Fundamenta Informaticae 128 (2013) 445–462
DOI 10.3233/FI-2013-953
IOS Press

State Elimination Heuristics for Short Regular Expressions*

Yo-Sub Han†
Department of Computer Science, Yonsei University
Seoul 120-794, Republic of Korea
emmous@cs.yonsei.ac.kr

Abstract. State elimination is an intuitive and easy-to-implement algorithm that computes a regular expression from a finite-state automaton (FA). It is very hard to compute the shortest regular expression for a given FA in general and we cannot avoid the exponential blow-up. This implies that state elimination cannot avoid the exponential blow-up either. Nevertheless, since the size of a regular expression by state elimination depends on the state removal sequence, we can have a shorter regular expression if we choose a better removal sequence for state elimination. This observation motivates us to examine state elimination heuristics based on the structural properties of the input FA and implement state elimination using the heuristics that run in polynomial time. We demonstrate the effectiveness of our algorithm by experiment results.

Keywords: finite-state automata, regular expressions, state elimination, heuristics

1. Introduction

It is well known that finite-state automata (FAs) have the same expressive power as regular expressions: it is the Kleene theorem [19]. This well-known statement is proved by showing that we can construct FAs from regular expressions and that we can compute regular expressions from FAs. There are many algorithms for constructing FAs such as the Thompson construction [26], the position construction [11, 21] or the follow construction [17]. For the other direction, we can obtain regular expressions from FAs

Address for correspondence: Department of Computer Science, Yonsei University, Seoul 120-794, Republic of Korea
*A preliminary version of this paper appeared in *Proceedings of the 14th International Conference on Implementation and Application of Automata*, CIAA 2009, Lect. Notes Comput. Sci. 5642, 178–187, Springer-Verlag, 2009.
†Han was supported by Basic Sci. Research Program through NRF funded by MEST (2010-0009168, 2012R1A1A2044562).

using the linear equation technique [8, 16] or the state elimination approach [5, 27]. When we transform a regular expression into an FA, we can have a polynomial-size FA with respect to the size of the input regular expression (for instance, all three FA constructions above satisfy this property). However, we often have an exponential-size regular expression from an FA: For example, given an n-state FA (an FA with n states) over k letter alphabet, the size of a corresponding regular expression can be $O(nk4^n)$ in the worst-case [9, 16]. Therefore, one can say that FAs are better representations than regular expressions for describing regular languages when we measure its size. Nevertheless, we often need regular expressions instead of FAs and, thus, it is a very crucial operation to compute a regular expression from an FA both in theory and practice. One evidence is that most automata manipulation libraries such as Grail [24] or JFLAP [25] do have such function.

Jiang and Ravikumar [18] proved that it is PSPACE-complete to compute a minimal regular expression from an (normal) FA and NP-complete to find a minimal regular expression for an acyclic FA that accepts only a finite language. The regular expression minimization problem is, in general, also PSPACE-complete [22]. Ellul et al. [9] showed that if an n-state FA is a planar graph, then we can obtain a regular expression whose size is less than $e^{O(\sqrt{n})}$. Recently, based on this work, Gruber and Holzer [12] demonstrated that we can compute a regular expression whose size is $O(1.742^n)$ for an n-state deterministic FA. However, since the running time for these algorithms are exponential they are not suitable for practical use. Han and Wood [15] found that some structural properties in an FA can lead to shorter regular expression in state elimination. They also designed an algorithm that identifies such properties in polynomial time. Delgado and Morais [7] proposed the state weight technique, which can be implemented in polynomial time as well, and demonstrated that it often gives a shorter regular expression.

We examine some known polynomial-runtime heuristics that may lead to shorter regular expressions from NFAs. We revisit the decomposition technique by Han and Wood [15] (this technique was not implemented before), improve some theoretical results, and implement[1] the technique and run experiments to see how good the heuristic is in practice.

In Section 2, we define some basic notions. In Section 3, we briefly describe state elimination and demonstrate the importance of state removal sequence. Then, we revisit known heuristics and related issues in Section 4. Based on these heuristics, we design a new state elimination algorithm, implement the algorithm in Grail+[2] and show experiment results in Section 5. We mention the future direction of our algorithm for state elimination and conclude the paper in Section 6.

2. Preliminaries

Let Σ denote a finite alphabet of characters and Σ^* denote the set of all strings over Σ. The size $|\Sigma|$ of Σ is the number of characters in Σ. A language over Σ is a subset of Σ^*. The symbol $\emptyset$ denotes the empty language and the symbol λ denotes the null string.

An FA A is specified by a tuple $(Q, \Sigma, \delta, s, F)$, where Q is a finite set of states, Σ is an input alphabet, $\delta : Q \times \Sigma \to 2^Q$ is a transition function, $s \in Q$ is the start state and $F \subseteq Q$ is a set of final states. If F consists of a single state f, then we use f instead of $\{f\}$ for simplicity. Let $|Q|$ be the number

[1] Email the corresponding author for a working software

[2] Grail+ is a symbolic computation environment for finite-state FAs, regular expressions and finite languages based on Grail [24]. For details, visit the homepage: `http://www.csd.uwo.ca/Research/grail/`

of states in Q and $|\delta|$ be the number of transitions in δ. Then, the size of A is $|A| = |Q| + |\delta|$. For a transition $\delta(p, a) = q$ in A, we say that p has an *out-transition* and q has an *in-transition*. Furthermore, we say that A is *non-returning* if the start state of A does not have any in-transitions and A is *non-exiting* if all final states of A do not have any out-transitions. If $\delta(q, a)$ has a single element q', then we denote $\delta(q, a) = q'$ instead of $\delta(q, a) = \{q'\}$ for simplicity.

A string x over Σ is accepted by A if there is a labeled path from s to a final state such that this path spells out x. We call this path an *accepting path*. Then, the language $L(A)$ of A is the set of all strings spelled out by accepting paths in A. We say that a state of A is *useful* if it appears in an accepting path in A; otherwise, it is *useless*. Unless otherwise mentioned, in the following we assume that all states of A are useful.

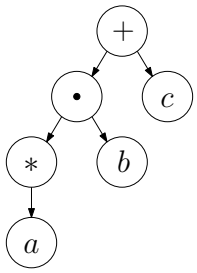

Figure 1: The syntax tree representation for $E = a^*b + c$. Then, the size $|E|$ of E is the number of nodes in the tree; namely, $|E| = 6$.

We define the size of a regular expression E to be the number of characters of Σ and the number of operations.[3] Note that we often omit the catenation symbol in a regular expression. For instance, we write ab instead of $a \cdot b$ and the sizes of both regular expressions are 3 by our definition. In other words, $|E|$ is the number of internal and external nodes in the corresponding syntax tree. Fig. 1 gives an example.

For complete background knowledge in automata theory, the reader may refer to textbooks [16, 27].

