Simple-Regular Expressions and Languages [*]

Yo-Sub Han, Gerhard Trippen and Derick Wood
Department of Computer Science
The Hong Kong University of Science and Technology
{emmous, trippen, dwood}@cs.ust.hk

Abstract

We define simple-regular expressions and languages. Simple-regular languages provide a necessary condition for a language to be outfix-free. We design algorithms that compute simple-regular languages from finite-state automata. Furthermore, we investigate the complexity blowup from a given finite-state automaton to its simple-regular language automaton and show that there is an exponential blowup. In addition, we present a finite-state automata construction for simple-regular expressions based on state expansion.

1 Introduction

It is well known that the family of languages specified by finite-state automata (FAs) is the same as the family of languages described by regular expressions [13]. It can be proved by showing that we can construct FAs from regular expressions and that we can construct regular expressions from FAs. Additionally, some automata constructions preserve the structural properties of given regular expressions; for example, the Thompson construction [17] or the position construction [8, 14]. Recently, Giammarresi et al. [7] introduced *Thompson languages* and *simple Thompson languages* based on the structural properties of Thompson automata. One interesting property of simple Thompson languages is that for a given Thompson automaton A, a string in a simple Thompson language of A corresponds to a simple path from the start state to the final state of A.

FAs are often used to represent codes since codes are sets of strings. A code can be classified by properties such as *prefix-freeness, suffix-freeness, infix-freeness* and *outfix-freeness* [12]. The conditions that classify code types define proper subfamilies of given language families. For regular languages, for example, outfix-freeness defines the family of outfix-free regular languages, which is a proper subfamily of regular languages. Then, we can classify FAs based on these conditions according to the languages that FAs define. We observe that an FA must have some structural properties to satisfy a certain condition. For example, a deterministic finite-state automaton (DFA) should not have any out-transitions from a final state if the DFA defines a prefix-free language [9]. Given a nondeterministic finite-state automaton (NFA) A, if $L(A)$ is outfix-free, then there are no cycles in A since

[*]Han and Wood were supported under the Research Grants Council of Hong Kong Competitive Earmarked Research Grant HKUST6197/01E and Trippen was supported under RGC/HKUST Direct Allocation Grant DAG03/04.EG05.

an outfix-free regular language is always finite [12]. In other words, all accepting paths of an outfix-free regular language in an FA must be simple. Furthermore, since an outfix-free regular language L is finite, there is an acyclic deterministic finite-state automaton for L. Han and Wood [10] designed an algorithm for determining the outfix-freeness of a given acyclic deterministic finite-state automaton based on the structural properties of outfix-free languages.

FAs are directed graphs with labels on edges and simple paths of FAs are already used in the literature [7, 12]. We examine FAs with respect to simple paths. In Section 2, we define some basic notions. In Section 3, we introduce *simple-regular expressions* and *languages* and design algorithms for computing simple-regular expressions and languages. The algorithm for computing simple-regular language of a given FA is based on the algorithm [16] for enumerating all simple paths. Then, we investigate the complexity blowup from an FA A to its simple-regular language FA A'. We also consider when A is a Thompson automaton or a position automaton since these two constructions are popular.

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. Given two strings x and y in Σ^*, x is said to be an *outfix* of y if there is a string w such that $y = x_1 w x_2$, where $x = x_1 x_2$, For example, *abe* is an outfix of *abcde*. Given a set X of strings over Σ, X is *outfix-free* if no string in X is an outfix of any other string in X.

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 $|A|$ of A is $|Q| + |\delta|$. 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 over 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 assume that A has only *useful* states; that is, each state of A appears on some path from the start state to some final state.

3 Simple-regular languages

3.1 Simple-regular languages from regular expressions and FAs

Definition 1. *Given a regular expression E, we define the* simple-regular expression $\mathcal{S}(E)$ *of E as follows:*

1. $\mathcal{S}(\emptyset) = \emptyset$.

2. $\mathcal{S}(\lambda) = \lambda$.

3. $\mathcal{S}(a) = a$, *for $a \in \Sigma$.*

4. *$\mathcal{S}(E+F)=\mathcal{S}(E)+\mathcal{S}(F)$, where E and F are regular expressions.*

5. *$\mathcal{S}(E\cdot F)=\mathcal{S}(E)\cdot\mathcal{S}(F)$.*

6. *$\mathcal{S}(E^*)=\lambda+E$.*

Given a regular expression E, $L(\mathcal{S}(E))$ is the corresponding simple-regular language of $L(E)$. A language is often given by an FA. We define a simple-regular language from a FA as follows.

