On NFA reductions

Lucian Ilie[1,⋆,⋆⋆] Gonzalo Navarro[2,⋆⋆⋆] and Sheng Yu[1,†]

[1] Department of Computer Science, University of Western Ontario
N6A 5B7, London, Ontario, CANADA
ilie|syu@csd.uwo.ca

[2] Department of Computer Science, University of Chile
Blanco Encalada 2120, Santiago, CHILE
gnavarro@dcc.uchile.cl

Abstract. We give faster algorithms for two methods of reducing the number of states in nondeterministic finite automata. The first uses equivalences and the second uses preorders. We develop restricted reduction algorithms that operate on position automata while preserving some of its properties. We show empirically that these reductions are effective in largely reducing the memory requirements of regular expression search algorithms, and compare the effectiveness of different reductions.

1 Introduction

Regular expression handling is at the heart of many applications, such as linguistics, computational biology, pattern recognition, text retrieval, and so on. An elegant theory gives the support to easily and efficiently solve many complex problems by mapping them to regular expressions, then obtaining a nondeterministic finite automaton (NFA) that recognizes it, and finally making it deterministic (a DFA). However, a severe obstacle in any real implementation of the above scheme is the size of the DFA, which can be exponential in the length of the original regular expression.

Although a simple algorithm for minimizing DFAs exists [5], it has the problem of requiring prior construction of the DFA to later minimize it. This can be infeasible because of main memory requirements and construction cost.

A much more promising (and more challenging) alternative is that of directly reducing the NFA before converting it into a DFA. This has the advantage of working over a much smaller structure (of size polynomial in the length of the regular expression) and of building the smaller DFA without the need to go through a larger one first.

However, the NFA state minimization problem is very hard (PSPACE-complete, [10]) and therefore algorithms such as [11, 13, 14] cannot be used in practice.

⋆ Research partially supported by NSERC.

⋆⋆ Corresponding author

⋆⋆⋆ Supported in part by Fondecyt grant 1-020831.

† Research partially supported by NSERC.

There are also algorithms which build small NFAs from regular expressions, see [7, 4], but they consider the total size, that is, they count both states and transitions, and they increase artificially the number of states to reduce the number of transitions. As the implementation crucially depends on the number of states, such algorithms may not help.

The approach we follow is reducing the size of a given NFA. The idea of reducing the size of NFAs by merging states was first introduced by Ilie and Yu [8] who used equivalence relations. Later, Champarnaud and Coulon [2] modified the idea to work for preorders. In this paper we give fast algorithms to compute these two reductions. We show that the algorithm based on equivalences can be implemented in $O(m \log n)$ time on an NFA with n states and m transitions, while that based on preorders can run in $O(mn)$ time. Both results improve the previous work.

When starting from a regular expression, the initial NFA, which we want to reduce, is the position automaton. Navarro and Raffinot [17, 18] showed that its special properties permit a more compact DFA representation. Our modified reductions are restricted to preserve those properties and hence may produce NFAs with more states than the original reductions.

Finally, we empirically evaluate the impact of the reduction algorithms. We show that the number of NFA states can be reduced by 10%–40%. Those reductions translate into huge reductions in the DFA size, with factors of up to 10^{-6}. We also compare the alternatives of full reduction versus restricted reduction, since the former yields less NFA states but the latter permits a more compact DFA representation. The results show that full reduction is preferable in most cases of interest.

2 Basic notions

We recall here the basic definitions we need throughout the paper. For further details we refer to [6] or [22].

Let A be an alphabet and A^* the set of all words over A; ε denotes the empty word. A *language* over A is a subset of A^*. A *nondeterministic finite automaton* (*NFA*) is a tuple $M = (Q, A, \delta, I, F)$, where Q is the set of states, $I \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final states, and $\delta : Q \times A \to 2^Q$ is the transition mapping; δ is extended to $\delta : 2^Q \times A^* \to 2^Q$ by $\delta(S, a) = \bigcup_{q \in S} \delta(q, a)$ and $\delta(S, \varepsilon) = S$, $\delta(S, aw) = \delta(\delta(S, a), w)$, for $S \subseteq Q$, $w \in A^*$. The *language* recognized by M is $\mathcal{L}(M) = \{w \in A^* \mid \delta(I, w) \cap F \neq \emptyset\}$. For a state $q \in Q$, we denote

$$\mathcal{L}_L(M, q) = \{w \in A^* \mid q \in \delta(I, w)\},$$
$$\mathcal{L}_R(M, q) = \{w \in A^* \mid \delta(q, w) \cap F \neq \emptyset\};$$

when M is understood, we write simply $\mathcal{L}_L(q)$ and $\mathcal{L}_R(q)$, resp. The *reversed* automaton of M is $M^r = (Q, A, \delta^r, F, I)$, where $q \in \delta^r(p, a)$ iff $p \in \delta(q, a)$.

3 NFA reduction with equivalences

The idea of reducing the size of NFAs by merging state was investigated first by Ilie and Yu [8]; see also [9]. We describe it briefly in this section.