3. State Elimination

Given an FA $A = (Q, \Sigma, \delta, s, F)$, we transform A into a new equivalent FA A' such that A' is non-returning and non-exiting, and has a single final state. Namely, $A' = (Q', \Sigma, \delta', s', f')$, where

- $Q' = Q \cup \{s', f'\}$
- $\delta'(q, a) = \begin{cases} \delta(q, a), \text{ when } q \in Q \text{ and } a \in \Sigma \\ f', \text{ when } q \in F \text{ and } a = \lambda \\ s, \text{ when } q = s' \text{ and } a = \lambda \end{cases}$

Note that $L(A) = L(A')$. This construction makes state elimination easier to understand and implement. Moreover, if there are more than one transition between two states, then we merge all transition labels

[3] Grail+ defines the size of regular expression in the same way.

using + (union) operation symbol. For instance, when we have $\delta(p, a) = q$ and $\delta(p, b) = q$ for $a, b \in \Sigma$, we merge them as $\delta(p, a+b) = q$. This FA now has regular expressions instead of characters as transition labels; we call such FAs *expression automata* [14]. From now on, we assume that an input FA is non-returning and non-exiting, and has a single final state. We also assume that there exists a single transition label, which can be a regular expression, between two states.

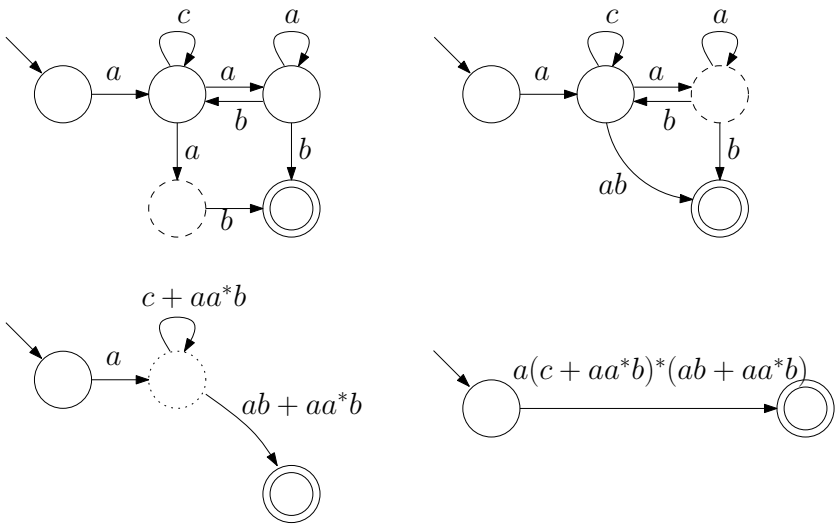

Figure 2: An example of state elimination. The dotted states are being removed.

Assume that an FA $A = (Q, \Sigma, \delta, s, f)$ has $m + 1$ states, and $q_0 = s$ and $q_m = f$. We formally define the *state elimination* of q_i, for $i \neq 0, m$ in A to be the bypassing of state q_i, q_i's in-transitions, q_i's out-transitions and q_i's self-looping transition with equivalent expression transition sequences. Let α_j be the in-transition label from q_j to q_i, γ_k be the out-transition label from q_i to q_k and β be the self-looping transition label of q_i. Then, the state elimination of q_i is that for each α_j, $0 \leq j \leq m$ and $j \neq i$, and each γ_k, $0 \leq k \leq m$ and $k \neq i$, we construct a new transition

$$\delta(q_j, \alpha_j \beta^* \gamma_k) = q_k.$$

If there exists a transition (q_j, ν, q_k) in δ for some regular expression ν, then we merge the two transitions to give the bypass transition $\delta(q_j, \alpha_j \beta^* \gamma_k + \nu) = q_k$. We then remove q and all transitions into and out of q from δ. See Fig. 2 for an example. State elimination was introduced by Brzozowski and McCluskey, Jr. [5] and later was more precisely formulated by Wood [27]. The main idea of the method is to maintain the successive automata, resulting of the elimination of a state, accepting the language of the original automaton. For more details on state elimination, refer to the literature [5, 27]. State elimination can be used in other applications. For example, Giammarresi and Montalbano [10] proposed a method of obtaining a generalized automaton [8], which has strings as transition labels rather than characters, from an FA using state elimination. They restricted state elimination of a subset of states, which does not induce a cycle or a self-loop. Another application is that Brüggemann-Klein and Wood [3] used state elimination to transform Thompson automata [26] into equivalent position automata [11, 21].

State elimination is intuitive and easy to understand and implement because of its simple procedure. However, one problem for state elimination is that the resulting regular expression depends on the state removal sequence. This means that depending on the chosen sequence of states to remove we may have a shorter or longer regular expression for the same FA. Fig. 3 illustrates this idea.

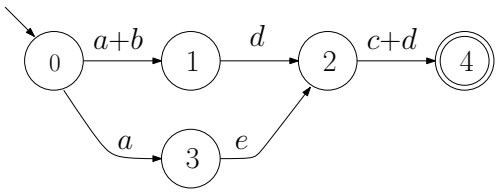

Figure 3: An example of different regular expressions by different removal sequences for a given FA. $E_1 = ae(c+d) + (a+b)d(c+d)$ is the output of state elimination in $1 \to 2 \to 3$ order and $E_2 = ((a+b)d+ae)(c+d)$ is the output of state elimination in $1 \to 3 \to 2$ order, where $|E_1| = 17, |E_2| = 13$ and $L(E_1) = L(E_2)$. Notice two copies of $(c+d)$ in E_1.

For an n-state FA A, there are $(n-2)!$ removal sequences. It is undesirable to try all possible sequences for shorter regular expressions. Instead, we use the structural properties of A and design a fast heuristic for state elimination that can give place to shorter regular expressions.

4. Heuristics for State Elimination

There are several heuristics for finding a removal sequence for state elimination. For example, Gruber and Holzer [12] suggested graph separator techniques and Delgado and Morais [7] relied on the state weight. Recently, Moreira and Reis [23] presented an $O(n^2 \log n)$ time algorithm that obtains an $O(n)$ size regular expressions from an n-state series-parallel acyclic FA. Gulan and Fernau [13] proposed a construction of regular expression from a restricted NFA via extended automata. Note that some heuristics for state elimination run in exponential time. Since we aim to compute a shorter regular expression from an FA quickly, we only consider polynomial runtime heuristics for our implementation. We use the decomposition heuristic by Han and Wood [15] and the state weight approach by Delgado and Morais [7]. Both approaches run in polynomial time.

4.1. The Decomposition Heuristic

Han and Wood [15] suggested two decomposition approaches based on the structural properties of a given FA A: The first is a horizontal decomposition that is to decompose A into two subautomata A_1 and A_2 such that $L(A) = L(A_1) \cup L(A_2)$. The second is a vertical decomposition that is to decompose A into two subautomata A_1 and A_2 such that $L(A) = L(A_1) \cdot L(A_2)$.

First, we use the vertical decomposition that is useful to find a better removal sequence of states for state elimination. We say that, given an FA A, a removal sequence RS_1 for state elimination of A is better than a removal sequence RS_2 for state elimination of A if the regular expression obtained by RS_1 is shorter than the regular expression obtained by RS_2. For the vertical decomposition, we first identify bridge states.

Definition 4.1. We define a state q in an FA A to be a *bridge state* if it satisfies the following conditions:

1. State q is neither a start nor a final state.
2. For each string $w \in L(A)$, its path in A must pass through q at least once.
3. State q is not in any cycle except for the self-loop.

