Short Regular Expressions from Finite Automata: Empirical Results

Hermann Gruber[1], Markus Holzer[1], and Michael Tautschnig[2]

[1] Institut für Informatik, Universität Giessen
Arndtstraße 2, D-35392 Giessen, Germany
email: {hermann.gruber,holzer}@informatik.uni-giessen.de
[2] Fachbereich Informatik, Technische Universität Darmstadt,
Hochschulstraße 10, D-64289 Darmstadt, Germany
email: tautschnig@forsyte.cs.tu-darmstadt.de

Abstract. We continue our work [H. Gruber, M. Holzer: Provably shorter regular expressions from deterministic finite automata (extended abstract). In *Proc. DLT*, LNCS 5257, 2008] on the problem of finding good elimination orderings for the state elimination algorithm, one of the most popular algorithms for the conversion of finite automata into equivalent regular expressions. Here we tackle this problem both from the theoretical and from the practical side. First we show that the problem of finding optimal elimination orderings can be used to estimate the cycle rank of the underlying automata. This gives good evidence that the problem under consideration is difficult, to a certain extent. Moreover, we conduct experiments on a large set of carefully chosen instances for five different strategies to choose elimination orderings, which are known from the literature. Perhaps the most surprising result is that a simple greedy heuristic by [M. Delgado, J. Morais: Approximation to the smallest regular expression for a given regular language. In *Proc. CIAA*, LNCS 3317, 2004] almost always outperforms all other strategies, including those with a provable performance guarantee.

1 Introduction

The classical theorem of Kleene [15] implies that every n-state finite automaton over alphabet Σ admits an equivalent regular expression. This conversion problem has received quite some attention recently, see, e.g., [9–13]. One of the most popular algorithms for this conversion is the so called *state elimination* algorithm. There, states from the automaton are successively eliminated by rerouting the in- and out-going transitions, which leads to an automaton with transitions labeled by regular expressions. The sequence of states eliminated thereby is called an *elimination ordering* or *elimination sequence*. If state elimination is applied to an n-state finite automaton, the resulting expression is of size at most $|\Sigma| \cdot 4^n$. While this bound appears large, it is known that an exponential blowup is necessary in the worst case [5, 10].

These theoretical results pushed the on-going quest for heuristics finding good elimination orderings leading to short regular expressions [2, 11, 13]. Recently

improved upper bounds on the size of the regular expressions resulting from deterministic finite automata (DFA) over small alphabets were obtained [11]. The latter are based on algorithms with a combinatorial flavor, and the analysis is facilitated by results from extremal graph theory. Although all of the heuristics choose an ordering for state elimination, only the algorithms from [11] lend themselves to a theoretical analysis at all. Thus a theoretical comparison of the different approaches would appear rather difficult.

In the present paper we continue our research on good elimination orderings for the state elimination algorithm from the theoretical as well as the practical side by doing experiments on a large dataset. On the theoretical side, we investigated the possibility whether designing an efficient approximation algorithm would be within reach, but our theoretical results are somewhat negative. Already weak approximation algorithms for the elimination orderings would constitute a major step towards a resolving the approximability of the undirected cycle rank problem, which is not completely understood yet [7]. Therefore we implemented some of the heuristics and compared their performance on a large but carefully chosen set of test instances. In short, the main empirical observations are the following: (1) Even the easiest heuristics provide a huge advantage over randomly chosen elimination orderings, thus substantiating an observation made earlier in [2] on very few instances. (2) Larger alphabets, and hence more transitions, in the given DFAs result in larger regular expressions. (3) For ε-NFAs obtained from regular expressions using the standard construction [14], the transformation back into regular expressions is much easier than for random DFAs. (4) Simplifying intermediate regular expressions on-the-fly as they appear during the conversion appears not to have a striking effect on the result on the average. Perhaps most surprisingly, it turned out that the simple greedy heuristic by Delgado and Morais [2] almost always outperforms the algorithms with provable performance guarantee from [11].

