Outfix-Free Regular Languages and Prime Outfix-Free Decomposition*

Yo-Sub Han and Derick Wood

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

Abstract. A string x is an outfix of a string y if there is a string w such that $x_1wx_2 = y$, where $x = x_1x_2$ and a set X of strings is outfix-free if no string in X is an outfix of any other string in X. We examine the outfix-free regular languages. Based on the properties of outfix strings, we develop a polynomial-time algorithm that determines the outfix-freeness of regular languages. We consider two cases: A language is given as a set of strings and a language is given by an acyclic deterministic finite-state automaton. Furthermore, we investigate the prime outfix-free decomposition of outfix-free regular languages and design a linear-time prime outfix-free decomposition algorithm for outfix-free regular languages. We demonstrate the uniqueness of prime outfix-free decomposition.

1 Introduction

Codes play a crucial role in many areas such as information processing, date compression, cryptography, information transmission and so on [14]. They are categorized with respect to different conditions (for example, *prefix-free, suffix-free, infix-free* or *outfix-free*) according to the applications [11,12,13,15]. Since a code is a set of strings, it 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 language, which is a proper subfamily of regular languages.

Based on such subfamilies of regular language, researchers have investigated properties of these languages as well as their decomposition problems. A decomposition of a language L is a catenation of several languages $L_1, L_2, \ldots, L_k$ such that $L = L_1L_2 \cdots L_k$ and $k \geq 2$. If L cannot be further decomposed except for $L \cdot \{\lambda\}$ or $\{\lambda\} \cdot L$, where λ is the null-string, we say that L a *prime* language.

Czyzowicz et al. [5] studied prefix-free regular languages and the prime prefix-free decomposition problem. They showed that the prime prefix-free decomposition of a prefix-free language is unique and demonstrated the importance of prime prefix-free decomposition in practice. Prefix-free regular languages are often used in the literature: to define the determinism of generalized automata [6] and of expression automata [10], and to represent a pattern set [9].

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

D.V. Hung and M. Wirsing (Eds.): ICTAC 2005, LNCS 3722, pp. 96–109, 2005.

Recently, Han et al. [8] studied infix-free regular languages and developed an algorithm to determine whether or not a given regular expression defines an infix-free regular language. They also designed an algorithm for computing the prime infix-free decomposition of infix-free regular languages and showed that the prime infix-free decomposition is not unique. Infix-free regular languages give rise to faster regular-expression text matching [2]. Infix-free languages are also used to compute forbidden words [1,4].

As a continuation of our investigations of subfamilies of regular languages, it is natural to examine outfix-free regular languages and the prime outfix-free decomposition problem. Note that Ito and his co-researchers [12] showed that an outfix-free regular language is finite and Han et al. [7] demonstrated that the family of outfix-free regular languages is a proper subset of the family of simple-regular languages. On the other hand, there was no known efficient algorithm to determine whether or not a given finite set of strings is outfix-free apart from using brute force. Furthermore, the decomposition of a finite set of strings is not unique and the computation of the decomposition is believed to be NP-complete [17]. Therefore, our goal is to develop an efficient algorithm for determining outfix-freeness of a given finite language and to investigate the prime outfix-free decomposition and its uniqueness.

We define some basic notions in Section 2 and propose an efficient algorithm to determine outfix-freeness in Section 3. Then, in Section 4, we show that an outfix-free regular language has a unique prime outfix-free decomposition and the unique decomposition can be computed in linear time in the size of the given finite-state automaton. We suggest some open problems and conclude this paper in Section 5.

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 a string $x = x_1 \cdots x_n$, $|x|$ is the number of characters in x and $x(i,j) = x_i x_{i+1} \cdots x_j$ is the substring of x from position i to position j, where $i \leq j$. 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 $x_1 w x_2 = y$, 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. Given a string x, let x^R be the reversal of x, in which case $X^R = \{x^R \mid x \in X\}$.

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 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 Outfix-Free Regular Languages

We first define outfix-free regular expressions and languages, and then present an algorithm to determine whether or not a given language is outfix-free. Since prefix-free, suffix-free, infix-free and outfix-free languages are related to each other, we define all of them and show their relationships.