Let $M = (Q, A, \delta, I, F)$ be an NFA. We define $\equiv_R$ as the coarsest equivalence relation over Q that satisfies:

(P_1) $\equiv_R \cap (F \times (Q - F)) = \emptyset$,
(P_2) for any $p, q \in Q, a \in A$, $\big(p \equiv_R q \Rightarrow \forall q' \in \delta(q, a), \exists p' \in \delta(p, a), q' \equiv_R p'\big)$.

The equivalence $\equiv_R$ is the largest equivalence over Q which is right-invariant w.r.t. M; see [8, 9]. Given $\equiv_R$, the algorithm to reduce the automaton M using it is trivial: simply merge all states in the same equivalence class and modify the transitions accordingly. Here is an example.

Example 1 The NFA in Fig. 4 is reduced using $\equiv_R$ as shown in Fig. 1; the equivalence classes are also shown.

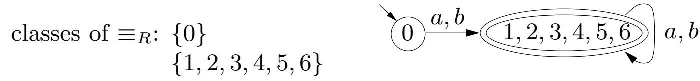

Fig. 1. $\mathbf{A}_{\mathrm{R}}(\tau) = \mathbf{A}_{\mathrm{pos}}(\tau)/_{\equiv_R}$ for $\tau = (a + b)(a^* + ba^* + b^*)^*$

Symmetrically, the relation $\equiv_L$ can be defined using the reversed automaton. The automaton M can be reduced according to either equivalence. As examples in [9] show, M can be reduced more using both equivalences but the problem of finding the best way to do the reduction is open. Fig. 2 gives an example (from [9]) where there is no unique way to reduce optimally using both $\equiv_R$ and $\equiv_L$.

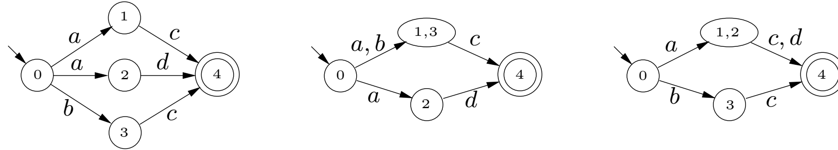

Fig. 2. An NFA and its corresponding quotients modulo $\equiv_R$ and $\equiv_L$

4 Computing equivalences

The algorithm in [8] for computing $\equiv_R$ runs in low polynomial time but the problem of finding a fast algorithm was left open. We show here that an old very fast algorithm of Paige and Tarjan [19] can be used to solve the problem.

Recall some definitions from [19]. For a binary relation E over a finite set U we denote, for any subset $S \subseteq U$, $E^{-1}(S) = \{x \mid \exists y \in S \text{ such that } xEy\}$. A subset $B \subseteq U$ is called *stable w.r.t.* S if either $B \subseteq E^{-1}(S)$ or $B \cap E^{-1}(S) = \emptyset$. A partition P of U is *stable w.r.t.* S if all the blocks of P are stable w.r.t. S. P is *stable* if it is stable w.r.t. each of its own blocks. The *relational coarsest partition problem* is that of finding, for a given relation E and a partition P over a set U, the coarsest stable refinement of P. Paige and Tarjan [19] gave an algorithm for this problem which runs in time $\mathcal{O}(m \log n)$ and space $\mathcal{O}(m + n)$, where $n = \mathsf{card}(U)$, $m = \mathsf{card}(E)$. They remarked that the algorithm works also for several relations.

This algorithm applies to our problem of finding $\equiv_R$ as follows. For any $a \in A$, denote $\delta_a = \{(p, q) \in Q \times Q \mid q \in \delta(p, a)\}$. Then $\equiv_R$ is the coarsest stable refinement of the partition $\{F, Q - F\}$ w.r.t. all relations δ_a, $a \in A$.

Therefore, if the number of states in our automaton is n and the number of transitions is m, we have the following theorem.

Theorem 1 *The equivalences* $\equiv_R$ *and* $\equiv_L$ *can be computed in time* $\mathcal{O}(m \log n)$ *and space* $\mathcal{O}(m + n)$.

It is interesting to notice that, to reduce NFAs by equivalences, we employed an idea from deterministic finite automata (DFA) reduction and then, to make it fast, we used an algorithm which was inspired itself from Hopcroft's algorithm [5] to reduce DFAs.

5 NFA reduction with preorders

Champarnaud and Coulon [2] noticed that a better reduction can be obtained if the axioms (P_1) and (P_2) above are used to construct a preorder relation instead of an equivalence. Let us denote the largest (w.r.t. inclusion) preorder which verifies (P_1) and (P_2) by $\subseteq_R$. It is then immediate that $p \subseteq_R q$ implies $\mathcal{L}_R(p) \subseteq \mathcal{L}_R(q)$.

As in the case of equivalences, the relation $\subseteq_L$ is symmetrically defined using the reversed automaton. Then, $p \subseteq_L q$ implies $\mathcal{L}_L(p) \subseteq \mathcal{L}_L(q)$.