2 Definitions

We assume the reader to be familiar with basic notions in formal language theory, in particular with those of ε-NFAs, NFAs, DFAs, regular expressions, and the languages they denote. Here we follow exactly the notational conventions from [14], with the following additions: The *size* or *alphabetic width* of a regular expression r over the alphabet Σ, denoted by $\mathrm{alph}(r)$, is defined as the total number of occurrences of letters of Σ in r. For a regular language L, we define its alphabetic width, $\mathrm{alph}(L)$, as the minimum alphabetic width among all regular expressions describing L. When working with ε-NFAs, we will often assume that the given automaton A is *normalized* in the sense that A has the state set $Q \cup \{s, t\}$ where s is the start state and has no incoming transitions, and t is the sole accepting state and has no outgoing transitions. This can be achieved by a simple construction if needed. As usual, two finite automata are called *equivalent* if they accept the same language.

Now we present an algorithm scheme that became known as *state elimination*, cf. [19]. Let Q be the state set of a finite automaton A. For a subset U of Q and an input word $w \in \Sigma^*$, we say that A can go on input w from state j through U to state k, if it has a computation on input w taking A from state j to k without going through any state outside U. Here, by "going through a state," we mean both entering and leaving. Now let L_{jk}^U be the set of words on which A can go from state j to state k through U. Observe that in particular for a normalized finite automaton A then holds $L_{st}^Q = L(A)$. The state elimination algorithm scheme proceeds as follows: We maintain a working set U and a matrix the entries of which are regular expressions r_{jk}^U denoting the languages L_{jk}^U. The algorithm proceeds in rounds: Beginning with $U = \emptyset$, we enlarge the set U by adding a new state $i \in Q \setminus U$ in each round. The round consists of computing the new entries denoting the languages $L_{jk}^{U \cup \{i\}}$, for each j, k satisfying $j, k \notin U \cup \{i\}$, by letting $r_{jk}^{U \cdot i} = r_{ji}^U \cdot (r_{ii}^U)^* \cdot r_{ik}^U$, where $U \cdot i$ denotes the ordering induced by U followed by i (cf. [11]). Here it is understood that the resulting expression on the left-hand side equals $\emptyset$ if r_{ji}^U or r_{ik}^U denotes the empty set. If the given automaton was normalized, we finally end up with a regular expression describing L_{st}^Q, a set equal to $L(A)$. Observe that the above algorithm requires an ordering in which the states i are to be processed one after another; such an ordering on Q is called an *elimination ordering*. It is well known that the choice of ordering can greatly influence the size of the resulting regular expressions, cf. [2, 19].

3 A Theoretical Result on Elimination Orderings

This section is devoted to the question whether we can find an optimum, or at least an approximately optimum elimination ordering in polynomial time. Sakarovitch stated that this is probably a hard combinatorial problem [19]. Although we cannot provide proper evidence that this problem is algorithmically intractable (such as **NP**-hardness), our result indicates that even designing an approximation algorithm with a reasonable performance guarantee is a challenging research problem.

Definition 1. *The* cycle rank *of a digraph $G = (V, E)$, denoted by $cr(G)$, is inductively defined as follows: (1) If G is acyclic, then $cr(G) = 0$. (2) If G is strongly connected and not acyclic, then $cr(G) = 1 + \min_{v \in V}\{cr(G - v)\}$. (3) If G is not strongly connected, then $cr(G)$ equals the maximum cycle rank among all strongly connected components of G. The* undirected *cycle rank of G is defined as the cycle rank of its symmetric closure.*

We will relate the undirected cycle rank to elimination orderings in the following. To this end, recall the following lemma from [11]:

Lemma 2. *Let A be a normalized ε-*NFA *with state set $\{s, t\} \cup Q$, and let G be the digraph underlying the transition structure of A. Assume $U \subseteq Q$ can be partitioned into two sets T_1 and T_2 such that the induced subdigraph $G[U]$ falls apart into mutually disconnected components $G[T_1]$ and $G[T_2]$. Then for*

the expression $r_{jk}^{T_1 \cdot T_2}$ *obtained by elimination of the the vertices in* T_1 *followed by elimination of the vertices in* T_2 *it holds* $r_{jk}^{T_1 \cdot T_2} \cong r_{jk}^{T_1} + r_{jk}^{T_2}$*, for all states* $j, k \in Q \setminus U$*.*

Using this lemma, we can prove that the undirected cycle rank of the underlying graph is a parameter that renders the problem of converting ε-NFAs into regular expressions fixed-parameter tractable—not in the usual sense of *computational*, but rather of *descriptional* complexity. We omit the proof of the next two statements due to space constraints.

