Fast Regular Expression Search

Gonzalo Navarro[1] and Mathieu Raffinot[2]

[1] Dept. of Computer Science, University of Chile. Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl. Partially supported by Fondecyt grant 1-990627.
[2] Institut Gaspard Monge, Cité Descartes, Champs-sur-Marne, 77454 Marne-la-Vallée Cedex 2, France. raffinot@monge.univ-mlv.fr

Abstract. We present a new algorithm to search regular expressions, which is able to skip text characters. The idea is to determine the minimum length ℓ of a string matching the regular expression, manipulate the original automaton so that it recognizes all the reverse prefixes of length up to ℓ of the strings accepted, and use it to skip text characters as done for exact string matching in previous work. As we show experimentally, the resulting algorithm is fast, the fastest one in many cases of interest.

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].

The traditional technique [16] 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 to search 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 [13] 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 and manage to update all them in a single operation. In this case, the vector of active and inactive states is stored as bits of a computer word. Instead of (ala Thompson [16]) 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 set of active states (another computer word). 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, we may resort to pure bit-parallelism and forget about the modules. This was done in [19] by Wu and Manber, and included in their software *Agrep* [18]. 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 [16] is used, all the arrows carry 1's from bit positions i to $i+1$, except for the ϵ-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, and second, the ϵ-transitions are carried out. As ϵ-transitions follow arbitrary paths, an $E : 2^m \rightarrow 2^m$ function is precomputed and stored, where $E(w)$ is the ϵ-closure of w. 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 $E(w_1 w_2) = E(w_1)$ *or* $E(w_2)$. This can be thought of as an alternative decomposition scheme, instead of Myers' modules.

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 for the regular expression to match. In [17, 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 in the language denoted by the regular expression and uses the Commentz-Walter algorithm [7] to search them. Another technique of this kind is used in *Gnu Grep v2.0*, which extracts a single string (the longest) which 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 [9, 14]. These are simple pattern search algorithms whose main idea is to build an automaton able to recognize the reverse prefixes of the pattern, and examine backwards a window of length m on the text. This automaton helps 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 match of the regular expression. Hence, this transformed automaton is a compact device to achieve the same multipattern searching, at much less space. The automata are simulated using bit-parallelism. Our experimental results show that, when the regular expression does not match too short or too frequent strings, our algorithm is among the fastest, faster than all those unable to skip characters and in most cases faster than those based on multipattern matching. An extra contribution is our bit-parallel simulation, which differs from Agrep's in that less bits are used and no ϵ-transitions exist, although the transitions on letters are arbitrary and therefore a separate table per letter is needed (the tables can be horizontally split in case of space problems). Our simulation turns out to be faster than Agrep and the fastest in most cases.

Some definitions that are used in this paper follow. A *word* is a string or sequence of characters over a finite alphabet Σ. 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. $p = xu$), $u \in \Sigma^*$. We call R our pattern (a regular expression), which is of length m. We note $L(R)$ the set of words generated by R. Our text is of size n.

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 sub-expressions one after the other. Parentheses are used to change the precedence, which is normally "$*$", ".", "|". We adapt 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 letters between (and including) c^1 and c^2. Finally, the period (.) represents any character.

2 The reverse factor search approach

In this section we describe the general reverse factor search approach currently used for a single pattern [12, 9, 14] or multiple patterns [8, 15].

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

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 1) for any factor of any length-ℓ prefix of our set of patterns (if we search 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 the window position 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 1. The search for a factor fail on a letter σ, i.e σu is not a factor of a length-ℓ 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 just one pattern, then we have recognized it and we report the occurrence. Otherwise, we just recognized a length-ℓ prefix of one or more patterns. We verify directly in the text if there is a match of a pattern, 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 according to position *last*.