Note that the bridge state condition is more restricted than the original condition in Han and Wood [15][4]. This is because we found a counter example that does not guarantee an optimal solution under the original conditions. In Fig. 4, the removal sequence $1 \to 2$ gives $E_1 = cd(b + ad)^*a$ whereas the removal sequence $2 \to 1$ gives $E_2 = c(db^*a)^*(db^*a)$. Note that $|E_1| < |E_2|$.

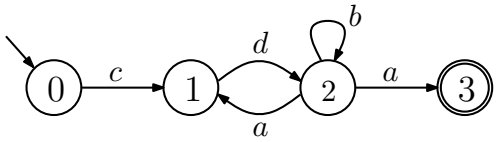

Figure 4: State 1 satisfies the bridge conditions in Han and Wood [15]. However, it is not a bridge state by Definition 4.1.

Han and Wood [15] presented an algorithm that finds bridge states in linear time in the size of a given FA based on the DFS algorithm. We can slightly modify the algorithm and find all bridge states according to Definition 4.1 in linear time as well. Let $\mathcal{E}(A)$ be the total size of all transition labels in A; that is

$$\mathcal{E}(A) = \sum_{i,j} |E_{i,j}|, \text{ where } \delta(q_i, E_{ij}) = q_j \text{ and } q_i, q_j \in Q.$$

Proposition 4.2. There exists an optimal removal sequence for state elimination that eliminates all nonbridge states excluding the start state and the final state before bridge states.

Proof:
Let an FA $A = (Q, \Sigma, \delta, s, f)$ be an input expression automaton. Given an optimal removal sequence OPT of state elimination for A that eliminates a bridge state q_i before eliminating a nonbridge state, we compute a new removal sequence that eliminates all nonbridge states before bridge states and gives the same size of regular expression.

Without loss of generality, we assume that OPT starts with q_i. (If not, then we eliminate all states before q_i in OPT and use the resulting FA as the new input FA and the remaining removal sequence as the new OPT.)

Let

$$\text{OPT} = q_i \to q_j \to \cdots \to q_k,$$

where q_i is the first bridge state to be removed before a nonbridge state in the sequence and $k = m - 1$.

[4]The original condition allows q to be in a cycle.

Then we make nOPT as follows:

$$\text{nOPT} = q_j \to \cdots \to q_k \to q_i.$$

Namely, the new removal sequence nOPT eliminates all states in the same order as OPT except for q_i. Then nOPT removes q_i at the end. Note that nOPT is not the final removal sequence yet. There may be some more bridge states eliminated before a nonbridge state in the sequence. Simply nOPT has one less such bridge state compared with OPT. Let A_{q_i} be the resulting FA after the state elimination of q_i. We observe that

$$\mathcal{E}(A) + c \le \mathcal{E}(A_{q_i}) \text{ and } c = \begin{cases} 3 & \text{if } \beta \neq \lambda, \\ 1 & \text{if } \beta = \lambda. \end{cases},$$

where β is the self-loop transition label of q_i. (See Fig. 6 for example.)

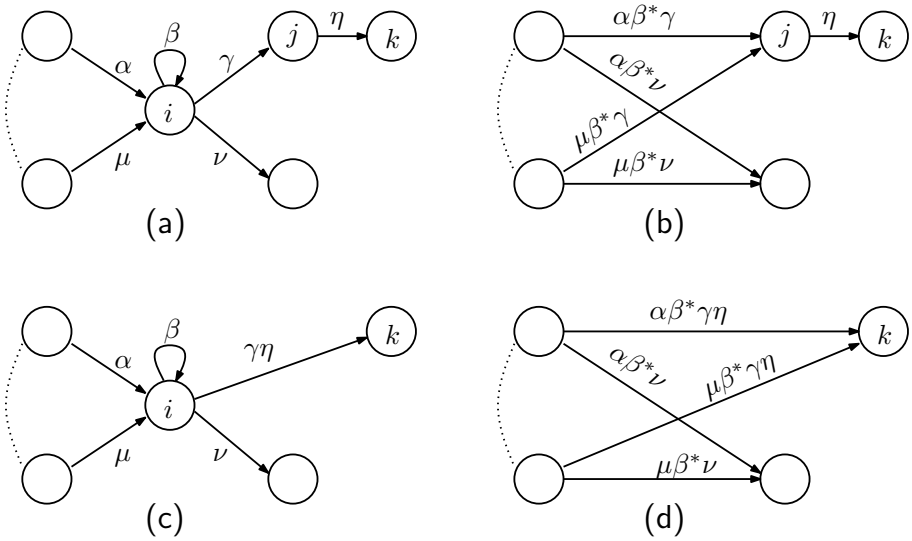

Figure 5: An example FA in which a bridge state q_i has a target state q_j. We draw two source states and two target states for q_i and one target state for q_j for simplicity. In general, they can have arbitrary numbers of states. Four FAs in the figure are:

(a) part of a given FA A
(b) the resulting FA A_{q_i} after eliminating q_i from A
(c) the resulting FA A_{q_j} after eliminating q_j from A
(d) the resulting FA A_{q_i,q_j} after eliminating q_j from A_{q_i}

Note that $\mathcal{E}(A) + c \le \mathcal{E}(A_{q_i})$ and $\mathcal{E}(A_{q_j}) + c \le \mathcal{E}(A_{q_i,q_j})$.

Now consider the next state, say q_j, to be eliminated in OPT. There are three cases for q_j:

(1) q_j is a target state of q_i (Fig. 5 (a)):
In A_{q_i}, state q_j has at least the same number of in-transitions compared to the number of in-transitions of q_j in A and each in-transition has a longer expression. This implies that $\mathcal{E}(A_{q_j}) + c \le \mathcal{E}(A_{q_i,q_j})$ as shown in Fig. 5 (c) and (d).

(2) q_j is a source state of q_i:
Based on a similar argument that we have used for case (1), we know that $\mathcal{E}(A_{q_j}) + c < \mathcal{E}(A_{q_i,q_j})$.

(3) q_j is neither a target state nor a source state of q_i:
The state elimination of q_j produces the same new expressions in both A and A_{q_i}. Then, since $\mathcal{E}(A) + c \leq \mathcal{E}(A_{q_i})$, $\mathcal{E}(A_{q_j}) + c \leq \mathcal{E}(A_{q_i,q_j})$.

Let A_{OPT} be the FA computed by OPT and A' be the corresponding FA that we have obtained by nOPT before removing q_i. Then, by the same argument above, it is always true that

$$\mathcal{E}(A') + c \leq \mathcal{E}(A_{OPT}).$$

Note that A' has three states, s, f and q_i as illustrated in Fig. 6.

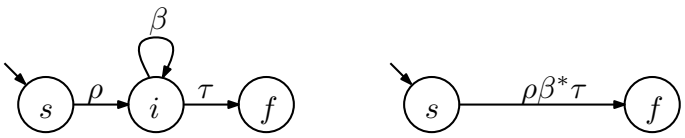

Figure 6: An example of A' (left) and A'_{q_i} (right).