Definition 2. *Given an FA A, we define the* simple-regular language $\mathcal{S}(L(A))$ *of $L(A)$ to be a subset of $L(A)$ such that each string is accepted by a simple path in A.*

3.2 Simple-regular languages from finite-state automata

We define a path to be *simple* if it does not have a cycle in A. Brüggemann-Klein and Wood [2] defined the *orbit* of a state q and its *gate* states to characterize one-unambiguous regular languages. The orbit $\mathcal{O}(q)$ of q in A is the strongly connected component of q; that is, it is the set of states of A that can be reached from q and from which q can be reached. We say that $\mathcal{O}(q)$ is *trivial* if it consists of only q and there are no transitions from q to itself in A. A state q of A is a gate of its orbit $\mathcal{O}(q)$ if q is a final state or q has an out-transition to a state outside $\mathcal{O}(q)$. Note that an orbit is a cycle. If we have (p,a,q) in δ, where $p\notin\mathcal{O}(q)$, $q\in\mathcal{O}(q)$ and $a\in\Sigma$, we call p an *entry state* of an orbit. Thus, we compute simple paths for each pair of an entry state and a gate state in an orbit in A.

Given an FA $A=(Q,\Sigma,\delta,s,F)$, we transform A into a new FA A' such that $L(A')=L(A)$ and A' is non-exiting and non-returning as follows: $A'=(Q\cup\{s',f'\},\Sigma,\delta\cup\{(s',\lambda,s)\}\cup\{(f_i,\lambda,f')\mid f_i\in F\},s',f')$. Furthermore, if there is more than one transition between two state p and q in A, we combine these transitions as a single transition. For example, if (p,a,q) and (p,b,q) in δ, we modify these two transitions as $(p,a+b,q)$ in δ.

We now have regular expressions instead of single characters in a transition of A. We call an FA with regular expressions *expression automaton* (EA). Thus, we have an EA $A=(Q,\Sigma,\delta,s,f)$ such that there is at most one transition between two states in Q and A is non-exiting and non-returning. If a regular expression E in δ is not simple, then we modify E as its simple-regular expression E'. If there are self-loops in A, we remove self-loops since a simple path cannot pass through a self-loop. For a formal definition and more details on EAs, refer to Han and Wood [9]. EAs are a generalization of FAs; they allow regular languages on transitions instead of single characters. If we only allow characters on transitions, then A is an FA and if we only allow strings, then A is a generalized automaton [4, 5].

Since the strongly connected components of a directed graph can be computed in linear time [1], we can identify all orbits of A in linear time in $|A|$. Once we identify orbits, then we compute all simple paths for each pair of an entry state and a gate state.

3.3 Computing all simple paths

We design an algorithm that computes all simple paths between two vertices in a graph based on the algorithm of Rubin [16]. Let $G=(V,E)$ be a directed graph without multiple edges or self-loops, where V is a set of vertices and E is a set of edges. Let $|V| = n$ be the number of vertices of V and $|E| = m$ be the number of edges of E. A *path* of length k in G is a nonempty sequence of vertices $p = (v_0, v_1, \ldots, v_k)$ such that (v_i, v_{i+1}) is an edge of G. We say a path p is *simple* if all of its vertices are distinct.

Let I_i be the n-bit boolean vector, where 0 is in the position i and 1 otherwise. Let $\wedge$ and $\vee$ be the boolean *and* and *or* operations. Given a path $p = (v_0, v_1, \ldots, v_k)$, we define the *vertex vector* of p to be an n-bit boolean vector such that the bits in positions $v_0, v_1, \ldots, v_k$ are 1 and the others are 0. We define the *edge vector* of p to be an m-bit boolean vector such that the bit j is either 1 if the edge e_j in p or 0 otherwise. The *path descriptor* of p is an ordered pair $d(p) = (v, e)$, where v is the vertex vector and e is the edge vector of p.

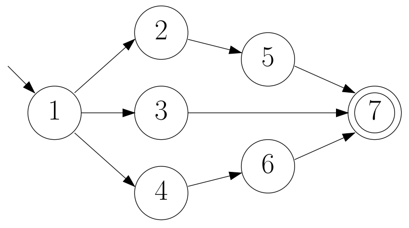

Figure 1: An example of an FA.

