Shorter Regular Expressions from Finite-State Automata*

Yo-Sub Han and Derick Wood

Department of Computer Science,
The Hong Kong University of Science and Technology
{emmous, dwood}@cs.ust.hk

Abstract. We consider the use of state elimination to construct shorter regular expressions from finite-state automata. Although state elimination is an intuitive method for computing regular expressions from finite-state automata, the resulting regular expressions are often very long and complicated. We examine the minimization of finite-state automata to obtain shorter expressions first. Then, we introduce vertical chopping based on bridge states and horizontal chopping based on the structural properties of given finite-state automata. 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.

Note that we have omitted almost all proofs in this preliminary version.

1 Introduction

It is well known that the family of languages defined by finite-state automata (FAs) is the same as the family of languages described by regular expressions [1]. This result is proved by showing that we can construct FAs from regular expressions and that we can compute regular expressions from FAs.

There are a number of FA constructions; for example, the Thompson construction [2], the position construction [3, 4] and the follow construction [5]. These constructions are inductive and, therefore, preserve the structural properties of regular expressions. For instance, the size of a Thompson automaton is bounded by the size of a given regular expression [6] and the number of states in a position automaton is the number of character appearances in the corresponding regular expression plus one [7].

When converting FAs into regular expressions, we can use either linear equations [8] or state elimination [9]. We consider state elimination. State elimination was already in use in the 1960's, in particular by Brzozowski and McCluskey, Jr. [9] and was carefully formulated by Wood [10]. The idea behind state elimination is simple. We keep removing states, except the start and the final states for

* The authors were supported under the Research Grants Council of Hong Kong Competitive Earmarked Research Grant HKUST6197/01E.

J. Farré, I. Litovsky, and S. Schmitz (Eds.): CIAA 2005, LNCS 3845, pp. 141–152, 2006.
© Springer-Verlag Berlin Heidelberg 2006

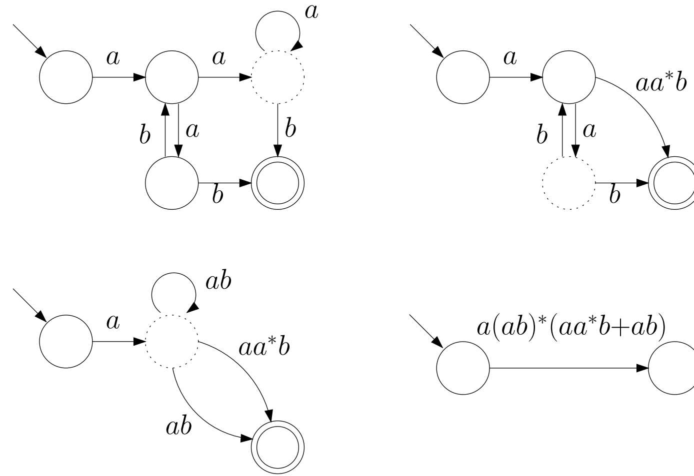

Fig. 1. An example of state elimination. The dotted states are being removed.

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.

In Section 2, we define some basic notions. In Section 3, we describe state elimination and suggest two ways to obtain smaller finite-state automata. Then, we introduce vertical chopping and horizontal chopping of a given FA in Sections 4 and 5. Furthermore, we show 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.

A finite-state automaton 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 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 a finite-state automaton 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

We define the *state elimination* of $q \in Q \setminus \{s, f\}$ in A to be the bypassing of state q, q's in-transitions, q's out-transitions and q's self-looping transition with equivalent expression transition sequences. For each in-transition (p_i, α_i, q), $1 \leq i \leq m$, for some $m \geq 1$, for each out-transition (q, γ, r_j), $1 \leq j \leq n$, for some $n \geq 1$, and for the self-looping transition (q, β, q) in δ, construct a new transition $(p_i, \alpha_i \cdot \beta^* \cdot \gamma_j, r_j)$. If there exists transition (p, ν, r) in δ for some expression ν, then we merge two transitions to give the bypass transition $(p, (\alpha_i \cdot \beta^* \cdot \gamma_j)+\nu, r)$. We then remove q and all transitions into and out of q in δ. We denote the resulting automaton by $A_q = (Q \setminus \{q\}, \Sigma, \delta_q, s, F)$. State elimination maintains the language accepted by a given automaton while removing states. Note that we have regular expressions instead of single characters on a transition of A_q. We say that a finite-state automaton with regular expressions on transitions is an *expression automaton* (EA) [9, 11].

