International Journal of Foundations of Computer Science
Vol. 17, No. 2 (2006) 379–393

World Scientific
www.worldscientific.com

INFIX-FREE REGULAR EXPRESSIONS AND LANGUAGES

YO-SUB HAN*
Department of Computer Science,
The Hong Kong University of Science and Technology,
Clear Water Bay, Kowloon, Hong Kong SAR
emmous@cs.ust.hk

YAJUN WANG†
Department of Computer Science,
The Hong Kong University of Science and Technology,
Clear Water Bay, Kowloon, Hong Kong SAR
yalding@cs.ust.hk

DERICK WOOD*
Department of Computer Science,
The Hong Kong University of Science and Technology,
Clear Water Bay, Kowloon, Hong Kong SAR
dwood@cs.ust.hk

Received 21 April 2005
Accepted 1 August 2005
Communicated by Arto Salomaa

ABSTRACT

We study infix-free regular languages. We observe the structural properties of finite-state automata for infix-free languages and develop a polynomial-time algorithm to determine infix-freeness of a regular language using state-pair graphs. We consider two cases: 1) A language is specified by a nondeterministic finite-state automaton and 2) a language is specified by a regular expression. Furthermore, we examine the prime infix-free decomposition of infix-free regular languages and design an algorithm for the infix-free primality test of an infix-free regular language. Moreover, we show that we can compute the prime infix-free decomposition in polynomial time. We also demonstrate that the prime infix-free decomposition is not unique.

1. Introduction

Codes play a crucial role in many areas such as information processing, date compression, cryptography, information transmission and so on [13]. They are

---

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

†The author was supported under the Research Grants Council of Hong Kong Competitive Earmarked Research Grant HKUST6206/02E.

categorized with respect to different conditions (for example, *prefix-free, suffix-free, infix-free* or *outfix-free*) according to the applications [8, 10, 11, 12, 14]. Since codes deal with sets of strings, they are closely related to formal language theory: a code is a *language*. The conditions that classify code types define proper subfamilies of given language families. For regular languages, for example, prefix-freeness defines the family of prefix-free regular languages, which is a proper subfamily of regular languages.

While infix-free languages have not been studied to the extent of prefix-free languages in the literature, infix-free languages have been used in text searching [2, 6] and computing forbidden words [1, 4]. Ito et al. [11] showed that it is decidable whether or not a given regular language is infix-free and recently, Béal et al. [1] proposed a polynomial-time algorithm to determine infix-freeness for a given deterministic finite-state automaton (DFA). On the other hand, infix-freeness of context-free languages is undecidable as Jürgensen and Konstantinidis [13] had shown. We develop a different algorithm from the algorithm of Béal et al. [1] that can determine infix-freeness of regular languages specified by nondeterministic finite-state automata (NFAs). Moreover, we investigate infix-freeness when languages are given by regular expressions.

Recently, Mateescu et al. [15, 16] examined the prime decomposition of regular languages and showed that it is decidable whether or not a given regular language has a decomposition and the prime decomposition is not unique. Czyzowicz et al. [5] studied the prime decomposition of prefix-free regular languages and proved that the prime prefix-free decomposition is unique. Since the family of infix-free (regular) languages is a proper subfamily of (regular) languages and also of prefix-free (regular) languages, we investigate the prime infix-free decomposition of infix-free regular languages and uniqueness of prime decomposition.

In Section 2, we define some basic notions. We then, in Section 3, define state-pair graphs and develop a polynomial-time algorithm that determines infix-freeness of regular languages. In Section 4, we propose an $O(m^3)$ worst-case algorithm to compute a prime infix-free decomposition for a minimal DFA, where m is the number of states. We also demonstrate that the decomposition is not unique.

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 $|A|$ of A is $|Q| + |\delta|$. Given a transition (p, a, q) in δ, where $p, q \in Q$ and $a \in \Sigma$, we say 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 state in F such that this path spells out the string x. Thus, the language $L(A)$ of a finite-state automaton A is the set of all strings that are spelled out by paths from s to a final state in F. We say that A is *non-returning* if the start state of A does not have any in-transitions and A is *non-exiting* if the final state of A does not have any out-transitions. 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.

Given two strings x and y over Σ, x is a *prefix* of y if there exists $z \in \Sigma^*$ such that $xz = y$ and x is a *suffix* of y if there exists $z \in \Sigma^*$ such that $zx = y$. Furthermore, x is said to be a *substring* or an *infix* of y if there are two strings u and v such that $uxv = y$. Given a set X of strings over Σ, X is *infix-free* if no string in X is an infix of any other string in X. Given a string x, let x^R be the reversal of x, in which case $X^R = \{x^R \mid x \in X\}$. We define a (regular) language L to be infix-free if L is an infix-free set. A regular expression E is infix-free if $L(E)$ is infix-free. We can define prefix-free and suffix-free languages in a similar way.

3. Infix-free regular languages

A regular language is represented by a finite-state automaton or described by a regular expression. We present algorithms that determine whether or not a given regular language L is infix-free based either on finite-state automata or on regular expressions. We assume that $\lambda \notin L$, where $L \neq \{\lambda\}$. Otherwise, L is not infix-free since λ is an infix of any strings.