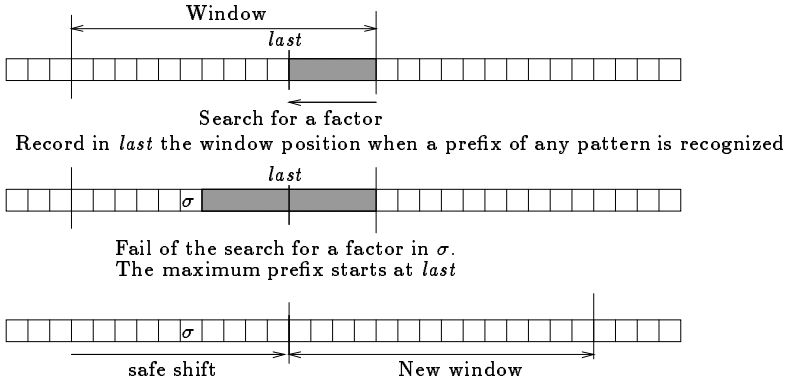

Fig. 1. The reverse factor search approach.

This simple approach leads to very fast algorithms in practice, such as BDM [9] and BNDM [14]. For a single pattern, this is optimal on average, matching Yao's bound [20] of $O(n \log(\ell)/\ell)$ (where n is the text size and ℓ 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.

2.1 A linear worst case algorithm

The main idea used in [9, 14, 8, 15] 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 [11] (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 ℓ. 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 2. 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.

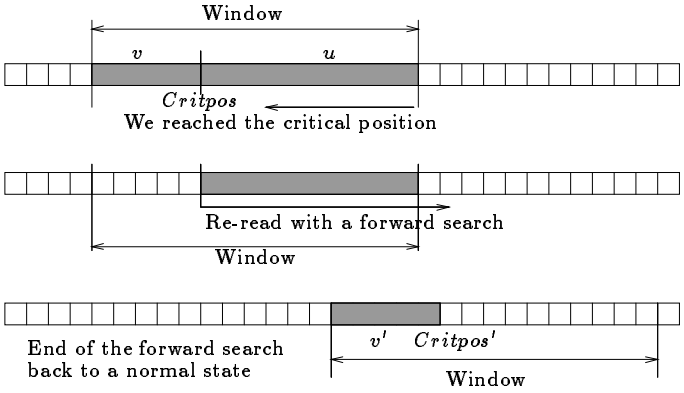

Fig. 2. The critical position is reached, in the linear-time algorithm.

(ii) We do not reach the critical position. This case is shown in Figure 3. 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.

3 Extending the approach to regular expression searching

In this section we explain how to adapt the general approach of Section 2 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

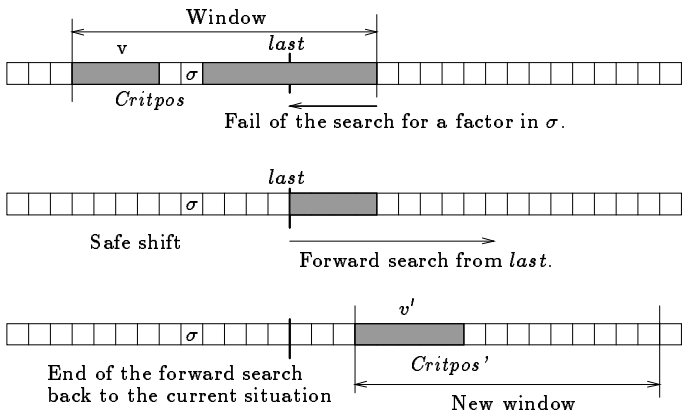

Fig. 3. The critical position is not reached, in the linear-time algorithm.

for a regular expression called R which is of size m and generates the language $L(R)$.

3.1 Basic approach

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

We use the general approach of Section 2, consisting of a backward and, if necessary (i.e we reached the beginning of the window), a forward scan. To adapt this scheme to regular expression search, we need two modifications:

(i) The backward search step of the general approach imposes here that we are able recognize any factor of the reverse prefixes of length ℓ of $L(R)$. Moreover, we mark in a variable *last* the longest prefix of $L(R)$ recognized (of course this prefix will be of length less than ℓ).
(ii) The forward search (if we reached the beginning of the window) verifies that there is a match of the regular expression starting at the beginning of the window (however, the match can be much longer than ℓ).

We detail now the steps of the preprocessing and searching phases. Complexities will be discussed in Section 4 because they are related to the way the automata are built.

Preprocessing The preprocessing consists of 3 steps:

1. Build the automaton that recognizes R. We note it $F(R)$, and its specific construction details are deferred to the next section.