Theorem 3. *Let* A *be a normalized* ε-NFA *with state set* $\{s, t\} \cup Q$*, let* c *be a positive integer, and let* G *be its underlying (di)graph. If* $U \subseteq Q$ *is such that* $G[U]$ *has undirected cycle rank at most* c*, then there is an elimination ordering for* U *which yields, for all states* j, k *in* $Q \setminus U$*, regular expressions* r_{jk}^{U} *of size at most* $|\Sigma| \cdot 4^c \cdot |U|$*.*

The problem in transforming the above result into an algorithm is that determining the undirected cycle rank of a graph or digraph is **NP**-complete and the best known approximation algorithm has, for a given graph with n vertices and (unknown) undirected cycle rank c, a performance ratio of $O(\sqrt{\log c} \cdot \log n)$, see [7]. It turns out that merely estimating the *order of magnitude* of the expression size resulting from an optimum ordering is by no means easier:

Lemma 4. *Given an undirected graph* G *on* n *vertices and of (unknown) cycle rank* c*, we can construct in polynomial time a* DFA A *such that the optimum elimination ordering for* A *yields an equivalent regular expression* r *with*

$$\frac{1}{3} \cdot c - 2 \leq \log \operatorname{alph}(r) \leq 2 \cdot c + \log n.$$

This shows that the optimum ordering expression size can be used as a pretty good estimate for the cycle rank, and already weak approximation algorithms for the former problem would constitute a major step towards a more complete understanding of the approximability of the undirected cycle rank problem, compare [7].

4 Algorithms for Choosing Elimination Orderings

When eliminating a state with m entering and n exiting transitions, the resulting digraph has up to $(m-1) \cdot (n-1)$ newly added edges. Intuitively, we want to keep the intermediate digraphs produced by the elimination process as sparse as possible. Thus it may be advisable to delay the elimination of heavily trafficked states as long as possible, as noted already by different authors [2, 13, 19]. An extremely simple strategy is to order the states by a measure that is defined as the number of ingoing edges times the number of outgoing edges (Algorithm **0A**). An easy observation is that this measure can of course change as the elimination proceeds, and a refined strategy recomputes these measures on the intermediate

digraphs after each elimination round (Algorithm **0B**). A further refinement devised in [2] works with a measure function, which also takes the size of the intermediate regular expressions into account (Algorithm **DM**)—we refer to [2] for details.

Recently, two new ordering algorithms were discovered in [11], which were also the first ones to come with a provably better performance guarantee on the resulting regular expressions, at least in case the given automata are deterministic and over not too large alphabets. Turán's Theorem in extremal graph theory states that sparse (di)graphs have independent sets of linear size, and that these can be eliminated at low cost and can be found by a simple greedy algorithm. This gives rise to the following algorithm (Algorithm **IS**): First, we find a huge independent set S in the graph underlying the automaton. Then we order the states in S arbitrarily, and eliminate them. For any remaining states we again find a huge independent set in the resulting digraph, and so on. It is shown in [11] that this algorithm is guaranteed to produce regular expressions of size at most $O(2.602^n)$, when given a DFA over binary alphabet.

A recent generalization of Turán's theorem by Edwards and Farr [3] concerns induced subgraphs of (undirected) treewidth at most 2 instead of independent sets. There the guaranteed size is three times larger, and again these can be eliminated at low cost and can be found by a simple greedy algorithm. Large induced subgraphs of low treewidth are useful, because it was proved in that these admit orderings, similar to independent set, such that eliminating them in the beginning can incur an increase in intermediate expression size bounded by a polynomial factor. The proof of that fact does not rely directly on tree decompositions, but proceeds by finding small balanced separators, and then recurring on the separated subgraphs. Following [11], this suggests the following algorithm (Algorithm **B3S**): First, we find a huge induced subgraph S of treewidth at most 2 in the graph underlying the automaton. Then we order the set S by finding a balanced 3-way separator X for S. If C_1, C_2 and C_3 are the parts of S separated by X, the resulting ordering is of the form C_1, C_2, C_3, X, where the ordering for the component C_i is found recursively, by finding a balanced 3-separator for C_i, and so on. Then we eliminate S. Finally, we eliminate $Q \setminus S$ with an "arbitrary" ordering; to optimize the latter, we used the heuristic **DM** on $Q \setminus S$. For finding huge independent sets and induced subgraphs of treewidth 2, we used a software library developed by Kerri Morgan [17]. It was shown in [11] that this approach allows for a guaranteed performance of $O(1.742^n)$ on DFAs over binary alphabet. We also note that all of the algorithms run in output polynomial time. In particular, they run in polynomial time provided they produce a regular expression of polynomial size.

