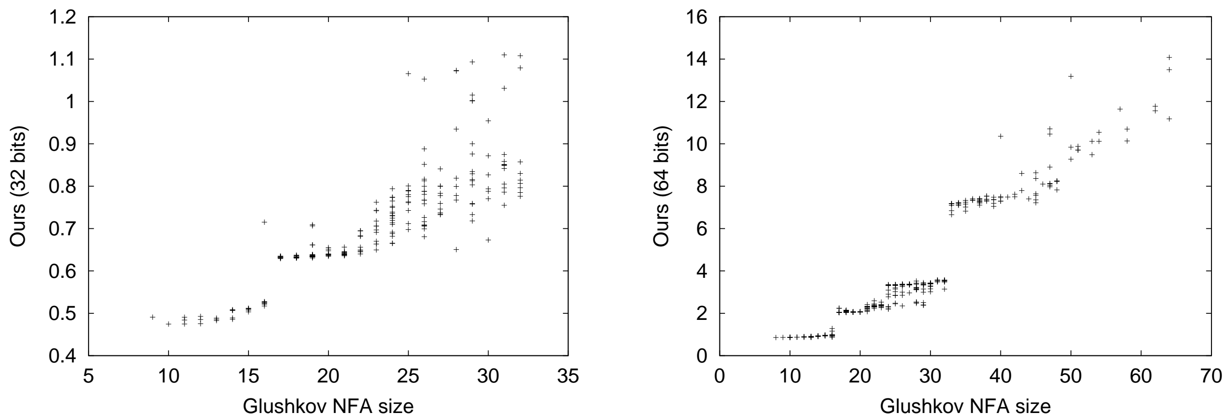


Figure 12: Our search times as a function of the NFA size $(m+1)$, using 32-bit masks (left) and 64-bit masks (right).

Let us compare those times against other approaches. We start with the full DFA implementation, that is, the DFA is completely built and later used for searching. The code was written by ourselves and built over Glushkov's NFA. We removed all the patterns that requested more than 64 Mb of memory for the automaton because they needed more than 10 minutes to be searched for. This left us with only 215 patterns. Figure 13 shows the relative performance as a function of the number of states in the NFA. We are considering only NFAs of up to 32 states in this case, so we compare against our 32-bit version. As it can be seen, our technique is always faster than a full DFA implementation, which takes an unreasonable amount of time for a significant number of patterns as their length grow. At the best moment of the full DFA implementation, it takes about 1.5 times the time we need for the search.

Only 7 patterns of more than 32 states can be handled by the full DFA implementation using less than 64 Mb. From these 7 patterns, 6 were 4 to 12 times slower than our 64-bit version, so we can say that this method cannot handle in general patterns of more than 32 states. That is, the full DFA method does not generalize to long regular expressions, and for short ones, we are clearly faster.

In order to test an external DFA implementation, we considered *Grep*. Since *Grep* took more

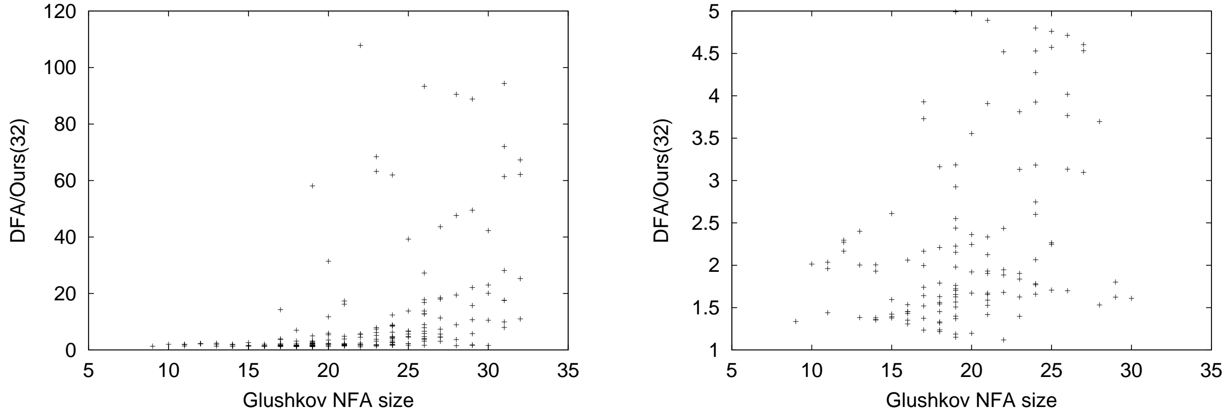


Figure 13: Comparison on PROSITE patterns between our algorithm and the full DFA implementation for NFAs of up to 32 states. On the right we show in detail the patterns where the full DFA implementation is up to 5 times slower than our algorithm.

than 10 minutes on NFAs of more than 20 states, we considered only those of up to 20 states. This left us with 98 patterns. We compared against our 32-bit version, as it was more than enough for the patterns that *Grep* could handle.