We first consider the representation of a regular language L by an NFA A. If a final state in A has an out-transition, then $L(A)$ is not prefix-free and, therefore, not infix-free. Similarly, if s of A has an in-transition, then $L(A)$ is not suffix-free and, therefore, not infix-free. Thus, we assume that A is non-returning and non-exiting. Furthermore, if A is non-exiting and has several final states, then all final states are equivalent and, therefore, are merged into a single final state.

Given an NFA $A = (Q, \Sigma, \delta, s, f)$, we assign a unique number for each state from 1 to m, where m is the number of states in Q. We use q_i, for $1 \leq i \leq m$, to denote the corresponding state in A; for example, q_1 denotes s and q_m denotes f.

If $L(A)$ is not infix-free, then there are two distinct strings s_1 and s_2 accepted by A and s_1 is an infix of s_2. It implies that there are two distinct paths in A that spell out s_1 and s_2, respectively, and the path for s_2 has a subpath that spells out s_1. For example, in Fig. 1, the given finite-state automaton accepts $s_1 = abba$ and $s_2 = aabbab$ and the subpath $q_2 \rightarrow q_5 \rightarrow q_6 \rightarrow q_7 \rightarrow q_8$ of the path for s_2 also spells out s_1.

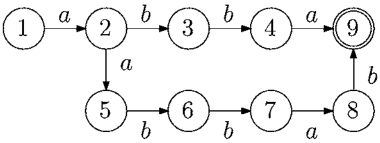


Figure 1: Two strings *abba* and *aabbab* are spelled out by two distinct paths in a given finite-state automaton.

We introduce the *state-pair graph* that is able to identify the case when two distinct paths in A spell out s_1 and s_2, and s_1 is an infix of s_2 as illustrated in Fig. 1.

Definition 1 *Given an NFA* $A = (Q, \Sigma, \delta, s, f)$, *we define the state-pair graph* $G_A = (V, E)$, *where* V *is a set of nodes and* E *is a set of edges, as follows:*

$$V = \{(i,j) \mid q_i \text{ and } q_j \in Q\} \text{ and}$$

$$E = \{((i,j), a, (x,y)) \mid (q_i, a, q_x) \text{ and } (q_j, a, q_y) \in \delta \text{ and } a \in \Sigma\}.$$

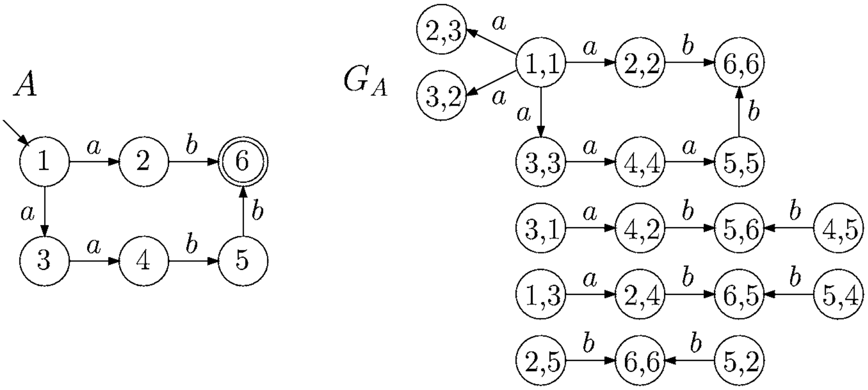


Figure 2: An example of a state-pair graph G_A for a given finite-state automaton A. We omit all nodes that have no out-transitions in G_A.

Fig. 2 illustrates the state-pair graph for a given finite-state automaton A. $L(A) = \{ab, aabb\}$ is not infix-free since ab is an infix of $aabb$. Note that the infix ab appears on the path from $(1,3)$ to $(6,5)$ in G_A.

Lemma 1 *If there is no path from* $(1,i)$ *to* (m,j) *in* G_A *except from (1,1) to* (m,m), *for* $1 \leq i,j \leq m$, *then* $L(A)$ *is infix-free.*

Proof. Assume that $L(A)$ is not infix-free. Then, there are two distinct strings s_1 and s_2 accepted by A, where s_1 is an infix of s_2; namely, $s_2 = w_1 s_1 w_2$. There are two cases to consider for s_2: 1) $w_1 \neq \lambda$ and 2) $w_1 = \lambda$ and $w_2 \neq \lambda$. We examine the two cases separately.

1. Since A accepts s_2, we arrive at some state q_i after reading the prefix w_1 of s_2. Note that $i \neq 1$ since A is non-returning. Since s_1 and s_2 are accepted by A, there should be two sequences of transitions, one of which is from q_1 (the start state) to q_m(the final state) and the other is from q_i to q_j, and both spell out the same string s_1. It implies that there is a path from $(1,i)$ to (m,j) in G_A, where $i \neq 1$ — a contradiction.

2. Since $w_1 = \lambda$, s_1 is a prefix of s_2. Then, there are two paths that spell out s_1 and s_2 in A and both paths have the same prefix s_1. Namely, there is a path from q_1 to q_m for s_1 and a path from q_1 to q_j for the prefix s_1 of s_2.