For example, we can represent the FA in Fig. 1 as a graph $G = (V, E)$ such that $V = \{1, 2, 3, 4, 5, 6, 7\}$ and $E = \{e_1 = (1, 2), e_2 = (1, 3), e_3 = (1, 4), e_4 = (2, 5), e_5 = (3, 7), e_6 = (4, 6), e_7 = (5, 7), e_8 = (6, 7)\}$. Note that we assign a unique index number for each state and assign a unique edge index number for each out-transition. We make all out-transition indices of a state to be consecutive in an edge vector of G; for example, the first three bits of an edge vector are out-transition indices of state 1 in Fig. 1. For a path $p = 1 \to 3 \to 7$, the path descriptor $d(p)$ is $(1010001, 01001000)$.

Lemma 1 (Rubin [16]). *Given a path descriptor (v, e) in $G = (V, E)$, we can compute the sequence of vertices of the path in $O(m)$ time, where $m = |E|$.*

Since G is an FA A, $m = O(n^2)$ if A is nondeterministic and we need $O(n^2)$ time to construct a path from a path descriptor. We propose a more efficient method for computing the sequence of vertices from a given path descriptor in an NFA. Given a path descriptor $d(p) = (v, e)$ of a simple path p in G, where v is a vertex vector and e is an edge vector, let us assume that we have computed the vertex sequences from its start vertex s to an intermediate vertex i and $i_o = i_1 i_2 \cdots i_k$ are the out-transition indices of i in e. Note that since p is a simple path only one bit in i_o must be 1 and the other bits must be 0. We search for the bit with 1.

We notice that bit operations (for example, *shift, and, or* and *not*) take constant time [15]. We use binary search method to find the bit with 1 from $i_o = i_1 i_2 \cdots i_k$ using these bit operations. Assume that k is even. We shift $i_1 i_2 \cdots i_k$ to the left by $k/2$ and append a $k/2$ number of 0s so that we have $i' = i_{k/2+1} \cdots i_k 000 \cdots 0$. If $i' = 0$, then the bit with 1 is in $i_1 i_2 \cdots i_{k/2}$. Otherwise, the bit with 1 is in $i_{k/2+1} \cdots i_k$. For example, if i is 00000100, then $i' = 01000000$ and we know that the bit with 1 must be in $i_5 i_6 i_7 i_8 = 0100$.

We repeat this procedure recursively until we find the bit with 1. It takes at most $\lceil \log k \rceil$ iterations to find the bit with 1 since the procedure is implemented based on binary search method, where k is the number of out-transitions from i in G. The length of a simple path can be at most n and the number of out-transitions from a vertex can be at most n.

Lemma 2. *Given a path descriptor (v, e) in $G = (V, E)$, we can compute the sequence of vertices of the path in $O(n \log n)$ worst-case time, where $n = |V|$.*

Since $m = O(n^2)$ in NFAs, Lemma 2 is an improvement from $O(n^2)$ time to $O(n \log n)$ time compared with Lemma 1.

The algorithm of Rubin [16] is based on matrix multiplication. He showed that we can enumerate all simple paths in a given directed graph G in $O(n^3)$ matrix operations[1], where n is the number of vertices in G.

```
EnumerateAllSimplePath ($G = (V, E)$)

Initialize a $n \times n$ matrix $D$, where $n = |V|$
for $(i, j) \in E$
  $D(i, j) = \{d((i, j))\}$

for $i = 1$ to $n$
  for $j = 1$ to $n$
    for $k = 1$ to $n$
      for each $(v, e) \in D(j, i)$ and $(w, f) \in D(i, k)$
        if $v \wedge w \wedge I_i = 0$ then
          add $(v \vee w, e \vee f)$ into $D(j, k)$
```

Figure 2: The algorithm of Rubin [16] that enumerates all simple paths of a given graph. Each entry $D(i, j)$ contains all simple paths descriptors from i to j in G.

3.4 Computing simple-regular languages

Given an EA $A = (Q, \Sigma, \delta, s, f)$, we compute the simple path matrix D for A using EnumerateAllSimplePath (EASP) in Fig. 2. Then, for each pair of an entry state i and a gate state j of an orbit, we compute all simple paths from $D(i, j)$. For each path descriptor in $D(i, j)$, we construct the corresponding simple path and compute the regular expression E for the simple path. Then, we add a new transition (i, E, j) into δ. After we complete to compute all simple paths for all pairs of an entry state and a gate state of an orbit $\mathcal{O}(j)$, we remove all states and transitions in $\mathcal{O}(j)$ except gate states.