2. Determine ℓ and compute the set $P_i(R)$ of all the nodes of $F(R)$ reachable in i steps or less from the initial state, for each $0 \leq i \leq \ell$ (so $P_i(R) \subseteq P_{i+1}(R)$). Both things are easily computed with a breadth-first search from the initial state until a final node is reached (being then ℓ the current depth at that point).
3. Build the automaton $B(R)$ that recognizes any factor of the reverse prefixes of length ℓ of $L(R)$. This is achieved by restricting the original automaton $F(R)$ to the nodes of $P_\ell(R)$, reversing the arrows, taking as (the only) terminal state the initial state of $F(R)$ and all the states as initial states.

The most interesting part of the above procedure is $B(R)$, which is a device to recognize the reverse factors of prefixes of length ℓ of $L(R)$. It is not hard to see that any such factor corresponds to a path in $F(R)$ that touches only nodes in $P_\ell(R)$. In $B(R)$ there exists the same path with the arrows reversed, and since all the states of $B(R)$ 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 $B(R)$ leads to its final state.

Note, however, that $B(R)$ can recognize more words than desired. For instance, if there are loops in $B(R)$ it can recognize words longer than ℓ. However, we can restrict more the set of words recognized by $B(R)$. The idea is that, if a state of $B(R)$ is active but it is farther than i positions to the final state of $B(R)$, 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 $B(R)$ with the set $P_i(R)$.

It is easy to see that, with this modification, the automaton recognizes exactly the desired prefixes, since if a state has not been "killed" with the intersection with $P_i(R)$ 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 ℓ of $L(R)$. However, this set may be large and the resulting scheme may need much more memory. The automaton $B(R)$ is a more compact device to obtain the same result.

Searching The search follows the general approach of Section 2. For each window position, we activate all the states of $B(R)$ and traverse the window backwards updating *last* each time the final state of $B(R)$ is reached (recall that after each step, we "kill" some states pf $B(R)$ using $P_i(R)$). If $B(R)$ runs out of active states we shift the window to position *last*. Otherwise, if we reached the beginning of the window, we start a forward scan using $F(R)$ from the beginning of the window until either a match is found[1], we reached the end of the text, or $F(R)$ runs out of active states. After the forward scan, we shift the window to position *last*.

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

3.2 Linear worst case extension

We also extended the general linear worst case approach (see Section 2.1) to the case of regular expression searching.

We transform the forward scan automaton $F(R)$ of the previous algorithm by adding a self-loop at its initial state, for each letter of Σ (so now it recognizes $\Sigma^* L(R)$). This is the normal automaton used for classical searching, and the one we use for the forward scanning.

The main difficulty to extend the general linear approach is 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 ℓ' characters ahead of the current text position (for $0 \leq \ell' \leq \ell$). To simplify our explanation, we call this ℓ' 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 $F(R)$ to a final state (this same idea has been used for multipattern matching in [8]). In this case, we need to recognize any reverse factor of $L(R)$ in the backward scan (not only the factors of prefixes of length ℓ)[2]. Each time ℓ' is large enough to be promising ($\ell' \geq \alpha\ell$, for some heuristically fixed α), we stop the forward scan and start a backward scan on a window of forward-length ℓ' (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 $L(R)$ have to be recognized, which makes the backward scans longer and the shifts shorter. Also, the window forward-length ℓ' is never larger than our previous ℓ, since the initial state of $F(R)$ 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 $P_i(R)$, 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 after reading that number

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

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 $B(R)$ automaton. Note that the previous approach applied to this case (with all active states belonging to $P_i(R)$) 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.

4 Building an NFA from a regular expression

There exist currently many different techniques to build an NFA from a regular expression R of size m. The most classical one is the Thomson construction [16]. It builds an NFA with at most $2m$ states that present somes particular properties. Some algorithms like that of Myers [13] and of Wu and Manber in Agrep [19, 18] make use of these properties.

A second one is the Glushkov's construction, popularized by Berry and Sethi in [4]. The NFA resulting of this construction has the advantage of having just $m + 1$ states (one per position in the regular expression). A lot of research on Glushov's construction has been pursued, like [5], where it is shown that the resulting NFA is quadratic in the number of edges in the worst case. In [10], a long time open question about the minimal number of edges of an NFA with linear number of states was answered, showing a construction with $O(m)$ states and $O(m(\log m)^2)$ edges, as well as a lower bound of $O(m \log m)$ edges. Hence, Glushkov construction is not space-optimal.