There is a corresponding path from (1,1) to (m,j) that spells out s_1 in G_A. Now we need to show that $j \neq m$. Since A accepts $s_2 = s_1 w_2$, there should be a transition sequence from q_j $(q_j, z_1, q_k)(q_k, z_2, q_{k+1}) \cdots (q_{k+l-2}, z_l, q_m)$, for some $l \geq 1$, such that $z_1 \cdots z_l = w_2$.

Therefore, if there is no path from $(1,i)$ to (m,j) in G_A apart from (1,1) to (m,m), for $1 \leq i,j \leq m$, then $L(A)$ is infix-free. □

Lemma 2 *If $L(A)$ is infix-free, then there is no path from $(1,i)$ to (m,j) except for the case $(1,1)$ to (m,m) in G_A, where $1 \leq i \leq m$ and $1 \leq j \leq m$.*

Proof. Assume that there is a path that spells out a string s_1 from $(1,i)$ to (m,j) in G_A. It implies that there exists two paths, one of which is from q_1 to q_m and the other is from q_i to q_j and both spell out s_1 in A. There are two cases: 1) $i \neq 1$ and 2) $i = 1$ and $j \neq m$.

1. Since A has only useful states, there should be a transition sequence from q_1 to q_i that spells out a string w_1, which cannot be λ since A is non-returning, and a transition sequence from q_j to q_m that spells out a string w_2, which can be λ when $j = m$. It implies that A accepts both s_1 and $w_1 s_1 w_2$ and s_1 is an infix of $w_1 s_1 w_2$ — a contradiction.
2. Since A has only useful states and $j \neq m$, there should be a transition sequence from q_j to q_m that spells out a string w_2, which cannot be λ. It implies that A accepts both s_1 and $s_1 w_2$ and s_1 is an infix of $s_1 w_2$ — a contradiction.

Therefore, if $L(A)$ is infix-free, then there is no path from $(1,i)$ to (m,j) apart from $(1,1)$ to (m,m) in G_A. □

From Lemmas 1 and 2, we obtain the following result.

Theorem 1 *A regular language $L(A)$ is infix-free if and only if the state-pair graph G_A for a given finite-state automaton A has no path from $(1,i)$ to (m,j) apart from $(1,1)$ to (m,m), where $1 \leq i \leq m$ and $1 \leq j \leq m$.*

Let us consider the complexity of the state-pair graph $G_A = (V,E)$ for a given finite-state automaton $A = (Q, \Sigma, \delta, s, f)$. It is clear that $V = |Q|^2$ from Definition 1. Let δ_i denote the set of out-transitions from a state q_i in A. Then, $|\delta| = \sum_{i=1}^{m} |\delta_i|$, where $m = |Q|$. Since a node (i,j) in G_A can have at most $|\delta_i| \times |\delta_j|$ out-transitions, $|E| = \sum_{i,j=1}^{m} |\delta_i| \times |\delta_j| \leq |\delta|^2$. Therefore, the complexity of G_A is at most $|Q|^2$ nodes and $|\delta|^2$ edges.

A sub-function DFS((i,j)) in Infix-Freeness (IF) shown in Fig. 3 is a depth-first search that starts at a node (i,j) in G_A. Note that although DFS((i,j)) is executed several times inside **for** loop in the algorithm, each node in G_A is visited at most twice. For details on DFS, refer to the textbook [3]. The construction of $G_A = (V,E)$ from A takes $O(|Q|^2 + |\delta|^2)$ time in the worst-case and DFS takes $O(|V| + |E|)$ time. Therefore, the total running time for IF is $O(|Q|^2 + |\delta|^2)$.

Theorem 2 *Given a finite-state automaton $A = (Q, \Sigma, \delta, s, f)$, we can determine whether or not $L(A)$ is infix-free in $O(|Q|^2 + |\delta|^2)$ worst-case time using IF in Fig. 3.*

```
Infix-Freeness($A = (Q, \Sigma, \delta, s, f)$)
/* we assume that $A$ is non-returning and non-exiting. */

Construct $G_A = (V, E)$ from $A$
for each node $(1, i)$ in $V$, where $2 \leq i \leq m$
    DFS($(1, i)$) in $G_A$
    if we meet a node $(m, j)$ for any $j$, $1 \leq j \leq m$
        then output $L(A)$ is not infix-free

DFS($(1, 1)$) in $G_A$
if we meet a node $(m, j)$ for any $j$, $2 \leq j < m$
    then output $L(A)$ is not infix-free

output $L(A)$ is infix-free
```

Figure 3: An infix-freeness checking algorithm for a given NFA.

Since $O(|\delta|) = O(|Q|^2)$ in the worst-case for NFAs, the running time of IF is $O(|Q|^4)$ in the worst-case. On the other hand, if a language is described by a regular expression, then we can choose a construction for finite-state automata that improves the worst-case running time. Since the complexity of the state-pair graph depends on the number of states and the number of transitions of a given automaton, we need a finite-state automata construction that gives fewer states and transitions. One possibility is to use the Thompson construction [17].