The reduction with preorders is more complicated than with equivalences. First, we can merge two states p and q as soon as any of the following conditions is met:

(i) $p \subseteq_R q$ and $q \subseteq_R p$,
(ii) $p \subseteq_L q$ and $q \subseteq_L p$,
(iii) $p \subseteq_R q$ and $p \subseteq_L q$.

However, after merging two states, the preorders $\subseteq_R$ and $\subseteq_L$ must be updated such that their relation with the languages $\mathcal{L}_R$ and $\mathcal{L}_L$ (see above) is preserved. For instance, in the case (i), assuming the merged state of p and q is denoted q, the update amounts to removing from $\subseteq_R$ all pairs (q, s) for which $p \not\subseteq_R s$. Case (ii) is handled similarly and (iii) does not need any update.

An open problem here is how to merge the states using the two preorders such that the reduction of the NFA is optimal; see the example in Fig. 2.

Since the preorder requirement is weaker than equivalence, $p \equiv_R q$ implies that $p \subseteq_R q$ and $q \subseteq_R p$. The converse is not true in general (see [2] for an example). Therefore, using preorders we have a chance to obtain a better reduction of the NFA. It remains to investigate how much better.

6 Computing preorders

We give here an algorithm to compute the preorders $\subseteq_R$ and $\subseteq_L$. Assuming that $Q = \{1, 2, \ldots, n\}$ and the number of transitions is m, our algorithm runs in time $\mathcal{O}(mn)$ and space $\mathcal{O}(n^2)$. The best algorithm given by Champarnaud and Coulon [2] runs in time $\mathcal{O}(mn^2)$.

We shall compute the complement $\not\subseteq_R$ of $\subseteq_R$ by the algorithm PREORDER(M) from Fig. 3; ω is the relation which is $\not\subseteq_R$ at the end. According to the definition of $\subseteq_R$, its complement $\not\subseteq_R$ is the smallest relation over Q such that

(P_1') $(F \times (Q - F)) \subseteq \not\subseteq_R$,
(P_2') for any $i, j \in Q, a \in A$, $\big(\exists i' \in \delta(i,a), \forall j' \in \delta(j,a), i' \not\subseteq_R j' \Rightarrow i \not\subseteq_R j\big)$.

So, we add (i, j) to $\not\subseteq_R$ based on the fact that there is $i' \in \delta(i, a)$ for which the number of those $j' \in \delta(j, a)$ with $i' \not\subseteq_R j'$ is precisely $\mathsf{card}(\delta(j, a))$; that is, all j's. Therefore, we shall compute some matrices of counters $N(a)$, for any $a \in A$; $N(a)$ is a $n \times n$ matrix such that

$$N(a)_{ij} = \mathsf{card}(\{\ell \in \delta(j,a) \mid i \not\subseteq_R \ell\}),$$

for all $i, j \in Q$. We start with all these counters set to zero and update them anytime there is new information on $\not\subseteq_R$; any new pair added to $\not\subseteq_R$ is enqueued (steps 9 and 19) and later dequeued (step 11) and processed such that all counters involved are adequately updated (step 14). Anytime such a counter $N(a)_{ik}$ reaches maximum value $\mathsf{card}(\delta(k, a))$ (step 15), all pairs (j, k) such that $i \in \delta(j, a)$ are added to $\not\subseteq_R$ if not already there (steps 16–18).

Let us show that the algorithm PREORDER(M) computes correctly the preorder $\not\subseteq_R$. First, it is clear that $\not\subseteq_R$ is obtained by adding all pairs in (P_1') and then using (P_2') as long as pairs can still be added. Assume then

$$\not\subseteq_R = \{(i_1, j_1), \ldots, (i_r, j_r), \ldots, (i_s, j_s)\},$$

where the first r pairs are added because of (P_1') and the remaining ones due to (P_2'). We show that the algorithm PREORDER(M) computes the same relation. Denote by ω the relation computed by the algorithm. Obviously, $\omega \subseteq \not\subseteq_R$. Assume there is (i_t, j_t) in $\not\subseteq_R$ but not in ω; consider such a pair with the lowest index t. It must be that $t > r$ since all pairs in $F \times (Q - F)$ are certainly added to ω. As (i_t, j_t) is in $\not\subseteq_R$, there must be an $i' \in \delta(i_t, a)$ such that (i_t, j_t) was added to $\not\subseteq_R$ because all pairs (i', j'), $j' \in \delta(j_t, a)$, were already in $\not\subseteq_R$. Thus, at least one of

```
PREORDER(M)
- given: an NFA M
- returns: ⊈_R
 1. for q ∈ Q, a ∈ A do                                   //1–4: preprocessing
 2.     compute δ^r(q, a) as a linked list
 3.     compute card(δ(q, a))
 4. initialize all N(a)s with 0s
 5. ω ← ∅, C ← NEWQUEUE()                                 //5–19: processing
 6. for i ∈ F do                                          //6–8: initialize ω
 7.     for j ∈ Q − F do                                  //ω will be ⊈_R at the end
 8.         ω ← ω ∪ {(i, j)}
 9.         ENQUEUE(C, (i, j))
10. while C ≠ ∅ do
11.     (i, j) ← DEQUEUE(C)                               //11–19: updates due to
12.     for a ∈ A do                                      //(i, j) being added to ω
13.         for k ∈ δ^r(j, a) do
14.             N(a)_ik ← N(a)_ik + 1                     //14: update counter
15.             if N(a)_ik = card(δ(k, a)) then           //15–18: update ω
16.                 for j ∈ δ^r(i, a) do                  //when a counter is maximal
17.                     if (j, k) ∉ ω then
18.                         ω ← ω ∪ {(j, k)}
19.                         ENQUEUE(C, (j, k))
20. return ω
```