Definition 1. *A language L is*

- prefix-free *if, for all distinct strings $x, y \in \Sigma^*$, $x \in L$ and $y \in L$ imply that x and y are not prefixes of each other.*
- suffix-free *if, for all distinct strings $x, y \in \Sigma^*$, $x \in L$ and $y \in L$ imply that x and y are not suffixes of each other.*
- bifix-free *if L is prefix-free and suffix-free.*
- infix-free *if, for all distinct strings $x, y \in \Sigma^*$, $x \in L$ and $y \in L$ imply that x and y are not substrings of each other.*
- outfix-free *if, for all distinct strings $x, y, z \in \Sigma^*$, $xz \in L$ and $xyz \in L$ imply $y = \lambda$.*
- hyper *if L is infix-free and outfix-free.*

For further details and definitions, refer to Ito et al. [12] or Shyr [18].

We say that a regular expression E is outfix-free if $L(E)$ is outfix-free. The language defined by such an outfix-free regular expression is called an *outfix-free regular language*. In a similar way, we can define prefix-free, suffix-free and infix-free regular expressions and languages.

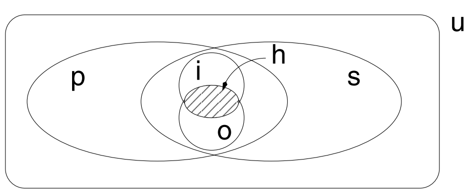

Fig. 1. A diagram to show inclusions of families of languages, where p,s,i,o and h denote prefix-free, suffix-free, infix-free , outfix-free and hyper families, respectively, and u denotes Σ^*. Note that the outfix-free family is a proper subset of the prefix-free and suffix-free families and the hyper family is the common intersection between the infix-free family and the outfix-free family.

Let $A = (Q, \Sigma, \delta, s, F)$ denote a deterministic finite-state automaton (DFA) for L. Han and Wood [10] showed that if A is non-exiting, then L is prefix-free. Han et al. [8] proposed an algorithm to determine whether or not a given regular expression E is infix-free in $O(|E|^2)$ worst-case time. This algorithm can also solve the prefix-free and suffix-free cases as well. Therefore, it is natural to design an algorithm to determine whether or not a given regular language is outfix-free. Since an outfix-free regular language L is finite [12,14], the problem is decidable by comparing all pairs of strings in L, although it is certainly undesirable to do so.

3.1 Prefix-Freeness

Since the family of outfix-free regular languages is a proper subfamily of prefix-free regular languages as shown in Fig. 1, we consider prefix-freeness of a finite language first.

Given a finite set of strings $W = \{w_1, w_2, \ldots, w_n\}$, where n is the number of strings in W, we construct a trie T for W. A trie is an ordered tree data structure that is used to store a set of strings and each edge in the tree has a single character label. For details on tries, refer to data structure textbooks [3,19]. Assume that w_i is a prefix of w_j, where $i \neq j$; it implies that $|w_i| < |w_j|$. Then, w_i and w_j must have the common path in T from the root to the ith node q that spells out w_i. Therefore, if we reach q while constructing the path for w_j in T, we recognize that w_i is a prefix of w_j. Let us consider the case when we construct a path for w_j first and, then, construct a path for w_i in T. The path for w_i ends at the $|w_i|$th node q that already has a child node for the path for w_j. Therefore, we know that w_i is a prefix of some other string. Note that we can construct a trie for W in $O(|w_1| + |w_2| + \cdots |w_n|)$ time, which is linear in the size of W.

Lemma 1. *Given a finite set W of strings, we can determine whether or not W is prefix-free in linear time in the size of W by constructing a trie for W. We can also determine suffix-freeness of W in the same runtime by constructing a trie for W^R.*

3.2 Outfix-Freeness

We now consider outfix-freeness. Assume that we have two distinct strings w_1 and w_2 and w_2 is an outfix of w_1. It implies that $w_1 = xyz$ for some strings x, y and z such that $w_2 = xz$ and $y \neq \lambda$. Moreover, w_1 and w_2 have the common prefix x and the common suffix z. Fig. 2 illustrates it.