5 Experiments

We have conducted experiments with the algorithms described in the previous section plus an additional random elimination ordering (**RA**). We have implemented the algorithms in C++, using the Automata Standard Template Library

(ASTL) [16] for representation and manipulation of automata. We have chosen the library ASTL to represent the NFAs and the intermediate results during state-elimination mainly because of the cursor concept and because it allows arbitrary input alphabets. This facilitates a direct implementation of the algorithms by programming appropriate iterators (*cursors*) over the state set. To gain performance, regular expressions are not stored as syntax trees, but as directed acyclic graphs, allowing for sharing common subexpressions. Similar ideas were used in [8]. Tests have shown that this allows us to compute regular expressions of very large alphabetic width—up to 10^{20}—while still having small memory footprint. The front end for reading the input NFAs or regular expressions is a `lex` and `yacc` generated parser. This resulted in an overall size of roughly 4000 lines of code. The tests are performed on a quad core Intel Xeon CPU E5345 with 2.33 GHz equipped with 16 GB RAM running Linux as an operating system. To limit the number of bugs in our program, unit-tests were performed with the help of the *Diagnostics* framework.[3]

Moreover, simplification of the regular expressions constructed during state elimination can be en- or disabled. Unless stated otherwise, all tests were run with simplification turned on. The simplification process is described by a term rewriting system (TRS) which works modulo ACIZ-identities[4] and some further identities, namely $r \cdot \varepsilon = \varepsilon \cdot r = r$, $a \cdot \emptyset = \emptyset \cdot r = \emptyset$, plus identities that deal with the Kleene star, which are $\emptyset^* = \varepsilon$, $\varepsilon^* = \varepsilon$, $(r^*)^* = r^*$, $(r + \varepsilon)^* = r^*$, $(r + s^*)^* = (r + s)^*$, and $(r^* s^*)^* = (r + s)^*$. They are similar to those used in [8]. Notice that the associativity laws can be built into the data structure by using list data types. For implementing commutativity we defined, apart from the above notion of equivalence, also an appropriate order on the subsorts of expressions.

Our test instances are chosen as follows: We used randomly generated DFAs for different numbers of states and alphabet sizes and regular expressions of varying alphabetic width. Moreover, we have also performed tests on special automata instances that appeared in the literature—we will discuss this issue in more detail below. To randomly generate DFAs (more precisely initially connected DFAs) we used the FAdo toolkit [1], while the random generation of regular expressions was done by GenRGenS [18]. The latter software was originally designed to randomly generate genome sequences and supports several classes of models, including context-free grammars. Observe that the generation of DFAs is uniformly at random. A running time limit for all tests was not established and all tests were finished after about 30 CPU days. All test instances and the source code are available online at `http://code.forsyte.de/automata` for

[3] *Diagnostics*, developed by the "Formal Methods in Systems Engineering" group at Technische Universität Darmstadt, is a unified framework for code annotation, logging, program monitoring, and unit-testing. Download and more information is available at `http://code.forsyte.de/diagnostics`.

[4] The set of equations $r + (s + t) = (r + s) = t$, $r + s = s + r$, $r + r = r$, and $r + \emptyset = r$ are commonly called ACIZ-identities or -axioms, where letter A is an abbreviation for associativity, C for commutativity, I for idempotency, and Z for zero absorption.

download. The unambiguous grammar used for generating regular expressions is included with the download.

For the DFA samples we used automata with $5 \leq n \leq 50$ states—in steps of 5 states—and input alphabets with $1 \leq k \leq 10$ symbols. We only show the diagrams for $k \in \{2, 3, 5, 10\}$. For each parameter n and k, a sample of 1000 random instances was generated and tested. The results are summarized in Figure 1 and can be interpreted as follows.