Fig. 3. Algorithm for computing preorders

those pairs (i', j') is not in ω. Since the index of (i', j') is strictly smaller than t, a contradiction is obtained.

The time complexity of the above algorithm is $\mathcal{O}(m+n^2)$ for the preprocessing and proportional to

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{a\in A}(\mathsf{card}(\delta^r(i,a)) + \mathsf{card}(\delta^r(j,a))) = \mathcal{O}(mn)$$

for processing. Therefore, the time complexity is $\mathcal{O}(mn)$. The space complexity is $\mathcal{O}(n^2)$. We have proved the following theorem.

Theorem 2 *The preorders $\subseteq_R$ and $\subseteq_L$ can be computed in time $\mathcal{O}(mn)$ and space $\mathcal{O}(n^2)$.*

7 Position automaton

We recall in this section the well-known construction of the position automaton[3], discovered independently by Glushkov [3] and McNaughton and Yamada [12].

[3] This automaton is sometimes called *Glushkov automaton*; e.g., in [18].

Let α be a regular expression. The basic idea of the position automaton is to assume that all occurrences of letters in α are different. For this, all letters are made different by marking each letter with a unique index, called its *position* in α. The set of positions of α is $\mathsf{pos}(\alpha) = \{1, 2, \ldots, |\alpha|_A\}$, where $|\alpha|_A$ is the number of letter occurrences in α. We shall denote also $\mathsf{pos}_0(\alpha) = \mathsf{pos}(\alpha) \cup \{0\}$. The expression obtained from α by marking each letter with its position is denoted $\overline{\alpha} \in \overline{A}^*$, where $\overline{A} = \{a_i \mid a \in A, 1 \le i \le |\alpha|_A\}$. For instance, if $\alpha = a(baa + b^*)$, then $\overline{\alpha} = a_1(b_2a_3a_4 + b_5^*)$. Notice that $\mathsf{pos}(\alpha) = \mathsf{pos}(\overline{\alpha})$. The same notation is also used for unmarking, that is, $\overline{\overline{a}} = a$.

Three mappings first, last, and follow are then defined as follows (see [3]). For any regular expression α and any $i \in \mathsf{pos}(\alpha)$, we have:

$$\begin{aligned} \mathsf{first}(\alpha) &= \{i \mid a_i w \in \mathcal{L}(\overline{\alpha})\}, \\ \mathsf{last}(\alpha) &= \{i \mid w a_i \in \mathcal{L}(\overline{\alpha})\}, \\ \mathsf{follow}(\alpha, i) &= \{j \mid u a_i a_j v \in \mathcal{L}(\overline{\alpha})\}. \end{aligned} \tag{1}$$

We extend follow by $\mathsf{follow}(\alpha, 0) = \mathsf{first}(\alpha)$. Also, let $\mathsf{last}_0(\alpha)$ stand for $\mathsf{last}(\alpha)$ if $\varepsilon \notin \mathcal{L}(\alpha)$ and $\mathsf{last}(\alpha) \cup \{0\}$ otherwise.

The *position automaton* for α is

$$\mathbf{A}_{\text{pos}}(\alpha) = (\mathsf{pos}_0(\alpha), A, \delta_{\text{pos}}, 0, \mathsf{last}_0(\alpha))$$

where

$$\delta_{\text{pos}} = \{(i, a, j) \mid j \in \mathsf{follow}(\alpha, i), a = \overline{a_j}\}.$$

Besides the property of accepting the language expressed by the original regular expression, that is, $\mathcal{L}(\mathbf{A}_{\text{pos}}(\alpha)) = \mathcal{L}(\alpha)$, the position automaton has two very important properties. First, the number of states is always $|\alpha|_A + 1$, which makes it work better then Thompson's automaton [20] in bit-parallel regular expression search algorithms [18]. Second, all transitions incoming to any given state are labelled by the same letter, a property exploited by Navarro and Raffinot [17, 18] in regular expression search algorithms to represent the DFA using $O(2^{|\alpha|_A} + |A|)$ bit-masks of length $|\alpha|_A$, rather than $O(2^{|\alpha|_A}|A|)$.