Given a regular expression E, the Thompson construction takes $O(|E|)$ time and the resulting Thompson automaton has $O(|E|)$ states and $O(|E|)$ transitions [9]; namely, $O(|Q|) = O(|\delta|) = O(|E|)$. Even though Thompson automata are a subfamily of NFAs, they define all regular languages. Therefore, we can use Thompson automata to determine infix-freeness of a regular language given by a regular expression. Since Thompson automata allow null-transitions, we include the null-transition case to construct the edges for a state-pair graph as follows:

$$V = \{(i, j) \mid q_i \text{ and } q_j \in Q\} \text{ and}$$

$$E = \{((i, j), a, (x, y)) \mid (q_i, a, q_x) \text{ and } (q_j, a, q_y) \in \delta \text{ and } a \in \Sigma \cup \{\lambda\}\}.$$

The complexity of the state-pair graph based on this new construction is the same as before; namely, $O(|Q|^2 + |\delta|^2)$. Therefore, we have the following result when checking regular expression infix-freeness.

Theorem 3 *Given a regular expression E, we can determine whether or not $L(E)$ is infix-free in $O(|E|^2)$ worst-case time.*

Proof. We construct the Thompson automaton A_T for E. Hopcroft and Ullman [9] showed that the number of states in A_T is $O(|E|)$ and also the number of transitions, $O(|Q|) = O(|\delta|) = O(|E|)$. Thus, we construct the state-pair graph based on the new construction that includes null-transitions and determine infix-freeness of $L(E)$ using IF in Fig. 3. Since $O(|Q|) = O(|\delta|) = O(|E|)$, the worst-case time complexity is $O(|E|^2)$. $\square$

We know that a regular language L is prefix-free if there are no out-transitions from a final state of a DFA for L [7]. However, if L is specified by an NFA A, then we have to use subset construction to compute a corresponding DFA from A and the subset construction takes exponential time in the worst-case [18]. By loosening the condition in Theorem 1, we can determine whether or not $L(A)$ is prefix-free in polynomial time.

Theorem 4 *Given a (nondeterministic) finite-state automaton* $A = (Q, \Sigma, \delta, s, f)$, *the regular language* $L(A)$ *is prefix-free if and only if there is no path from* $(1, 1)$ *to* (m, j), *for any* $j \neq m$, *in the state-pair graph* G_A *for* A. *Moreover, we can determine prefix-freeness in* $O(|Q|^2 + |\delta|^2)$ *worst-case time.*

Proof. Using similar arguments to those of Lemmas 1 and 2, we can show $L(A)$ is prefix-free if and only if G_A has no path from $(1, 1)$ to (m, j). Then, we run DFS$((1, 1))$ for G_A until either we meet (m, j) for any $1 \leq j < m$ or we visit all nodes in G_A. $\square$

Based on Theorem 4, we can determine suffix-freeness of $L(A)$ in $O(|Q|^2 + |\delta|^2)$ worst-case time as well. We define a language L to be *bifix-free* if L is prefix-free and suffix-free. Then, we can determine bifix-freeness by checking for prefix-freeness and suffix-freeness.

Theorem 5 *Given a (nondeterministic) finite-state automaton* $A = (Q, \Sigma, \delta, s, f)$, *we can determine prefix-freeness, suffix-freeness and bifix-freeness of* $L(A)$ *in* $O(|Q|^2 + |\delta|^2)$ *worst-case time.*

A language L over Σ is *p-infix-free* if two conditions $xuy \in L$ and $u \in L$ imply that $y = \lambda$, where u, x and y are strings over Σ. Similarly, L is *s-infix-free* if $xuy \in L$ and $u \in L$ imply that $x = \lambda$. For more details on these languages, refer to Ito et al. [11]. Then, we can determine whether or not a given regular language is p-infix-free using state-pair graphs.

Theorem 6 *A regular language* $L(A)$ *is p-infix-free if and only if the state-pair graph* G_A *for a given finite-state automaton* A *has no path from* $(1, i)$ *to* (m, j), *where* $1 < i \leq m$ *and* $1 \leq j < m$.

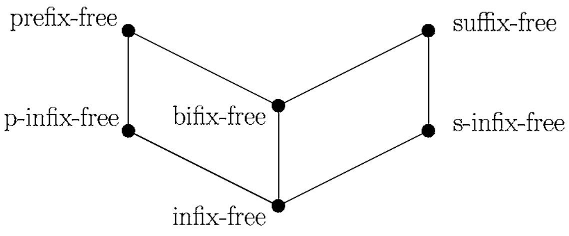


Figure 4: Some code languages and their relationships. For example, all p-prefix-free languages are prefix-free and all infix-free languages are p-infix-free.

Fig. 4 illustrates various code language relationships. For context-free languages, it is undecidable whether or not a given language L is prefix-free, suffix-free, bifix-free or infix-free [13]. The p-infix-free and s-infix-free cases are still open. On the other hand, it is decidable when L is a regular language although there were no known polynomial time algorithms for these decision problems. In this section, we have shown that we can solve these decision problems of all code languages in Fig. 4 in polynomial time using state-pair graphs of regular languages.