Given an FA $A = (Q, \Sigma, \delta, s, F)$ that is not non-returning and not non-exiting, we transform A into a new FA A' such that $L(A') = L(A)$ and A' is non-returning and non-exiting by introducing a new start state s' and a new final state f' as follows: $A' = (Q \cup \{s', f'\}, \Sigma, \delta \cup \{(s', \lambda, s)\} \cup \{(f_i, \lambda, f') \mid f_i \in F\}, s', f')$.

Lemma 1. *Let $A = (Q, \Sigma, \delta, s, f)$ be a non-returning and non-exiting expression automaton 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 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 expression automaton $A_{Q\setminus\{s,f\}} = (\{s, f\}, \Sigma, (s, E, f), s, f)$, where E is the corresponding regular expression for A.

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 remove q from the automaton A, then we have to introduce $O(mn)$ duplicate strings as new transition labels.

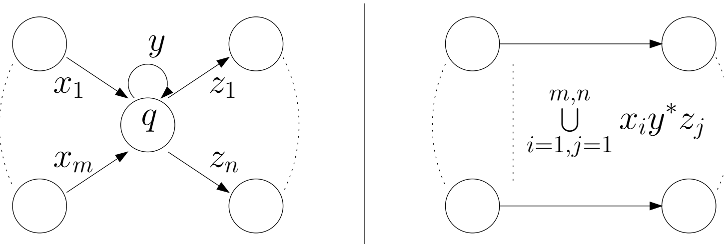

Fig. 2. An example of state elimination that produce many duplicate strings

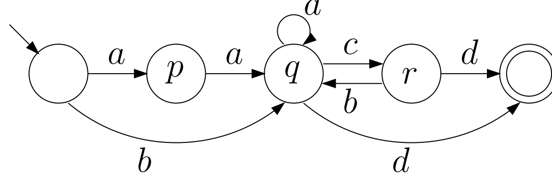

Fig. 3. An example of different regular expressions by different removal sequences for a given finite-state automaton. $E_1 = (aa + b)(a + cb)^*(cd + d)$ is the output of state elimination in $p \to r \to q$ order and $E_2 = (aa + b)a^*c(ba^*c)^*(ba^*d + d) + (aa + b)a^*d$ is the output of state elimination in $p \to q \to r$ order, where $L(E_1) = L(E_2)$.

Another problem with state elimination is that different removal sequences result in different regular expressions. Although we cannot always avoid exponential blow-up, we can still obtain shorter regular expressions by choosing a better removal sequence. Fig. 3 illustrates this idea.

Recently, Delgado and Morais [12] investigated heuristics for computing a shorter regular expression from a given finite-state automaton A. They define 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 a state q in A, 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 remove a state that has the lightest weight based on state 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 to compute an optimal removal sequence for a given automaton A. Then, if we have a smaller automaton A' such that $L(A) = L(A')$, then we can compute the optimal removal sequence more rapidly and the removal sequence will lead to a shorter regular expression.

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 $q \in F$ and 2) $(p, a, t) \in \delta$ if and only if $(q, a, t) \in \delta$, where $t \in Q$ and $a \in \Sigma$. 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 but it does reduce the size of A.

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.*

Now we consider the 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 $(t, a, q) \in \delta$, where $a \in \Sigma$, and p and q have no other in-transitions except from t as shown in Fig. 4. 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.*

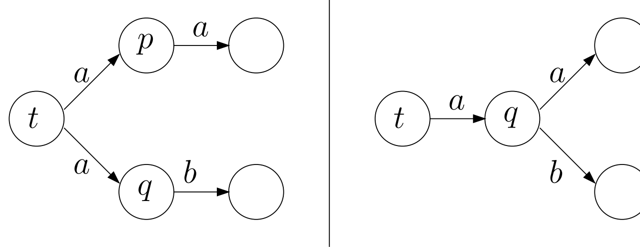

Fig. 4. Note that state t has the same out-transitions to two target states p and q. We make all out-transitions of p leave from q and remove p.