At first glance one observes that larger alphabet size, and hence more transitions, in the DFAs result in larger regular expressions. This is of course expected. Taking a closer look, one further observes that the Algorithm **0B** with the simple greedy strategy and the Algorithm **DM** with the more sophisticated measure function of Delgado and Morais [2] almost always outperforms the Algorithms **IS** and **B3S** with provable performance guarantees—indicated by fitted appropriate exponential functions—from [11], on the average. This was a nice surprise and

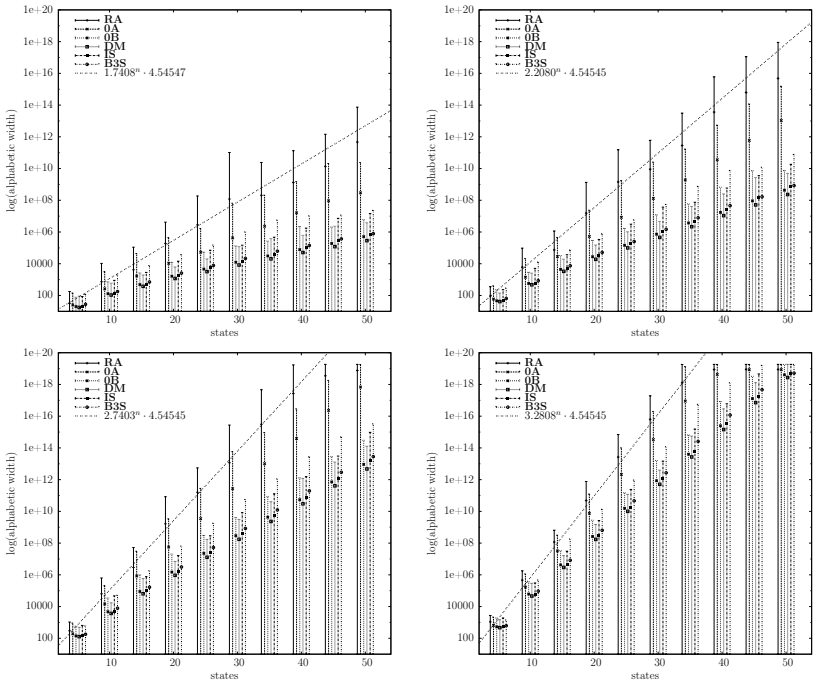

Fig. 1. Alphabetic size (y-axis, logarithmically scaled) in relation to the number of states (x-axis, linearly scaled) for DFAs with $5 \leq n \leq 50$ states—in steps of 5 states—and input alphabet size 2 (upper left), 3 (upper right), 5 (lower left), and 10 (lower right) for the random ordering **RA** and Algorithms **0A**, **0B**, **DM**, **IS**, and **B3S**. Here a vertical bar for an algorithm indicates the maximal occurring alphabetic width by its height. Moreover it also shows the alphabetic width on average indicated by the appropriate mark symbol.

not really expected. Apart from the random ordering **RA**, only Algorithm **0A** is significantly worse than the other tested algorithms, not only on the average, but also on the worst-case behavior. Again, not a real surprise, since this algorithm is too static, by not taking changes of the underlying graph during the elimination process into account. Moreover, another effect, not depicted here, was encountered during our test: Simplifying of intermediate regular expressions does not have any significant effect on the outcome of the conducted experiments. Possible reasons for this may be that we have run our tests on DFAs, not NFAs, and that we have excluded more powerful simplification rules such as $r \cdot s + r \cdot t = r \cdot (s + t)$. We plan to conduct further experiments in this direction.

Next we summarize our results on special instances, which were already discussed in the literature. First we have reproduced the experiments done in [2] on automata with transformation monoids from $\mathcal{POI}_n$ and $\mathcal{POPI}_n$ of all injective order preserving and orientation preserving, respectively, partial transformations on a chain with n elements. Moreover, we have considered DFAs whose transition structure is a $n \times n$ grid graph with a input alphabet of size 4, one letter for each direction. Recently, these automata, referred to as $Grid_n$, were used to prove lower bounds on the alphabetic width for the conversion of planar DFAs to regular expressions [10]. Finally, we also considered DFAs accepting the languages $L_n = \Sigma^* \setminus (\Sigma^* f_n \Sigma^*)$, where $\Sigma = \{a, b\}$ and f_n is the nth finite Fibonacci word defined by $f_0 = a$, $f_1 = ab$, and $f_n = f_{n-1} \cdot f_{n-2}$, for $n \geq 2$. These automata denoted by Fib_n were proposed in [6] as possibly difficult candidates for converting DFAs into regular expressions. Some of the obtained results are summarized in Table 1. Here a similar situation shows up as for random instances. The Algorithm **DM** is superior to the other algorithms. Furthermore, the automata Fib_n don't show the conjectured behavior as difficult candidates for converting DFAs into regular expressions. Here the grid automata $Grid_n$ are much more difficult as indicated by the enormously large alphabetic width of at most $1.1 \cdot 10^{17}$ produced by the Algorithm **IS**.