We characterize the family of infix-free (regular) languages in terms of closure properties.

Theorem 7 *The family of infix-free (regular) languages is closed under catenation and intersection but not under union, complement or star.*

Proof. We only prove the catenation case. The other cases can be proved straightforwardly.

Assume that $L = L_1 \cdot L_2$ is not infix-free whereas L_1 and L_2 are infix-free. Then, there are two strings $s = s_1 \cdot s_2$ and $w = w_1 \cdot w_2 \in L$, where s_1 and $w_1 \in L_1$, s_2 and $w_2 \in L_2$ and s is a substring of w. Since s is a substring of w, $w = vs_1s_2u$ and at least one of v and u is not null string. Note that w can be decomposed into w_1w_2 and therefore we should be able to partition vs_1s_2u into two substrings w_1 and w_2. Then, either s_1 is an infix of w_1 or s_2 is an infix of w_2 — a contradiction. □

4. Prime infix-free regular languages and decomposition

Decomposition is the reverse operation of catenation. If $L = L_1 \cdot L_2$, then L is the catenation of L_1 and L_2 and $L_1 \cdot L_2$ is a decomposition of L. We call L_1 and L_2 *factors* of L. Note that every language L has a decomposition, $L = \{\lambda\} \cdot L$, where L is a factor of itself. We call $\{\lambda\}$ a *trivial* language. We define a language L to be *prime* if $L \neq L_1 \cdot L_2$, for any non-trivial languages L_1 and L_2. Then, the prime decomposition of L is to decompose L into $L_1L_2 \cdots L_k$, where $L_1, L_2, \cdots, L_k$ are prime languages and $k \geq 1$ is a constant.

Mateescu et al. [15, 16] showed that the primality of regular languages is decidable and the decomposition of a regular language into prime regular languages is not unique. Recently, Czyzowicz et al. [5] considered prefix-free regular languages and showed that the prime prefix-free decomposition for a prefix-free regular language L is unique and can be computed in $O(m)$ worst-case time, where m is the size of the minimal DFA for L. Note that in prime prefix-free decomposition, all factors must be prefix-free.

We investigate prime infix-free regular languages and decomposition.

4.1. Prime infix-free regular languages

Definition 2 *We define a regular language L to be a* prime *infix-free language if $L \neq L_1 \cdot L_2$, for any non-trivial infix-free regular languages L_1 and L_2.*

From now on, when we say prime, we mean prime infix-free.

Definition 3 *We define a state b in a DFA A to be a* bridge state *if the following conditions hold:*

1. *State b is neither a start nor a final state.*
2. *For any string $w \in L(A)$, its path in A must pass through b at least once. Therefore, we can partition A at b into two subautomata A_1 and A_2 such that all out-transitions from b belong to A_2.*
3. *State b is not in any cycles in A.*
4. *$L(A_1)$ and $L(A_2)$ are infix-free.*

Given an infix-free DFA $A = (Q, \Sigma, \delta, s, f)$ with a bridge state $b \in Q$, we can 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 set 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 set of transitions of A that appear on some path from b to f in A and $\delta_1 = \delta \setminus \delta_2$. Fig. 5 illustrates the partition at a bridge state.

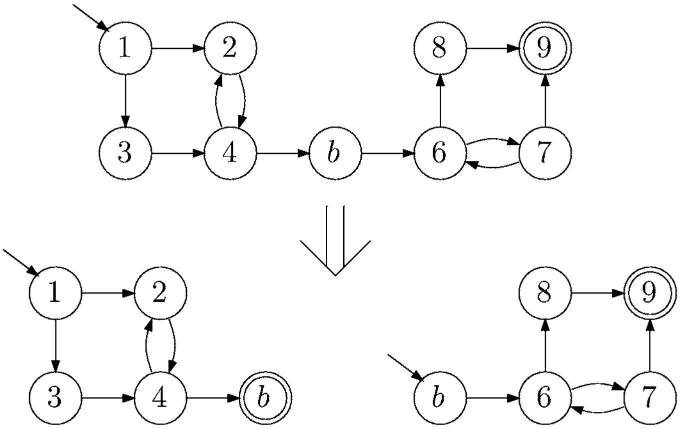


Figure 5: An example of the partitioning of an automaton at a bridge state b.

Lemma 3 *Given a DFA A and its subautomata A_1 and A_2 partitioned at a bridge state, $L(A) = L(A_1) \cdot L(A_2)$.*

Proof. Let $w_1 \in L(A_1)$ and $w_2 \in L(A_2)$. We process w_1w_2 with respect to A. Since $\delta_1 \subseteq \delta$, we reach state b after reading w_1. Again, we can process w_2 from b and reach the final state of A since $\delta_2 \subseteq \delta$. □

Lemma 3 shows that the second requirement in Definition 3 ensures that the decomposition of $L(A)$ is $L(A_1) \cdot L(A_2)$. The third requirement is based on the property that finite-state automata for infix-free regular languages must be non-returning and non-exiting.