Ilie et al. [13] adopted these ideas to minimize 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. Note that the nondeterministic finite-state automaton (NFA) minimization problem in general is known to be PSPACE-complete [14].

4 Vertical Chopping

Assume that we have a finite-state automaton A that cannot be minimized any further by using equivalent states. Then, we have to compute a removal sequence for A. One question arising from Fig. 3 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. 3 has some helpful properties.

Definition 1. *We define a state b in a DFA A to be a* bridge state *if it satisfies the following three 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. However, we may have several duplicate states and transitions in both A_1 and A_2 without the third requirement. Then, it does not give a smaller subautomaton in the worst-case. Fig. 5 illustrates this phenomenon.

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. 6 gives an example of 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'

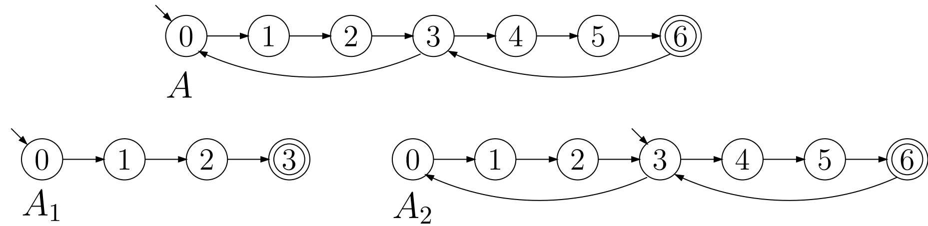

Fig. 5. State 3 satisfies both the first and second conditions in Definition 1 and, therefore, we can partition A into two subautomata A_1 and A_2, where $L(A) = L(A_1) \cdot L(A_2)$. However, A_2 has the same size as A, where state 3 is now the start state of A_2.

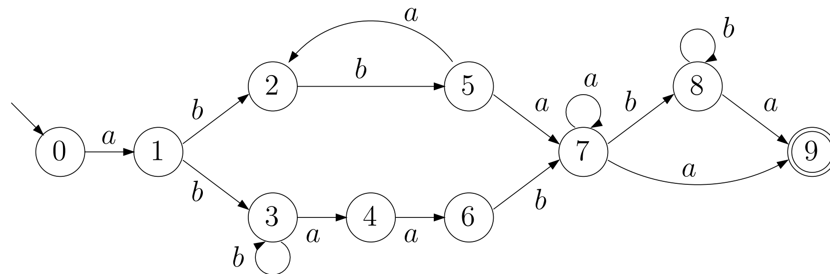

Fig. 6. States 1 and 7 are bridge states

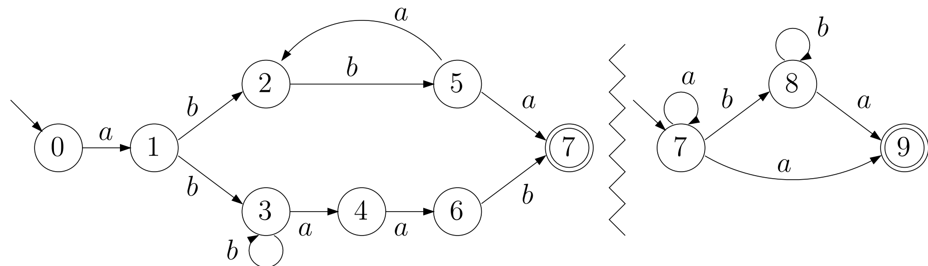

Fig. 7. An example of vertical chopping of the automaton in Fig. 6 at state 7

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 without visiting b twice 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. 7 illustrates partitioning at a bridge state.

Lemma 4. *Given an FA A, let A_1 and A_2 be subautomata of A that are partitioned at a bridge state of A. Then, $L(A) = L(A_1) \cdot L(A_2)$.*

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, as shown in Fig. 6, state 1 is a bridge state of A and is a bridge state of A_1, shown in Fig. 7, 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.

We say a path in A is *simple* if it does not have any cycles. Then, from the second requirement of bridge states in Definition 1, we establish the following statement.

