Obtaining Shorter Regular Expressions from Finite-State Automata*

Yo-Sub Han
System Technology Division, Korea Institute of Science and Technology
P.O.BOX 131, Cheongryang, Seoul, Korea, Email: emmous@kist.re.kr

Derick Wood
Department of Computer Science, The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong SAR, Email: dwood@cs.ust.hk

May 15, 2006

Abstract

We consider the use of state elimination to construct shorter regular expressions from finite-state automata (FAs). Although state elimination is an intuitive method for computing regular expressions from FAs, the resulting regular expressions are often very long and complicated. We examine the minimization of FAs to obtain shorter expressions first. Then, we introduce vertical chopping based on bridge states and horizontal chopping based on the structural properties of given FAs. We prove that we should not eliminate bridge states until we eliminate all non-bridge states to obtain shorter regular expressions. In addition, we suggest heuristics for state elimination that lead to shorter regular expressions based on vertical chopping and horizontal chopping.

1 Introduction

It is well known that the family of languages accepted by finite-state automata (FAs) is the same as the family of languages defined by regular expressions [15]. It can be proved by showing that we can construct FAs from regular expressions and that we can construct regular expressions from FAs.

There are a number of FA constructions; for example, the Thompson construction [17], the position construction [9, 16] and the follow construction [13]. These constructions are inductive and, therefore, preserve the structural properties of the corresponding regular expressions. For instance, the size of a Thompson automaton is bounded by the size of a given regular expression [8] and the number of states in a position automaton is the number of character appearances in the corresponding regular expression plus one [3].

When converting FAs into regular expressions, we can use either linear equations [6] or state elimination [2]. We consider state elimination. State elimination was already in use in the 1960s, in particular by Brzozowski and McCluskey, Jr. [2] and was carefully formulated by Wood [18]. The idea behind state elimination is simple. We keep removing states, except

*Part of this research was carried out while the first author was in HKUST. Both authors supported under the Research Grants Council of Hong Kong Competitive Earmarked Research Grant HKUST6197/01E.

for the start and the final states for a given FA, while maintaining the transition information of the automaton until there are no more states to eliminate. We illustrate state elimination in Fig. 1.

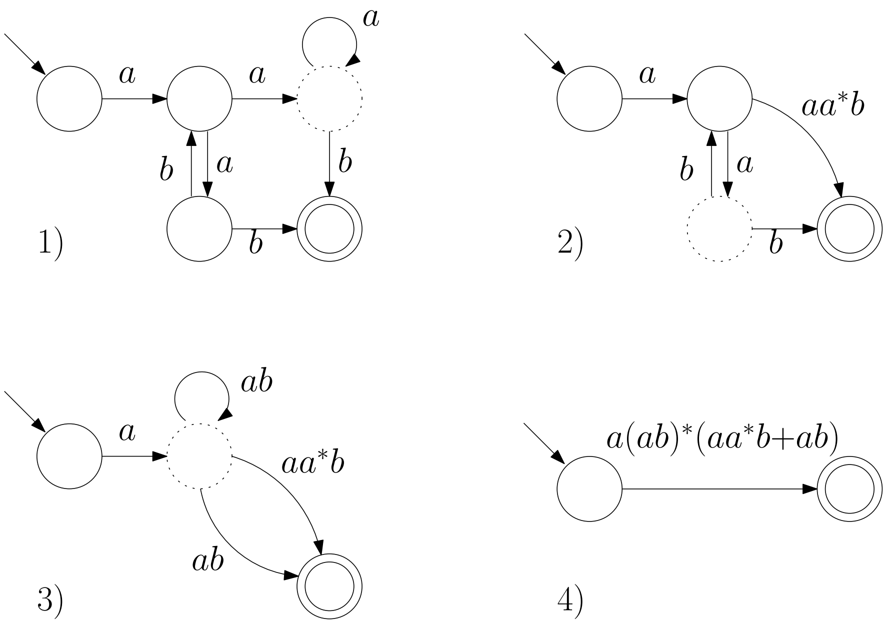

Figure 1: An example of state elimination. The dotted states are being removed.

Apart from computing regular expressions for FAs, state elimination also has other applications: Giammarresi and Montalbano [7] proposed a method of obtaining a generalized automaton [6], which has strings as transition labels rather than characters, from a given FA using state elimination. They restricted state elimination of a subset of states, which does not induce a cycle or a self-loop. Brüggemann-Klein and Wood [1] showed that Thompson automata can be transformed into position automata by eliminating states that have null transitions.

We investigate the structural properties of FAs that help to obtain shorter regular expressions in state elimination. We observe that if we decompose a given FA in certain ways, then we can construct a shorter regular expression compared with random selection of states in state elimination. Based on this observation, we propose two decomposition methods, vertical chopping and horizontal chopping.

In Section 2, we define some basic notions. In Section 3, we describe state elimination and suggest two ways to minimize FAs. Then, we introduce vertical chopping and horizontal chopping of an FA in Sections 4 and 5. Furthermore, we prove that we should not eliminate bridge states, which are defined in Section 4, until we eliminate all non-bridge states to obtain a shorter regular expression. Finally, we suggest some heuristics for state elimination that lead to shorter regular expressions.

2 Preliminaries

Let Σ denote a finite alphabet of characters and Σ^* denote the set of all strings over Σ. A language over Σ is any subset of Σ^*. The character $\emptyset$ denotes the empty language and the character 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, $\delta \subseteq Q \times \Sigma \times Q$ is a (finite) set of transitions, $s \in Q$ is the start state and $F \subseteq Q$ is a set of final states. Let $|Q|$ be the number of states in Q and $|\delta|$ be the number of transitions in δ. Then, the size of A is $|A| = |Q| + |\delta|$. Given a transition (p, a, q) in δ, where $p, q \in Q$ and $a \in \Sigma$, we say that p has an *out-transition* and q has an *in-transition*. Furthermore, p is a *source state* of q and q is a *target state* of p. A string x in Σ^* is accepted by A if there is a labeled path from s to a final state in F that spells out x. Thus, the language $L(A)$ of an FA A is the set of all strings spelled out by paths from s to a final state in F. We define A to be *non-returning* if the start state of A does not have any in-transitions and A to be *non-exiting* if a final state of A does not have any out-transitions. We assume that A has only *useful* states: that is, each state appears on some path from the start state to some final state.