Now we eliminate q_i from A' and denote the resulting FA by A'_{q_i}.

$$\mathcal{E}(A'_{q_i}) = \mathcal{E}(A') + c, \text{where } c = \begin{cases} 3 & \text{if } \beta \neq \lambda, \\ 1 & \text{if } \beta = \lambda. \end{cases}$$

Since $\mathcal{E}(A'_{q_i}) = \mathcal{E}(A') + c \leq \mathcal{E}(A_{OPT})$, nOPT is also an optimal removal sequence for state elimination with one less bridge state before a nonbridge state compared with OPT.

Now if nOPT has another bridge state eliminated before some nonbridge state, then we move the bridge state to the end. By the same argument that we have used for OPT and nOPT, we guarantee that the optimality still holds for the new sequence. This follows that there exists an optimal removal sequence for state elimination that eliminates all nonbridge states before bridge states. □

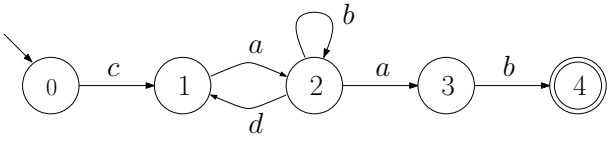

Figure 7: State 3 is a bridge state but both $3 \to 1 \to 2$ and $1 \to 2 \to 3$ removal sequences give the same shortest regular expression by state elimination for the FA.

The statement in Proposition 4.2 seems a bit weak since the existence of bridge states does not always guarantee a shorter regular expression by state elimination. This is because of a very special case when a bridge state has only one in-transition (or out-transition) and its source (target) state is not a bridge state as depicted in Fig. 7. Otherwise, for any regular expression obtained by state elimination whose

removal sequence eliminates a bridge state before eliminating all nonbridge states, there is a shorter regular expression by state elimination whose removal sequence eliminates all nonbridge states before eliminating any bridge states. Thus, bridge states are helpful to find a better removal sequence for state elimination.

Given an FA A, we find bridge states in linear time and apply the vertical decomposition. Once we obtain several decomposed subautomata for A, we check whether or not another decomposition, the horizontal decomposition, is feasible before computing a regular expression for each subautomaton.

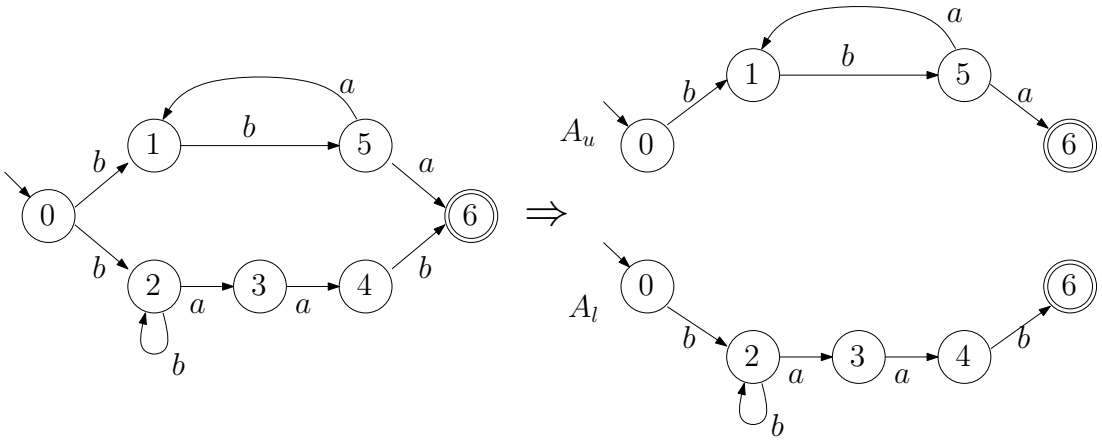

Figure 8: An example of a horizontal decomposition for an FA A without bridge states. Once we decompose A, we may have bridge states for new subautomata. Notice that new bridge states reveal a good removal sequence in its subautomaton. For example, 2, 3 and 4 are bridge states in A_l.

Proposition 4.3. (Han and Wood [15])
Given an FA $A = (Q, \Sigma, \delta, s, f)$, we can discover all subautomata that are disjoint from each other except s and f in $O(|Q| + |\delta|)$ time using DFS.

Fig. 8 gives an example of a horizontal decomposition. We notice that the horizontal decomposition is a good heuristic for state elimination since the removal sequence for each separated subautomaton does not interfere with other removal sequences for other subautomata. For example, in Fig. 8, the removal sequence $1 \to 5 \to 2 \to 3 \to 4$ and the removal sequence $2 \to 1 \to 3 \to 5 \to 4$ give the same regular expressions. Namely, we only look at each subautomaton to find a proper removal sequence and merge the resulting regular expressions using unions. Therefore, if possible, we always decompose the input FA into several horizontally disjoint subautomata, and compute the corresponding regular expressions for subautomata and merge them. Namely, we can use the horizontal decomposition for finding a short regular expression using state elimination. The horizontal decomposition does not affect the optimal removal sequence.

Moreover, as shown in Fig. 8, states 2, 3 and 4 become bridge states in A_l that are not bridge states in A. These new local bridge states in each subautomaton give a better removal sequence for the whole FA. In other words, we can repeat the vertical decomposition, if possible, and the horizontal decomposition again. Since there are only finite number of states, this process runs in polynomial time. Overall, the decomposition heuristic is a classical divide-and-conquer approach for state elimination.

Proposition 4.4. Given an FA $A = (Q, \Sigma, \delta, s, f)$, we can decompose A, if possible, into several subautomata in which both horizontal and vertical decomposition are not feasible in $O(|A|^2)$ worst-case time.

Proof:
Let m be the number of states and n be the number of transitions in A. We first run the vertical decomposition algorithm by Han and Wood [15] and it takes $O(m + n)$ time.

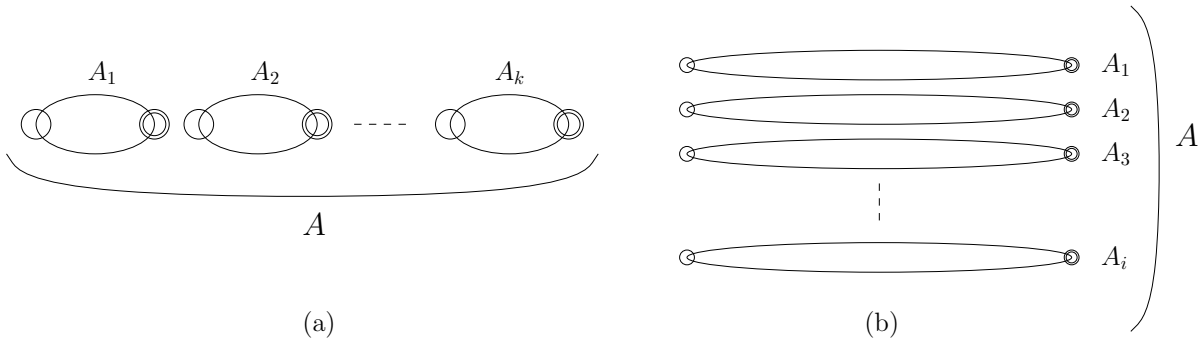

Figure 9: In the example, (a) is the vertical decomposition and (b) is the horizontal decomposition.

