Fundamenta Informaticae 81 (2007) 441–457
IOS Press

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

Yo-Sub Han†‡
Intelligence and Interaction Research Center, Korea Institute of Science and Technology
P.O.BOX 131, Cheongryang, Seoul, Korea
emmous@kist.re.kr

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

Abstract. A string x is an outfix of a string y if there is a string w such that $x_1wx_2 = y$ and $x = x_1x_2$. A set X of strings is outfix-free if no string in X is an outfix of any other string in X. Based on the properties of outfix strings, we develop a polynomial-time algorithm that determines outfix-freeness of regular languages. Note that outfix-free regular languages are always finite. We consider two cases: 1) a language is given as a finite set of strings and 2) a language is given by a finite-state automaton. Furthermore, we investigate the prime outfix-free decomposition of outfix-free regular languages and design a linear-time algorithm that computes prime outfix-free decomposition for outfix-free regular languages. We also demonstrate the uniqueness of prime outfix-free decomposition.

Keywords: regular languages, outfix-freeness, prime decomposition

*A preliminary version of this paper appeared in the *Proceedings of the Second International Colloquium on Theoretical Aspects of Computing*, ICTAC'05, 2005 [13]. Part of this research research was carried out while Han was in HKUST.
†Address for correspondence: Intelligence and Interaction Research Center, Korea Institute of Science and Technology, P.O.BOX 131, Cheongryang, Seoul, Korea
‡Han was supported by the KIST Tangible Space Initiative Grant 2E20050.
§Wood was supported by the Research Grants Council of Hong Kong Competitive Earmarked Research Grant HKUST6197/01E.

1. Introduction

Codes play a crucial role in many areas such as information processing, data compression, cryptography and information transmission [18]. They are categorized with respect to different conditions (for example, *prefix-free*, *suffix-free*, *infix-free* or *outfix-free*) according to the applications [7, 8, 15, 16, 17, 20, 23]. 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 languages, which is a proper subfamily of regular languages.

Based on such subfamilies of regular languages, researchers have investigated properties of these languages and their decomposition problems. A decomposition of a language L is a catenation of several languages $L_1, L_2, \ldots, L_k$ such that $L = L_1 L_2 \cdots L_k$ and $k \geq 2$. We call $L_1, L_2, \ldots, L_k$ *factors* of L. If L cannot be further decomposed except for $L \cdot \{\lambda\}$ or $\{\lambda\} \cdot L$, 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 [12], and to represent a pattern set [10].

Han et al. [11] 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. Note that the prime prefix-free decomposition requires each factor language to be prefix-free whereas the prime infix-free decomposition requires each factor language to be infix-free. 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 coresearchers [16] showed that an outfix-free regular language is finite and Han et al. [9] demonstrated that the family of outfix-free regular languages is a proper subset of the family of simple-regular languages. (A simple-regular language is a set of strings spelled out by simple paths in a given finite-state automaton.) 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 [22]. 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 two efficient algorithms that determine outfix-freeness in Section 3. The first algorithm takes a set of strings as input and determines outfix-freeness of the set by constructing tries. The second algorithm takes a (nondeterministic) finite-state automaton (FA) as input. Note that, given an FA A, we can determine outfix-freeness of $L(A)$ by checking whether or not $(L(A) \leftarrow \Sigma^+) \cap L(A)$ is empty, where $\leftarrow$ denotes the sequential insertion operation [19]. However, this approach does not consider the structural properties of A whereas our second algorithm is based on the structural properties of A. Moreover, our algorithm is much easier to understand and implement. 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 deterministic finite-state automaton (DFA). 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 symbol $\emptyset$ denotes the empty language and the symbol 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, X is *outfix-free* if no string in X is an outfix of any other string in X. Given a string x in a set X, let x^R be the reversal of x, in which case $X^R = \{x^R \mid x \in X\}$.

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, $\delta : Q \times \Sigma \to 2^Q$ is a transition function, $s \in Q$ is the start state and $F \subseteq Q$ is a set of final states. If F consists of a single state f, we use f instead of $\{f\}$ for simplicity. 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 $\delta(p, a) = q$, 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. 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 all final states of A do 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 and, therefore, there is no sink state and A may not be complete in general. 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. 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.