3 State Elimination

Given an FA $A = (Q, \Sigma, \delta, s, F)$, the *state elimination* of a state $q \in Q \setminus \{\{s\} \cup F\}$ is the bypassing of q, q's in-transitions, q's out-transitions and q's self-looping transition with equivalent expression transition sequences. Namely, for each in-transition (p_i, α_i, q), for $1 \leq i \leq m$ and $m \geq 1$, for each out-transition (q, γ, r_j), for $1 \leq j \leq n$ and $n \geq 1$, and for the self-looping transition (q, β, q) in δ, we construct a new transition $(p_i, \alpha_i \cdot \beta^* \cdot \gamma_j, r_j)$. If there is a transition (p, ν, r) in δ for some expression ν, then we merge two transitions into a single transition $(p, (\alpha_i \cdot \beta^* \cdot \gamma_j)+\nu, r)$. We then remove q and all in-transitions and out-transitions of q including the self-looping transition from δ. We denote the resulting automaton by $A_q = (Q \setminus \{q\}, \Sigma, \delta_q, s, F)$.

State elimination is a very primitive operation on FAs. It was introduced by Brzozowski and McCluskey, Jr. [2] to compute regular expressions from FAs. State elimination maintains the language accepted by a given FA while removing states. Note that we have regular expressions instead of single characters on a transition of A_q. We say that an FA with regular expressions on transitions is an *expression automaton* (EA) [2, 10]. EAs are a generalization of FAs. Given an EA A, if we only allow characters on transitions, then A is a traditional FA and if we only allow strings, then A is a generalized automaton [6, 7]. Therefore, EAs have more expressive power on each transition. On the other hand, they have the same expressive power as FAs overall [18].

It is easier to formulate state elimination for computing a regular expression if a given FA A is non-returning and non-exiting. If an FA $A = (Q, \Sigma, \delta, s, F)$ is not non-returning and not non-exiting, then we transform A into a new FA A' such that $L(A') = L(A)$ and A' is non-returning and non-exiting as follows:

- non-returning: we introduce a new start state s' and connect s' to s by a null transition, (s', λ, s).
- non-exiting: we introduce a new final state f' and connect $f_i \in F$ to f' by null transitions, (f_i, λ, f') for $f_i \in F$.

Now the resulting automaton

$$A' = (Q \cup \{s', f'\}, \Sigma, \delta \cup \{(s', \lambda, s)\} \cup \{(f_i, \lambda, f') \mid f_i \in F\}, s', f')$$

is non-returning and non-exiting. In addition, A' has a single final state.

Proposition 1 (Han and Wood [10])**.** *Let $A = (Q, \Sigma, \delta, s, f)$ be a non-returning and non-exiting FA with at least three states and q be a state in $Q \setminus \{s, f\}$. Then, $L(A_q) = L(A)$ and A_q is also non-returning and non-exiting.*

Once we eliminate all states in $Q \setminus \{s, f\}$ for A that is non-returning and non-exiting, we obtain an EA $A_{Q\setminus\{s,f\}} = (\{s, f\}, \Sigma, (s, E, f), s, f)$, where E is the corresponding regular expression for A by Proposition 1. Therefore, state elimination guarantees to construct regular expressions from FAs.

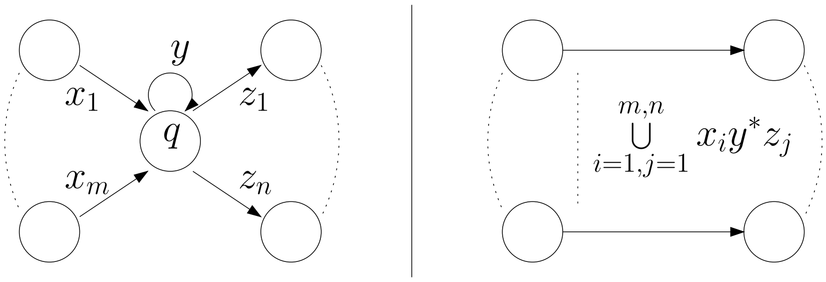

Figure 2: An example of state elimination that produces many duplicate strings.

One problem with state elimination is that it may increase the size of labels on transitions exponentially while removing states for a given automaton. For example in Fig. 2, if we eliminate q from the automaton, then we have to introduce $O(mn)$ duplicate strings as new transition labels. On the other hand, if a given automaton has a simpler structure, for instance see Fig. 3, then state elimination yields new transitions that are smaller.

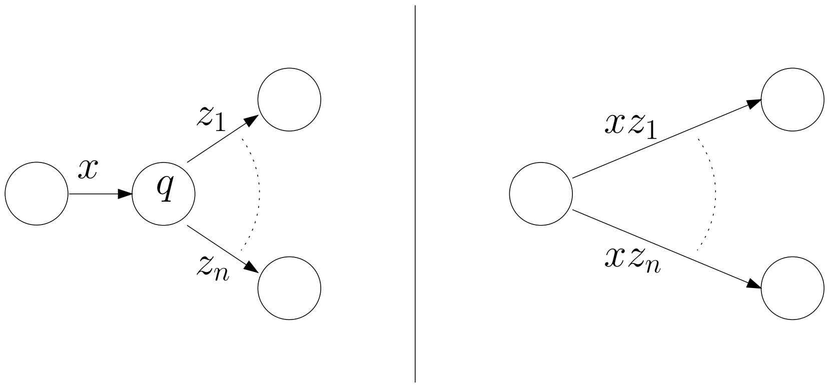

Figure 3: We require $n-1$ new copies of string x after the state elimination of q.

Another problem with state elimination is that different removal sequences give different regular expressions for the same language. Although we cannot avoid the exponential blow-up in state elimination, we can obtain a shorter regular expression by choosing a better removal sequence. Fig. 4 illustrates this idea. Therefore, it is natural to ask how to compute a better removal sequence from a given FA.