Theorem 8 *An infix-free regular language L is prime if and only if the minimal DFA A for L does not have any bridge states.*

Proof. Let s denote the start state and f denote the final state in A.
$\Longrightarrow$ Assume that A has a bridge state q. Then, we can separate A into two automata

A_1 and A_2 such that s is the start state and q is the final state of A_1 and q is the start state and f is the final state of A_2. Then, $L = L(A_1) \cdot L(A_2)$, where $L(A_1)$ and $L(A_2)$ are infix-free — a contradiction.
$\Longleftarrow$ Assume that L is not prime. Then, L can be represented as $L_1 \cdot L_2$, where L_1 and L_2 are infix-free; namely, $L = L_1 \cdot L_2$. Czyzowicz et al. [5] showed that given prefix-free languages A, B and C such that $A = B \cdot C$, A is regular if and only if B and C are regular. Thus, if L is regular, then L_1 and L_2 must be regular since all infix-free languages are prefix-free. Let A_1 and A_2 be minimal DFAs for L_1 and L_2, respectively. Since A_1 and A_2 are non-returning and non-exiting, there is only one start state and one final state for A_1 and A_2. We catenate A_1 and A_2 by merging the final state of A_1 and the start state of A_2 as a single state q. Then, the catenated automaton is the minimal DFA for $L(A_1) \cdot L(A_2) = L$ and has a bridge state q — a contradiction. □

4.2. Prime decomposition of infix-free regular languages

The prime decomposition for an infix-free regular language L is to represent L as a catenation of prime infix-free regular languages. If L is prime, then L itself is a prime decomposition. Thus, given L, we first check whether or not L is prime and decompose L if it is not prime. By the definition of bridge states, we can decompose L into $L(A_1)$ and $L(A_2)$ at bridge states. If both $L(A_1)$ and $L(A_2)$ are prime, a prime decomposition of L is $L(A_1) \cdot L(A_2)$. Otherwise, we repeat the preceding procedure for a non-prime language.

Let B denote the set of bridge states for a given minimal DFA A. Then, the number of states in B is at most m, where m is the number of states in A. Note that once we partition A at $b \in B$ into A_1 and A_2, then only states in $B \setminus \{b\}$ can be bridge states of A_1 and A_2[a]. Therefore, we can determine the primality of $L(A)$ by checking whether A has bridge states and compute a prime decomposition of $L(A)$ using these bridge states. Since there are at most m bridge states in an automaton for an infix-free regular language, we can compute a prime decomposition of $L(A)$ after a finite number of decompositions at bridge states.

First, we show how to compute bridge states and, then, present an algorithm to decompose a non-prime infix-free regular language using bridge states. Let $G(V, E)$ be a labeled directed graph for a given minimal DFA $A = (Q, \Sigma, \delta, s, f)$, where $V = Q$ and $E = \delta$. We say that a path in G is *simple* if it does not have a cycle.

Lemma 4 *Let $P_{s,f}$ be a simple path from s to f in G. Then, only the states on $P_{s,f}$ can be bridge states of A.*

Proof. Assume that a state q is a bridge state and is not on $P_{s,f}$. Then, it contradicts the second requirement of bridge states. □

Since the first three requirements in Definition 3 are based on the structural properties of a given automaton, we compute a set of states that satisfy these three requirements and check the last requirement for each state in the set. Assume that we have a simple path $P_{s,f}$ from s to f in $G = (V, E)$, which can be computed

---

[a] Sometimes, a state in $B \setminus \{b\}$ is not a bridge state anymore after partitioning. Fig. 7 gives an example.

in $O(|V| + |E|)$ worst-case time. All states on $P_{s,f}$ form a set of candidate bridge states (CBS); namely, $CBS = (s, b_1, b_2, \cdots, b_k, f)$.

We use DFS to explore G from s. We visit all states in CBS first. While exploring G, we maintain the following three values, for each state $q \in Q$,

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

min: The index i of a state $b_i \in CBS$ such that there is a path from q to b_i and there is no path from q to b_h for $h < i$ without visiting any state in CBS.

max: The index i of a state $b_i \in CBS$ such that there is a path from q to b_i and there is no path from q to b_j for $i < j$ without visiting any state in CBS.

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

When a state $q \in Q \setminus CBS$ is visited during 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 states in CBS and the other is a subset T_2 of $Q \setminus CBS$; namely, all states in T_1 are candidate bridge states and all states in T_2 are not candidate bridge states. Once we have explored all children of q, we update **min** and **max** of q as follows:

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

and

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

Fig. 6 provides an example of DFS after updating (**min**, **anc**, **max**) for all states in G.