Given an FA $A = (Q, \Sigma, \delta, s, F)$ and a state $q \in Q$, we define the *right FA* $A_{\overrightarrow{q}}$ to be $(Q, \Sigma, \delta, q, F)$; namely, we make q to be the start state. Then, the *right language* $L_{\overrightarrow{q}}$ of q is the set of strings accepted by $A_{\overrightarrow{q}}$.

For complete background knowledge in automata theory, the reader may refer to textbooks [14, 25].

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 regular 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.

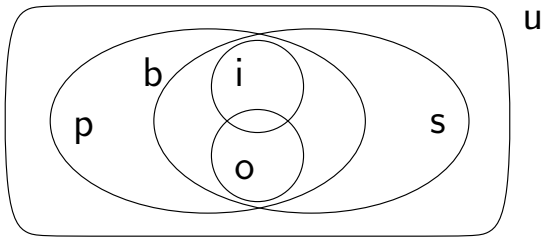

Figure 1. The diagram shows inclusions of families of languages, where p,s,b,i and o denote prefix-free, suffix-free, bifix-free, infix-free, and outfix-free families, respectively, and u denotes the set of all languages over Σ. Note that the outfix-free family is a proper subset of the prefix-free and suffix-free families.

Definition 3.1. 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 $y = \lambda$.

For further details and definitions, refer to Ito et al. [16] or Shyr [24].

We say that a regular expression E is outfix-free if $L(E)$ is outfix-free. The language defined by 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.

Let $A = (Q, \Sigma, \delta, s, F)$ be a DFA for a regular language L. Han and Wood [12] showed that A is non-exiting if and only if L is prefix-free. Moreover, Han et al. [11] proposed an algorithm that determines 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 [16, 18], the problem is decidable by first checking that L is finite and, then, 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. If a node q in T has an end-marker, then it means that the corresponding string from the root to q is in W. Fig. 2 gives an example. For details on tries, refer to data structure textbooks [3, 26].

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 3.1. 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.

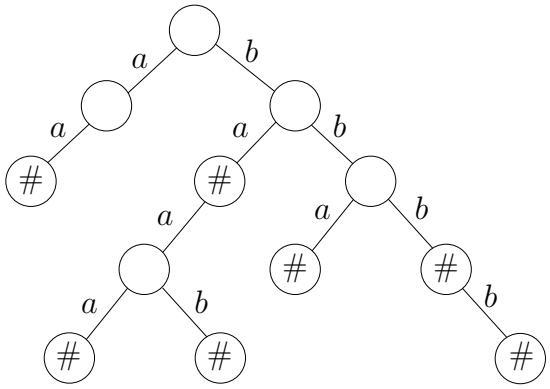

Figure 2. The trie for $W = \{aa, ba, baaa, baab, bba, bbb, bbbb\}$, where # denotes the end-marker for the corresponding string.

Proof:
We construct a trie T for W in linear time and check if any internal node has an end-marker while traversing T in liner time. If we identify any internal node with an end-marker, then W is not prefix-free. Otherwise, W is prefix-free. □

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. This 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. 3 illustrates such w_1 and w_2.

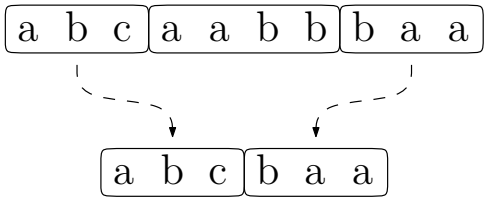

Figure 3. A graphical illustration of an outfix string; $abcbaa$ is an outfix of $abcaabbbaa$.

Based on the observations, we determine whether or not one string w_1 is an outfix of another string w_2, where $|w_1| \geq |w_2|$, as follows: 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 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. 3.

Lemma 3.2. 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.