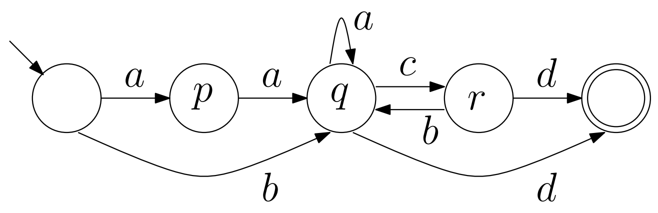

Figure 4: An example of different regular expressions by different removal sequences for a given FA. $E_1 = (aa + b)(a + cb)^*(cd + d)$ is the output of state elimination in $p \to r \to q$ order and $E_2 = (aa + b)a^*c(ba^*c)^*(ba^*d + d) + (aa + b)a^*d$ is the output of state elimination in $p \to q \to r$ order, where $L(E_1) = L(E_2)$.

Recently, Delgado and Morais [5] proposed heuristics for computing a smaller regular expression from a given FA A. They introduced the *weight* of a state q in A. Given a transition $t = (p, \alpha, q)$, the weight of t is the total number of character appearances in α. Then, the weight of q, which we call *state weight*, is defined as *the sum of in-transition weights + the sum of out-transition weights + the loop weight*. Then, they eliminate a state that has the lightest weight. Although this heuristic is better than random selection, it is straightforward to give examples in which the greedy choice does not lead to shorter regular expressions.

Assume that we have an algorithm that computes an optimal removal sequence for a given FA A. Since the algorithm deals with FAs, the size of FAs must be closely related to the runtime of the algorithm. Therefore, if we have a smaller FA A' such that $L(A) = L(A')$, then we can compute the optimal removal sequence more quickly and the removal sequence will lead to a shorter regular expression. On the other hand, since the nondeterministic finite-state automaton (NFA) minimization problem is known to be PSPACE-complete [14], we cannot expect to minimize a very different and smaller FA from a given FA for the same language. We examine NFA minimization methods that reduce the number of duplicate strings in state elimination.

We define two states p and q in an FA $A = (Q, \Sigma, \delta, s, F)$ to be *equivalent* if the following conditions hold:

1. $p \in F$ if and only if $q \in F$.

2. $(p, a, t) \in \delta$ if and only if $(q, a, t) \in \delta$ for any $a \in \Sigma$.

If we have two equivalent states, then we remove one of them, say p, and redirect all in-transitions of p into q. This does not change the language of A while it reduces the size of A.

State equivalence already plays an important role in the literature. If a given FA A is deterministic, then we obtain a minimal DFA after eliminating all equivalent states [11]. If A is a position automaton, then we have a follow automaton [13] after the elimination of all equivalent states.

Lemma 2. *If two source states of a current state q are equivalent, then we need fewer new transitions when eliminating q after merging the two states.*

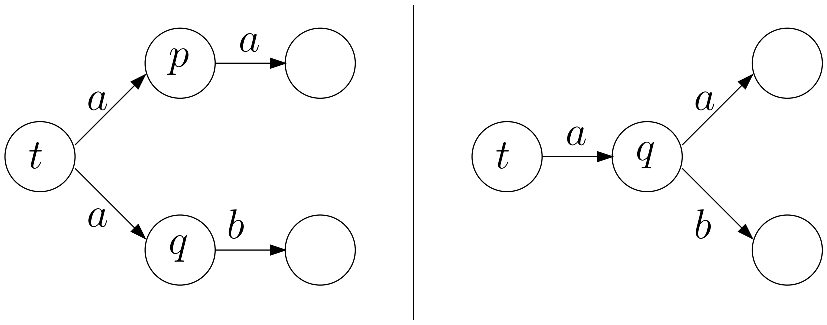

Figure 5: State t has two out-transitions with the same label to two distinct target states p and q. We make all out-transitions of p leave from q and remove p.

Now we consider target states of the current state $t \in Q$ of an FA $A = (Q, \Sigma, \delta, s, F)$. Assume that t has two target states p and q and two out-transitions of t have the same character; namely, $(t, a, p) \in \delta$ if and only if $(t, a, q) \in \delta$, for $a \in \Sigma$, and p and q have no other in-transitions except from t as shown in Fig. 5. Then, we delete p and attach all out-transitions of p to q so that all out-transitions are from q.

Lemma 3. *If the current state t, in an FA $A = (Q, \Sigma, \delta, s, F)$, has two target states that are reachable only from t via the same transition label, then we need fewer new transitions when removing q after merging the two states.*

Ilie et al. [12] adopted these ideas for minimizing NFAs and designed an $O(m \log n)$ time algorithm using $O(m + n)$ space that discovers equivalent states for a given FA A, where n is the number of states and m is the number of transitions of A.

4 Vertical Chopping

Assume that we have an FA A that cannot be minimized any further by using equivalent states. Now we have to compute a removal sequence for A. One question arising from Fig. 4 is why does removing the middle state at the last step lead to a shorter regular expression than when removing it at the second to last step. We observe that the middle state in Fig. 4 has some helpful properties.

Definition 4. *We define a state b in an FA A to be a* bridge state *if it satisfies the following conditions:*

1. *State b is neither a start nor a final state.*
2. *For each string $w \in L(A)$, its path in A must pass through b at least once.*
3. *Once w's path passes through state b for the first time, the path can never pass through any states that have been visited before apart from state b.*

Note that we can decompose A into two subautomata A_1 and A_2 such that $L(A) = L(A_1) \cdot L(A_2)$ from the first and the second requirements. On the other hand, we may have several duplicated states and transitions in both A_1 and A_2 without the third requirement and it does not give a smaller subautomaton in the worst-case. Fig. 6 illustrates this phenomenon.