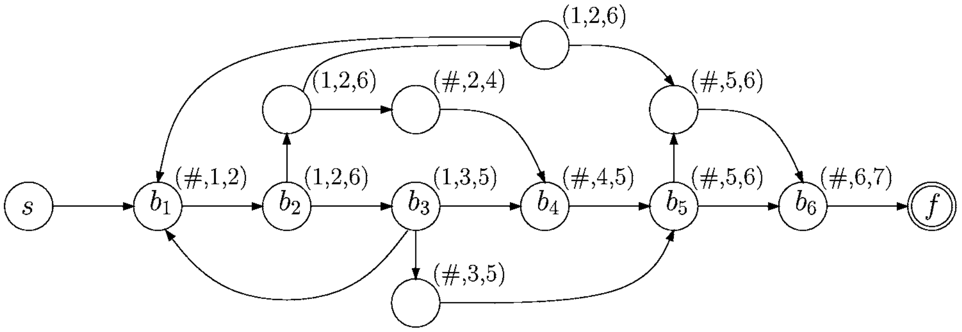


Figure 6: An example of DFS that computes (**min**, **anc**, **max**), for each state in G, for a given $CBS = (s, b_1, b_2, b_3, b_4, b_5, b_6, f)$, where # denotes the null index.

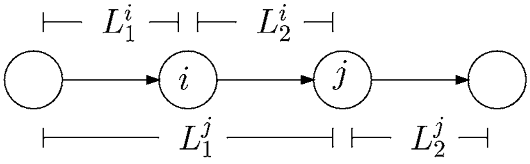
If a state $b_i \in CBS$ does not have any out-transitions except a transition to $b_{i+1} \in CBS$ (for example, b_6 in Fig. 6), then b_i has $(\#, i, i + 1)$ when DFS is completed, where # denotes the null index. Once we have completed DFS and computed (**min**, **anc**, **max**) for all states in G, we remove states from CBS that violate the first three requirements to be bridge states. Assume $b_i \in CBS$ has (h, i, j), where $h < i$ and $i < j$. First, we remove $b_h, b_{h+1}, \ldots, b_i$ from CBS since there is a path from b_i to b_h and, therefore, these states are in a cycle in A. If h is #, then we do not remove any states. Second, we remove $b_{i+1}, b_{i+2}, \ldots, b_{j-1}$ from CBS since there is a path from b_i to b_j; that is, there is another simple path from b_i to f. Finally, we remove s and f from CBS. For example, we have $\{b_6\}$ after removing states that violate the first three requirements from CBS in Fig. 6. This algorithm gives the following result.

Lemma 5 *We can compute a set of candidate bridge states that satisfy the first three requirements in Definition 3 for a given automaton* $A = (Q, \Sigma, \delta, s, f)$ *in* $O(|Q| + |\delta|)$ *worst-case time using DFS.*

Given a set of candidate bridge states CBS computed from a given minimal DFA A for $L(A)$, we check for each state $b_i \in CBS$ whether or not two subautomata A_1 and A_2 that are partitioned at b_i are infix-free using IF. If both A_1 and A_2 are infix-free, then L is not prime and we decompose L into $L(A_1) \cdot L(A_2)$ and continue to check and decompose for each A_1 and A_2 respectively using $CBS \setminus \{b_i\}$.

Lemma 6 *If a candidate state* $b_i \in CBS$ *is not a bridge state for a given minimal DFA* A*, then* b_i *cannot be a bridge state in a decomposed subautomaton after the decomposition at a bridge state* b_j*,* $i \neq j$*.*

Proof. Assume that b_i is not a bridge state in A but it becomes a bridge state in a subautomaton, say A_1, after decomposing A into two subautomata at $b_j \in CBS$, where $i < j$.



By the assumption, L_1^j and L_2^j are infix-free. Since b_i is a bridge state in A_1, L_1^i and L_2^i should be infix-free. However, if L_2^i is infix-free, then $L_2^i \cdot L_2^j$ must be infix-free since the catenation of infix-free regular languages is closed according to Theorem 7. It implies that $L(A) = L_1^i \cdot L_2^i L_2^j$ and, thus, b_i is a bridge state of A — a contradiction. Therefore, if b_i is not a bridge state in A, then b_i cannot be a bridge state in a decomposed subautomaton. □

Lemma 6 shows that once a candidate state in CBS is not a bridge state, then we do not need to consider the state as a candidate anymore even after a decomposition at some bridge state.

Theorem 9 *Given a minimal DFA* $A = (Q, \Sigma, \delta, s, f)$ *for an infix-free regular language* $L(A)$*, we can determine primality of* $L(A)$ *in* $O(m^3)$ *worst-case time and compute a prime decomposition of* $L(A)$ *in* $O(m^3)$ *worst-case time, where* m *is the number of states in* A*.*

Proof. Since there can be at most m candidate bridge states CBS after DFS and it takes $O(m^2)$ time for each candidate state to determine whether or not $L(A_1)$ and $L(A_2)$ are infix-free, the total running time for determining primality of $L(A)$ is $O(m) \times O(m^2) = O(m^3)$ in the worst-case. If a state $b_i \in CBS$ is not a bridge state, then we remove b_i from CBS since it can never be a bridge state by Lemma 6. Furthermore, once we find a bridge state b_j, then we partition A into A_1 and A_2 at b_j and repeat the procedure for $L(A_1)$ and $L(A_2)$, respectively, using the remaining candidate states in CBS. Since each candidate state in CBS can contribute a decomposition at most once, it takes $O(m^3)$ worst-case time to compute an infix-free decomposition for $L(A)$. □

Note that a bridge state $b_i \in CBS$ of a minimal DFA A can turn out not to be a bridge state after a decomposition at some other bridge state b_j of A. Fig. 7 illustrates this situation.