Figure 14 shows the results. As it can be seen, there were still 5 patterns, some with as few as 17 states, which were more than 40 times slower than our algorithm. For the majority, we are about 1.3 times faster for short patterns and about 1.1 times faster for patterns needing two table accesses. In 3 cases *Grep* was up to 5% faster than us. Observe, however, that *Grep* cannot handle longer patterns and that, even in the area shown, it can give rise to surprises with exceedingly long execution times.

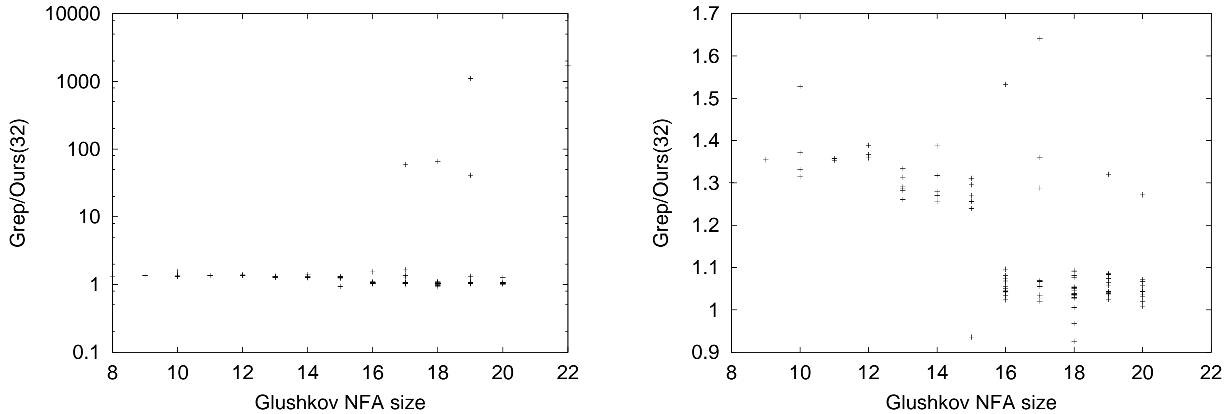


Figure 14: Comparison on PROSITE patterns between our algorithm and *Gnu Grep* for NFAs of up to 20 states. On the right we show in detail the patterns where *Gnu Grep* implementation is up to 2 times slower than our algorithm.

Let us now compare our algorithm against Myers' [19]. The code, from G. Myers, builds an NFA of DFAs based on Thompson's construction. The main purpose of this comparison is to compare two different mechanisms to handle large patterns, so we have used only the 64-bit version of our code. As it can be seen in Figure 15 (left), our table splitting technique is not only a simpler, but also a more efficient, mechanism to handle longer patterns than Myers'. Our technique is 2 to 5 times faster for NFAs of up to 32 states (where we handcode the use of 1 or 2 tables), and about 1.5 times faster for up to 64 states, where we use a normal loop to update the bit mask.

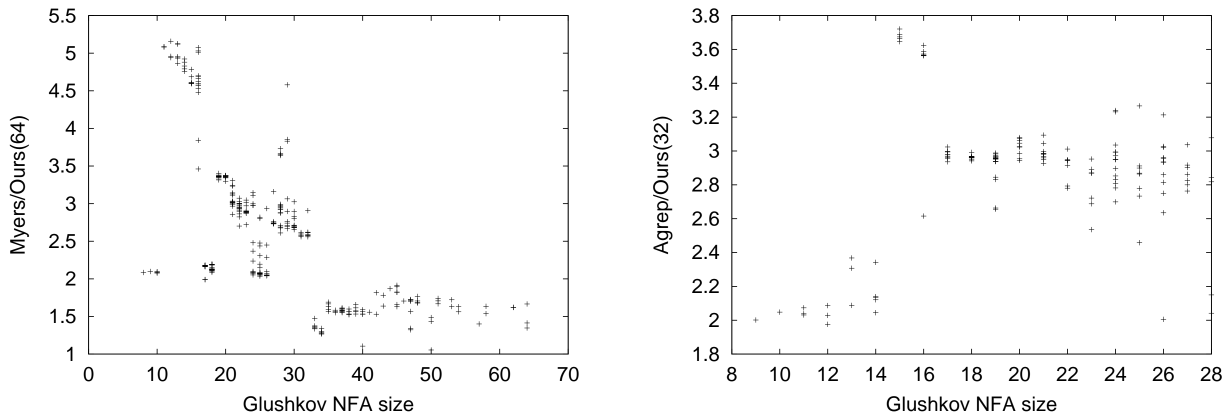


Figure 15: Comparison between ours and Myers' algorithm (left) and Agrep (right) for PROSITE patterns. Our algorithm uses masks of 64 bits against Myers and 32 bits against Agrep.