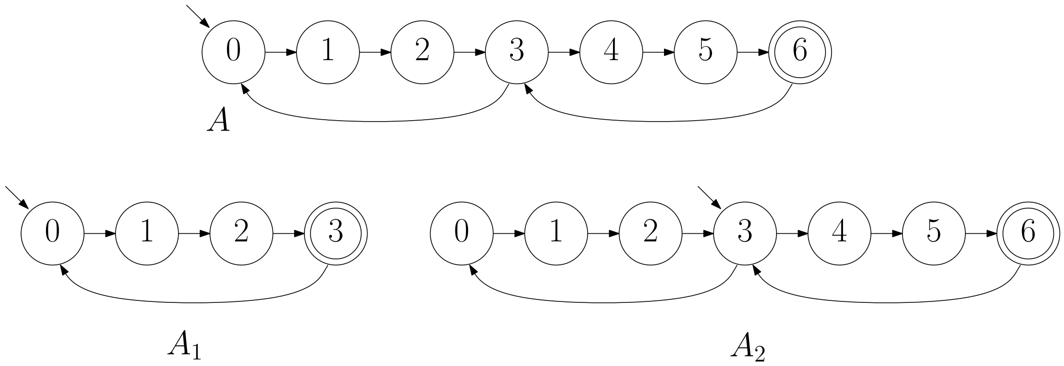

Figure 6: Since state 3 satisfies both the first and second conditions in Definition 4, we can partition A into two subautomata A_1 and A_2 such that $L(A) = L(A_1) \cdot L(A_2)$. Note that A_2 has the same size as A, where state 3 is now the start state of A_2.

The third requirement guarantees that if we partition A at a bridge state b into A_1 and A_2, then all out-transitions of b appear only in A_2. Therefore, A_1 and A_2 have only b as a common state between them. Fig. 7 gives an example of bridge states.

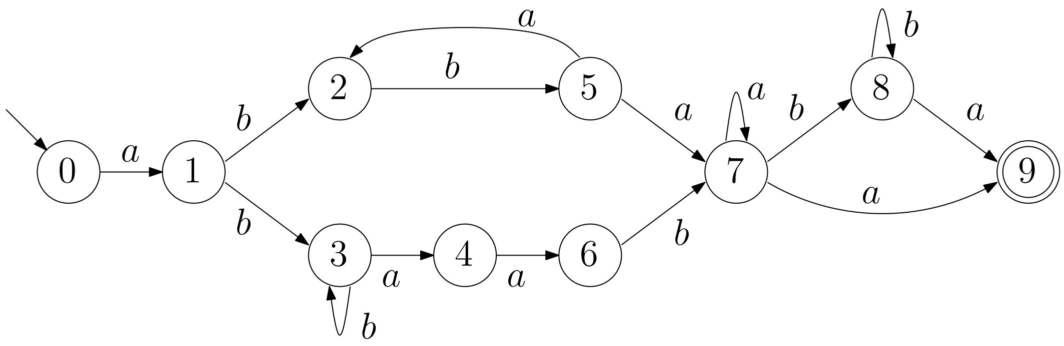

Figure 7: States 1 and 7 are bridge states.

Assume that there is only one final state in A. If there is more than one final state, then we introduce a new final state f' and connect all final states to f' by null transitions. Given an FA $A = (Q, \Sigma, \delta, s, f)$ and a bridge state $b \in Q$, we partition A into two subautomata A_1 and A_2 as follows: $A_1 = (Q_1, \Sigma, \delta_1, s, b)$ and $A_2 = (Q_2, \Sigma, \delta_2, b, f)$, where Q_1 is a subset of states of A that appear on some path from s and b in A, $Q_2 = Q \setminus Q_1 \cup \{b\}$, δ_2 is a subset of transitions of A that appear on some path from b to f in A and $\delta_1 = \delta \setminus \delta_2$. Fig. 8 illustrates partitioning at a bridge state.

Lemma 5. *Given an FA A, let A_1 and A_2 be subautomata of A that are partitioned at a bridge state of A. Then, $L(A) = L(A_1) \cdot L(A_2)$.*

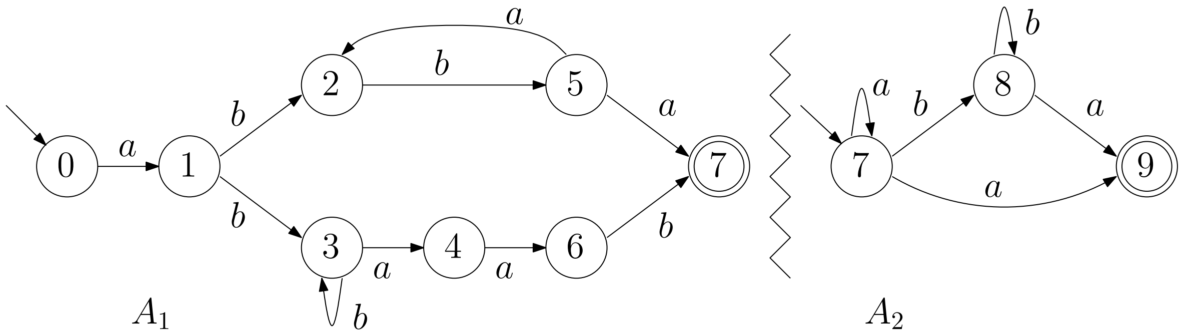

Figure 8: An example of vertical chopping of the automaton in Fig. 7 at a bridge state 7.

Proof. Let $w_1 \in L(A_1)$ and $w_2 \in L(A_2)$. We now process $w_1 w_2$ with respect to A. Since $\delta_1 \subseteq \delta$, we reach state b after reading w_1. Again, we can read out w_2 from b and reach a final state of A since $\delta_2 \subseteq \delta$. $\square$

Note that if states p and q are bridge states in A, then q is still a bridge state in one of the resulting subautomata after the partitioning of A at p. For example, state 1 is a bridge state of the FA A in Fig. 7 and is a bridge state of the FA A_1 in Fig. 8 after chopping at state 7. Let $B = \{b_1, b_2, \ldots, b_k\}$ be a set of bridge states in A, where k is the total number of bridge states in A. Then, $B \setminus \{b_i\}$ is the set of bridge states of A_1 and A_2 after chopping A at state b_i.