Let $A_1, A_2, \ldots, A_k$ be the decomposed subautomata from A as shown in Fig. 9 (a). Notice that $|A| + k - 1 = |A_1| + |A_2| + \cdots + |A_k|$ for some constant k. (If k is proportional to n, then the average number of states for each subautomaton A_j for $1 \leq j \leq k$ is constant.) This implies that the horizontal decomposition takes $O(m+n)$. Let $A = A_1 \cup A_2 \cup \cdots \cup A_i$ as shown in Fig. 9 (b). Similarly, $|A| + 2i - 2 = |A_1| + |A_2| + \cdots + |A_i|$ for some constant i. Thus, it takes $O(m+n)$ for the next vertical decomposition and so on. In each step, we consider at least one state less compared to the previous step. Therefore, the procedure eventually terminates and the total running time is

$$O(m + n) \times m = O(m^2 + mn) = O(|A|^2).$$ □

4.2. The State Weight Heuristic

Delgado and Morais [7] proposed the state weight heuristic. They defined a state weight be the size of new transition labels that are created by eliminating the state. We borrow their notion and define the weight of a state q in an FA $A = (Q, \Sigma, \delta, s, f)$ as follows:

$$\sum_{i=1}^{\mathsf{IN}} (\mathbb{W}_{\mathsf{in}}(i) \times \mathsf{OUT}) + \sum_{i=1}^{\mathsf{OUT}} (\mathbb{W}_{\mathsf{out}}(i) \times \mathsf{IN}) + \mathbb{W}_{\mathsf{loop}} \times (\mathsf{IN} \times \mathsf{OUT}), \tag{1}$$

where IN is the number of in-transitions excluding self-loop, OUT is the number of out-transitions excluding self-loop, $\mathbb{W}_{\mathsf{in}}(i)$ is the size of the transition label on the ith in-transition, $\mathbb{W}_{\mathsf{out}}(i)$ is the size of the transition label on the ith out-transition and $\mathbb{W}_{\mathsf{loop}}$ is the self-loop label size for q. Note that our weight definition is slightly different from Delgado and Morais [7]: We define the weight be to the total

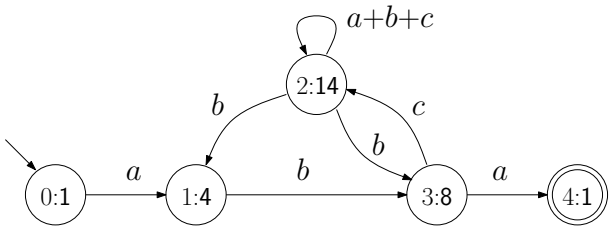

Figure 10: Each state has a state index and the state weight by Equation (1). In this FA, the state weight heuristic suggests $1 \to 3 \to 2$ removal sequence but it is not the best removal sequence.

size of transition labels after eliminating q. We can compute the weight of all states in A in polynomial time.

Delgado and Morais [7] noticed that the state weight heuristic does not guarantee the shortest regular expression. For instance, in Fig. 10, the state weight heuristic suggests $1 \to 3 \to 2$ removal sequence, which gives $E_1 = abc(((a + b + c) + (b + bb)c))^*(b + bb)a$ whereas the removal sequence $1 \to 2 \to 3$ gives $E_2 = ab(c(a + b + c)^*(b + bb))^*a$. Note that we can select a least weight state and remove it, and recompute the state weight and choose a new least weight state in the resulting FA. This approach does not guarantee the shortest regular expression either but it often gives shorter regular expressions compared with the one-time state weight heuristic. On the other hand, since we calculate state weight every step, it may take more time than the one-time state weight heuristic. We implement both approaches and analyze the experiment results in Section 5.

5. Implementation and Experiment Results

We have studied two main heuristics in Section 4; the decomposition heuristic and the state weight heuristic. Delgado and Morais [7] demonstrated the effectiveness of the state weight approach. We implement the decomposition heuristics by Han and Wood [15] and compare the heuristics with some other approaches to see how good they are.

Given an FA $A = (Q, \Sigma, \delta, s, F)$, we first remove all unreachable states, merge multiple transitions between two states into a single transition and make A to have a single final state using λ-transitions. This preprocessing takes $O(|A|)$ time. We use combinations of heuristics in Section 4 for state removal sequences. The following is the list of six different state elimination heuristics that we have implemented for experiments.

$\mathbb{C}$-I: We eliminate states in random order without any heuristics.

$\mathbb{C}$-II: We use both the vertical decomposition and the horizontal decomposition until both decompositions are not feasible. Once the decomposition step is over, we eliminate states in random order.

$\mathbb{C}$-III: We compute the state weight of all states and eliminate a state with least weight. Note that we compute the state weight only once.

$\mathbb{C}$-VI: We first use the vertical and horizontal decompositions and decide the removal sequence for each decomposed subautomaton using the state weight heuristic as $\mathbb{C}$-III.

$\mathbb{C}$-V: We select a least weight state and eliminate it. Then, we compute the state weight again for the resulting FA and eliminate the new least weight state. We repeat this until there is no more state to remove. Namely, we have to compute the state weight roughly $|Q|$ times, where $|Q|$ is the number of states in the input FA. Note that this case is different from $\mathbb{C}$-III.

$\mathbb{C}$-VI: We use the vertical and horizontal decompositions. Then, for each decomposed subautomaton, we use the repeated state weight heuristics to decide the removal order as $\mathbb{C}$-V.

For input NFAs, we use a random NFA generator recently developed by Almeida et al. [2]. The generator combines the van Zijl bit-stream method suggested by Champarnaud et al. [6] and Leslie [20]. Although there is no known uniform random generator for NFAs, the NFA generator by Almeida et al. [2] preserves certain properties of the uniform random generator for DFAs developed by the same authors [1]. For detail on the construction and the properties of the random generator, refer to Almeida et al. [1, 2]. We generate 20,000 sample NFAs for different numbers n of states and transition density d, where $3 \leq n \leq 10$ and $d = 0.2$ or 0.5.

Our implementation is based on Grail+. $\mathbb{C}$-I is the current state elimination algorithm implemented in Grail+ and JFLAP [25]. We implement the other 5 heuristics in Grail+. We run all tests in the same computer, an Intel® Core(TM) i7 CPU at 2.67GHz with 4 GM of RAM. We apply each of the six algorithms state elimination to 20,000 sample NFAs and compute the average size of the regular expressions and the average CPU runtime.

Tables 1 and 2 show the experiment results of the state elimination with six different heuristics. We run the state elimination for 20,000 sample NFAs and compute the average size of regular expressions and the average CPU runtime. Note that if the number n of states is larger or the transition density d is larger, then there are less numbers of bridge states. For instance, when $n = 6$ there are no more bridge states with $d = 0.5$. Since we mostly focus on the role of bridge states for state elimination, we mainly consider sample NFAs with some bridge states. Without bridge states, ($\mathbb{C}$-I, $\mathbb{C}$-II), ($\mathbb{C}$-III, $\mathbb{C}$-IV) and ($\mathbb{C}$-V, $\mathbb{C}$-VI) are the same for each other. Table 2 shows the experiment results for sample NFAs without bridge states and, thus, has only three cases.