Some research has been done also to try to construct directly a DFA from a regular expression, without constructing an NFA, such as [6].

For our purpose, when we consider bit-parallelism, the most interesting is to have a minimal number of states, because we manage computer words of a fixed length w to represent the set of possible states. Hence, we choose the original Gluskov's construction, which leads to an NFA with $m+1$ states and a quadratic (in the worst case) number of edges. The number of edges is unimportant for our case.

In Gluskov's construction, the edges have no simple structure, and we need a table which, for each current set of states and each current text character, gives the new set of states. On the other hand, the construction of Wu and Manber uses the regularities of Thompson's construction so that they need only a table for the ϵ-transitions, not for every character. In exchange, we have $m+1$ states instead of nearly $2m$ states, and hence their table sizes square ours. As we show later experimentally, our NFA simulation is faster than those based on the Thompson construction, so the tradeoff pays off.

5 Experimental results

We compare in this section our approach against previous work. 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. The goal of this comparison is just to show that our simulation is competitive. 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 is faster in general. Finally, we compare our backward search algorithm against other algorithms that are also able to skip characters.

We use an English text (writings of B. Franklin), filtered to lower-case and replicated until obtaining 10 Mb. 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, which were selected to illustrate different interesting cases rather than more or less "probable" cases. In fact we believe that common patterns have long exact strings and our algorithm would behave even better than in these experiments. 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 minimum length ℓ 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).

No.	Pattern	Size (# letters)	Minimum length ℓ	Prob. match (empirical)
1	`benjamin|franklin`	16	8	.00003586
2	`benjamin|franklin|writing`	23	7	.0001014
3	`[a-z][a-z0-9]*[a-z]`	3	2	.6092
4	`benj.*min`	8	7	.000007915
5	`[a-z][a-z][a-z][a-z][a-z]`	5	5	.2024
6	`(benj.*min)|(fra.*lin)`	15	6	.00003586
7	`ben(a|(j|a)*)min`	9	6	.009491
8	`be.*ja.*in`	8	6	.00001211
9	`ben[jl]amin`	8	8	.000007915
10	`(be|fr)(nj|an)(am|kl)in`	14	8	.00003586

Table 1. The patterns used on English text.

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 10 runs over the 10 Mb (the variance was very low). We include the time for preprocessing in the figures.

5.1 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 method we adopted. The purpose of this test is to show that our approach is competitive against the rest. We have tested the following algorithms for the forward scanning (the implementations are ours except otherwise stated). See the Introduction for detailed descriptions of previous work.

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

Thompson: simulates the nondeterministic automaton by keeping a list of active states which is updated for each character read (this does not mean that we build the automaton using Thompson's method).

BP-Thompson: same as before, but the set of active states is kept as a bit vector. Set manipulation is faster when many states are active.

Agrep: uses a bit mask to handle the active states [19]. The software [18] 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.

Myers: is the algorithm based on modules implemented as DFAs [13]. The code is from G. Myers.

Ours: our forward algorithm, similar to that of Agrep except because we eliminate the ϵ-transitions and have a separate transition table for each character (Section 4).

Except for Agrep and Myers, which have their own code, we use the NFA construction of Section 4. Table 2 shows the results on the different patterns. As it can be seen, the schemes that rely on nondeterministic simulation (Thompson variants) worsen when the combination of pattern size and matching probability increases. The rest is basically insensitive to the pattern, except because all worsen a little when the pattern matches very frequently. If the pattern gets significantly larger, however, the deterministic simulations worsen as well, as some of them are even exponentially depending on the automaton size. Agrep, Myers and Ours can adapt at higher but reasonable costs, proportional to the pattern length. This comes not only from the possible need to use many machine words but also because it may be necessary to cut the tables horizontally.