Now we present an algorithm that computes all bridge states for a given FA $A = (Q, \Sigma, \delta, s, f)$. We say that a path in A is *simple* if it does not have any cycles. Then, from the second requirement of bridge states in Definition 4, we establish the following statement.

Lemma 6. *Let P be a simple path from s to f in A. Then, only the states in P can be bridge states of A.*

Proof. Assume that a state q is a bridge state and it is not in P. Then, it immediately contradicts the second requirement of bridge states. $\square$

Since A is essentially a directed graph, our algorithm is based on Depth-First Search (DFS). For details on DFS, refer to the textbook [4]. First, we compute a simple path P from s to f of A using DFS. Let $\mathcal{C}_B = \{s, b_1, b_2, \ldots, b_k, f\}$ be the set of states in P. Our approach is to take all states in $\mathcal{C}_B$ as bridge state candidates and to identify all states that violate any requirements in Definition 4 and remove them from $\mathcal{C}_B$. Then, we have bridge states.

We use DFS to explore A from s. We maintain the following three values, for each state $q \in Q$,

anc: The index i of a state $b_i \in \mathcal{C}_B$ such that there is a path from b_i to q and there is no path from $b_j \in \mathcal{C}_B$ to q for $j > i$. The **anc** of b_i is i.

min: The index i of a state $b_i \in \mathcal{C}_B$ such that there is a path from q to b_i and there is no path from q to b_h for $h < i$ in A without visiting any states in $\mathcal{C}_B$.

max: The index i of a state $b_i \in \mathcal{C}_B$ such that there is a path from q to b_i and there is no path from q to b_j for $i < j$ in A without visiting any states in $\mathcal{C}_B$.

The **min** value of a state q means that there is a path from q to $b_{\mathbf{min}}$. If a state $b_i \in \mathcal{C}_B$ has a **min** value, then it implies that b_i is in a cycle. Similarly, if b_i has a **max** value and $\mathbf{max} \neq i+1$, then it means that there is another simple path from b_i to $b_{\mathbf{max}}$ without passing through b_{i+1}.

For computing three values for all states in A efficiently, we first visit all states of $\mathcal{C}_B$ in $s \rightarrow b_1 \rightarrow b_2 \rightarrow \cdots \rightarrow b_k \rightarrow f$ order before visiting any other states in DFS. When a state $q \in Q \setminus \mathcal{C}_B$ is discovered in DFS, q inherits **anc** of its preceding state. A state q has two types of child state: One type is a subset T_1 of $\mathcal{C}_B$ and the other is a subset T_2 of $Q \setminus \mathcal{C}_B$; namely, some child states of q are bridge state candidates and the other states of q are not. Once we have finished exploring all descendants of q, we update **min** and **max** of q as follows:

$$\mathbf{min} = \min(\min_{q \in T_1}(q.\mathbf{anc}), \min_{q \in T_2}(q.\mathbf{min}))$$

and

$$\mathbf{max} = \max(\max_{q \in T_1}(q.\mathbf{anc}), \max_{q \in T_2}(q.\mathbf{max})).$$

Fig. 9 provides an example of DFS result after updating (**min**, **anc**, **max**) for all states in a given FA.

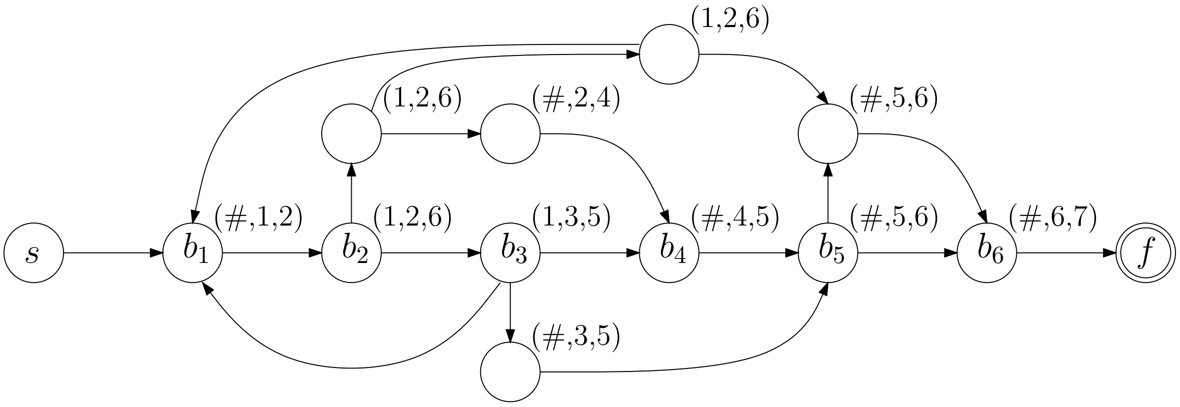

Figure 9: An example of DFS that computes (**min**, **anc**, **max**), for each state of A, for a given $\mathcal{C}_B = \{s, b_1, b_2, b_3, b_4, b_5, b_6, f\}$, where # denotes the null index.

If a state $b_i \in \mathcal{C}_B$ does not any out-transitions except for a transition to $b_{i+1} \in \mathcal{C}_B$ (for example, b_6 in Fig. 9), then b_i has $(\#, i, i+1)$ when DFS is complete, where # denotes the null index. Once we compute DFS and have (**min**, **anc**, **max**) for all states in A, we remove states that violate any requirements to be bridge states from $\mathcal{C}_B$. Assume $b_i \in \mathcal{C}_B$ has (h, i, j), where $h < i$ and $i < j$.

First, we remove $b_{h+1}, b_{h+2}, \ldots, b_i$ from $\mathcal{C}_B$ since there is a path from b_i to b_h and, therefore, all these states can be revisited after visiting b_i; it violates the third requirement in Definition 4. For example, b_2 in Fig. 9 is removed from $\mathcal{C}_B$ since its **min** is not #. If h is #, then we do not remove any states. Second, we remove $b_{i+1}, b_{i+2}, \ldots, b_{j-1}$ from $\mathcal{C}_B$ since there is a path from b_i to b_j; that is, there is another simple path from b_i to f without passing through these states. Finally, we remove s and f from $\mathcal{C}_B$.

