Compact DFA Representation for Fast Regular Expression Search*

Gonzalo Navarro[1] and Mathieu Raffinot[2]

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

[2] Equipe génome, cellule et informatique, Université de Versailles, 45 avenue des Etats-Unis, 78035 Versailles Cedex, E-mail: raffinot@genetique.uvsq.fr.

Abstract. We present a new technique to encode a deterministic finite automaton (DFA). Based on the specific properties of Glushkov's nondeterministic finite automaton (NFA) construction algorithm, we are able to encode the DFA using $(m+1)(2^{m+1}+|\Sigma|)$ bits, where m is the number of characters (excluding operator symbols) in the regular expression and Σ is the alphabet. This compares favorably against the worst case of $(m+1)2^{m+1}|\Sigma|$ bits needed by a classical DFA representation and $m(2^{2m+1}+|\Sigma|)$ bits needed by the Wu and Manber approach implemented in *Agrep*.
Our approach is practical and simple to implement, and it permits searching regular expressions of moderate size (which include most cases of interest) faster than with any previously existing algorithm, as we show experimentally.

1 Introduction and Related Work

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 RE our regular expression, which is of length m. This means that m is the total number of characters in RE, not counting operators symbols "|", "*" and parentheses. We note $L(RE)$ the set of words generated by RE and Σ the alphabet.

The traditional technique [10] to search a regular expression of length 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. 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 $O(2^m)$ states, which implies a preprocessing cost and extra space exponential in m.

Some techniques have been proposed to obtain a good tradeoff between both extremes. In 1992, Myers [7] presented a four-russians approach which obtains $O(mn/\log n)$ worst-case time and 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. 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

* Partially supported by ECOS-Sud project C99E04 and, for the first author, Fondecyt grant 1-990627.

(lower) positions in an amount given by the right argument. 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. $0^3 1 = 0001$.

In our case, the vector of active and inactive states is stored as bits of a computer word. Instead of (ala Thompson [10]) 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 on this direction, one may resort to pure bit-parallelism and forget about the modules. This was done in [13] by Wu and Manber, and included in their software *Agrep* [12]. 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 [10] is used, then all the arrows carry 1's from bit positions i to $i+1$, except for the ε-transitions. Then, a generalization of Shift-Or [2] (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 $B : \Sigma \to 2^{O(m)}$, 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 ε-transitions are carried out. As ε-transitions follow arbitrary paths, a table $E : 2^{O(m)} \to 2^{O(m)}$ is precomputed, where $E[D]$ is the ε-closure of D. To move from the state set D to the new D' after reading text character c, the action is

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

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, much less known, NFA construction algorithm is Glushkov's [6, 3]. 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 ε-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 $m(2^{2m+1} + |\Sigma|)$ bits (E plus B tables).

In [8], we proposed the use of Glushkov's construction instead of Thompson's. The table had then 2^{m+1} entries, but unfortunately the structural property that arrows were either forward or ε-transitions did not hold anymore. As a result, we needed a table $M : 2^{m+1} \times \Sigma \to 2^{m+1}$ indexed by the current state and text character, for a total space requirement of $(m+1)2^{m+1}|\Sigma|$ bits. The transition action was simply $D' \leftarrow M[D, c]$, just as for a classical DFA implementation. We showed experimentally that the Glushkov based construction was normally faster than the one based on Thompson, but not better than a classical DFA.

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.

The net result is a simple algorithm for regular expression searching which uses normally less space and has faster preprocessing and search time (albeit all are $O(n)$ search time, a smaller DFA representation implies more locality of reference). We show experimentally that we are at least 10% faster than any previous algorithm for searching regular expressions of moderate size, which include most cases of interest.

The algorithms reviewed are called "forward scanning" algorithms because they inspect all the text characters, one by one. It should be noted that there exist algorithms able to skip text characters, which

are based on discarding text areas that cannot contain a match and using a classical algorithm on the rest. Example of these algorithms are that based on multi pattern matching (Watson [11, chapter 5]), on filtering using necessary substrings (*Gnu Grep v2.0*) and on reversing the arrows of the DFA to search reversed factors of the regular expression (Navarro and Raffinot [8]). Those algorithms are in some cases faster than ours, but all them need a forward scan algorithm to search the text areas that cannot be discarded. Hence, a better forward scanning algorithm is always welcome. Moreover, many interesting regular expressions cannot be efficiently searched using backward scanning algorithms. In particular, we show how to use our compact representation to obtain an improved version of our previous algorithm in [8].