Proof:
The proof is straightforward from the definition of outfix. □

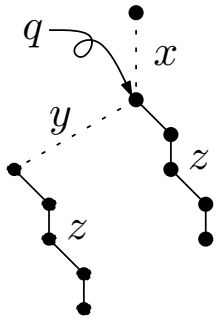

Figure 4. An example of a trie for strings $w_1 = xyz$ and $w_2 = xz$. Note that both paths end with the same subpath sequence in the trie since w_1 and w_2 have the common suffix z.

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. 4, 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 3.1. The running time is linear in the the size of $T(q)$.

3.3. Complexity of outfix-freeness

The subfunction is_prefix-free(T) in Fig. 5 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 by Lemma 3.1. Thus, the total running time for the algorithm Outfix-freeness (OFF) in Fig. 5 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

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

Figure 5. An outfix-freeness checking algorithm for a given finite set of strings.

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. 6 illustrates a worst-case example.

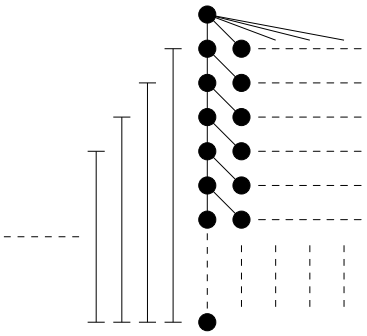

Figure 6. All suffixes of a string w in T are used to determine the outfix-freeness by OFF. The size of the sum of all suffixes of w is $O(|w|^2)$.

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 3.1. 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.

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

Theorem 3.2. 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.

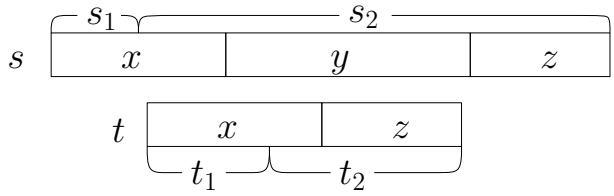

Figure 7. The figure illustrates the first case in the proof of Theorem 3.2, where s_i and $t_i \in L_i$ for $i = 1, 2$. Since s_1 is a prefix of t_1, L_1 is not outfix-free.

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 a 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. 7. 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.

For cases 1 and 2, L_1 is not prefix-free and, therefore, not outfix-free — a contradiction. For cases 3 and 4, L_2 is not suffix-free and, therefore, not outfix-free — a contradiction. Thus, the family of outfix-free languages is closed under catenation. Ito et al. [16] gave a different proof. □

3.4. Outfix-freeness of finite-state automata

We design an algorithm that determines the outfix-freeness of a given FA A using the structural properties. Assume that two strings $w_1 = xyz$ and $w_2 = xz$ are accepted by A, where $y \neq \lambda$ and w_2 is an outfix of w_1. Let p and q be the states that we reach after reading x from w_1 and w_2, respectively in A (p and q may be the same state). Since w_1 and w_2 are in $L(A)$, $yz \in L_{\overrightarrow{p}}$ and $z \in L_{\overrightarrow{q}}$ in A. This follows that $L_{\overrightarrow{p}} \cup L_{\overrightarrow{q}}$ is not suffix-free. We use *state-pair graphs* to check the existence of such w_1 and w_2 in $L(A)$.

Definition 3.2. Given an FA $A = (Q, \Sigma, \delta, s, F)$, we define the state-pair graph $G_A = (V_G, E_G)$, where V_G is a set of nodes and E_G is a set of edges, as follows:

$V_G = \{(i, j) \mid i \text{ and } j \in Q\}$ and

$E_G = \{((i, j), a, (x, y)) \mid \delta(i, a) = x \text{ and } \delta(j, a) = y \text{ and } a \in \Sigma\}$.

The crucial property of state-pair graphs is that if there is a string w spelled out by two distinct paths in A, for example, one path is from i to x and the other path is from j to y, then, there is a path from (i, j) to (x, y) in G_A that spells out the same string w. Note that state-pair graphs do not require given FAs to be deterministic. The state-pair graph $G_A = (V_G, E_G)$ of an FA $A = (Q, \Sigma, \delta, s, F)$ has at most $|Q|^2$ nodes and $|\delta|^2$ edges.