We obtain two bridge states b_1 and b_6 after removing states from $\mathcal{C}_B = \{s, b_1, \ldots, b_6, f\}$ in Fig. 9 that violate any requirements to be bridge states in Fig. 9. Since it takes constant time to compute three values for each state and DFS runs in linear time in the size of a given FA, we establish the following result.

Theorem 7. *We can compute a set of bridge states for a given FA $A = (Q, \Sigma, \delta, s, f)$ in $O(|Q| + |\delta|)$ time using DFS.*

Now we demonstrate how bridge states help to compute a shorter regular expression from a given automaton A. Note that we use state elimination for computing regular expressions. As we have mentioned previously, the removal sequence for state elimination is crucial when we wish to compute a shorter regular expression.

Lemma 8. *If all states in a given automaton $A = (Q, \Sigma, \delta, s, f)$ are bridge states, then state elimination gives the same regular expression whatever the removal sequence of states of A we use.*

Proof. Since all states are bridge states, they are in the same simple path from the start state to the final state of A. Assume that we assign unique number for each state from s to f as $q_1, q_2, \ldots q_m$, where m is the number of states in A. Then, the state elimination of state q_i makes a new transition

$$\delta(q_{i-1}, q_{i+1}) = \delta(q_{i-1}, q_i) \cdot \delta(q_i, q_i)^* \cdot \delta(q_i, q_{i+1})$$

from q_{i-1} to q_{i+1}. Note that it does not introduce any new duplicated strings except the Kleene star for a self-loop. Thus, the state elimination of A gives a simple catenation. Therefore, it always gives the same regular expression independently of the removal sequence for the states in A. □

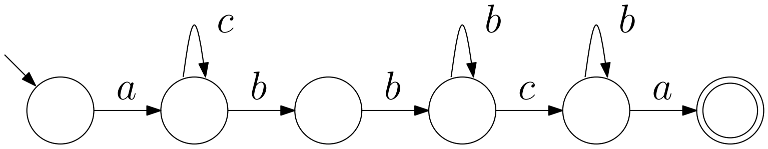

Figure 10: An example of an FA whose states are all bridge states. Note that state elimination always gives $ac^*bbb^*cb^*a$ whatever removal sequence we use.

Proposition 9. *Given an EA $A = (Q, \Sigma, \delta, s, f)$, all and only regular expressions in transitions will appear in the corresponding regular expression when we compute the expression using state elimination.*

Proof. Since state elimination is based on catenations of in-transitions and out-transitions of the current state to be eliminated, the statement is true. □

Now we answer the question arising in Fig. 4. We assume that there are no three consecutive bridge states in A. If there are, then we delete the middle bridge state by state elimination. Given an EA $A = (Q, \Sigma, \delta, s, f)$, let $\mathbb{C}(A)$ be the total number of character appearances in transitions of A; that is,

$$\mathbb{C}(A) = \sum_{i,j} |e_{ij}|, \text{ for each } (q_i, e_{ij}, q_j) \in \delta, \text{ where } q_i, q_j \in Q.$$

For example, if A is $(\{s, f\}, \Sigma, (s, E, f), s, f)$, which is the final expression automaton of state elimination for computing a corresponding regular expression, then $\mathbb{C}(A) = |E|$.

Theorem 10. *Given an expression automaton $A = (Q, \Sigma, \delta, s, f)$ and a set B of bridge states of A, the optimal removal sequence must eliminate all states in $Q \setminus B$ before eliminating any bridge states.*

Proof. Without loss of generality, we assume that we have an optimal removal sequence OPT of state elimination for A that eliminate a bridge state b first. We prove that there is a shorter regular expression using a different removal sequence and, therefore, OPT is not an optimal sequence.

Since we assume that there are no three consecutive bridge states in A, either a target state or a source state of b must be not a bridge state. Let us assume that a target state is not a bridge state. Fig. 11 illustrates an example of the state elimination of b in A.

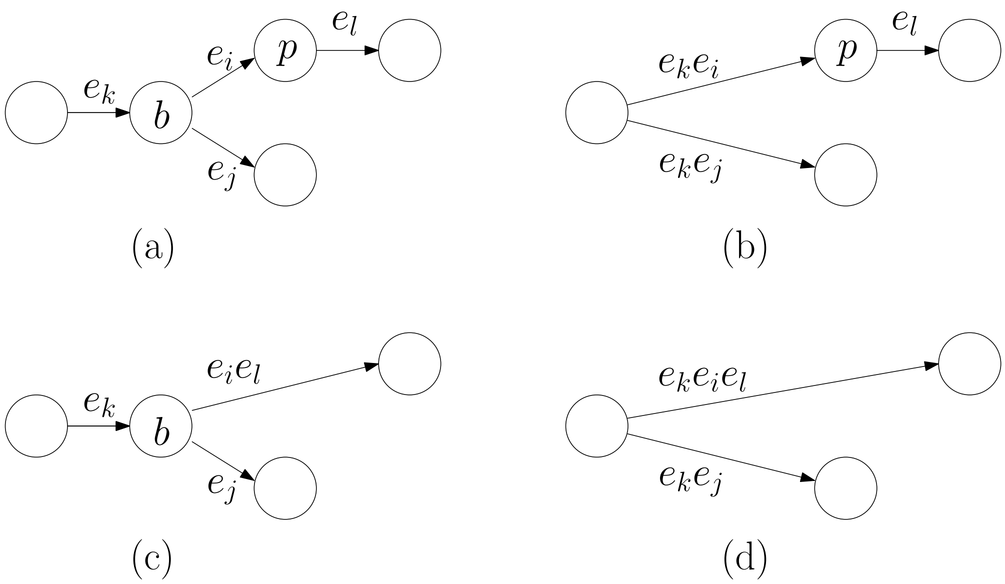

Figure 11: (a) is an example of the state elimination of a bridge state b. Note that $\mathbb{C}(A) < \mathbb{C}(A_b)$. (b) is when OPT eliminates a target state of b from A_b. Note that $\mathbb{C}(A_p) < \mathbb{C}(A_{bp})$.