2 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 [10], 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 [10], Myers [7] and Wu and Manber [13, 12].

Another particularly interesting NFA construction algorithm is by Glushkov [6], popularized by Berry and Sethi in [3]. 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 our bit-parallel representation (it just means denser bit masks). We present this construction in depth.

2.1 Glushkov construction

The construction begins by marking the positions of the characters of Σ 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 σ, we need to know which positions $\{j_1 \ldots j_k\}$ we can reach from j by σ. 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 σ_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 *$Follow(x) = \{y \in Pos(\overline{RE}),\ \exists u, v \in \overline{\Sigma}^*,\ u\sigma_x\sigma_y v \in L(\overline{RE})\}$, 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 *The function $Empty_{RE}$ is $\{\varepsilon\}$ if ε belongs to $L(RE)$ and $\emptyset$ 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).

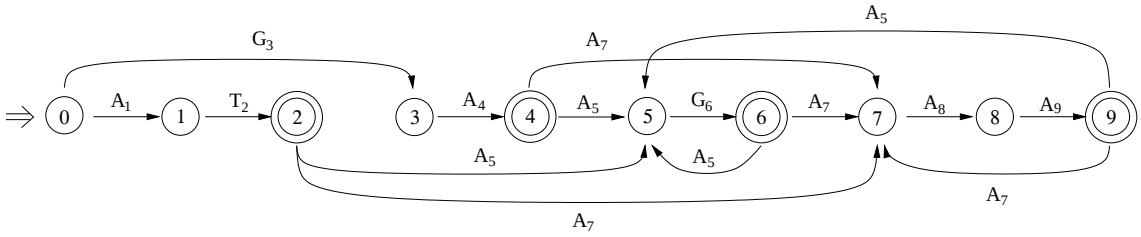

Fig. 1. Marked Glushkov automaton built on the marked regular expression $(A_1T_2|G_3A_4)((A_5G_6|A_7A_8A_9)*)$. The state 0 is initial. Double-circled states are final.

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, $F = Last(\overline{RE}) \cup (Empty_{RE} \cdot \{0\})$. 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 ε belongs to $L(\overline{RE})$, in which case $Empty_{RE} = \{\varepsilon\}$ and hence $Empty_{RE} \cdot \{0\} = \{0\}$. If not, $Empty_{RE} = Empty_{RE} \cdot \{0\} = \emptyset$.
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 σ_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.

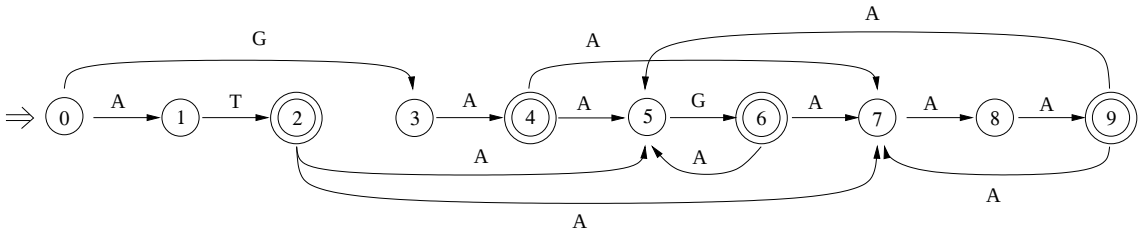

Fig. 2. Glushkov automaton built on the regular expression `(AT|GA)((AG|AAA)*)`. The state 0 is initial. Double-circled states are final.

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 [4, 5]. However, when using bit parallelism, the complexity is directly reduced to $O(m^2)$ by manipulating all the states in a register (see Section 3).