Example 2 Consider the regular expression $\tau = (a + b)(a^* + ba^* + b^*)^*$. The marked version of τ is $\overline{\tau} = (a_1 + b_2)(a_3^* + b_4a_5^* + b_6^*)^*$. The values of the mappings first, last, and follow for τ and the corresponding position automaton $\mathbf{A}_{\text{pos}}(\tau)$ are given in Fig. 4.

The position automaton can be computed easily in cubic time using the inductive definitions of first, last, and follow, but Brüggemann-Klein [1] showed how to compute it in quadratic time.

8 Reducing the position automaton

In this section we show how the position automaton can be reduced using equivalences and/or preorders such that its essential properties are preserved.

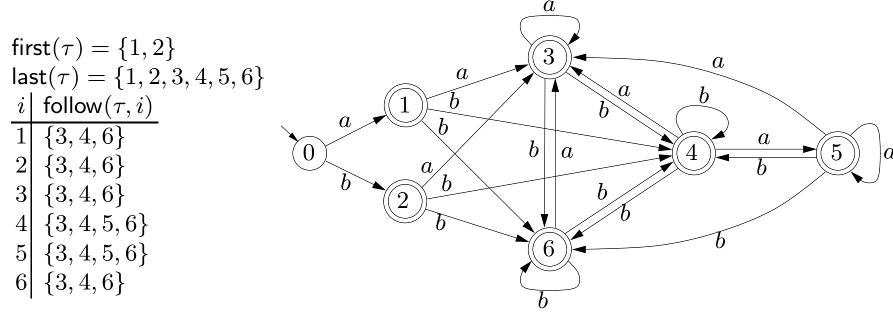

Fig. 4. $\mathbf{A}_{\text{pos}}(\tau)$ for $\tau = (a + b)(a^* + ba^* + b^*)^*$

Consider a regular expression α and define the equivalence $\sim_\ell$ over $\mathsf{pos}_0(\alpha)$ by $i \sim_\ell j$ iff the letter labelling all transitions incoming to i is the same as the one for j.

The idea is to reduce the position automaton such that the transitions incoming to a given state are still labelled the same. Therefore, any states we merge must be in $\sim_\ell$. Using equivalences, say $\equiv_R$, we merge according to the equivalence $\equiv_R \cap \sim_\ell$. Fig. 5 shows an example.

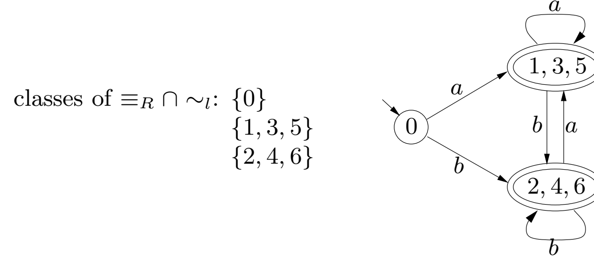

Fig. 5. $\mathbf{A}_{\text{pos}}(\tau)/_{\equiv_R \cap \sim_l}$ for $\tau = (a + b)(a^* + ba^* + b^*)^*$

Using preorders, we do just as before with the restriction imposed by $\sim_\ell$.

9 Experimental results

In this section we aim at establishing how significant is the reduction obtained using equivalences, and its relevance to regular expression search algorithms. In particular, we are interested in comparing two choices: *full right-equivalence*, where the properties of the position automaton are not preserved (Section 3), and *restricted right-equivalence*, where those properties are preserved (Section 8).

While full right-equivalence can potentially yield larger reductions in number of NFA states, it requires a representation of $O(2^{n_f}|A|)$ cells for the DFA (n_f is the number of reduced NFA states, A is the alphabet). Restricted right-equivalence may yield more states, but permits a more compact representation in $O(2^{n_r}+|A|)$ cells for the DFA (n_r is the number of NFA states after the restricted reduction).

We have tested regular expressions on DNA, having 10 to 100 alphabet symbols, averaging over 10,000 expressions per length, with density of operators from 0.1 to 0.4 (Section 9.1 gives more details on the generation process). For each such expression, we built its position automaton (Section 7), and then applied full and restricted reduction. Figure 6 shows the reductions obtained, as a fraction of the original number of states (which was always $n = 1 + |\alpha|_A$ because of Glushkov's construction). There is a second version of the full reduction, where the NFA is previously modified to include a self-loop at the initial state, for search purposes. This permits no further restricted reduction, but allows a slightly better full reduction, as it can be seen. Both reduction factors tend to stabilize as n grows, being better for higher density of operators in the regular expression. It is also clear that full reduction gives substantially better reductions compared to restricted reduction (10%-20% better).