Based on these observations, we determine whether or not one string w_1 is an outfix of another string w_2 for two given strings w_1 and w_2, where $|w_1| \geq |w_2|$. We compare two characters, one from w_1 and the other from w_2, from left to right (from 1 to $|w_2|$) until two compared characters are different; say the ith characters are different. If we completely read w_2, then we recognize that w_2 is a prefix of w_1 and, therefore, w_2 is an outfix of w_1. We repeat these character-by-character comparisons from right to left (from $|w_2|$ to 1) until we have two

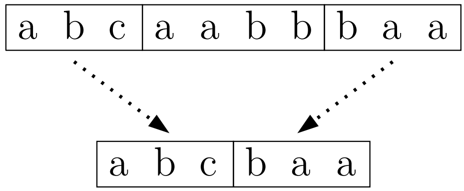

Fig. 2. A graphical illustration of an outfix string; *abcbaa* is an outfix of *abcaabbbaa*

different characters. Assume that the jth characters are different. If $i > j$, then w_2 is an outfix of w_1. Otherwise, w_2 is not an outfix of w_1. For example, $i = 4$ and $j = 3$ in Fig. 2.

Lemma 2. *Given two strings w_1 and w_2, where $|w_1| \geq |w_2|$, w_2 is an outfix of w_1 if and only if there is a position i such that $w_2(1, i)$ is a prefix of w_1 and $w_2(i + 1, |w_2|)$ is a suffix of w_1.*

Let us consider the trie T for w_1 and w_2. Since w_1 and w_2 have the common prefix, both strings share the common path from the root to a node q of height i that spells out $w_2(1, i)$. Moreover, the path for $w_2(i+1, |w_2|)$ in T is a suffix-path for $w_1(i + 1, |w_1|)$ in T. For example, in Fig. 3, the path for x is the common prefix-path and the path for z is the common suffix-path. Thus, if a given finite set W of strings is not outfix-free, then there is such a pair of strings. Since a node $q \in T$ gives the common prefix for all strings that pass through q, we only need to check whether some path from q to a leaf is a suffix-path for some other path from q to another leaf.

Let $T(q)$ be the subtree of T rooted at $q \in T$. Then, we can determine whether or not a path from q is a suffix-path for another path from q in $T(q)$ by determining the suffix-freeness of all paths from q to a leaf in $T(q)$ based on the same algorithm for Lemma 1. The running time is linear in the the size of $T(q)$.

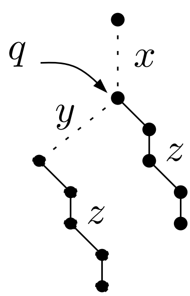

Fig. 3. An example of a trie for strings $w_1 = xyz$ and $w_2 = xz$. Note that both paths end with the same subpath sequence in the trie since w_1 and w_2 have the common suffix z.

3.3 Complexity of Outfix-Freeness

The subfunction is_prefix-free(T) in Fig. 4 determines whether or not the set of strings represented by a given trie T is prefix-free. Note that is_prefix-free(T) runs in $O(|T|)$ time, where $|T|$ is the number of nodes in T.

Given a finite set $W = \{w_1, w_2, \ldots, w_n\}$ of strings, we can construct a trie T in $O(\sum_{i=1}^{n} |w_i|)$ time and space, which is linear in the size of W, where $n \geq 1$. Prefix-freeness and suffix-freeness can be verified in linear time. Thus, the total running time for the algorithm Outfix-freeness (OFF) in Fig. 4 is

$$O(|T|) + \sum_{q \in T} |T(q)|,$$

where q is a node that has more than one child. In the worst-case, we have to examine all nodes in T; for example, T is a complete tree, where each internal node has the same number of children. To compute the size of $\sum |T(q)|$, let us consider a string $w_i \in W$ that makes a path P from the root to a leaf in T. If a node $q \in T$ of height j in path P has more than one child, then the suffix $w_i(j+1, |w_i|)$ of w_i that starts from q is used in is_suffix-free($T(q)$) in OFF. In the worst-case, all suffixes of w_i can be used by is_suffix-free($T(q)$). Therefore, w_i contributes $O(|w_i|^2)$ to the total running time of OFF. Fig. 5 illustrates a worst-case example.