Lemma 5. *Let P be a simple path from s to f in A. Then, only the states in P can be bridge states of A.*

Since A is essentially a directed graph, we can compute all bridge states for A using Depth-First Search (DFS) based on Lemma 5.

Theorem 1. *We can compute a set of bridge states for a given automaton $A = (Q, \Sigma, \delta, s, f)$ in $O(|Q| + |\delta|)$ time using DFS.*

Now we demonstrate how bridge states can 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 6. *If all states in a given automaton $A = (Q, \Sigma, \delta, s, f)$ are bridge states, then state elimination results in the same regular expression whatever the removal sequence of states of A we use.*

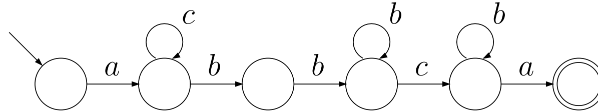

Fig. 8. An example of an FA whose states are all bridge states. Note that state elimination always gives $ac^*bbb^*cb^*a$ no matter which removal sequence we use.

Now we answer the question arising in Fig. 3. 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 expression automaton $A = (Q, \Sigma, \delta, s, f)$, 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 2. *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 (sketch of proof). Without loss of generality, we assume that we have an optimal removal sequence OPT of state eliminations for A that eliminates 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 not be a bridge state. Let us assume that a target state is not a bridge state. 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 neither a target state nor a source state of b.

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 of q in A_b has a longer expression than the regular expression of the corresponding in-transitions of q in A. 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.
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 $\mathbb{C}(A) < \mathbb{C}(A_b)$, we conclude that $\mathbb{C}(A_p) < \mathbb{C}(A_{bp})$.

Let A_{OPT} be the expression automaton computed by OPT and A' be the corresponding expression automaton that we construct 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.

Now we eliminate b from A' and denote the resulting expression automaton by A'_b. 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 non-bridge states before eliminating any bridge states. □

Theorem 2 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 give a regular expression for A. Note that each subautomaton is disjoint from every other subautomaton 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 automaton 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 removal sequences of A to compute such a sequence. For example, we can partition A, shown in Fig. 9, into two subautomata A_u and A_l and 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.

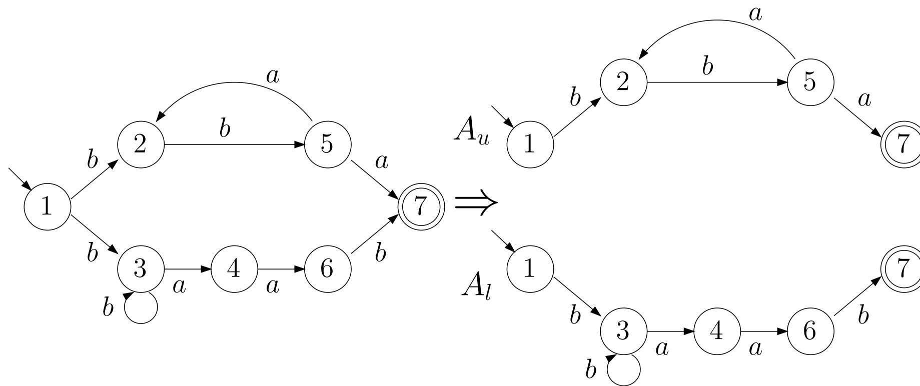

Fig. 9. An example of horizontal chopping for a given automaton without bridge states

Another interesting observation is as follows. Assume that an optimal removal sequence is $5 \to 3 \to 4 \to 6 \to 2$ for the given FA in Fig. 9. 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 when we compute an optimal removal sequence for a given FA A, we can compute optimal removal sequences for subautomata and combine 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 all subautomata are disjoint from each other except s and f. Our algorithm is based on DFS. When exploring A, we maintain a group index for each state 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, (then it must have been visited for the first time) 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 have visited all states in A.

Fig. 10 illustrates how DFS identifies groups from a given automaton. Note that when we visit state q from state p, we merge group 1 and group 2 into a single group.