Note that an outfix-free FA is always non-exiting. If an FA is non-exiting and has several final states, then all final states can be merged into a single final state since they are all equivalent. Therefore, we assume that a given FA is non-exiting and has a single final state. Given an FA $A = (Q, \Sigma, \delta, s, f)$, we demonstrate how to determine the outfix-freeness of $L(A)$ using its state-pair graph. We first computer its state-pair graph G_A, where $m = |Q|$. Next, we define a new DFA A' from G_A by making $(1, 1)$ to be the start state and (m, m) to be the final state and removing all non-reachable states from $(1, 1)$ in G_A. Note that, by construction, A' is deterministic. We call A' the *state-pair DFA* of A.

Lemma 3.3. Given an FA $A = (Q, \Sigma, \delta, s, f)$, $L(A)$ is outfix-free if and only if $L_{\overrightarrow{p}} \cup L_{\overrightarrow{q}}$ is suffix-free for all pair states $(p, q) \in Q'$ of its state-pair DFA $A' = (Q', \Sigma, \delta', s', f')$, where $L_{\overrightarrow{q}}$ is the right language of state q in A.

Proof:
$\Longrightarrow$ Assume that $L_{\overrightarrow{p}} \cup L_{\overrightarrow{q}}$ is not suffix-free. Then, there are two strings w_1 and w_2 in $L_{\overrightarrow{p}} \cup L_{\overrightarrow{q}}$, where w_2 is a suffix of w_1. There are four cases to consider:

1. $w_1, w_2 \in L_{\overrightarrow{p}}$.

   Since p is reachable from s in A, there is a path from s to p and the path spells out a string x. This implies that A accepts both xw_1 and xw_2, where xw_2 is an outfix of xw_1 — a contradiction. Note that (p, p) is also a state of A'.

2. $w_1 \in L_{\overrightarrow{p}}$ and $w_2 \in L_{\overrightarrow{q}}$.

   Since all states of A' are reachable from $(1, 1)$ and A' is deterministic, there is a path from $(1, 1)$ to (p, q) that spells out a string x. This implies that A accepts both xw_1 and xw_2, where xw_2 is an outfix of xw_1 — a contradiction.

3. $w_1 \in L_{\overrightarrow{q}}$ and $w_2 \in L_{\overrightarrow{p}}$.

   This case is symmetric to the second case.

4. $w_1, w_2 \in L_{\overrightarrow{q}}$.

   This case is symmetric to the first case.

Therefore, if $L(A)$ is outfix-free, then $L_{\overrightarrow{p}} \cup L_{\overrightarrow{q}}$ is suffix-free.

$\Longleftarrow$ 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. Since w_1 and w_2 are spelled out by A, there are two accepting paths for w_1 and w_2, respectively. Let p and q be the states that we reach after reading x from the two accepting paths. This follows that the pair (p, q) is reachable from $(1, 1)$ in the state-pair graph of A and, thus, (p, q) is a state of its state-pair DFA A'. Furthermore, from the accepting paths for w_1 and w_2, we know that $yz \in L_{\overrightarrow{p}}$ and $z \in L_{\overrightarrow{q}}$ in A. Thus, $L_{\overrightarrow{p}} \cup L_{\overrightarrow{q}}$ is not suffix-free — a contradiction.

Therefore, if $L_{\overrightarrow{p}} \cup L_{\overrightarrow{q}}$ is suffix-free, then $L(A)$ is outfix-free. □

Given an FA $A = (Q, \Sigma, \delta, s, f)$ and its state-pair DFA $A' = (Q', \Sigma, \delta', s', f')$, we now need to check whether or not $L_{\overrightarrow{p}} \cup L_{\overrightarrow{q}}$ is suffix-free for all state $(p, q) \in Q'$. Since $L_{\overrightarrow{q}}$ in A is computed from its right FA $A_{\overrightarrow{q}}$, we can construct an FA $B = (Q_B, \Sigma, \delta_B, s_B, f_B)$ for $L_{\overrightarrow{p}} \cup L_{\overrightarrow{q}}$ from $A_{\overrightarrow{p}} = (Q_p, \Sigma, \delta_p, p, f)$ and $A_{\overrightarrow{q}} = (Q_q, \Sigma, \delta_q, q, f)$, where

$$Q_B = \{s_B, f_B\} \cup Q_p \cup Q_q,$$

$$\delta_B = \{(s_B, \lambda, p), (s_B, \lambda, q), (f, \lambda, f_B)\} \cup \delta_p \cup \delta_q.$$