In Fig. 3, we construct the simple path matrix using EASP and compute all simple paths from 1 to 2 and from 1 to 3. Note that there is only one nontrivial orbit in the EA A

[1]Note that it is not the total running time to compute all simple paths. There can be an exponential number of simple paths in G.

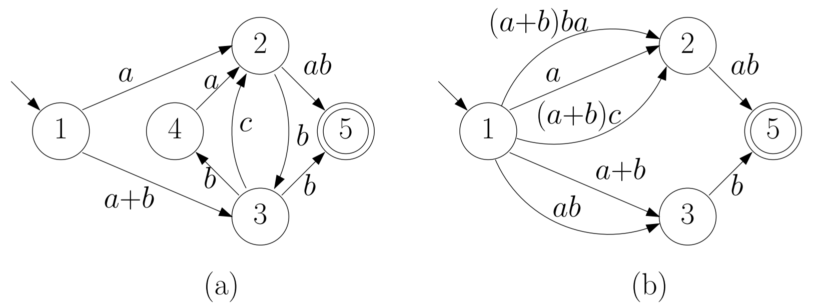

Figure 3: An example of computing the simple-regular language of a given FA using EASP. The set of states {2,3,4} is an orbit $\mathcal{O}(2)$ and {2,3} are gate states of $\mathcal{O}(2)$, where state 1 is an entry state of $\mathcal{O}(2)$.

in Fig. 3 (a), where state 1 is an entry state and states 2 and 3 are gate states. Fig. 3 (b) illustrates the resulting EA A' for the simple-regular language of $L(A)$.

Note that all regular expressions are catenations of regular expression in A; it implies that there are no Kleene stars since A has no Kleene stars in transitions. Furthermore, we can transform an EA into a traditional FA using state expansion.

3.5 State expansion

State elimination was introduced by Brzozowski and McCluskey, Jr. [3] to compute regular expressions from FAs. State elimination maintains the language accepted by a given automaton while removing states; EAs can be used as a data structure for state elimination. *State expansion* is the reverse operation of state elimination.

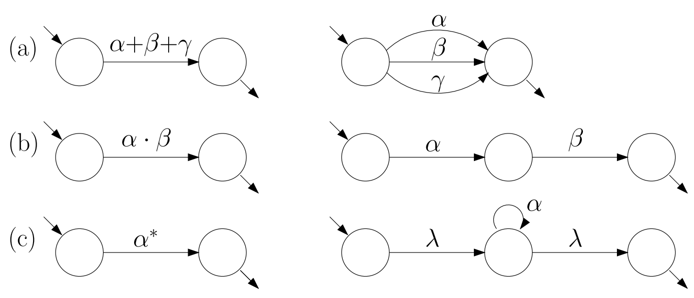

Figure 4: Inductive state expansion procedures; (a) $E = \alpha + \beta + \gamma$, (b) $E = \alpha \cdot \beta$ and (c) $E = \alpha^*$, where α, β and γ are regular expressions.

Fig. 4 shows the inductive state expansion. We can transform an EA A into a (traditional) FA by the sequence of state expansions. We expand each regular expression in a transition of A until the expression is either a single character or the null-string. Ilie and Yu [11] adopted this idea of state expansion and proposed an NFA construction with λ, which is a compact Thompson construction[2].

[2]Ilie and Yu [11] called the construction the ϵNFA construction, where ϵ denotes the null-string λ.

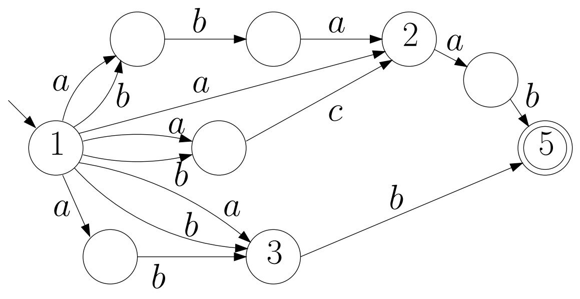

Figure 5: An example of state expansion; the state expansion of the EA in Fig. 3 (b).

Fig. 5 attracts our attention for the complexity issues; the size of the automaton has hardly increased from the automaton in Fig. 3 (b). We investigate the complexity between regular expressions and simple-regular expressions and languages. Let the size $|E|$ of a regular expression E be the number of character appearances in E and k be the number of Kleene stars in E.