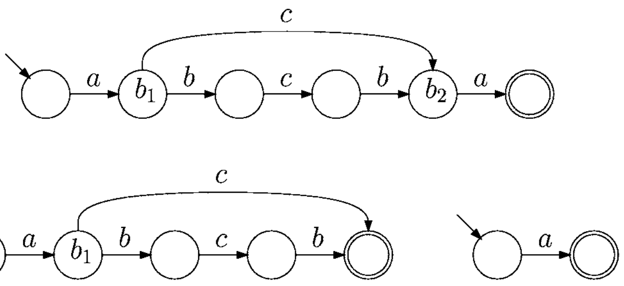


Figure 7: States b_1 and b_2 are bridge states for a given minimal DFA A. However, once we decompose A at b_2, then b_1 is no longer a bridge state anymore in A_1. Similarly, if we decompose A at b_1, then b_2 is not a bridge state in A_2.

Czyzowicz et al. [5] demonstrated that the prime prefix-free decomposition for a prefix-free regular language is unique. However, it turns out that the prime infix-free decomposition for an infix-free regular language is not unique. Example 1 gives an example of non-uniqueness.

Example 1 *The following is an example of non-uniqueness for the infix-free decomposition.*

$$L(a(bcb+c)a) = \begin{cases} L_1(a(bcb+c)) \cdot L_2(a). \\ L_2(a) \cdot L_3((bcb+c)a). \end{cases}$$

The language L is infix-free but not prime and it has two different prime decompositions, where L_1, L_2 and L_3 are prime infix-free languages.

5. Conclusions

We have investigated infix-free regular languages and their prime decomposition. We have designed algorithms to determine whether or not a given regular language L is infix-free based on state-pair graphs, where L is either specified by an NFA or given by a regular expression. It turns out that state-pair graphs are an appropriate tool for the decision problems of various codes in regular languages. Furthermore, we have provided an algorithm to determine the primality for a given minimal DFA of an infix-free regular language and compute a prime infix-free decomposition in $O(m^3)$ time, where m is the number of states in the minimal DFA. In addition, we have shown that the prime infix-free decomposition is not unique.

We conclude this paper by mentioning two interesting open problems in the literature.

1. Is it decidable whether or not a given context-free language is p-infix-free [13]?
2. Is it NP-complete to determine whether or not a given (finite) language has a decomposition [16]?

Acknowledgments

We wish to thank the referee for the careful reading of the paper and many valuable suggestions. As usual, however, we alone are responsible for any remaining sins of omission and commission.

References

1. M.-P. Béal, M. Crochemore, F. Mignosi, A. Restivo, and M. Sciortino. Computing forbidden words of regular languages. *Fundamenta Informaticae*, 56(1-2):121–135, 2003.
2. C. L. A. Clarke and G. V. Cormack. On the use of regular expressions for searching text. *ACM Transactions on Programming Languages and Systems*, 19(3):413–426, 1997.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.
4. M. Crochemore, F. Mignosi, and A. Restivo. Automata and forbidden words. *Information Processing Letters*, 67(3):111–117, 1998.
5. J. Czyzowicz, W. Fraczak, A. Pelc, and W. Rytter. Linear-time prime decomposition of regular prefix codes. *International Journal of Foundations of Computer Science*, 14:1019–1032, 2003.
6. Y.-S. Han, Y. Wang, and D. Wood. Prefix-free regular-expression matching. In *Proceedings of CPM'05*, 298–309. Springer-Verlag, 2005. Lecture Notes in Computer Science 3537.
7. 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.
8. Y.-S. Han and D. Wood. Outfix-free regular languages and prime outfix-free decomposition. To appear in Proceedings of ICTAC'05, 2005.
9. J. Hopcroft and J. Ullman. *Formal Languages and Their Relationship to Automata.* Addison-Wesley, Reading, MA, 1969.
10. M. Ito, H. Jürgensen, H.-J. Shyr, and G. Thierrin. N-prefix-suffix languages. *International Journal of Computer Mathematics*, 30:37–56, 1989.
11. M. Ito, H. Jürgensen, H.-J. Shyr, and G. Thierrin. Outfix and infix codes and related classes of languages. *Journal of Computer and System Sciences*, 43:484–508, 1991.

12. H. Jürgensen. Infix codes. In *Proceedings of Hungarian Computer Science Conference*, 25–29, 1984.
13. 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.
14. D. Y. Long, J. Ma, and D. Zhou. Structure of 3-infix-outfix maximal codes. *Theoretical Computer Science*, 188(1-2):231–240, 1997.
15. A. Mateescu, A. Salomaa, and S. Yu. On the decomposition of finite languages. Technical Report 222, TUCS, 1998.
16. A. Mateescu, A. Salomaa, and S. Yu. Factorizations of languages and commutativity conditions. *Acta Cybernetica*, 15(3):339–351, 2002.
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.