Note that $O(|B|) = O(|A_{\overrightarrow{p}}| + |A_{\overrightarrow{q}}|) = O(|A|)$. Recently, Han et al. [11] proposed algorithms to determine prefix-freeness, suffix-freeness, bifix-freeness and infix-freeness of a given (nondeterministic) FA A in $O(|A|^2)$ time. We use the algorithm to check suffix-freeness for each B. Given an FA $A = (Q, \Sigma, \delta, s, f)$, there are at most $|Q|^2$ pair states in its state-pair DFA and, for each pair state, we can decide the suffix-freeness of B in $O(|B|^2)$ worst-case time.

Theorem 3.3. Given an FA $A = (Q, \Sigma, \delta, s, f)$, we can determine the outfix-freeness of $L(A)$ in $O(|Q|^4 + |Q|^2|\delta|^2)$ worst-case time.

Proof:
Since there are at most $|Q|^2$ pair states in its state-pair DFA and, for each pair state, it takes $O(|B|^2)$ time for checking the suffix-freeness of $L(B)$, the total running time is

$$|Q|^2 \times O(|B|^2) = |Q|^2 \times O(|A|^2) = |Q|^2 \times O(|Q|^2 + |\delta|^2) = O(|Q|^4 + |Q|^2|\delta|^2).$$

Therefore, we can check the outfix-freeness of a given FA in polynomial time. □

We note that if a given FA A is deterministic, then we can speed up the algorithm in Theorem 3.3 by skipping the construction of the state-pair graph. Note that we compute the state-pair graph and its state-pair DFA to find a common prefix x of two strings w_1 and w_2 because the input FA is nondeterministic. However, if A is deterministic and w_1 and w_2 have a common string x, then both accepting paths must have the same path for x. Therefore, we only need to check whether or not $L_{\overrightarrow{q}}$ is suffix-free for each q in A. Since it takes $O(|Q|^2 + |\delta|^2)$ time for each state to check suffix-freeness and there are $|Q|$ states, the total runtime for determining outfix-freeness of A is $O(|Q|^3 + |Q||\delta|^2)$. Since $O(|Q|) = O(|\delta|)$ in DFAs, we establish the following result.

Proposition 3.1. Given a DFA $A = (Q, \Sigma, \delta, s, f)$, we can determine the outfix-freeness 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. [21, 22] showed that the primality of regular languages is decidable and the prime decomposition of a regular language is not unique even for finite languages. 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. [11] 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 [16], the primality test for finite languages is believed to be NP-complete [22]. Thus, the decomposition problem for finite languages is beyond trivial. We design a linear-time algorithm to determine whether or not a given finite language is prime outfix-free. We also investigate prime outfix-free decomposition and its uniqueness.

4.1. Prime outfix-free regular languages

Definition 4.1. 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 FAs for such languages. We call DFAs without back-edges *acyclic DFAs* (ADFAs). Furthermore, outfix-free FAs are always non-exiting since they are prefix-free. Note that if an FA is non-exiting and has several final states, then all final states are equivalent and, therefore, can be merged into a single final state.

Definition 4.2. We define a state b in an ADFA A to be a *bridge state*[1] 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 as described below.

Given an ADFA $A = (Q, \Sigma, \delta, s, f)$ 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:

- $A_1 = (Q_1, \Sigma, \delta_1, s, b)$,

[1]The definition of bridge states in this paper is different from the definition of bridge states in Han et al. [11] although both definitions have similar conditions.

Q_1 is a set of states that appear on some path from s to b in A including both s and b.

δ_1 is a set of transitions that appear on some path from s to b in A.