Therefore, the total time complexity is $O(|w_1|^2 + |w_2|^2 + \cdots + |w_n|^2)$ in the worse case. If the size of w_i is $O(k)$, for some k, then the running time is $O(k^2 n)$. On the other hand, the all-pairs comparison approach gives $O(kn^2)$ worst-case running time. Note that the size of each string in W is usually much smaller than the number of strings in W; namely, $k \ll n$.

Theorem 1. *Given a finite set $W = \{w_1, w_2, \ldots, w_n\}$ of strings, we can determine whether or not W is outfix-free in $O(\sum_i^n |w_i|^2)$ time using $O(\sum_i^n |w_i|)$ space in the worse-case.*

Outfix-freeness($W = \{w_1, w_2, \ldots, w_n\}$)

Construct a trie T for W

if (is_prefix-free(T) = no)
  then return no
if (is_suffix-free(T) = no)
  then return no

for each $q \in T$ that has more than one child
  if (is_suffix-free($T(q)$) = no)
    then return no

return yes

Fig. 4. An outfix-freeness checking algorithm for a given finite set of strings

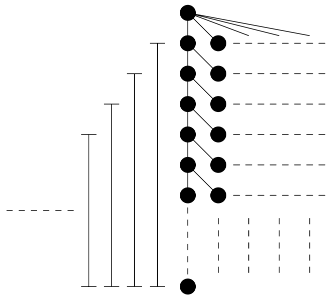

Fig. 5. All suffixes of a string w in T are used to determine the outfix-freeness by OFF. The size of the sum of all suffixes is $O(|w|^2)$.

Now we characterize the family of outfix-free (regular) languages in terms of closure properties.

Theorem 2. *The family of outfix-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 outfix-free whereas L_1 and L_2 are outfix-free. Then, there are two distinct strings s and $t \in L$, where t is an outfix of s. Namely, $s = xyz$, $t = xz$ and $y \neq \lambda$. Since s and t are catenation of two strings from L_1 and L_2, s and t can be partitioned into two parts; $s = s_1 s_2$ and $t = t_1 t_2$, where $s_i, t_i \in L_i$ for $i = 1, 2$. From the assumption that t is an outfix of s, s and t have the common prefix and the common suffix as shown in Fig. 6. If we decompose s and t into $s_1 s_2$ and $t_1 t_2$, then we have one of the following four cases:

1. s_1 is a prefix of t_1.
2. t_1 is a prefix of s_1.
3. s_2 is a suffix of t_2.
4. t_2 is a suffix of s_2.

Let us consider the first case as illustrated in Fig. 6. Since s_1 is a prefix of t_1 and $s_1, t_1 \in L_1$, L_1 is not outfix-free — a contradiction. We can use a similar argument for the other three cases. □

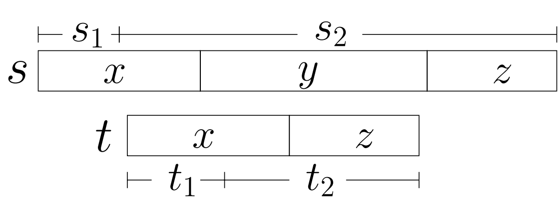

Fig. 6. The figure illustrates the first case in the proof of Theorem 2, where s_i and $t_i \in L_i$ for $i = 1, 2$. Since s_1 is a prefix of t_1, L_1 is not outfix-free.

3.4 Outfix-Freeness of Acyclic Deterministic Finite-State Automata

Acyclic deterministic finite-state automata (ADFAs) are a proper subfamily of DFAs that define finite languages. For example, a trie is an ADFA. Since ADFAs represent finite languages, they are often used to store a finite number of strings. Moreover, ADFAs require less space than tries. For instance, we use $O(|\Sigma|^5)$ space to store all strings of length 5 over Σ in a trie. On the other hand, we use 6 states with $5 \times |\Sigma|$ transitions in an ADFA. We consider outfix-freeness of a language given by an ADFA $A = (Q, \Sigma, \delta, s, f)$. Given A and a state $q \in Q$, we define the *right language* $L_{\overrightarrow{q}}$ to be the set of strings spelled out by paths from q to f.