Lemma 3. $|\mathcal{S}(E)| = |E| + k$.

Proof. The proof is clear from Definition 1. For each starred subexpression F in E, we replace F with $\lambda + F$. For example, if $E = \alpha \cdot \beta^* \cdot \gamma$, then $\mathcal{S}(E) = \alpha \cdot (\beta + \lambda) \cdot \gamma$. Thus, $|\mathcal{S}(E)|$ increases by k if there are k Kleene stars. $\square$

We now consider the size of an FA for $\mathcal{S}(E)$ based on state expansion. Since $\mathcal{S}(E)$ does not include any Kleene stars, we use only union and catenation constructions, see Fig. 4 (a) and (b).

Theorem 1. *Given a regular expression E, let $A_{\mathcal{S}(E)}$ be the FA of $\mathcal{S}(E)$ constructed by state expansion as shown in Fig. 4. Then, $|A_{\mathcal{S}(E)}| \leq |E| + 2$.*

Proof. Let $|E_\Sigma|$ be the number of characters over Σ, $|E_\cdot|$ be the number of catenation operations, $|E_+|$ be the number of union operations and $|E_*|$ be the number of Kleene stars in E, respectively. It implies that $|E| = |E_\Sigma| + |E_\cdot| + |E_+| + |E_*|$. Then, $|\mathcal{S}(E)| = |E| + |E_*|$ by Lemma 3. Let us compare E and its simple-regular expression $\mathcal{S}(E)$. If a character $a \in \Sigma$ appears in E, a also appears in $\mathcal{S}(E)$ and for each Kleene star in E, there is a corresponding λ in $\mathcal{S}(E)$. In other words, $|\mathcal{S}(E)_\Sigma| = |E_\Sigma| + |E_*|$ and $|\mathcal{S}(E)_\cdot| = |E_\cdot|$.

Note that state expansion requires a new state for each catenation operation and a new transition for each character from Σ as shown in constructions (a) and (b) of Fig. 4. Therefore, $|A_{\mathcal{S}(E)}| = |\mathcal{S}(E)_\Sigma| + |\mathcal{S}(E)_\cdot| + 2 = |E_\Sigma| + |E_*| + |E_\cdot| + 2 \leq |E| + 2$. $\square$

The constant two in Theorem 1 is from the construction. First, we have an EA that has one start state, one final state and one transition between them with a given regular expression. Theorem 1 shows that state expansion gives an FA with size at most $|E| + 2$ instead of $O(|E|)$, where E is a simple-regular expression.

3.6 Complexity issues

We already know that there can be an exponential number of simple paths between two vertices in a graph in the worst-case. On the other hand, an FA that has a polynomial number of states can accept an exponential number of strings. See Fig. 6 for an example.

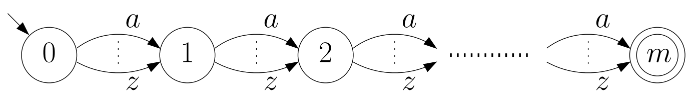

Figure 6: The FA A accepts all strings of length m over $\Sigma = \{a, b, c, \ldots, z\}$; in other words, A accepts n^m strings while the size of A is $m + 1 + mn = O(mn)$.

Thus, one interesting question is that whether or not we need an exponential number of states for representing an FA of simple-regular language of a given FA. Let us consider a DFA $A = (Q, \Sigma, \delta, s, f)$ such that for any two distinct states p and q in Q, (p, a, q) is in δ and each transition label in δ is unique. See Fig. 7 for such an example.

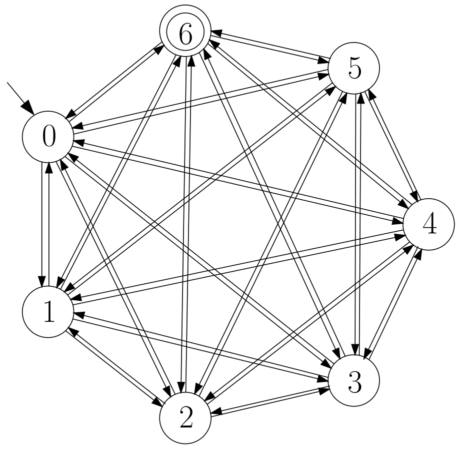

Figure 7: An example of an FA that has an exponential blowup for computing its simple-regular language. The transition label from p to q is pq and, thus, each label is unique. (For example, the transition label from state 1 to state 4 is 14.)