- $A_2 = (Q_2, \Sigma, \delta_2, b, f)$,

  Q_2 is a set of states that appear on some path from b to f in A including both b and f.

  δ_2 is a set of transitions that appear on some path from b to f in A.

Fig. 8 illustrates a partitioning at a bridge state.

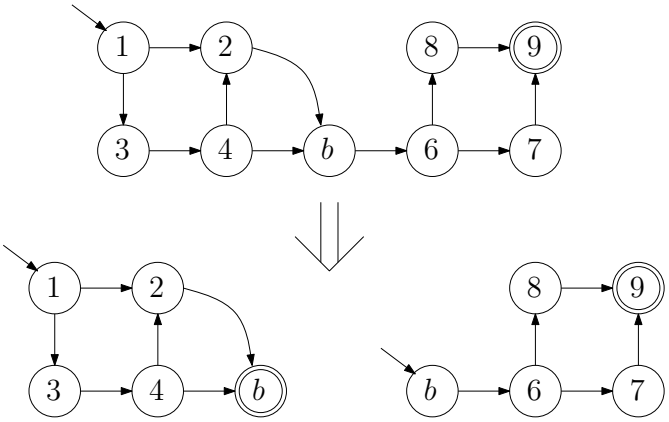

Figure 8. An example of partitioning of an FA at a bridge state b.

It is easy to verify that $L(A) = L(A_1) \cdot L(A_2)$ from the second requirement in Definition 4.2. Namely, bridge states are closely related to the decomposition of an FA.

Lemma 4.1. 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 and $L(A) = L(A_1)L(A_2)$. Ito et al. [16] showed that if $L(A) = L(A_1)L(A_2)$ is outfix-free, then $L(A_1)$ and $L(A_2)$ are both outfix-free. Therefore, if A has a bridge state, then $L(A)$ is not prime. □

Lemma 4.2. 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 4.1 as follows: If L is prime, then its minimal DFA does not have any bridge states. Then, from Lemmas 4.1 and 4.2, we obtain the following result.

Theorem 4.1. An outfix-free regular language L is prime if and only if the minimal DFA for L does not have any bridge states.

Lemma 4.1 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 the bridge state. In addition, if we have a set B of bridge states in A and decompose A at a bridge state $b \in B$, then $B \setminus \{b\}$ is the set of bridge states for the resulting two automata after the decomposition.

Theorem 4.2. 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 4.1. Since A'' has no bridge states, $L'' = L(A'')$ is prime by Theorem 4.1. 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. □

Ito et al. [16] showed that if an outfix-free regular language L is written as $L = L_1 L_2$ for two regular languages L_1 and L_2, then both L_1 and L_2 must be outfix-free. Then, the minimal DFA for L must have a corresponding bridge state for L_1 and L_2 by Lemma 4.1. This shows that L can be decomposed if and only if its minimal DFA has a bridge state. Therefore, a set of bridge states define the unique prime decomposition of a given minimal DFA for an outfix-free regular language. From this observation and Theorem 4.2, we establish the following result.

Proposition 4.1. The prime outfix-free decomposition for an outfix-free regular language is always unique.

We now demonstrate how to compute a set of bridge states defined in Definition 4.2 from a 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 4.3. Let $P_{s,f}$ be arbitrary simple path from s to f in G. Then, all bridge states of A are states on $P_{s,f}$.

Proof:
Assume that a state q is a bridge state and is not on $P_{s,f}$. This assumption immediately 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 using Depth-First Search (DFS). For details on DFS, refer to the textbook [3]. All states on $P_{s,f}$ form a set of candidate bridge states; namely, $\mathcal{C}_B = \{s, b_1, b_2, \ldots, b_k, f\}$. Our approach is to take all states in $\mathcal{C}_B$ as bridge state candidates and to identify all states that violate any requirements in Definition 4.2 and remove them from $\mathcal{C}_B$. Consequently, the remaining states are the bridge states.