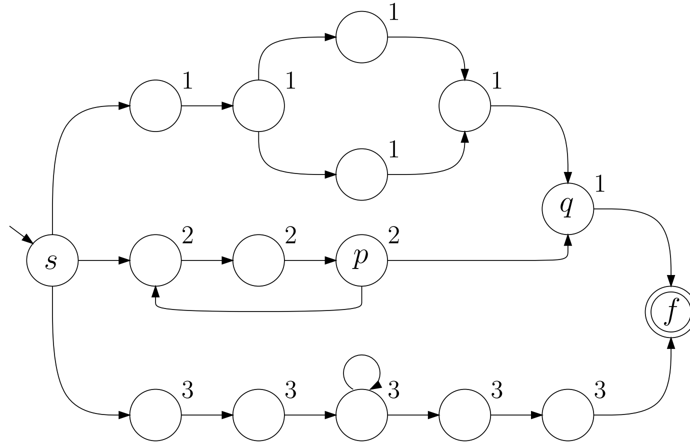

Fig. 10. An example of DFS that identify groups. The label outside a state is its group index. Note that group 1 and group 2 belong to the same group because of q. Therefore, there are two disjoint subautomata that we can use horizontal chopping.

Theorem 3. *Given a finite-state automaton $A = (Q, \Sigma, \delta, s, f)$, we can discover all subautomata that are disjoint from each other except s and f in $O(|Q| + |\delta|)$ time using DFS.*

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. 9. 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.

6 Conclusions

There are several FA constructions from regular expressions and each construction has different properties [7, 6, 3, 5, 4, 2]. On the other hand, there are only two main methods to compute a regular expression from a given FA; namely,

linear equations [8] and state elimination [9]. 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 obtained by state elimination depends on the removal sequence of states. If we choose a good removal sequence, then we obtain a shorter regular expression. On the other hand, we have to try all possible sequences to find the optimal sequence, which is undesirable since there are $O(m!)$ sequences, where m is the number of states. Moreover, state elimination blows up the sizes of regular expressions in transitions. These observations 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 showed 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. Kleene, S.: Representation of events in nerve nets and finite automata. In Shannon, C., McCarthy, J., eds.: Automata Studies, Princeton, NJ, Princeton University Press (1956) 3–42
2. Thompson, K.: Regular expression search algorithm. Communications of the ACM **11** (1968) 419–422
3. Glushkov, V.: The abstract theory of automata. Russian Mathematical Surveys **16** (1961) 1–53
4. McNaughton, R., Yamada, H.: Regular expressions and state graphs for automata. IEEE Transactions on Electronic Computers **9** (1960) 39–47
5. Ilie, L., Yu, S.: Follow automata. Information and Computation **186** (2003) 140–162
6. Giammarresi, D., Ponty, J.L., Wood, D., Ziadi, D.: A characterization of Thompson digraphs. Discrete Applied Mathematics **134** (2004) 317–337
7. Caron, P., Ziadi, D.: Characterization of Glushkov automata. Theoretical Computer Science **233** (2000) 75–90
8. Eilenberg, S.: Automata, Languages, and Machines. Volume A. Academic Press, New York, NY (1974)
9. Brzozowski, J., McCluskey, Jr., E.: Signal flow graph techniques for sequential circuit state diagrams. IEEE Transactions on Electronic Computers **EC-12** (1963) 67–76

10. Wood, D.: Theory of Computation. John Wiley & Sons, Inc., New York, NY (1987)
11. Han, Y.S., Wood, D.: The generalization of generalized automata: Expression automata. In: Proceedings of CIAA'04, Springer-Verlag (2004) 156–166 Lecture Notes in Computer Science 3317.
12. Delgado, M., Morais, J.: Approximation to the smallest regular expression for a given regular language. In: Proceedings of CIAA'04, Springer-Verlag (2004) 312–314 Lecture Notes in Computer Science 3317.
13. Ilie, L., Navarro, G., Yu, S.: On NFA reductions. In Karhumaki, J., Maurer, H., Paun, G., Rozenberg, G., eds.: Theory is Forever (Salomaa Festschrift). Lecture Notes in Computer Science 3113, Springer-Verlag, Heidelberg (2004) 112–124
14. Jiang, T., Ravikumar, B.: Minimal NFA problems are hard. SIAM Journal on Computing **22** (1993) 1117–1141