3 DFA representation and search algorithm

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 [7, 13] are built on this idea. We present in this section a new bit-parallel DFA representation based on Glushkov's construction. As we make heavy use of this construction and its properties, we start by presenting a bit-parallel implementation of Glushkov's construction.

3.1 Bit-parallel Glushkov construction

All along the Glushkov algorithm we manipulate sets of NFA states. As it is useful for the search algorithm, 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.

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 Σ 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 $First(v)$, $Last(v)$, $Follow(v, x)$ and $Empty_v$ are computed in postfix order. 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 bitmasks $B : \Sigma \rightarrow 2^{m+1}$, 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.

3.2 Properties of Glushkov's construction

We present now some properties of the Glushkov construction which are necessary for our compact DFA representation. All them are very easy to prove.

A first property should be obvious at this point, but it is important because it makes our problem totally different from that of a Thompson's construction. Since we do not build any ε-transitions, we have

Property *Glushkov's NFA is ε-free.*

That is, in the approach of Wu and Manber [13], the ε-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.

```
Glushkov_variables(v_RE, lpos)
1.    If v = [ | ] (v_l, v_r) OR v = [·] (v_l, v_r) Then
2.        lpos ← Glushkov_variables(v_l, lpos)
3.        lpos ← Glushkov_variables(v_r, lpos)
4.    Else If v = * Then lpos ← Glushkov_variables(v_*, lpos)
5.    End of if
6.    If v = (ε) Then
7.        First(v) ← 0^{m+1}, Last(v) ← 0^{m+1}, Empty_v ← TRUE
8.    Else If v = (C) , C ⊆ Σ Then
9.        lpos ← lpos + 1
10.       For σ ∈ C Do B[σ] ← B[σ] | 0^{m−lpos}10^{lpos}
11.       First(v) ← 0^{m−lpos}10^{lpos}, Last(v) ← 0^{m−lpos}10^{lpos}
12.       Empty_v ← FALSE , Follow(lpos) ← 0^{m+1}
13.   Else If v = [ | ] (v_l, v_r) Then
14.       First(v) ← First(v_l) | First(v_r), Last(v) ← Last(v_l) | Last(v_r)
15.       Empty_v ← Empty_{v_l} OR Empty_{v_r}
16.   Else If v = [·] (v_l, v_r) Then
17.       First(v) ← First(v_l), Last(v) ← Last(v_r)
18.       If Empty_{v_l} = TRUE Then First(v) ← First(v) | First(v_r)
19.       If Empty_{v_r} = TRUE Then Last(v) ← Last(v_l) | Last(v)
20.       Empty_v ← Empty_{v_l} AND Empty_{v_r}
21.       For x ∈ Last(v_l) Do Follow(x) ← Follow(x) | First(v_r)
22.   Else If v = [*] (v_*) Then
23.       First(v) ← First(v_*), Last(v) ← Last(v_*), Empty_v ← TRUE
24.       For x ∈ Last(v_*) Do Follow(x) ← Follow(x) | First(v_*)
25.   End of if
26.   Return lpos
```

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

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 the arrows that arrive at state y is precisely σ_y. This also holds if we consider that σ_y is in fact a subset of Σ.

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 σ. Let $Follow(x)$ be the set of states that can follow state x in one transition, by Glushkov's construction. Let $\delta : S \times \Sigma \to \mathcal{S}$ the transition function of the 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 σ. Then, it holds*

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

Proof. The lemma follows from Lemma 1. Let $y \in \delta(x, \sigma)$. This means that y can be reached from x by σ 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 σ and it can be reached from x. But Lemma 1 implies that every arrow leading to y is labeled by σ, including the one departing from x, and hence $y \in \delta(x, \sigma)$.

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 $Follow$).

With these properties in mind we can design now a compact DFA representation.

3.3 A compact DFA 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) such that

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

tells which states can be reached from an active state in D, no matter by which character.