Assume that two strings $w_1 = xyz$ and $w_2 = xz$ are accepted by A, where w_2 is an outfix of w_1. Note that w_1 and w_2 have the common prefix x and the common suffix z and there is a unique path from s to a state q that spells out x in A since A is deterministic. Then, yz and z are accepted by $A_{\overrightarrow{q}}$. It means that $L_{\overrightarrow{q}}$ is not suffix-free.

Lemma 3. *Given an ADFA $A = (Q, \Sigma, \delta, s, f)$, $L(A)$ is outfix-free if and only if $L_{\overrightarrow{q}}$ is suffix-free for any state $q \in Q$.*

Proof.
$\Longrightarrow$ Assume that $L_{\overrightarrow{q}}$ is not suffix-free. Then, there are two strings w_1 and w_2 in $L_{\overrightarrow{q}}$, where w_2 is a suffix of w_1. Since A has only useful states, there must be a path from s to q that spells out a string x. It implies that A accepts both xw_1 and xw_2, where xw_2 is an outfix of xw_1 — a contradiction. Therefore, if $L(A)$ is outfix-free, then $L_{\overrightarrow{q}}$ is suffix-free for any state $q \in Q$.

$\Longleftarrow$ Assume that $L(A)$ is not outfix-free. Then, there are two strings $w_1 = xyz$ and $w_2 = xz$ accepted by A, where w_2 is an outfix of w_1. There is a unique path from s to q that spells out x in A. Then, there are two distinct paths, one is for yz and the other is for z, from q since A accepts w_1 and w_2. It implies that $A_{\overrightarrow{q}}$ accepts yz and z and $L_{\overrightarrow{q}}$ is not suffix-free — a contradiction. Therefore, if $L_{\overrightarrow{q}}$ is suffix-free for any state $q \in Q$, then $L(A)$ is outfix-free. □

Recently, Han et al. [8] proposed algorithms to determine prefix-freeness, suffix-freeness, bifix-freeness and infix-freeness of a given a (nondeterministic) finite-state automaton $A = (Q, \Sigma, \delta, s, f)$ in $O(|Q|^2 + |\delta|^2)$ time. We use their algorithm to check suffix-freeness for each state. Given an ADFA $A = (Q, \Sigma, \delta, s, f)$ and a state $q \in Q$, the size of $A_{\overrightarrow{q}}$ is at most the size of A; namely, $|A_{\overrightarrow{q}}| \leq |A|$. Since it takes $O(|Q|^2 + |\delta|^2)$ time for each state to check suffix-freeness and there are $|Q|$ states, the total time complexity to determine outfix-freeness of A is $O(|Q|^3 + |Q||\delta|^2)$. Since a DFA has a constant number of out-transitions from a state, we obtain the following result.

Theorem 3. *Given an ADFA $A = (Q, \Sigma, \delta, s, f)$, we can determine outfix-freeness of $L(A)$ in $O(|Q|^3)$ worst-case time.*

Furthermore, we determine infix-freeness of $L(A)$ after an outfix-freeness test. If $L(A)$ is infix-free and outfix-free, then $L(A)$ is hyper. Since the time complexity for the infix-freeness test is $O(|Q|^2)$ for A [8], we can determine hyperness of $L(A)$ in $O(|Q|^3)$ time as well.

Theorem 4. *Given an ADFA $A = (Q, \Sigma, \delta, s, f)$, we can determine hyperness of $L(A)$ in $O(|Q|^3)$ worst-case time.*

4 Prime Outfix-Free Regular Languages and Prime 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 two non-trivial languages. Then, the prime decomposition of L is to decompose L into $L_1 \cdot L_2 \cdot \ldots \cdot L_k$, where $L_1, L_2, \ldots, L_k$ are prime languages and $k \geq 1$ is a constant.