We use DFS to explore G from s. We visit all states in $\mathcal{C}_B$ first. While exploring G, we maintain the following two values, for each state $q \in Q$,

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

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

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 $\mathbf{max} \neq i{+}1$, then it means that there is another simple path from b_i to $b_{\mathbf{max}}$ without passing through b_{i+1}.

When a state $q \in Q \setminus \mathcal{C}_B$ 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 $\mathcal{C}_B$ and the other is a subset T_2 of $Q \setminus \mathcal{C}_B$; 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:

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

where $q.\mathbf{anc}$ denotes the **anc** value of q and $q.\mathbf{max}$ denotes the **max** value of q.

Fig. 9 provides an example of DFS after updating (**anc**, **max**) for all states in G.

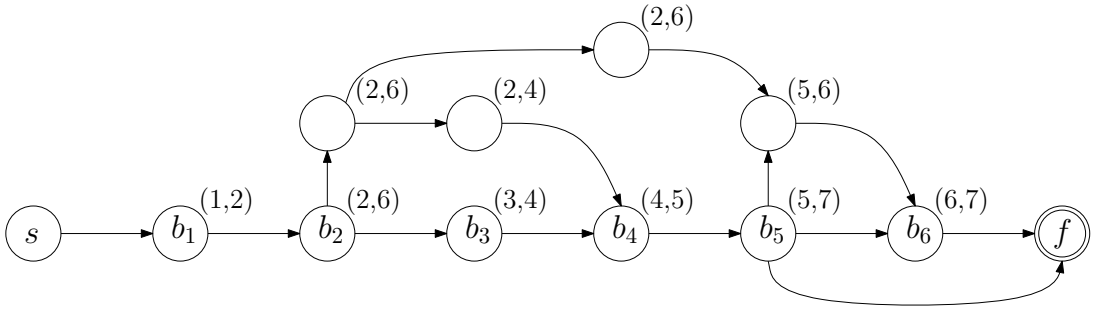

Figure 9. An example of DFS that computes (**anc**, **max**), for each state in G, for a given $\mathcal{C}_B = \{s, b_1, b_2, b_3, b_4, b_5, b_6, f\}$.

If a state $b_i \in \mathcal{C}_B$ does not have any out-transitions except for a transition to $b_{i+1} \in \mathcal{C}_B$ (for example, b_6 in Fig. 9), then b_i has $(i, i{+}1)$ when DFS is completed. Once we have completed DFS and computed (**anc**, **max**) for all states in G, we remove states from $\mathcal{C}_B$ that violate the requirements to be bridge states. Assume $b_i \in \mathcal{C}_B$ has (i, j), where $i{+}1 < j$. We remove $b_{i+1}, b_{i+2}, \ldots, b_{j-1}$ from $\mathcal{C}_B$ since that there is a path from b_i to b_j; that is, there is another simple path from b_i to f without visiting $b_{i+1}, b_{i+2}, \ldots, b_{j-1}$.

Then, we remove s and f from $\mathcal{C}_B$. For example, we have $\{b_1, b_2\}$ after removing states that violate the requirements from $\mathcal{C}_B$ in Fig. 9. This algorithm gives the following result.

Theorem 4.3. 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.

Proof:
Since we use DFS twice to compute $\mathcal{C}_B$ and (**anc**, **max**) for all states in A, the runtime is $O(m)$. Once we have computed (**anc**, **max**) for all states, then we remove states that violate the requirements from $\mathcal{C}_B$. It takes linear time in the size of $\mathcal{C}_B$, which is at most m. Therefore, the total runtime for computing bridge states of A is $O(m)$. Then, by Theorems 4.1 and 4.2, the two results are true. □

5. Conclusions

We have investigated the outfix-free regular languages. First, we suggested an efficient 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 FA.

Second, 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. 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.

Acknowledgements

We wish to thank the referee for the careful reading and many valuable suggestions. Especially, the referee's comments help to improve the algorithm for Theorem 3.3.

References

[1] Béal, M.-P., Crochemore, M., Mignosi, F., Restivo, A., Sciortino, M.: Computing forbidden words of regular languages., *Fundamenta Informaticae*, **56**(1-2), 2003, 121–135.