We also studied the setup when starting with a regular expression instead of a DFA. For the conversion from a regular expression to a finite automaton we have implemented Thompson's algorithm [14]. Again Algorithm **DM** outperforms all

	Instance							
Algorithm	(A)	(B)	(C)	(D)	$Grid_3$	$Grid_{12}$	Fib_3	Fib_{12}
RA	1304	93688	7426	1404252	7516	$\leq 1.3 \cdot 10^{19}$	7	2119
0A	1121	54625	2425	819934	1988	$\leq 4.5 \cdot 10^{18}$	11	70877
0B	634	25816	882	13240	634	$\leq 1.1 \cdot 10^{18}$	5	377
DM	491	8989	704	11528	622	$\leq 7.7 \cdot 10^{17}$	5	377
IS	535	9750	929	14701	634	$\leq 1.1 \cdot 10^{17}$	8	1584
B3S	768	11663	793	13062	958	$\leq 1.2 \cdot 10^{19}$	9	1586

Table 1. Results on the alphabetic width for some specific DFAs instances that appeared already in the literature [2, 6, 10]. In particular, (A), (B), (C), (D) denote the automata $Min(\mathcal{POI}_4[1, 20])$, $Min(\mathcal{POI}_5[1, 125])$, $Min(\mathcal{POPI}_4[1, 60])$ and $Min(\mathcal{POPI}_5[1, 70])$ as they appear in [2].

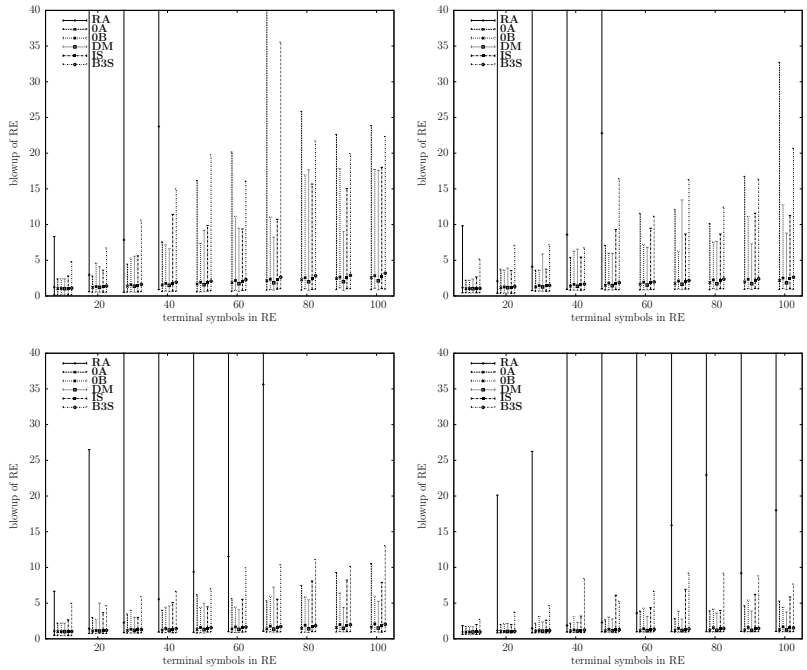

Fig. 2. Blowup (y-axis, linearly scaled) in relation to the length n (x-axis, linearly scaled) of the regular expression (RE) for $10 \leq n \leq 100$ in steps of 10 caused by the transformation RE $\rightarrow$ NFA $\rightarrow$ RE for input alphabet size 2 (upper left), 3 (upper right), 5 (lower left), and 10 (lower right) for the random ordering **RA** and Algorithms **0A**, **0B**, **DM**, **IS**, and **B3S**. The length of a regular expression is defined to be the number of terminal symbols. Here a vertical bar for an algorithm indicates the maximal occurring blowup. Moreover it also shows the average bloat factor indicated by the corresponding mark symbol. Missing lines for **RA** indicate that even the average blowup is greater than 40.

the other algorithms; but note that the resulting expressions are much smaller than for random DFAs. For regular expressions of length $10 \leq n \leq 100$ in steps of 10 and input alphabet size $1 \leq k \leq 10$ the size blowup is depicted in Figure 2—again only the diagrams for $k \in \{2, 3, 5, 10\}$ are shown.