Table 1 and Fig. 11 show that $\mathbb{C}$-VI is best followed by $\mathbb{C}$-V $\rightarrow$ $\mathbb{C}$-IV $\rightarrow$ $\mathbb{C}$-III $\rightarrow$ $\mathbb{C}$-II $\rightarrow$ $\mathbb{C}$-I. Moreover the runtime of checking the existence of bridge states is negligible and bridge states often improve the overall runtime since the size of the resulting regular expressions are shorter and, thus, requires less computation time. Therefore, we claim that it is better to use the decomposition heuristics and the state weight heuristics for state elimination in practice. We, next, consider the sample NFAs with bridge states only to determine whether or not bridge states are indeed helpful for obtaining shorter regular expressions in practice.

We observe that there are less number of bridge states when the size of NFAs is larger. We also notice that if the transition density is larger, then there are less number of bridge states as shown in Table 1. However, if we start with regular expressions and transform into FAs using the standard FA construction methods [11, 21, 26], then there are often lots of bridge states. Thus, it is still worth to examine the NFAs with some bridge states for state elimination.

Fig. 12 shows that $\mathbb{C}$-VI is the best. Furthermore, bridge states help to reduce the size of the resulting regular expression significantly when the input NFAs are getting larger. This is because a bridge state essentially divides the input NFA into two subNFAs, which are smaller than the original NFA, compute

Experimental Results for All Sample NFAs													
	$\mathbb{C}$-I		$\mathbb{C}$-II		$\mathbb{C}$-III		$\mathbb{C}$-IV		$\mathbb{C}$-V		$\mathbb{C}$-VI		
n, d	time	size	time	size	time	size	time	size	time	size	time	size	# of B_NFAs
3, 0.2	0.0003	6.6	0.0003	5.7	0.0003	5.5	0.0003	5.4	0.0003	5.4	0.0003	5.4	2531
3, 0.5	0.0012	79.3	0.0012	76.6	0.0008	46.6	0.0008	46.0	0.0007	45.3	0.0007	45.1	1133
5, 0.2	0.0051	386.0	0.0050	375.4	0.0022	140.1	0.0022	137.5	0.0017	119.6	0.0017	118.2	1673
5, 0.5	0.0334	2630.1	0.0334	2630.1	0.0184	1388.3	0.0184	1388.3	0.0132	1190.8	0.0132	1190.8	1
6, 0.2	0.0178	1351.8	0.0176	1335.9	0.0055	386.2	0.0054	382.4	0.0038	294.2	0.0038	292.4	1278
6, 0.5	0.1397	11037.3	0.1397	11037.3	0.0732	5649.6	0.0732	5649.6	0.0487	4420.9	0.0487	4420.9	0
7, 0.2	0.0707	5456.6	0.0705	5441.1	0.0174	1269.4	0.0173	1264.7	0.0102	826.3	0.0102	824.6	429
7, 0.5	0.5737	44805.7	0.5737	44805.7	0.2905	22362.5	0.2905	22362.5	0.1736	15792.9	0.1736	15792.9	0
8, 0.2	0.2848	21930.0	0.2846	21909.2	0.0563	4224.4	0.0562	4218.1	0.0267	2290.1	0.0267	2288.3	201
8, 0.5	2.3904	183167.2	2.3904	183167.2	1.1977	91047.0	1.1977	91047.0	0.6462	58306.6	0.6462	58306.6	0
9, 0.2	1.1363	86880.2	1.1360	86856.2	0.1903	14408.7	0.1902	14400.9	0.0716	6305.3	0.0715	6303.4	84
9, 0.5	9.7079	732355.3	9.7079	732355.3	4.8369	362908.7	4.8369	362908.7	2.3449	209654.5	2.3449	209654.5	0
10, 0.2	4.8132	362571.2	4.8129	362559.1	0.7258	54579.7	0.7254	54550.8	0.2142	19047.5	0.2142	19044.3	43
10, 0.5	39.3740	2947500.6	39.3740	2947500.6	19.6842	1466585.1	19.6842	1466585.1	8.7356	778921.4	8.7356	778921.4	0

Experimental Results for Sample NFAs with Bridge States													
	$\mathbb{C}$-I		$\mathbb{C}$-II		$\mathbb{C}$-III		$\mathbb{C}$-IV		$\mathbb{C}$-V		$\mathbb{C}$-VI		
n, d	time	size	time	size	time	size	time	size	time	size	time	size	# of B_NFAs
3, 0.2	0.0003	13.5	0.0003	6.1	0.0002	6.7	0.0003	6.1	0.0002	6.7	0.0003	6.1	2531
3, 0.5	0.0010	76.3	0.0005	29.2	0.0006	37.7	0.0006	26.2	0.0004	30.3	0.0006	26.2	1133
5, 0.2	0.0029	220.4	0.0016	94.0	0.0015	103.6	0.0012	72.6	0.0013	86.8	0.0010	70.7	1673
5, 0.5	0.0160	968.0	0.0150	929.0	0.0000	796.0	0.0160	757.0	0.0000	637.0	0.0150	607.0	1
6, 0.2	0.0073	539.9	0.0043	291.3	0.0031	234.6	0.0025	174.3	0.0026	187.2	0.0023	159.4	1278
7, 0.2	0.0227	1735.0	0.0135	1010.2	0.0098	710.9	0.0070	495.3	0.0061	488.6	0.0052	411.1	429
8, 0.2	0.0811	6300.0	0.0542	4226.0	0.0287	2200.7	0.0203	1574.6	0.0157	1296.2	0.0126	1121.6	201
9, 0.2	0.3170	24876.3	0.2401	19154.6	0.1049	8124.7	0.0793	6275.6	0.0418	3697.2	0.0347	3247.6	84
10, 0.2	1.2964	100314.9	1.1897	94661.5	0.4041	31162.0	0.2230	17750.6	0.1129	10232.5	0.0933	8720.8	43

Table 1: Experiment results of the state elimination with 6 different approaches, where n denotes the number of states and d denotes the transition density. The table above shows the average size of regular expressions and the average CPU runtime of 20,000 NFAs for each case. The table below is the experiment results for NFAs with bridge states from the sample NFAs. The last column (# of B_NFAs) shows the number of NFAs with bridge states out of 20,000 NFAs.

Experimental Results for Sample NFAs without Bridge States							
	$\mathbb{C}$-I & $\mathbb{C}$-II		$\mathbb{C}$-III & $\mathbb{C}$-IV		$\mathbb{C}$-V & $\mathbb{C}$-VI		
n, d	time	size	time	size	time	size	# of sample data
3, 0.2	0.0003	5.6	0.0003	5.3	0.0004	5.3	17469
3, 0.5	0.0012	79.4	0.0008	47.2	0.0007	46.2	18867
5, 0.2	0.0053	401.1	0.0023	143.4	0.0017	122.6	18327
5, 0.5	0.0334	2630.2	0.0184	1388.4	0.0132	1190.8	19999
6, 0.2	0.0185	1407.2	0.0056	396.6	0.0039	301.5	18722
6, 0.5	0.1397	11037.3	0.0732	5649.6	0.0487	4420.9	20000
7, 0.2	0.0718	5538.2	0.0175	1281.6	0.0103	833.7	19571
7, 0.5	0.5737	44805.7	0.2905	22362.5	0.1736	15792.9	20000
8, 0.2	0.2869	22088.7	0.0566	4244.9	0.0268	2300.2	19799
8, 0.5	2.3904	183167.2	1.1977	91047.0	0.6462	58306.6	20000
9, 0.2	1.1397	87141.7	0.1907	14435.2	0.0717	6316.3	19916
9, 0.5	9.7079	732355.3	4.8369	362908.7	2.3449	209654.5	20000
10, 0.2	4.8207	363136.3	0.7265	54630.1	0.2144	19066.5	19957
10, 0.5	39.3740	2947500.6	19.6842	1466585.1	8.7356	778921.4	20000

Table 2: Experiment results of the state elimination with 6 different approaches for NFAs without bridge states from the sample NFAs, where n denotes the number of states and d denotes the transition density. For each case, we compute the average CPU runtime and the average size of regular expressions.