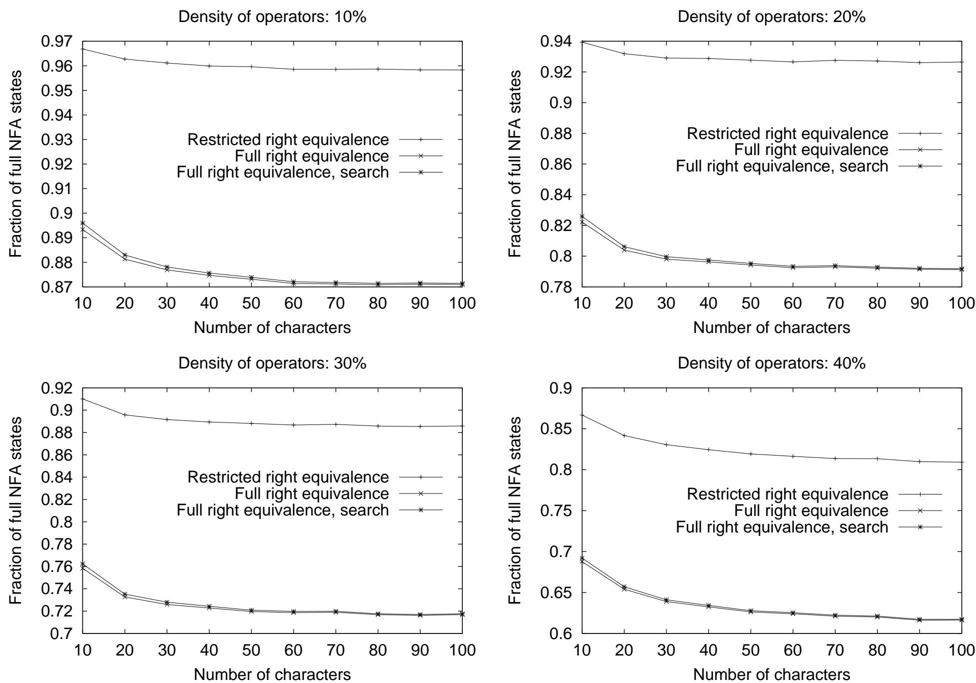

Fig. 6. Reduction factors in number of states obtained over position automata built from regular expressions of lengths 10 to 100 and density of operators from 0.1 to 0.4, built from DNA text.

As explained, the above does not immediately mean that full reduction is better, because its DFA representation must have one table per alphabet letter. Figure 7 shows the reduction fraction in the representation of the DFA, compared to that of the position automaton. This time the difference between the search and the original automaton are negligible. As it can be seen, the restricted reduction is convenient only for $n \leq 10$ to $n \leq 30$, depending on the operator density. Note, on the other hand, that those short expression lengths imply that even the original position automaton is not problematic in terms of space or construction cost. That is, full reduction becomes superior precisely when the space problem becomes important.

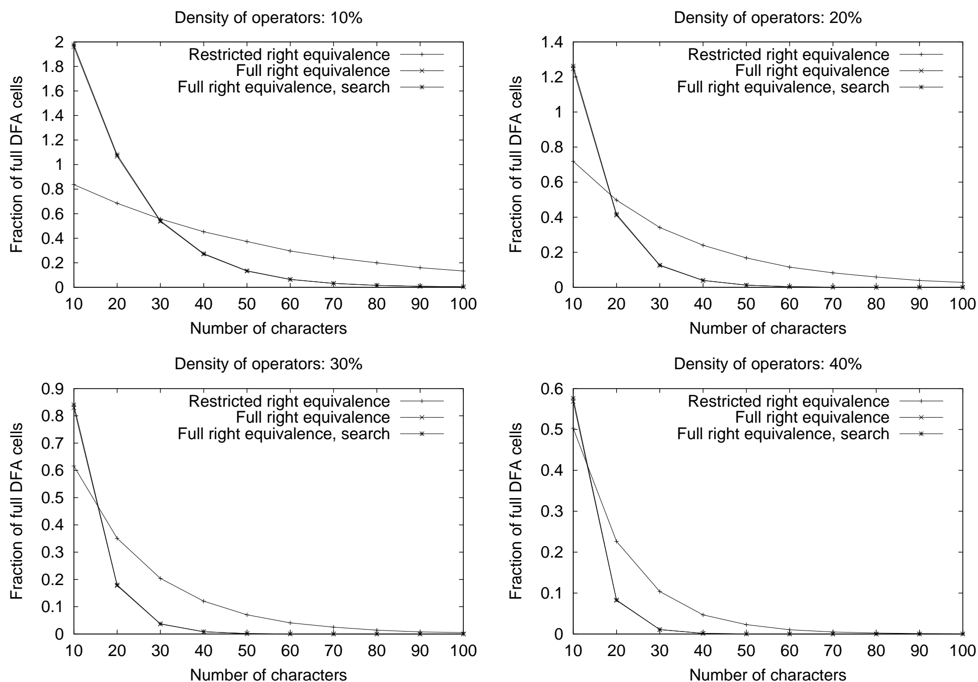

Fig. 7. Reduction factors in DFA sizes obtained over position automata built from regular expressions of lengths 10 to 100 and density of operators from 0.1 to 0.4, built from DNA text.

Figure 8 shows the same results in logarithmic scale, to show how large are the savings due to full reductions when n becomes large. The second version of NFA (for searching) is omitted since the difference in DFA size is unnoticeable.