With respect to the comparison, we point out that our scheme is competitive, being the fastest in many cases, and always at most 5% over the performance of the fastest. DFA is the best in the other cases. Our algorithm can in fact be seen as a DFA implementation, where our state identifier is the bit mask and the transition table is the one we use. However, the DFA has less states, since most of the bit combinations we store are in fact unreachable[3]. On the other hand,

[3] We do not build the transition table for unreachable states, but we do not compact reachable states in consecutive table positions as the DFA implementation does. This is the essence of the bit-parallel implementation.

Pattern	DFA	Thompson	BP-Thompson	Agrep	Myers	Ours	Ours/best
1	0.70	4.47	4.19	1.88	5.00	**0.68**	1.00
2	**0.73**	4.13	5.10	1.89	8.57	0.76	1.04
3	1.01	18.2	3.75	**0.98**	2.19	0.99	1.01
4	0.71	4.16	3.17	0.97	2.17	**0.68**	1.00
5	0.87	18.7	4.32	1.05	2.18	**0.82**	1.00
6	0.76	4.25	4.06	1.87	4.94	**0.72**	1.00
7	0.73	4.67	2.82	0.99	2.17	**0.72**	1.00
8	**0.72**	4.93	3.40	0.96	2.18	0.73	1.01
9	**0.66**	4.75	3.11	1.00	2.16	0.68	1.03
10	**0.71**	4.36	3.97	1.86	5.01	0.73	1.03

Table 2. Forward search times on English, in seconds for 10 Mb.

the bit-parallel implementation is much more flexible when it comes to adapt it for backward searching or to extend it to handle extended patterns or to allow errors.

We have left aside lazy deterministic automata implementations. However, as we show in Section 5.3, these also tend to be slower than ours.

5.2 Forward versus backward scanning

We compare now our new forward scan algorithm (called **Fwd** in this section and **Ours** in Section 5.1) against backward scanning. There are three backward scanning algorithms. The simplest one, presented in Section 3.1, is called **Bwd**. The two linear variations presented in Section 3.2 are called **LBwd-All** (that recognizes all the reverse factors) and **LBwd-Pref** (that recognizes reverse factors of length-ℓ prefixes). The linear variations depend on an α parameter, which is between 0 and 1. We have tested the values 0.25, 0.50 and 0.75 for α, although the results change little.

Table 3 shows the results. We obtained improvements in 7 of the 10 patterns (and very impressive in four cases). In general, the linear versions are quite bad in comparison with the simple one, although in some cases they are faster than forward searching. It is difficult to determine which of the two versions is better in which cases, and which is the best value for α.

5.3 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:

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	0.68	**0.28**	0.44	0.43	0.46	0.47	0.49	0.50
2	0.76	**0.65**	1.17	1.00	1.09	0.93	0.98	0.95
3	**0.99**	2.37	3.30	2.59	3.01	2.56	2.56	2.56
4	0.68	**0.56**	1.70	1.68	1.71	0.94	0.93	0.92
5	**0.82**	2.02	2.05	2.40	2.09	2.13	2.15	2.18
6	0.72	**0.70**	1.82	1.85	1.84	1.10	1.12	1.09
7	0.72	**0.30**	0.46	0.45	0.47	0.51	0.52	0.51
8	**0.73**	0.91	1.75	1.85	1.87	1.33	1.45	1.47
9	0.68	**0.24**	0.37	0.37	0.39	0.41	0.39	0.41
10	0.73	**0.29**	0.42	0.45	0.44	0.47	0.46	0.48

Table 3. Backward search times on English, in seconds for 10 Mb.

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 v2.3*, which implements this idea. Where the filter cannot be applied, *Grep* uses a forward scanning algorithm which is 30% slower than our *Fwd*[4]. Hence, we plot this value only where the idea can be applied. We point out that *Grep* also abandons a line when it finds a first match in it.

Multiple pattern: this idea was presented in [17]. A length $\ell' < \ell$ is selected, and all the possible suffixes of length ℓ' of $L(R)$ are generated and searched for. The choice of ℓ' is not obvious, since longer strings make the search faster, but there are more of them. Unfortunately, the code of [17] is not public, so we have used the following procedure: first, we extract by hand the suffixes of length ℓ' for each regular expression; then we use the multipattern search of *Agrep* [18], which is very fast, to search 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 ℓ' value for each regular expression. The resulting algorithm is quite similar to the idea of [17].