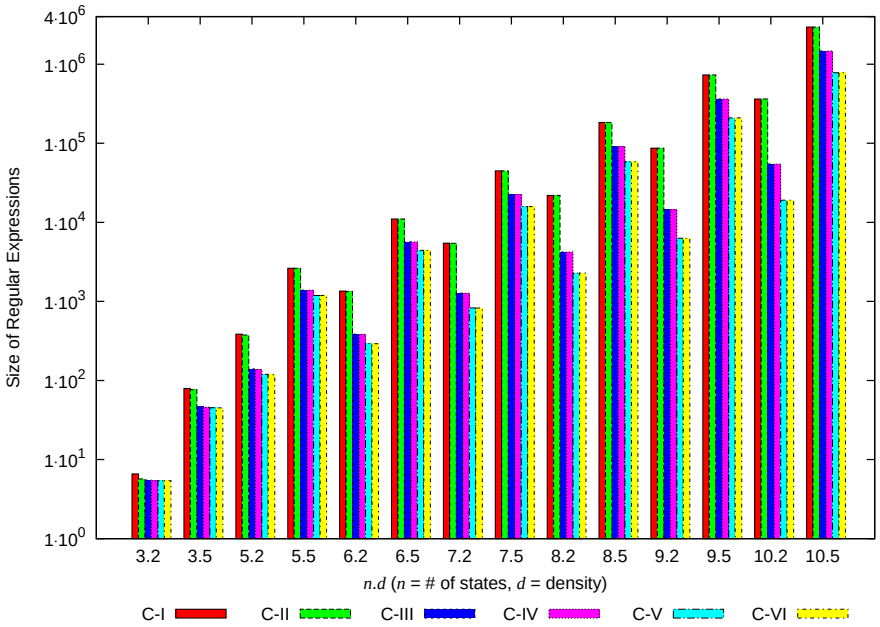

Figure 11: The average size of resulting regular expressions of state elimination. The X-axis denotes the number of states and the transition density and the Y-axis denotes the average size of the resulting regular expressions for each case in log scale.

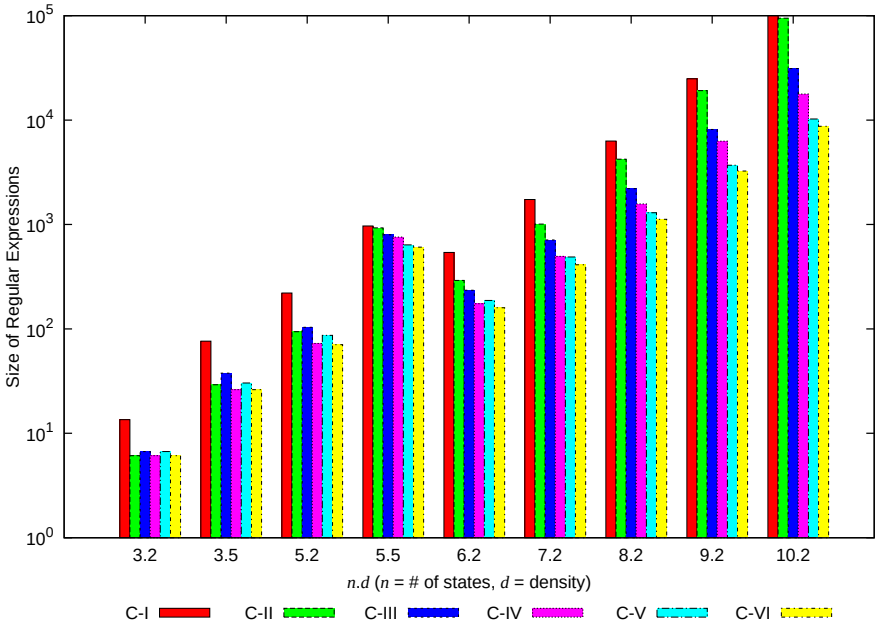

Figure 12: The average size of resulting regular expressions of state elimination. We only consider sample NFAs with bridge states.

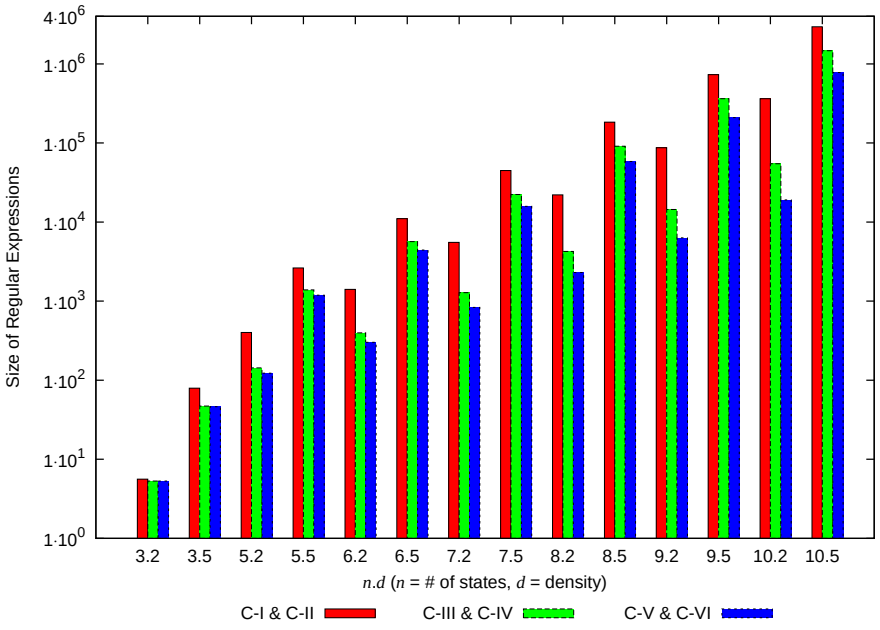

Figure 13: The average size of resulting regular expressions of state elimination for sample NFAs without bridge states.

regular expressions for both NFAs separately and catenate the resulting regular expressions. In addition, we notice that the combination of the decomposition heuristics and the state weight heuristic gives rise to shorter regular expressions.

We also examine the sample NFAs without bridge states in Table 2 and in Fig. 13. The results reconfirm that the state weight heuristic is practical and helpful as mentioned in Delgado and Morais [7]. Note that when there are no bridge states, ($\mathbb{C}$-I, $\mathbb{C}$-II), ($\mathbb{C}$-III, $\mathbb{C}$-IV) and ($\mathbb{C}$-V, $\mathbb{C}$-VI) are the same with each other.