We first construct a DFA $A' = (Q', \Sigma, \delta', s', f')$ of the simple-regular language of $L(A)$ such that A' accepts all strings that are spelled out by simple paths in A. Note that each simple path in A spells out a unique string since A has a unique label for each transition. On the other hand, all accepted strings must start with one of out-transition labels of s. For example, all strings must start with one of $\{01, 02, 03, 04, 05, 06\}$ in Fig. 7. Let m be the number of states of A. We add $m-1$ out-transitions, which are out-transitions of s in A, to the start state s' and each out-transition label is different from each other. Now let us consider a target state q of s', where (s', a, q) is in δ'. We can find the corresponding state $q^\natural$ in A; $q^\natural$ has $m-1$ out-transitions and one of them is to s. Then, with a similar argument, there are $m-2$ out-transitions from q in A' and all out-transition labels are different. Fig. 8 illustrates this procedure for the FA in Fig. 7, where $m = 7$.

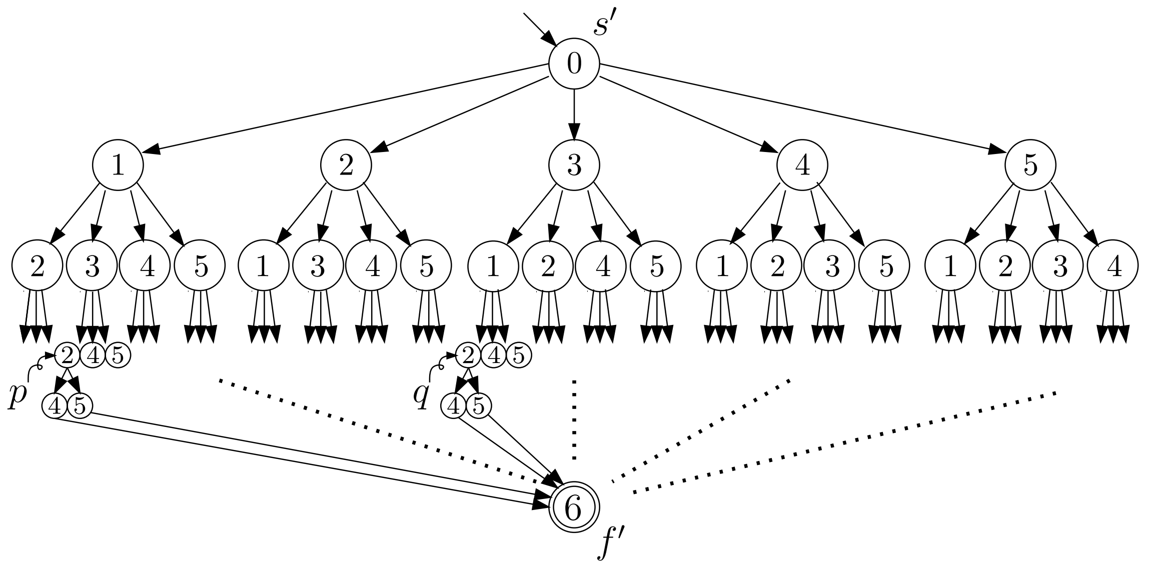

Figure 8: An example of a DFA for a simple-regular language. The indices inside states denote the corresponding states in the FA in Fig. 7. We omit to show a transition from each state to the final state. Note that states p and q are equivalent.

By the construction of A', it is clear that A' is deterministic and $L(A')$ is the simple-regular language of $L(A)$. We define the level of a state q in A' to be the minimal number of transitions from s to q. Given a DFA A', let L_p denote the regular language, where we make p the start state of A'. Since A' can accept a string with length $m-1$, L_p of a state whose level is i accepts a string with length $m-1-i$.

We say that a DFA A' is minimal if and only if $L_p \neq L_q$ for any two states in A' [18]. If $L_p = L_q$, then we say p and q are *equivalent*; for example, two states p and q at the third level in Fig. 8 are equivalent and, therefore, A' is not minimal.