Mateescu et al. [16,17] showed that the primality of regular languages is decidable and the prime decomposition of a regular language is not unique even for finite languages. Furthermore, they pointed out that no star language L ($L = K^*$, for some K) can possess a prime decomposition. 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 the unique decomposition for L can be computed in $O(m)$ worst-case time, where m is the size of the minimal DFA for L. Recently, Han et al. [8] investigated the prime infix-free decomposition of infix-free regular languages and demonstrated that the prime infix-free decomposition is not unique.

We examine prime outfix-free regular languages and decomposition. Even though outfix-free regular languages are finite [12], the primality test for finite languages is believed to be NP-complete [17]. Thus, the decomposition problem for finite languages is not trivial at all. We design a linear-time algorithm to determine whether or not a given finite language L is prime outfix-free. We investigate prime outfix-free decompositions and uniqueness.

4.1 Prime Outfix-Free Regular Languages

Definition 2. *A regular language L is a* prime *outfix-free language if $L \neq L_1 \cdot L_2$ for any outfix-free regular languages L_1 and L_2.*

From now on, when we say prime, we mean prime outfix-free. Since we are dealing with outfix-free regular languages, there are no back-edges in finite-state automata for such languages. Furthermore, these finite-state automata are always non-exiting and non-returning. Note that if a finite-state automaton is non-exiting and has several final states, then all final states are equivalent and, therefore, are merged into a single final state.

Definition 3. *We define a state b in a DFA A to be a* bridge state *if the following two 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. Therefore, we can partition A at b into two subautomata A_1 and A_2.*

Given a DFA $A = (Q, \Sigma, \delta, s, f)$ and a bridge state $b \in Q$, where $L(A)$ is outfix-free, 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 to 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. 7 illustrates a partition at a bridge state.

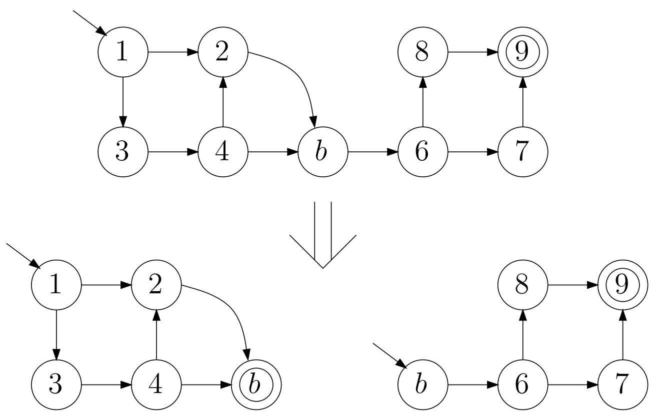

Fig. 7. An example of partitioning of an automaton at a bridge state b

It is easy to verify that $L(A) = L(A_1) \cdot L(A_2)$ from the second requirement in Definition 3.

Lemma 4. *If a minimal DFA A has a bridge state, where $L(A)$ is outfix-free, then $L(A)$ is not prime.*

Proof. Since A has a bridge state b, we can partition A into A_1 and A_2 at b. We establish that $L(A_1)$ and $L(A_2)$ are outfix-free and, therefore, $L(A)$ is not prime. Assume that $L(A_1)$ is not outfix-free. Then, there are two distinct strings u and v accepted by A_1, where v is an outfix of u; namely, $u = xyz$ and $v = xz$ for some strings x, y and z. Let w be a string from $L(A_2)$. Since $L(A) = L(A_1) \cdot L(A_2)$, both $uw = xyzw$ and $vw = xzw$ are in $L(A)$. It contradicts the assumption that $L(A)$ is outfix-free. Therefore, if $L(A)$ is outfix-free, then $L(A_1)$ should be outfix-free as well. With a similar argument, we can show that $L(A_2)$ should be outfix-free. Hence, if A has a bridge state, then $L(A)$ can be decomposed as $L(A_1) \cdot L(A_2)$, where $L(A_1)$ and $L(A_2)$ are outfix-free, and, therefore, $L(A)$ is not prime. □

Lemma 5. *If a minimal DFA A does not have any bridge states and $L(A)$ is outfix-free, then $L(A)$ is prime.*

Proof. Assume that L is not prime. Then, L can be decomposed as $L_1 \cdot L_2$, where L_1 and L_2 are outfix-free. 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 outfix-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 are only one start state and one final state for each of them. 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 b. Then, the catenated automaton is the minimal DFA for $L(A_1) \cdot L(A_2) = L$ and has a bridge state b — a contradiction. □

We can rephrase Lemma 4 as follows: If L is prime, then its minimal DFA does not have any bridge states. Then, from Lemmas 4 and 5, we obtain the following result.