Finally, we show a comparison against *Agrep* software [29, 28] by S. Wu and U. Manber, where we have respected its original decisions about splitting tables. **Agrep** is limited to Thompson NFAs of less than 32 states, so only some patterns could be tested and we only show our 32-bit code. Figure 15 (right) shows the comparison, where it can be seen that we are twice as fast on short patterns and around 3 times faster for longer patterns. This shows the superiority of Glushkov's over Thompson's based constructions.

It is important to notice that the results on bit-parallel algorithms improve with technological progress, whereas those on classical algorithms do not. For example, if we ran our experiments on a 64-bit machine, our search times on 64-bit masks would be as good as those on 32-bit masks (apart from the effects on the number of tables used).

# 8 Conclusions

We have presented new techniques and algorithms for faster regular expression searching. We have shown how to represent a deterministic finite automaton (DFA) in a compact form and how to manipulate an automaton in order to permit skipping text characters at search time. These techniques yield fast and simple search algorithms, which we have shown experimentally to be faster than previous approaches in most cases.

Bit-parallelism has been an essential tool in recent research on regular expression searching, and our work is not an exception. The flexibility of bit-parallelism permits extending the search problem,

for example to permit a limited number of errors in the occurrences of the regular expression, and it has been demonstrated in pattern matching software such as *Agrep* [28] and *Nrgrep* [20]. The latter is built on the ideas presented in this paper. Also, it extends *Grep*'s idea of selecting a necessary string, so it chooses the most promising subgraph of the NFA for the scanning.

The preprocessing time is a subject of future work. In our experiments the patterns were reasonably short and the simple technique of using one transition table was the best choice. However, longer patterns would need the use of the table splitting technique, which worsens the search times. Also, it seems difficult to split the table when we are trying to build only the reachable states.

Finally, we point out that there is some recent work on reducing the number of states of NFAs [7, 15], and on restricting these types of reductions in order to retain the useful properties of Glushkov construction (especially that of Lemma 2), which permits applying the compact DFA representation we propose in this paper [14].

## References


[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

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

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

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

[6] J.-M. Champarnaud. Subset construction complexity for homogeneous automata, position automata and ZPC-structures. *Theoretical Computer Science*, 267:17–34, 2001.

[7] J.-M. Champarnaud and F. Coulon. NFA reduction algorithms by means of regular inequalities. In *Proc. DLT 2003*, LNCS 2710, pages 194–205, 2003.

[8] C.-H. Chang and R. Paige. From regular expression to DFA's using NFA's. In *Proc. of the 3rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS v. 664, pages 90–110, 1992.

[9] B. Commentz-Walter. A string matching algorithm fast on the average. In *Proc. of the 6th International Colloquium on Automata, Languages and Programming (ICALP)*, number 6 in LNCS, pages 118–132. Springer-Verlag, 1979.

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

[11] 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.

[12] V-M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1961.

[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] L. Ilie, G. Navarro, and S. Yu. On NFA reductions. In *Theory is Forever (Salomaa Festschrift)*, LNCS 3113, pages 112–124. Springer-Verlag, Heidelberg, 2004.

[15] L. Ilie and S. Yu. Reducing NFAs by invariant equivalences. *Theoretical Computer Science*, 306:373–390, 2003.

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

[17] T. Lecroq. Experimental results on string matching algorithms. *Software Practice and Experience*, 25(7):727–765, 1995.

[18] M. Mohri. String matching with automata. *Nordic Journal of Computing*, 4(2):217–231, 1997.

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

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

[21] G. Navarro and M. Raffinot. Fast regular expression search. In *Proceedings of the 3rd Workshop on Algorithm Engineering (WAE'99)*, LNCS v. 1668, pages 199–213, 1999.

[22] 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`.

[23] G. Navarro and M. Raffinot. Compact DFA representation for fast regular expression search. In *Proceedings of the 5th Workshop on Algorithm Engineering (WAE'01)*, LNCS 2141, pages 1–12, 2001.

[24] G. Navarro and M. Raffinot. Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. *Journal of Computational Biology*, 10(6):903–923, 2003. Earlier version in *Proc. RECOMB 2001*.

[25] Mathieu Raffinot. On the multi backward dawg matching algorithm (MultiBDM). In *Proc. of the 4th South American Workshop on String Processing (WSP)*, pages 149–165. Carleton University Press, 1997.

[26] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[27] B. Watson. *Taxonomies and Toolkits of Regular Language Algorithms.* Phd. dissertation, Eindhoven University of Technology, The Netherlands, 1995.

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

[29] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, October 1992.

[30] A. Yao. The complexity of pattern matching for a random string. *SIAM Journal on Computing*, 8:368–387, 1979.