Let A_b be the resulting expression automaton after the state elimination of b. Then, $\mathbb{C}(A) < \mathbb{C}(A_b)$ by Fig. 2. Let q be the next state to be eliminated after b by OPT. We consider two cases: Case 1 is when q is a target or a source state of b and Case 2 is when q is not.

1. If q is a target or a source state of b: Assume that q is a target state of b. In A_b, q has at least the same number of in-transitions compared to q in A and each in-transition has a longer expression. Therefore, $\mathbb{C}(A_p) < \mathbb{C}(A_{bp})$. Moreover, a target state of p in A_{bp} has longer expressions of in-transitions than the corresponding expression of in-transitions in A_p as shown in Fig. 11 (b).

2. If q is neither a target nor a source state of b: The state elimination of q produces the same new expressions in both A and A_b. Then, since $\mathbb{C}(A) < \mathbb{C}(A_b)$, $\mathbb{C}(A_p) < \mathbb{C}(A_{bp})$.

Let A_{OPT} be the expression automaton computed by OPT and A' be the corresponding expression automaton that we make by eliminating the same state as OPT does except for b. Then, by the same argument, it is always true that $\mathbb{C}(A') < \mathbb{C}(A_{OPT})$. Once OPT completes state elimination, then $\mathbb{C}(A') < \mathbb{C}(A_{OPT})$ and A' has three states, s, f and b. Note that $\mathbb{C}(A_{OPT})$ is the size of the regular expression computed by OPT.

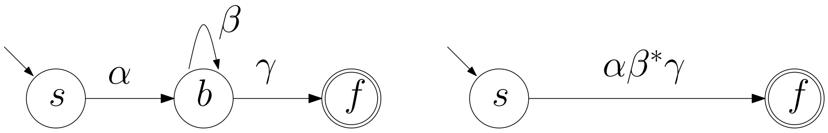

Figure 12: An example of state elimination of a bridge state. Note that the numbers of character appearances are same for both automata.

Now we eliminate b from A' and denote the resulting expression automaton by A'_b as illustrated in Fig. 12. Note that $\mathbb{C}(A'_b) = \mathbb{C}(A')$ is the size of the corresponding regular expression that we have computed. Since $\mathbb{C}(A'_b) = \mathbb{C}(A') < \mathbb{C}(A_{OPT})$, we have computed a regular expression that is shorter than the regular expression computed by OPT — a contradiction. Therefore, the optimal removal sequence must eliminate all states in $Q \setminus B$ before eliminating any bridge states. $\square$

Theorem 10 suggests that given an automaton A, we identify all bridge states of A, chop A into several subautomata using bridge states, compute corresponding regular expressions for each subautomaton and catenate the resulting regular expressions to obtain a regular expression for A. Note that each subautomaton is disjoint from every other subautomata except for bridge states. Thus, vertical chopping is a divide-and-conquer approach based on the structural properties of A.

5 Horizontal chopping

Now we have an FA A without any bridge states and, therefore, we can assume that there is only one start state and one final state in A. Although we cannot avoid computing a removal sequence for A, we can sometimes avoid examining all of A to compute such a sequence. For example, we can partition A, shown in Fig. 13, into two subautomata A_u and A_l. We can compute corresponding regular expressions e_u and e_l for A_u and A_l, respectively. Then, a regular expression for A is $e_u + e_l$, which does not increase the number of character appearances.

Another interesting observation is as follows. Assume that an optimal removal sequence is $5 \to 3 \to 4 \to 6 \to 2$ for the FA in Fig. 13. Then, a removal sequence, $3 \to 4 \to 6 \to 5 \to 2$ gives the same regular expression as before since state elimination of a state in the upper subautomaton does not affect expressions in the lower subautomaton. It implies that sometimes we can compute an optimal removal sequence for a given FA A by computing optimal removal sequences for subautomata and combining them. This approach is also a divide-and-conquer approach. Since we partition A horizontally, we call it *horizontal chopping*.

For horizontal chopping of a given FA $A = (Q, \Sigma, \delta, s, f)$, we have to identify subautomata of A such that each subautomaton is disjoint from each other except for s and f. Our algorithm is based on DFS. When exploring A, we maintain a group index for each state

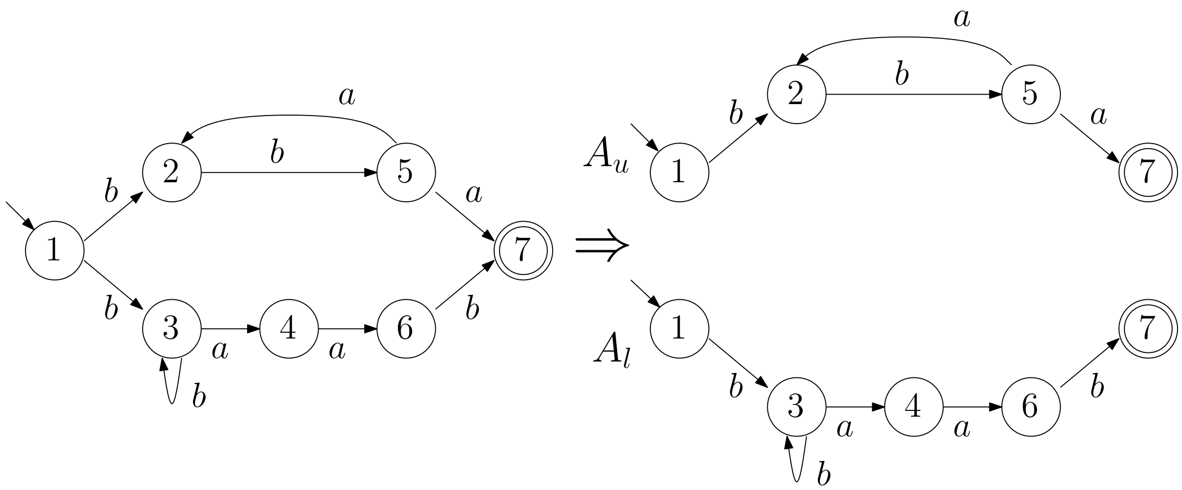

Figure 13: An example of horizontal chopping for a given FA without bridge states.

of A. First, we assign a different group index for each child of s in A. Assume p is the current state with group index i and q is the next state to visit in DFS. If q does not have a group index (that means q is discovered just now), then q inherits the group index i from p. Otherwise, q already has a group index j and we combine two group indices i and j and regard them as the same group. We continue to explore until we visit all states in A.