As noted in [17, 18], DFA space and construction cost can be traded for search cost as follows: A single table of $O(2^n)$ entries can be split into k tables of size $O(2^{n/k})$ each, so that each such table has to be accessed for each text character in order to build the original entry. Hence the search cost becomes

$O(k)$ per text character. If main memory is limited, a huge DFA actually means larger search time, and a reduction in its size translate into better search times. We have computed the number of tables needed with and without reductions assuming that we dedicate 4 megabytes of RAM to the DFA. For the highest operator density we have obtained speedups of up to 50% in search times, that is, we need 2/3 of the tables needed by the original automaton.

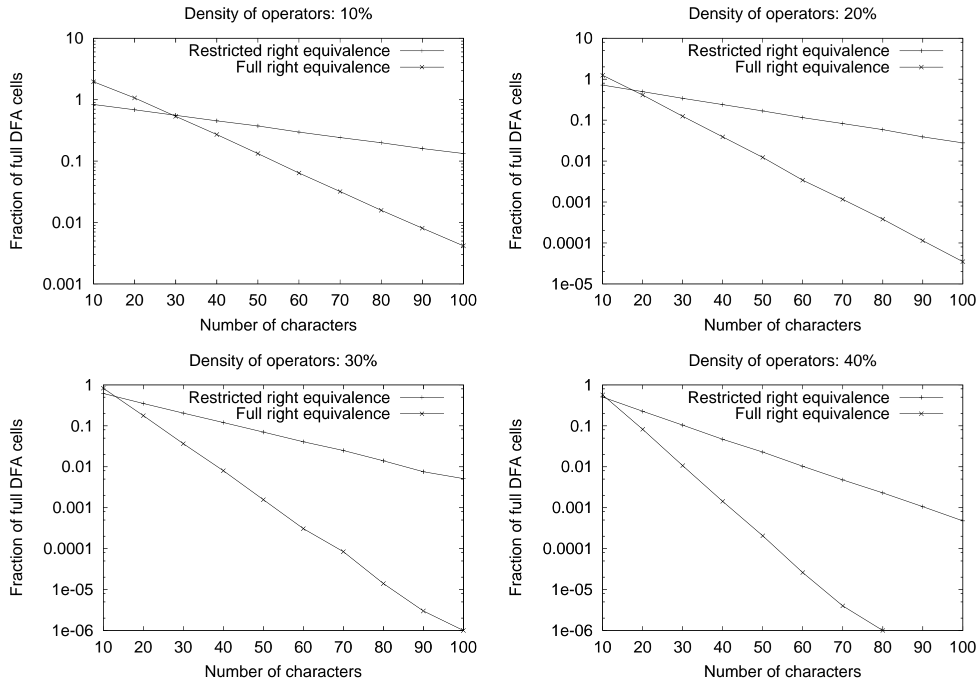

Fig. 8. Reduction factors in DFA sizes obtained over position automata built from regular expressions of lengths 10 to 100 and density of operators from 0.1 to 0.4, built from DNA text. Note the logarithmic scale.

9.1 Generating Regular Expressions

The choice of test patterns is always problematic when dealing with regular expressions, since there is no clear concept of what a random regular expression is and, as far as we know, there is no public repository of regular expressions available, except for a dozen of trivial examples. We have chosen to generate random regular expressions as follows:

1. We choose a base real-world text, in this case DNA from Homo Sapiens.
2. We choose n and pick a random text substring of length n.

3. We choose an *operator density* $0 \leq \gamma \leq 1$.
4. We apply a recursive procedure to convert a string of length ℓ into a regular expression:
   (a) An empty string is converted into an empty regular expression. In the rest, we assume a nonempty string.
   (b) With probability $1 - \gamma$ we choose that the expression will be the concatenation of two subexpressions: a left part of ℓ' characters and a right part of $\ell - \ell'$ characters, where ℓ' is chosen uniformly in the range $1 \leq \ell' \leq \ell - 1$. We recursively convert both subparts into regular expressions e_1 and e_2. The resulting expression is $e_1 \cdot e_2$. If $\ell = 1$ we simply write down the string character.
   (c) Otherwise, if the parent in the recursion has just generated a Kleene closure operator "$*$", we choose to add a union operator "$|$", if not, we choose with the same probability between a Kleene closure and a union.
   (d) If we chose that the expression will have a union operator, we choose a left part of ℓ' characters and a right part of $\ell - \ell'$ characters, where ℓ' is chosen uniformly in the range $0 \leq \ell' \leq \ell$. We recursively convert both subparts into regular expressions e_1 and e_2. The resulting expression is $e_1|e_2$.
   (e) If we chose to add a Kleene closure operator "$*$" at the end of the string, we recursively generate a regular expression e_1 for the string. The resulting expression is e_1*.