Whether using a different conversion algorithm than Thompson's can affect the obtained results is not clear and has to be verified by further experiments. Finally, we mention that we believe that all of the algorithms under consideration would equally benefit from the preprocessing techniques presented in [13]; in particular we do not expect that they have a noticeable effect on random DFA input.

Acknowledgment. We are indebted to Loek Cleophas, Manuel Delgado, Vincent Le Maout, and Kerri Morgan for helping us to effectively build on their previous research.

References

1. Almeida, M., Moreira, N., Reis, R. Enumeration and generation with a string automata representation. *Theor. Comput. Sci.* 387(2):93–102, 2007.
2. Delgado, M., Morais, J. Approximation to the smallest regular expression for a given regular language. In *CIAA 2004*, *LNCS*, vol. 3317, pp. 312–314. Springer, 2004.
3. Edwards, K., Farr, G. E. Planarization and fragmentability of some classes of graphs. *Discrete Math.*, 308(12):2396–2406, 2008.
4. Eggan, L. C. Transition graphs and the star height of regular events. *Mich. Math. J.*, 10:385–397, 1963.
5. Ehrenfeucht, A., Zeiger, H. P. Complexity measures for regular expressions. *J. Comput. Syst. Sci.* , 12(2):134–146, 1976.
6. Ellul, K., Krawetz, B., Shallit, J., Wang, M. Regular expressions: New results and open problems. *J. Autom. Lang. Comb.*, 10(4):407–437, 2005.
7. Feige, U., Hajiaghayi, M., Lee, J. R. Improved approximation algorithms for minimum weight vertex separators. *SIAM J. Comput.*, 38(2):629-657, 2008.
8. Frishert, M., Cleophas, L. G., Watson, B. W. The effect of rewriting regular expressions on their accepting automata. In *CIAA 2003*, *LNCS*, vol. 2759, pp. 304–305. Springer, 2003.
9. Gelade, W., Neven, F. Succinctness of the complement and intersection of regular expressions. In *STACS 2008*, *Dagstuhl Seminar Proceedings*, vol. 08001, pp. 325–336. IBFI Schloss Dagstuhl, 2008.
10. Gruber, H., Holzer, M. Finite automata, digraph connectivity, and regular expression size. In *ICALP 2008*, *LNCS*, vol. 5126, pp. 39–50. Springer, 2008.
11. Gruber, H., Holzer, M. Provably shorter regular expressions from deterministic finite automata (extended abstract). In *DLT 2008*, *LNCS*, vol. 5257, pp. 383–395. Springer, 2008.
12. Gruber, H., Johannsen, J. Optimal lower bounds on regular expression size using communication complexity. In *FoSSaCS 2008*, *LNCS*, vol. 4962, pp. 273–286. Springer, 2008.
13. Han, Y., Wood, D. Obtaining shorter regular expressions from finite-state automata. *Theor. Comput. Sci.*, 370(1-3):110–120, 2007.
14. Hopcroft, J. E., Ullman, J. D. *Introduction to automata theory, languages and computation.* Addison-Wesley, 1979.
15. Kleene, S. C. Representation of events in nerve nets and finite automata. In *Automata studies*, pp. 3–42. Princeton University Press, 1956.
16. Le Maout, V. Cursors. In *CIAA 2000*, *LNCS*, vol. 2088, pp. 195–207. Springer, 2000.
17. Morgan, K. Approximation algorithms for the maximum induced planar and outerplanar subgraph problems. *Bachelor with honors thesis.* Monash University, Australia, 2005.
18. Ponty, Y., Termier, M., Denise, A.: GenRGenS: software for generating random genomic sequences and structures. *Bioinformatics* 22(12):1534–1535, 2006.
19. Sakarovitch, J. The language, the expression, and the (small) automaton. In *CIAA 2000*, *LNCS*, vol. 3845, pp. 15–30. Springer, 2005.