By Lemma 2, it holds that

$$\delta(D, \sigma) \;=\; T[D] \;\&\; B[\sigma]$$

(where we are using bit mask representation for sets). Hence instead of the complete transition table $\delta : \; 2^{m+1} \times \Sigma \to 2^{m+1}$ we build and store only $T : \; 2^{m+1} \to 2^{m+1}$ and $B : \; \Sigma \to 2^{m+1}$. The number of bits required in 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^{m+1}
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
```

Fig. 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.

3.4 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* the regular expression. This is achieved by appending Σ^* 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
```

Fig. 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.

Compared to Wu and Manber's algorithm [13], 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 [8], 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 as a classical DFA). Figure 6 shows the recursive construction of this table, which is invoked with $D = 0^m 1$, the initial state, and assumes that T is initialized with zeros and that B, $Follow$ and m are already computed.

4 Experimental results

We compare in this section our approach against previous work. We use two different texts: an English one (writings of B. Franklin, filtered to lower-case) and a DNA sequence (h.influenzae). Both were replicated until obtaining 10 Mb.

```
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.    For σ ∈ Σ Do
5.        If T[N & B[σ]] = 0^{m+1} Then /* not built yet */
6.            BuildTrec (N & B[σ])
7.    End of for
```

Fig. 6. Recursive construction of table T. We fill only the reachable cells.

A major problem when presenting experiments on regular expressions is that there is not a concept of "random" regular expression, so it is not possible to search, say, 1,000 random patterns. Lacking such good choice, we fixed a set of 10 patterns for English and 10 for DNA, which were selected to illustrate different interesting cases rather than more or less "probable" cases. Therefore, the goal is not to show what are the typical cases in practice but to show how the scheme behaves under different characteristics of the pattern.

The patterns are given in Table 1. We also show their number of letters, which is closely related to the size of the automata recognizing them. The period (.) in the patterns matches any character except the end of line (lines have approximately 70 characters). We also use $[c_1...c_k]$ (where c_i are characters) as 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 letters between (and including) c^1 and c^2.

No.	DNA Pattern	m	No.	English Pattern	m
1	`AC((A|G)T)*A`	6	1	`benjamin|franklin`	16
2	`AGT(TGACAG)*A`	10	2	`benjamin|franklin|writing`	23
3	`(A(T|C)G)|((CG)*A)`	7	3	`[a-z][a-z0-9]*[a-z]`	3
4	`GTT|T|AG*`	6	4	`benj.*min`	8
5	`A(G|CT)*`	4	5	`[a-z][a-z][a-z][a-z][a-z]`	5
6	`((A|CG)*|(AC(T|G))*)AG`	9	6	`(benj.*min)|(fra.*lin)`	15
7	`AG(TC|G)*TA`	7	7	`ben(a|(j|a)*)min`	9
8	`[ACG][ACG][ACG][ACG][ACG][ACG]T`	7	8	`be.*ja.*in`	8
9	`TTTTTTTTTT[AG]`	11	9	`ben[jl]amin`	8
10	`AGT.*AGT`	7	10	`(be|fr)(nj|an)(am|kl)in`	14

Table 1. The patterns used on DNA and English.

Our machine is a Sun UltraSparc-1 of 167 MHz, with 64 Mb of RAM, running Solaris 2.5.1. We measured CPU times in seconds, averaging 100 runs over the 10 Mb (the variance was very low).

We have tested the following forward scanning algorithms (the implementations are ours except otherwise stated). See the Introduction for detailed descriptions of previous work. We have left aside some algorithms that are clearly not competitive, such as Thompson's [10].

DFA: builds the classical deterministic automaton and runs it over the text. We have not minimized the automaton.

Agrep: uses a bit mask to handle the active states [13]. The software [12] is from S. Wu and U. Manber, and has an advantage on frequently occurring patterns because it abandons a line as soon as it finds the

pattern on it. We forced it to build one table, except for the English pattern #2, where two tables were faster.

Myers: is the algorithm based on modules implemented as DFAs [7]. The code is from G. Myers.

Ours (old): our previous forward algorithm of [8] (see Section 2). It builds only the reachable states, just like the classical DFA algorithm.

Ours (naive): our new algorithm building the whole table T with **BuildT**.

Ours (optim): our new algorithm where we build only the T mask for the reachable states, using **BuildTrec**.