Proposal 5.1. We propose the following suggestions for the state elimination implementation based on our experiment results:

1. It is better to apply the decomposition heuristic first and the state weight heuristic later for each decomposed subautomaton.

2. For NFAs without bridge states, the state weight heuristic is practical and helpful for state elimination. Especially, the repeated state weight heuristic ($\mathbb{C}$-V) outperforms the single state weight heuristic ($\mathbb{C}$-III).

3. Heuristics for state elimination enable to obtain a regular expression faster compared with state elimination without heuristics although they require additional processing time. The additional processing time is negligible in practice. This is because heuristics help to keep smaller size of regular expressions in transitions when running state elimination.

We notice that most FAs do not have vertical or horizontal decompositions. However, as demonstrated in Table 1, the decomposition verification time in practice is very fast whereas its gain, if feasible, is very large. Therefore, both decomposition heuristics are still useful in practice for implementing state elimination. For randomly generated NFAs, there are fewer bridge states when NFAs are getting larger. However, if we have modified FAs constructed by standard construction methods such as the Thompson construction [26] or the position construction [11, 21] first, then we may expect to have more frequent decompositions. This is because these standard constructions maintain the structural properties of catenation (vertical decomposition) and union (horizontal decomposition) in regular expressions.

6. Conclusions

There are several heuristics for state elimination. We have chosen some polynomial runtime heuristics in the literature and implemented them in Grail+, which is one of the most well-known and well-used FA libraries. We have re-examined the decomposition heuristics by Han and Wood [15] and improved some theoretical results in their work and implemented the technique. We have also adopted the state weight heuristic by Delgado and Morais [7] and implemented it in Grail+.

We have demonstrated that the best combination of the decomposition heuristics and the state weight heuristic is to apply the decomposition heuristics as much as possible and, then, apply the repeated state weight approach (the $\mathbb{C}$-VI case in Section 5). We have also observed that the additional running time for heuristics is negligible, and the heuristics give rise to shorter transition labels and speed up the overall running time in practice.

Our future direction of state elimination implementation is to introduce the on-the-fly regular expression minimization when updating transition label in state elimination. For example, we can use simple factorization technique (for instance, $ab + ac \rightarrow a(b + c)$). We can also use the orbit property by Brüggemann-Klein and Wood [4] since it can be checked in polynomial time and gives a Kleene star operator. Furthermore, the orbit property separates the input FA and, thus, reduces the problem size for state elimination. For both theoretical and practical research, as hinted in Fig. 11, Fig. 12 and Fig. 13, the regular expression by state elimination is more closely related to the transition density: namely, if we have a larger number of transitions, then we have a longer regular expression for a fixed number of states. In the literature, there is little work done for the number of transitions and the size of regular expressions. Thus, it is an interesting problem to investigate.

Acknowledgements

We wish to thank Yu and his group for providing us the source code of Grail+. We also thank Almeida for sharing his NFA random generator.

We wish to thank the referees for the care they put into reading the previous version of this manuscript. Their comments including the references on random FA generators [1, 2, 6, 20] were invaluable in depth and detail. The current version owes much to their efforts.

References

[1] M. Almeida, N. Moreira, and R. Reis. Enumeration and generation with a string automata representation. *Theoretical Computer Science*, 387(2):93–102, 2007.

[2] M. Almeida, N. Moreira, and R. Reis. On the performance of automata minimization algorithms. Technical Report DCC-2007-03, DCC - FC & LIACC, Universidade do Porto, June 2007.

[3] A. Brüggemann-Klein and D. Wood. The validation of SGML content models. *Mathematical and Computer Modelling*, 25:73–84, 1997.

[4] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 140:229–253, 1998.

[5] J. Brzozowski and E. McCluskey, Jr. Signal flow graph techniques for sequential circuit state diagrams. *IEEE Transactions on Electronic Computers*, EC-12:67–76, 1963.

[6] J.-M. Champarnaud, G. Hansel, T. Paranthoën, and D. Ziadi. Random generation models for nfas. *Journal of Automata, Languages and Combinatorics*, 9(2/3):203–216, 2004.

[7] M. Delgado and J. Morais. Approximation to the smallest regular expression for a given regular language. In *Proceedings of CIAA'04*, Lecture Notes in Computer Science 3317, 312–314, 2004.

[8] S. Eilenberg. *Automata, Languages, and Machines*, volume A. Academic Press, New York, NY, 1974.

[9] K. Ellul, B. Krawetz, J. Shallit, and M.-W. Wang. Regular expressions: New results and open problems. *Journal of Automata, Languages and Combinatorics*, 10:407–437, 2005.

[10] D. Giammarresi and R. Montalbano. Deterministic generalized automata. *Theoretical Computer Science*, 215:191–208, 1999.

[11] V. Glushkov. On a synthesis algorithm for abstract automata. *Ukr. Matem. Zhurnal*, 12(2):147–156, 1960. In Russian.

[12] H. Gruber and M. Holzer. Provably shorter regular expressions from deterministic finite automata. In *Proceedings of DLT'08*, Lecture Notes in Computer Science 5257, 383–395, 2008.

[13] S. Gulan and H. Fernau. Local elimination-strategies in automata for shorter regular expressions. In *Proceedings of SOFSEM'08*, 46–57, 2008.

[14] Y.-S. Han and D. Wood. The generalization of generalized automata: Expression automata. *International Journal of Foundations of Computer Science*, 16(3):499–510, 2005.

[15] Y.-S. Han and D. Wood. Obtaining shorter regular expressions from finite-state automata. *Theoretical Computer Science*, 370(1-3):110–120, 2007.

[16] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, 2 edition, 1979.

[17] L. Ilie and S. Yu. Follow automata. *Information and Computation*, 186(1):140–162, 2003.

[18] T. Jiang and B. Ravikumar. Minimal NFA problems are hard. *SIAM Journal on Computing*, 22(6):1117–1141, 1993.

[19] S. Kleene. Representation of events in nerve nets and finite automata. In C. Shannon and J. McCarthy, editors, *Automata Studies*, 3–42, Princeton, NJ, 1956. Princeton University Press.

[20] T. K. S. Leslie. Efficient approaches to subset construction. Master's thesis, Waterloo, Ontario, Canada, 1995.

[21] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, 9:39–47, 1960.

[22] A. Meyer and L. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential time. In *Proceedings of the Thirteenth Annual IEEE Symposium on Switching and Automata Theory*, 125–129, 1972.

[23] N. Moreira and R. Reis. Series-parallel automata and short regular expressions. *Fundamenta Informaticae*, 91(3-4):611–629, 2009.

[24] D. Raymond and D. Wood. Grail: Engineering automata in C++. Technical Report 345, Department of Computer Science, University of Western Ontario, London, Ontario, Canada, 1993.

[25] S. H. Rodger and T. W. Finley. *JFLAP: An Interactive Formal Languages and Automata Package*. Jones & Bartlett Pub, 2006.

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

[27] D. Wood. *Theory of Computation*. John Wiley & Sons, Inc., New York, NY, 1987.