Fig. 14 illustrates DFS running to identify groups from a given automaton. Note that when we visit state q from state p, we unite group 1 and group 2 to a single group. From this algorithm, we establish the following result.

Theorem 11. *Given an FA $A = (Q, \Sigma, \delta, s, f)$, we can discover all subautomata that are disjoint from each other except for s and f in $O(|Q| + |\delta|)$ time using DFS.*

Moreover, once we partition A horizontally, some states become bridge states of subautomata. For example, state 2 is a bridge state of A_u and states 3, 4 and 6 are bridge states of A_l in Fig. 13. Note that these states are not bridge states of A. Therefore, we can compute bridge states for each subautomaton and perform vertical chopping if there are bridge states; then, again we can repeat horizontal chopping. We continue chopping until no further chopping is possible, and, then compute a removal sequence. Note that state elimination using horizontal chopping and vertical chopping works well for FAs that preserve the structural properties of corresponding regular expressions. For example, for each catenation operation of a given regular expression that is not enclosed by a Kleene star, there is a bridge state in the corresponding Thompson automaton and position automaton. Similarly, for each union operation that is not enclosed by a Kleene star, we can find a horizontal chopping in the corresponding Thompson automaton. On the other hand, we might not be able to perform any vertical chopping or horizontal chopping in the worst-case. However, then it implies that such an FA is already complex and barely preserves any structural properties of the possible regular expressions. In this case, we can only choose brute force.

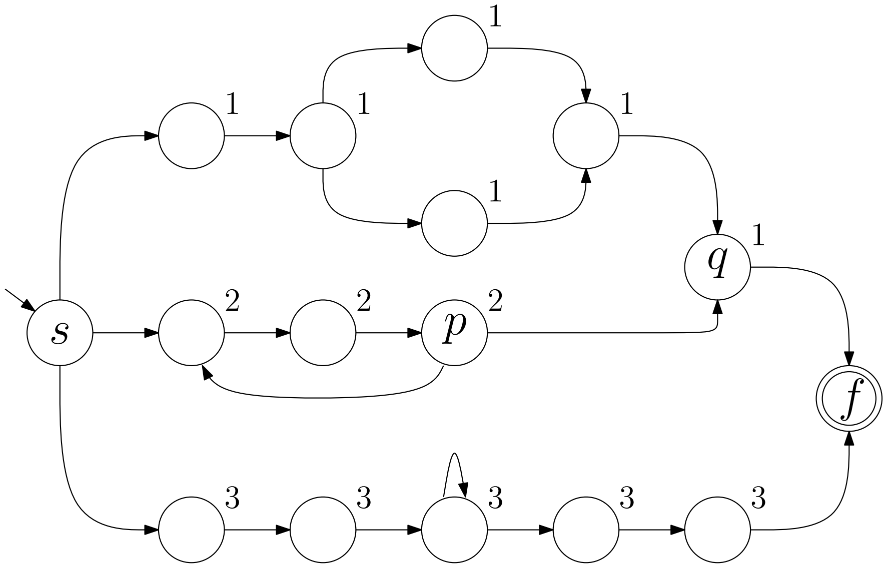

Figure 14: An example of DFS that identifies groups. The label outside a state is its group index. Note that group 1 and group 2 belong to the same group because of q. Therefore, there are two disjoint subautomata that we can use horizontal chopping.

6 Conclusions

There are several number of FA constructions from regular expressions and each construction has different properties in the literature [3, 8, 9, 13, 16, 17]. On the other hand, there are few methods to capture a regular expression from a given FA such as linear equations [6] and state elimination [2]. State elimination is an intuitive construction: we compute a regular expression by removing states in a given automaton while maintaining expressions in transitions. The resulting regular expression of state elimination depends on the removal sequence of states. Thus, if we choose a good removal sequence, then we have a shorter regular expression. However, we have to try all possible sequences to find out an optimal sequence and there are $O(m!)$ sequences, where m is the number of states. Moreover, state elimination blows up the size of regular expressions in transitions much. Thus, these attract us to investigate state elimination for reducing the size of regular expressions and computing a better removal sequence that ensures to have a shorter regular expression.

We have examined NFA minimization to reduce the number of character appearances based on state equivalence. Furthermore, we have investigated the properties of bridge states of an FA and proved that bridge states must be eliminated after eliminating all non-bridge states in A in order to have a shorter regular expression. We can perform vertical chopping of A using bridge states. We have also discovered that we can use horizontal chopping that ensures to compute a state removal sequence of A quickly: once we partition A horizontally, then we can repeat vertical chopping for each subautomaton. We have designed two algorithms for identifying vertical chopping and horizontal chopping of A based on DFS. Both algorithms have a linear running time in the size of A. The combination of vertical chopping and horizontal chopping suggests a divide-and-conquer heuristic for computing a better removal sequence of states of A.

References

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

[2] 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.

[3] P. Caron and D. Ziadi. Characterization of Glushkov automata. *Theoretical Computer Science*, 233(1–2):75–90, 2000.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

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

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

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

[8] D. Giammarresi, J.-L. Ponty, D. Wood, and D. Ziadi. A characterization of Thompson digraphs. *Discrete Applied Mathematics*, 134:317–337, 2004.

[9] V. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1961.

[10] 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.

[11] J. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*, 189–196, New York, NY, 1971. Academic Press.

[12] L. Ilie, G. Navarro, and S. Yu. On NFA reductions. In J. Karhumaki, H. Maurer, G. Paun, and G. Rozenberg, editors, *Theory is Forever (Salomaa Festschrift)*, Lecture Notes in Computer Science 3113, 112–124. Springer-Verlag, Heidelberg, 2004.

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

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

[15] 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.

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

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

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