The goal of showing two versions of our algorithm is as follows. Our normal algorithm 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 to show that our technique is superior. Our optimized algorithm builds only the reachable states and it permits comparing against DFA (the classical algorithm) and our old algorithm. The disadvantage of our optimized algorithm is that it does not permit splitting the tables (neither does DFA), while our "naive" algorithm and that of Agrep do.

Tables 2 and 3 show the results on the different patterns, where we have separated preprocessing and search time. As it can be seen, Myers' algorithm is out of competition (it should be a good choice for much larger regular expressions). Our new algorithm (naive version) 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. It works quite well except on large patterns like the natural language pattern #2. Our optimized algorithm behaves well in those situations too, and compares favorably against the classical DFA algorithm and our old bit parallel algorithm, which scan the text at 14.4 and 14.6 Mb/sec, respectively. This means that our new algorithm is at least 10% faster than any alternative approach.

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

#	DFA	Agrep	Myers	Ours (old)	Ours (naive)	Ours (optim)
1	$0.034 + 0.643n$	$0.104 + 0.756n$	$0.086 + 2.201n$	$0.007 + 0.631n$	$0.009 + 0.584n$	$0.005 + 0.575n$
2	$0.006 + 0.624n$	$0.133 + 0.754n$	$0.090 + 4.965n$	$0.011 + 0.630n$	$0.049 + 0.566n$	$0.000 + 0.576n$
3	$0.028 + 0.796n$	$0.095 + 0.758n$	$0.080 + 2.182n$	$0.007 + 0.803n$	$0.007 + 0.759n$	$0.087 + 0.775n$
4	$0.025 + 0.883n$	$0.101 + 0.760n$	$0.082 + 2.141n$	$0.008 + 0.865n$	$0.012 + 0.788n$	$0.029 + 0.807n$
5	$0.018 + 0.831n$	$0.089 + 0.757n$	$0.088 + 2.205n$	$0.007 + 0.814n$	$0.005 + 0.755n$	$0.008 + 0.777n$
6	$0.007 + 0.658n$	$0.126 + 0.762n$	$0.089 + 2.148n$	$0.008 + 0.652n$	$0.014 + 0.584n$	$0.004 + 0.592n$
7	$0.004 + 0.634n$	$0.104 + 0.750n$	$0.092 + 2.173n$	$0.005 + 0.635n$	$0.015 + 0.571n$	$0.040 + 0.567n$
8	$0.004 + 0.646n$	$0.101 + 0.831n$	$0.099 + 2.132n$	$0.012 + 0.638n$	$0.008 + 0.583n$	$0.071 + 0.582n$
9	$0.006 + 0.621n$	$0.096 + 0.694n$	$0.085 + 5.304n$	$0.024 + 0.626n$	$0.005 + 0.568n$	$0.007 + 0.565n$
10	$0.038 + 0.639n$	$0.108 + 0.748n$	$0.098 + 2.109n$	$0.018 + 0.645n$	$0.011 + 0.560n$	$0.036 + 0.562n$

Table 2. Search times on English, in the form of $a + bn$, where a is the preprocessing time and b is the search time per megabyte, all in tenths of seconds.

5 Conclusions

We have presented a new technique for compact DFA representation based on the properties of Glushkov's NFA construction, as opposed to the much better known Thompson's. As a result, we can represent the DFA using $(m+1)(2^{m+1} + |\Sigma|)$ bits (where m is the number of normal characters in the pattern and Σ is