[2] Clarke, C. L. A., Cormack, G. V.: On the use of regular expressions for searching text, *ACM Transactions on Programming Languages and Systems*, **19**(3), 1997, 413–426.

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

[4] Crochemore, M., Mignosi, F., Restivo, A.: Automata and Forbidden Words., *Information Processing Letters*, **67**(3), 1998, 111–117.

[5] Czyzowicz, J., Fraczak, W., Pelc, A., Rytter, W.: Linear-Time Prime Decomposition Of Regular Prefix Codes, *International Journal of Foundations of Computer Science*, **14**, 2003, 1019–1032.

[6] Giammarresi, D., Montalbano, R.: Deterministic Generalized Automata, *Theoretical Computer Science*, **215**, 1999, 191–208.

[7] Golomb, S., Gordon, B., Welch, L.: Comma-Free Codes, *The Canadian Journal of Mathematics*, **10**, 1958, 202–209.

[8] Han, Y.-S., Salomaa, K., Wood, D.: Intercode Regular Languages, *Fundamenta Informaticae*, **76**(1-2), 2007, 113–128.

[9] Han, Y.-S., Trippen, G., Wood, D.: Simple-regular expressions and languages, *Proceedings of DCFS'05*, 2005, 146–157.

[10] Han, Y.-S., Wang, Y., Wood, D.: Prefix-Free Regular-Expression Matching, *Proceedings of CPM'05*, Lecture Notes in Computer Science 3537, 2005, 298–309.

[11] Han, Y.-S., Wang, Y., Wood, D.: Infix-free Regular Expressions and Languages, *International Journal of Foundations of Computer Science*, **17**(2), 2006, 379–393.

[12] Han, Y.-S., Wood, D.: The Generalization of Generalized Automata: Expression Automata, *International Journal of Foundations of Computer Science*, **16**(3), 2005, 499–510.

[13] Han, Y.-S., Wood, D.: Outfix-free Regular Languages and Prime Outfix-free Decomposition, *Proceedings of ICTAC'05*, Lecture Notes in Computer Science 3722, 2005, 96–109.

[14] Hopcroft, J., Ullman, J.: *Introduction to Automata Theory, Languages, and Computation*, 2 edition, Addison-Wesley, Reading, MA, 1979.

[15] Ito, M., Jürgensen, H., Shyr, H.-J., Thierrin, G.: N-prefix-suffix languages, *International Journal of Computer Mathematics*, **30**, 1989, 37–56.

[16] Ito, M., Jürgensen, H., Shyr, H.-J., Thierrin, G.: Outfix and Infix Codes and Related Classes of Languages, *Journal of Computer and System Sciences*, **43**, 1991, 484–508.

[17] Jürgensen, H.: Infix codes, *Proceedings of Hungarian Computer Science Conference*, 1984, 25–29.

[18] Jürgensen, H., Konstantinidis, S.: Codes, in: *Word, Language, Grammar* (G. Rozenberg, A. Salomaa, Eds.), vol. 1 of *Handbook of Formal Languages*, Springer-Verlag, 1997, 511–607.

[19] Kari, L.: On Language Equations with Invertible Operations., *Theoretical Computer Science*, **132**(2), 1994, 129–150.

[20] Long, D. Y., Ma, J., Zhou, D.: Structure of 3-infix-outfix maximal codes, *Theoretical Computer Science*, **188**(1-2), 1997, 231–240.

[21] Mateescu, A., Salomaa, A., Yu, S.: *On the Decomposition of Finite Languages*, Technical Report 222, TUCS, 1998.

[22] Mateescu, A., Salomaa, A., Yu, S.: Factorizations of Languages and Commutativity Conditions, *Acta Cybernetica*, **15**(3), 2002, 339–351.

[23] Shyr, H., Yu, S.: Intercodes and some related properties., *Soochow J. Math.*, **16**(1), 1990, 95–107.

[24] Shyr, H.-J.: *Lecture Notes: Free Monoids and Languages*, Hon Min Book Company, Taichung, Taiwan R.O.C, 1991.

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

[26] Wood, D.: *Data structures, algorithms, and performance*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.