The above procedure is just one of the many possible alternatives to generate random regular expressions one could argue for, but it has a couple of advantages. First, it permits determining the length n (number of characters of A) in advance. Second, it takes the characters from the text, respecting its distribution. Third, it permits us to choose expressions with more or less operators by varying γ. We show experiments with $\gamma = 0.10$ to $\gamma = 0.40$. Examples obtained from our tests, with $n = 10$, are "$ACAT(T|\varepsilon)TT * AG(T|\varepsilon)$" and "$A(CAT(\varepsilon|T*) * ((\varepsilon|\varepsilon|T) * TA*) * (\varepsilon|\varepsilon|GT))*$", respectively.

10 Conclusion

We have developed faster algorithms to implement two existing NFA reduction techniques. We have also adapted them to work over position automata while preserving their properties that allow a compact DFA representation. Finally, we have empirically assessed the practical impact of the reductions, as well as the convenience of preserving or not the position automata properties.

Future work involves empirically evaluating the impact of using preorders instead of equivalences. The former are more complex and slower to compute, and it is not clear which is the optimal way to apply the different reductions, hence the importance of determining their practical value.

References

1. Brüggemann-Klein, A., Regular expressions into finite automata, *Theoret. Comput. Sci.* **120** (1993) 197 – 213.
2. Champarnaud, J.-M., and F. Coulon, NFA reduction algorithms by means of regular inequalities, in: Z. Ésik, Z. Fülöp, eds., *Proc. of DLT 2003* (Szeged, 2003), Lecture Notes in Comput. Sci. **2710**, Springer-Verlag, Berlin, Heidelberg, 2003, 194 – 205.
3. Glushkov, V.M., The abstract theory of automata, *Russian Math. Surveys* **16** (1961) 1 – 53.
4. Hagenah, C., and Muscholl, A., Computing ϵ-free NFA from regular expressions in $O(n\log^2(n))$ time, *Theor. Inform. Appl.* **34** (4) (2000) 257 – 277.
5. Hopcroft, J., An $n\log n$ algorithm for minimizing states in a finite automaton, *Proc. Internat. Sympos. Theory of machines and computations*, Technion, Haifa, 1971, Academic Press, New York, 1971, 189–196.
6. Hopcroft, J.E., and Ullman, J.D., *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Mass., 1979.
7. Hromkovic, J., Seibert, S., and Wilke, T., Translating regular expressions into small ϵ-free nondeterministic finite automata, *J. Comput. System Sci.* **62** (4) (2001) 565 – 588.
8. Ilie, L., and Yu, S., Algorithms for computing small NFAs, in: K. Diks, W. Rytter, eds., *Proc. of the 27th MFCS*, (Warszawa, 2002), Lecture Notes in Comput. Sci., **2420**, Springer-Verlag, Berlin, Heidelberg, 2002, 328 – 340.
9. Ilie, L., and Yu, S., Reducing NFAs by invariant equivalences, *Theoret. Comput. Sci.* **306** (2003) 373 – 390.
10. Jiang, T., and Ravikumar, B., Minimal NFA problems are hard, *SIAM J. Comput.* **22**(6) (1993), 1117 – 1141.
11. Kameda, T., and Weiner, P., On the state minimization of nondeterministic finite automata, *IEEE Trans. Computers* **C-19**(7) (1970) 617 – 627.
12. McNaughton, R., and Yamada, H., Regular expressions and state graphs for automata, *IEEE Trans. on Electronic Computers* **9** (1) (1960) 39 – 47.
13. Melnikov, B. F., A new algorithm of the state-minimization for the nondeterministic finite automata, *Korean J. Comput. Appl. Math.* **6**(2) (1999) 277 – 290.
14. Melnikov, B. F., Once more about the state-minimization of the nondeterministic finite automata, *Korean J. Comput. Appl. Math.* **7**(3) (2000) 655–662.
15. Navarro, G., NR-grep: a Fast and Flexible Pattern Matching Tool, *Software Practice and Experience* **31** (2001) 1265 – 1312.
16. Navarro, G., and Raffinot, M., Fast Regular Expression Search, *Proc. WAE'99*, Lecture Notes Comput. Sci. **1668**, Springer-Verlag, Berlin, Heidelberg, 1999, 198 – 212.
17. Navarro, G., and Raffinot, M., Compact DFA Representation for Fast Regular Expression Search, *Proc. WAE'01*, Lecture Notes Comput. Sci. **2141**, Springer-Verlag, Berlin, Heidelberg, 2001, 1 – 12.
18. Navarro, G., and Raffinot, M., *Flexible Pattern Matching in Strings. Practical On-Line Search Algorithms for Texts and Biological Sequences*, Cambridge University Press, Cambridge, 2002.
19. Paige, R., and Tarjan, R.E, Three Partition Refinement Algorithms, *SIAM J. Comput.* (1987) **16**(6) 973 – 989.
20. Thompson, K., Regular expression search algorithm, *Comm. ACM* **11** (6) (1968) 419 – 422.

21. Wu, S., and Mamber, U., Fast text searching allowing errors, *Comm. ACM* **35**(10) (1992) 83 – 91.
22. Yu, S., Regular Languages, in: G. Rozenberg, A. Salomaa, *Handbook of Formal Languages, Vol. I*, Springer-Verlag, Berlin, 1997, 41 – 110.