Lemma 4. *If two states p and q are at different levels in A', then $L_p \neq L_q$.*

Proof. Without loss of generality, we assume that the level i of p is smaller than the level j of q. Note that L_q can accept a string with length at most $m-1-j$ and, therefore, a string with length $m-1-i$ cannot be accepted by L_q whereas L_p can accept a string with length $m-1-i$. Therefore, $L_p \neq L_q$. $\square$

Lemma 4 shows that only states at the same level can be equivalent. Then, we construct the minimal DFA for A' by identifying equivalent states at each level and merging them. Consider states p and q in Fig. 8. Let $\Phi(p)$ be a set of states from s' to p in A' and $\Phi(p)^\natural$ be a set of the corresponding states of $\Phi(p)$ in A; for example, $\Phi(p)^\natural = \{0, 1, 3, 2\}$. We observe that $\Phi(p)^\natural = \Phi(q)^\natural$ and $p^\natural = q^\natural$.

Lemma 5. *Two states p and q in A' are equivalent if and only if $\Phi(p)^\natural = \Phi(q)^\natural$ and $p^\natural = q^\natural$.*

Proof.
$\Longrightarrow$ Assume that $\Phi(p)^\natural \neq \Phi(q)^\natural$ or $p^\natural \neq q^\natural$.

1. If $\Phi(p)^\natural \neq \Phi(q)^\natural$, then it implies that the level of p and the level of q are different. Then, by Lemma 4, $L_p \neq L_q$ and, therefore, p and q are not equivalent — a contradiction.

2. If $p^\natural \neq q^\natural$, then it implies that the label of transition from p to the final state f' and the label of transition from q to f' are different and, therefore, $L_p \neq L_q$ — a contradiction.

Therefore, if p and q are equivalent, then $\Phi(p)^\natural = \Phi(q)^\natural$ and $p^\natural = q^\natural$.
$\Longleftarrow$ Let w be a string in L_p. Since $w \in L_p$, there is a simple and unique path for w from $p^\natural$ in A. Furthermore, this path does not visit any states in $\Phi(p)^\natural$. Note that $p^\natural = q^\natural$ and $\Phi(p)^\natural = \Phi(q)^\natural$ and, hence, $w \in L_q$. Therefore, if $\Phi(p)^\natural = \Phi(q)^\natural$ and $p^\natural = q^\natural$, then $L_p = L_q$ and p and q are equivalent. □

We investigate the complexity of the minimal DFA $M(A')$ for A'. A DFA minimization is based on identifying all equivalent states and merging them into a single state and, thus, a state in $M(A')$ is a set of equivalent states of A'. We compute how many equivalent sets of states in A' to count the number of states in $M(A')$.

Lemma 6. *For $k \geq 2$, there are $\frac{(m-2)(m-3)\cdots(m-k-1)}{(k-1)!}$ sets of equivalent states at level k in A'.*

Proof. For two states p and q in A', if $\Phi(p)^\natural = \Phi(q)^\natural$, then we denote it by $p \equiv^\natural q$.

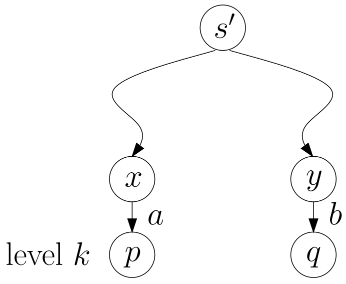

Figure 9: If two states p and q are equivalent, then $x \equiv^\natural y$ and $p^\natural = q^\natural$ by Lemma 5.

At level k in A', there are $(m-2)(m-3)\cdots(m-k-1)$ states. Now two states p and q at the same level are equivalent if and only if $x \equiv^\natural y$ and $p^\natural = q^\natural$ by Lemma 5, where x is the source state of p and y is the source state of q. Note that in A', each state has a unique source state except s' and f'. Let us consider the source state x of p. The level of x is $k-1$ and it has $m-k$ out-transitions. There are $(k-1)!$ states such that for any pair of states x and y, $x \equiv^\natural y$ since there are $(k-1)!$ permutations for given $k-1$ characters. Furthermore, if (x, a, p) is in δ', then there is (y, b, q) in δ' such that $p^\natural = q^\natural$, where a and b are not necessary to be same. Note that such p and q are equivalent. In other words, there are $(k-1)!$ equivalent states of p at level k including itself. Therefore, there are $\frac{(m-2)(m-3)\cdots(m-k-1)}{(k-1)!}$ sets of equivalent states at level k.