Theorem 5. *An outfix-free regular language L is prime if and only if the minimal DFA for L does not have any bridge states.*

Lemma 4 shows that if a minimal DFA A for an outfix-free regular language L has a bridge state, then we can decompose L into a catenation of two outfix-free regular languages using bridge states. In addition, if we have a set B of bridge states for A and decompose A at b, then $B \setminus \{b\}$ is the set of bridge states for the resulting two automata after the decomposition.

Theorem 6. *Let A be a minimal DFA for an outfix-free regular language that has k bridge states. Then, $L(A)$ can be decomposed into $k+1$ prime outfix-free regular languages, namely, $L(A) = L_1 L_2 \cdots L_{k+1}$ and $L_1, L_2, \ldots, L_{k+1}$ are prime.*

Proof. Let $(b_1, b_2, \ldots, b_k)$ be the sequence of bridge states from s to f in A. We prove the statement by induction on k. It is sufficient to show that $L(A) = L'L''$ such that L' is accepted by a DFA A' with $k-1$ bridge states and L'' is a prime outfix-free regular language.

We partition A into two subautomata A' and A'' at b_k. Note that $L(A')$ and $L(A'')$ are outfix-free languages by the proof of Lemma 4. Since A'' has no bridge states, $L'' = L(A'')$ is prime by Theorem 5. By the definition of bridge states, all paths must pass through $(b_1, b_2, \ldots, b_{k-1})$ in A' and, therefore, A' has $k-1$ bridge states. Thus, if A has k bridge states, then $L(A)$ can be decomposed into $k+1$ prime outfix-free regular languages. □

Note that Theorem 6 guarantees the uniqueness of prime outfix-free decomposition. Furthermore, finding the prime decomposition of an outfix-free regular language is equivalent to identifying bridge states of its minimal DFA by Theorems 5 and 6.

We now show how to compute a set of bridge states defined in Definition 3 from a given minimal DFA A in $O(m)$ time, where m is the size of A. 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 6. *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. □

Assume that we have a simple path $P_{s,f}$ from s to f in $G = (V, E)$, which can be computed in $O(|V| + |E|)$ worst-case time. All states on $P_{s,f}$ form a set of candidate bridge states (CBS); namely, $CBS = (s, b_1, b_2, \ldots, b_k, f)$.

We use DFS to explore G from s. We visit all states in CBS first. While exploring G, we maintain the following two 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.
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 **max** value of a state q means that there is a path from q to $b_{\mathbf{max}}$. If b_i has a **max** value and **max** $\neq i + 1$, then it means that there is another simple path from b_i to $b_{\mathbf{max}}$ without passing through b_{i+1}.

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 **max** of q as follows:

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

Fig. 8 provides an example of DFS after updating (**anc**, **max**) for all states in G.

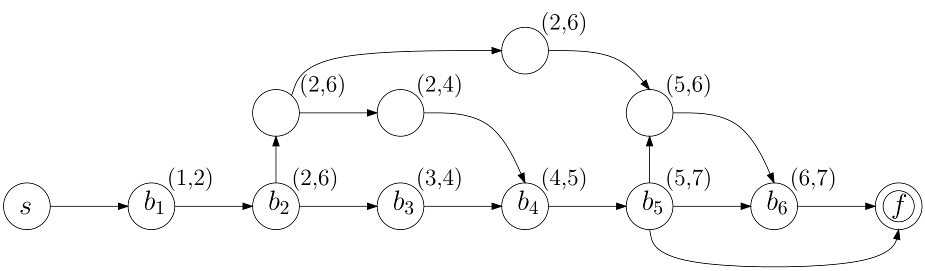

Fig. 8. An example of DFS that computes (**anc**, **max**), for each state in G, for a given $CBS = (s, b_1, b_2, b_3, b_4, b_5, b_6, f)$