Our algorithms are called **Fwd** and **Bwd** and correspond to those of the previous sections. Table 4 shows the results. The single pattern filter is a very effective trick, but it can be applied only in a restricted set of cases. In some cases its improvement over our backward search is modest. 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 (an exception is the 2nd pattern, where we have a costly preprocessing).

[4] Which shows that our implementation is faster than a good lazy deterministic automaton implementation.

Pattern	Fwd	Bwd	Single pattern filter	Multipattern filter
1	0.68	**0.28**	—	0.31
2	0.76	0.65	—	**0.37**
3	**0.99**	2.37	—	1.65
4	0.68	0.56	**0.17**	0.87
5	**0.82**	2.02	—	2.02
6	0.72	**0.70**	—	1.00
7	0.72	0.30	**0.26**	0.44
8	0.73	0.91	**0.63**	0.66
9	0.68	0.24	**0.19**	0.31
10	0.73	**0.28**	0.98	0.35

Table 4. Algorithm comparison on English, in seconds for 10 Mb.

6 Conclusions

We have presented a new algorithm for regular expression searching able to skip characters. It is based on an extension of the backward DAWG matching approach, where the automaton is manipulated to recognize reverse prefixes of strings of the language. We also presented two more complex variants which are of linear time in the worst case. The automaton is simulated using bit-parallelism.

We first show that the bit-parallel implementation is competitive (at most 5% over the fastest one in all cases). The advantage of bit-parallelism is that the algorithm can easily handle extended patterns, such as classes of characters, wild cards and even approximate searching. We then compare the backward matching against the classical forward one, finding out that the former is superior when the minimum length of a match is not too short and the matching probability is not too high.

Finally, we compare our approach against others able to skip characters. These are based on filtering the search using multipattern matching. The experiments show that our approach is faster in many cases, although there exist some faster hybrid algorithms which can be applied in some restricted cases. Our approach is more general and performs reasonably well in all cases.

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. More work on minimizing the NFA could improve the average case.

Being able to skip characters and based on an easily generalizable technique such as bit-parallelism permits to extend our scheme to deal with other cases, such as searching a regular expression allowing errors, and being still able to skip characters. This is also a subject of future work.

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. *CACM*, 35(10):74–82, October 1992.
4. G. Berry and R. Sethi. From regular expression to deterministic automata. *Theor. Comput. Sci.*, 48(1):117–126, 1986.
5. A. Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213, November 1993.
6. C.-H. Chang and R. Paige. From regular expression to DFA's using NFA's. In *Proc. of the CPM'92*, number 664 in LNCS, pages 90–110. Springer-Verlag, 1992.
7. B. Commentz-Walter. A string matching algorithm fast on the average. In *Proc. ICALP'79*, number 6 in LNCS, pages 118–132. Springer-Verlag, 1979.
8. 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.
9. 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.
10. Juraj Hromkovič, Sebastian Seibert, and Thoma s Wilke. Translating regular expression into small ε-free nondeterministic automata. In *STACS 97*, LNCS, pages 55–66. Springer-Verlag, 1997.
11. D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):323–350, 1977.
12. T. Lecroq. Experimental results on string matching algorithms. *Softw. Pract. Exp.*, 25(7):727–765, 1995.
13. E. Myers. A four-russian algorithm for regular expression pattern matching. *J. of the ACM*, 39(2):430–448, 1992.
14. G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *Proc. CPM'98*, LNCS v. 1448, pages 14–33, 1998.
15. Mathieu Raffinot. On the multi backward dawg matching algorithm (MultiBDM). In *Proc. WSP'97*, pages 149–165. Carleton University Press, 1997.
16. K. Thompson. Regular expression search algorithm. *CACM*, 11(6):419–422, 1968.
17. B. Watson. *Taxonomies and Toolkits of Regular Language Algorithms.* Phd. dissertation, Eindhoven University of Technology, The Netherlands, 1995.
18. S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. of USENIX Technical Conference*, pages 153–162, 1992.
19. S. Wu and U. Manber. Fast text searching allowing errors. *CACM*, 35(10):83–91, October 1992.
20. A. Yao. The complexity of pattern matching for a random string. *SIAM J. on Computing*, 8:368–387, 1979.