#	DFA	Agrep	Myers	Ours (old)	Ours (naive)	Ours (optim)
1	$0.010 + 0.633n$	$0.114 + 0.779n$	$0.090 + 5.293n$	$0.006 + 0.633n$	$0.074 + 0.569n$	$0.009 + 0.563n$
2	$0.022 + 0.629n$	$0.112 + 1.583n$	$0.090 + 8.452n$	$0.009 + 0.699n$	$20.61 + 0.575n$	$0.019 + 0.569n$
3	$0.003 + 0.932n$	$0.106 + 0.769n$	$0.090 + 2.103n$	$0.050 + 0.897n$	$0.007 + 0.856n$	$0.022 + 0.898n$
4	$0.068 + 0.639n$	$0.100 + 0.755n$	$0.080 + 2.111n$	$0.023 + 0.631n$	$0.008 + 0.567n$	$0.000 + 0.578n$
5	$0.009 + 0.879n$	$0.095 + 0.871n$	$0.090 + 2.110n$	$0.063 + 0.727n$	$0.050 + 0.664n$	$0.000 + 0.684n$
6	$0.242 + 0.645n$	$1.494 + 0.775n$	$0.090 + 5.204n$	$0.083 + 0.640n$	$0.043 + 0.569n$	$0.013 + 0.567n$
7	$0.007 + 0.631n$	$0.122 + 0.755n$	$0.090 + 2.140n$	$0.006 + 0.625n$	$0.006 + 0.572n$	$0.001 + 0.578n$
8	$0.081 + 0.628n$	$0.103 + 0.755n$	$0.090 + 2.133n$	$0.023 + 0.624n$	$0.048 + 0.562n$	$0.027 + 0.556n$
9	$0.000 + 0.627n$	$0.102 + 0.704n$	$0.086 + 2.100n$	$0.008 + 0.626n$	$0.011 + 0.567n$	$0.002 + 0.565n$
10	$0.012 + 0.632n$	$0.774 + 0.789n$	$0.081 + 5.244n$	$0.007 + 0.643n$	$0.018 + 0.567n$	$0.005 + 0.561n$

Table 3. Search times on English, in the form of $a + bn$, where a is the preprocessing time and b is the search time per megabyte, all in tenths of seconds.

the alphabet). This compares favorably against previous techniques which needed either $(m+1)2^{m+1}|\Sigma|$ or $m(2^{2m+1} + |\Sigma|)$ bits.

The representation is quite practical. We are not only still able of searching in $O(n)$ time using the compact DFA, but thanks to more locality of reference we can search faster in practice than any previous approach, as we show experimentally.

Despite that we have presented a forward scan algorithm, our approach can be adapted to character skipping algorithms as well. For example, our algorithm presented in [8] modified the automaton by reversing its arrows, making all the states initial and making the initial state final, so as to recognize reverse prefixes of the original language $L(RE)$. This algorithm is used to extend BNDM [9] so as to obtain a fast character skipping algorithm for regular expression search.

Reversing the arrows means that the property that all arrows arriving to a state have the same label does not hold anymore (once we reverse the arrows, the result is not a Glushkov NFA). Rather, all the arrows *departing* from a state have now the same label. Once again, we can represent the DFA in a compact form by noting that $\delta(D, \sigma) = T\ [D\ \&B[\sigma]]$, where T is the deterministic $Follow$ table of the reversed automaton and B is the character table of the original automaton. That is, we first keep the states of D that can originate arrows labeled by σ, and once they are obtained we find all the target states.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.
2. R. Baeza-Yates and G. Gonnet. A new approach to text searching. *CACM*, 35(10):74–82, October 1992.
3. G. Berry and R. Sethi. From regular expression to deterministic automata. *Theoretical Computer Science*, 48(1):117–126, 1986.
4. A. Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213, November 1993.
5. C.-H. Chang and R. Paige. From regular expression to DFA's using NFA's. In *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching*, LNCS v. 664, pages 90–110, 1992.
6. V-M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1961.
7. E. Myers. A four-russian algorithm for regular expression pattern matching. *J. of the ACM*, 39(2):430–448, 1992.
8. G. Navarro and M. Raffinot. Fast regular expression search. In *Proceedings of the 3rd Workshop on Algorithm Engineering*, LNCS v. 1668, pages 199–213, 1999.
9. 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`.
10. K. Thompson. Regular expression search algorithm. *CACM*, 11(6):419–422, 1968.

11. B. Watson. *Taxonomies and Toolkits of Regular Language Algorithms.* Phd. dissertation, Eindhoven University of Technology, The Netherlands, 1995.
12. S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. of USENIX Technical Conference*, pages 153–162, 1992.
13. S. Wu and U. Manber. Fast text searching allowing errors. *CACM*, 35(10):83–91, October 1992.