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. 8), then b_i has $(i, i+1)$ when DFS is completed. Once we have completed DFS and computed (**anc**, **max**) for all states in G, we remove states from CBS that violate the requirements to be bridge states. Assume $b_i \in CBS$ has (i, j), where $i < j$. We remove $b_{i+1}, b_{i+2}, \ldots, b_{j-1}$ from CBS since that there is a path from b_i to b_j; that is, there is another simple path from b_i to f. Then, we remove s and f from CBS. For example, we have $\{b_1, b_2\}$ after removing states that violate the requirements from CBS in Fig. 8. This algorithm gives the following result.

Theorem 7. *Given a minimal DFA A for an outfix-free regular language:*

1. *We can determine the primality of $L(A)$ in $O(m)$ time,*
2. *We can compute the unique outfix-free decomposition of $L(A)$ in $O(m)$ time if $L(A)$ is not prime,*

where m is the size of A.

5 Conclusions

We have investigated the outfix-free regular languages. First, we suggested an algorithm to verify whether or not a given set $W = \{w_1, w_2, \ldots, w_n\}$ of strings is outfix-free. We then established that the verification takes $O(\sum_{i=1}^{n} |w_i|^2)$ worst-case time, where n is the number of strings in W. We also considered the case when a language L is given by an ADFA. Moreover, we have extended the algorithm to determine hyperness of L by checking infix-freeness using the algorithm of Han et al. [8].

We have demonstrated that an outfix-free regular language L has a unique outfix-free decomposition and the unique decomposition can be computed in $O(m)$ time, where m is the size of the minimal DFA for L.

As we have observed, outfix-free regular languages are finite sets. However, this observation does not hold for the context-free languages. For example, the non-regular language, $\{w \mid w = a^i c b^i, i \geq 1\}$ is context-free, outfix-free and infinite. The decidability of outfix-freeness for context-free languages is open as is the prime decomposition problem. Moreover, there are non-context-free languages that are outfix-free; for example, $\{w \mid w = a^i b^i c^i, i \geq 1\}$. Thus, it is reasonable to investigate the properties and the structure of the family of outfix-free languages.

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. D. Giammarresi and R. Montalbano. Deterministic generalized automata. *Theoretical Computer Science*, 215:191–208, 1999.
7. Y.-S. Han, G. Trippen, and D. Wood. Simple-regular expressions and languages. In *Proceedings of DCFS'05*, 146–157, 2005.
8. Y.-S. Han, Y. Wang, and D. Wood. Infix-free regular expressions and languages. To appear in International Journal of Foundations of Computer Science, 2005.
9. 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.
10. Y.-S. Han and D. Wood. The generalization of generalized automata: Expression automata. *International Journal of Foundations of Computer Science*, 16(3):499–510, 2005.
11. M. Ito, H. Jürgensen, H.-J. Shyr, and G. Thierrin. N-prefix-suffix languages. *International Journal of Computer Mathematics*, 30:37–56, 1989.
12. 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.
13. H. Jürgensen. Infix codes. In *Proceedings of Hungarian Computer Science Conference*, 25–29, 1984.
14. 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.
15. D. Y. Long, J. Ma, and D. Zhou. Structure of 3-infix-outfix maximal codes. *Theoretical Computer Science*, 188(1-2):231–240, 1997.
16. A. Mateescu, A. Salomaa, and S. Yu. On the decomposition of finite languages. Technical Report 222, TUCS, 1998.
17. A. Mateescu, A. Salomaa, and S. Yu. Factorizations of languages and commutativity conditions. *Acta Cybernetica*, 15(3):339–351, 2002.
18. H.-J. Shyr. *Lecture Notes: Free Monoids and Languages*. Hon Min Book Company, Taichung, Taiwan R.O.C, 1991.
19. D. Wood. *Data structures, algorithms, and performance*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.