Note that when $k = 1$, there are $m-1$ sets of equivalent states instead of $m-2$ because of the final state f'. □

Theorem 2. *Given a DFA $A = (Q, \Sigma, \delta, s, f)$, the size of its minimal DFA of the simple-regular language of $L(A)$ can be exponential in the size of A in the worst-case.*

Proof. Assume that A has a similar structure to the FA in Fig. 7, where $|Q| = m$. Let $M(A')$ be the minimal DFA of the simple-regular language of $L(A)$. Then, there are $\frac{(m-2)(m-3)\cdots(m-k-1)}{(k-1)!}$ states at each level k in $M(A')$ for $1 \leq k \leq m-2$. (To be precise, the final state is at level 1 but we count the start state and the final state, separately.) Therefore, the number of total states in $M(A')$ is

$$\begin{aligned} & 2 + \sum_{k=1}^{m-2} \frac{(m-2)(m-3)\cdots(m-k-1)}{(k-1)!} \\ = \; & 2 + \sum_{k=1}^{m-2} \binom{m-2}{k-1}(m-k-1) > \sum_{k=1}^{m-2} \binom{m-2}{k-1} = O(2^m). \end{aligned}$$

Hence, there are $O(2^m)$ states in $M(A')$. □

Theorem 2 shows that there is an exponential blowup for computing the simple-regular language of L when L is given by an FA A. On the other hand, if L is given by a regular expression E, then $|\mathcal{S}(E)| = O(|E|)$ by Definition 1. It leads us to consider the case when A preserves the structural properties of the corresponding regular expression. For example, the Thompson automata [17] are a proper subfamily of FAs although they represent all regular languages.

Theorem 3. *Given a Thompson automaton A_T, we can compute the corresponding Thompson automaton A'_T such that $L(A'_T)$ is the simple-regular language of $L(A_T)$ by deleting back-edges using DFS.*

Note that we can also compute the simple-regular language of a position automaton since both the Thompson automata [17] and the position automata [8, 14] essentially have the same structure for same regular expressions [6].

4 Conclusions

We have introduced simple-regular expressions and languages. Given an outfix-free regular language L and an FA A for L, all strings in L must be spelled out by simple paths. On the other hand, not all simple-regular languages are outfix-free. For example, $L(abe+abcde)$ is a simple-regular language but not outfix-free. Thus, simple-regular languages are a proper subfamily of regular languages and a proper superfamily of outfix-free regular languages.

We have designed algorithms that compute an FA A' from a given FA A such that $L(A')$ is the simple-regular language of $L(A)$. Since there can be an exponential number of simple paths between two vertices in a graph, we cannot avoid the exponential running time. On the other hand, we know that an FA with a polynomial number of states can accept an exponential number of strings. We have investigated the complexity blowup between A and A' and have shown that given an FA A, the size of its minimal DFA for A' is exponential in the size of A in the worst-case, where $L(A')$ is the simple-regular language of $L(A)$. We have also considered the case when A preserves certain structural properties such as the Thompson automata or the position automata and proved that we can compute the simple-regular languages efficiently without enumerating all simple paths. Since a regular language can be defined by a regular expression, we have presented an efficient FA construction for simple-regular expressions based on state expansion.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.

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

[3] 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.

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

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

[6] D. Giammarresi, J.-L. Ponty, and D. Wood. The Glushkov and Thompson constructions: A synthesis. Unpublished manuscript, July 1998. `http://www.cs.ust.hk/tcsc/RR/1998-11.ps.gz`.

[7] D. Giammarresi, J.-L. Ponty, and D. Wood. Thompson languages. In *Jewels are Forever, Contributions on Theoretical Computer Science in Honor of Arto Salomaa*, 16–24, 1999.

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

[9] Y.-S. Han and D. Wood. The generalization of generalized automata: Expression automata. In *Proceedings of CIAA'04*, 156–166. Springer-Verlag, 2004. Lecture Notes in Computer Science 3317.

[10] Y.-S. Han and D. Wood. Outfix-free regular expressions and languages. manuscript submitted for publication, 2005.

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

[12] H. Jürgensen and S. Konstantinidis. Codes. In G. Rozenberg and A. Salomaa, editors, *word, language, grammar*, volume 1 of *Handbook of Formal Languages*, 511–607. Springer-Verlag, 1997.

[13] 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.

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

[15] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, 1999.

[16] F. Rubin. Enumerating all simple paths in a graph. *IEEE Transactions on Circuits and Systems*, 25:641–642, 1978.

[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.