# THE FORMAL LANGUAGE THEORY COLUMN

BY

**ARTO SALOMAA**

Turku Centre for Computer Science and,
Department of Mathematics, University of Turku
FIN-20014 Turku, Finland
asalomaa@utu.fi

# DECISION ALGORITHMS FOR SUBFAMILIES OF REGULAR LANGUAGES USING STATE-PAIR GRAPHS

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

**Abstract**

We survey recent results on decision algorithms for subfamilies of regular languages. In particular, we look at the decision algorithms using state-pair graphs constructed from finite-state automata. The algorithms rely on the structural property of a finite-state automaton that is preserved in its state-pair graph. We also review applications of state-pair graphs in different subfamilies of regular languages.

*Han was supported by the KIST Tangible Space Initiative Grants 2E20050 and 2Z03050.

*Bulletin of the EATCS no 93, pp. 118–133, October 2007*

# 1 Introduction

There are many subfamilies of formal languages. Example are recursively enumerable languages, context-sensitive languages, context-free languages and regular languages. A subfamily sometimes includes another subfamily. For instance, among the four example subfamilies, each subfamily includes the following subfamilies in order and this gives rise to the Chomsky hierarchy [6]. Furthermore, a subfamily has many (often infinite) subfamilies depending on how to define subfamilies. For context-free languages, for example, there is an infinite hierarchy for $LL(k)$ languages and all $LL(k)$ languages are a proper subfamily of context-free languages [1]. Given a family $\mathcal{L}$ of languages and a language $L$, the *decision problem* of $L$ with respect to $\mathcal{L}$ is to decide whether or not $L$ belongs to $\mathcal{L}$.

In this column, we consider the decision problem of subfamilies of regular languages. Regular languages have many different subfamilies; for example, finite languages, one-unambiguous regular languages [4], block-deterministic regular languages [11] and so on. We investigate subfamilies of regular languages that are defined by code properties such as prefix-freeness, suffix-freeness or infix-freeness. Note that codes have been used in many different areas; for example, information processing, data compression, cryptography and information transmission [24]. Codes are categorized with respect to different conditions according to the applications. For instance, prefix-freeness establishes prefix-free codes[1]. In regular languages, prefix-freeness defines a subfamily $\mathcal{L}_{(p,r)}$, prefix-free regular languages, where all languages are prefix-free sets and regular. Namely,

$$\mathcal{L}_{(p,r)} = \{L \mid L \text{ is regular and prefix-free.}\}$$

Similarly, we can define suffix-free, bifix-free, infix-free and outfix-free regular languages. Most of the decision problems related to code properties are decidable for regular languages whereas they often become undecidable for context-free languages [24]. We study decision algorithms for these subfamilies of regular languages. We examine algorithms that use a particular graph, *state-pair graph*.

A state-pair graph is a directed graph computed from a finite-state automaton $A$ using pairs of states and pairs of transitions in $A$. We review the basic concepts of state-pair graphs and decision algorithms for various subfamilies of regular languages that are defined by code properties. We emphasize the link between the structural property of state-pair graphs and these subfamilies of regular languages.

We define some basic notions in Section 2 and recall the formal definition of a state-pair graph in Section 3. We review decision algorithms for prefix-free, suffix-free and infix-free regular languages in Section 4. Then, we look at $k$-intercode regular languages in two different cases in Section 5; 1) $k$ is fixed and

[1] In the literature, prefix-free codes are often called prefix codes.

2) $k$ is unknown, where $k$ is an index. Lastly, we turn to two finite languages, hypercodes and outfix-free regular languages in Section 6 and conclude the column in Section 7.

# 2 Preliminaries

Let $\Sigma$ be a finite alphabet of characters and $\Sigma^*$ be the set of all strings over $\Sigma$. The number of characters in $\Sigma$ is denoted by $|\Sigma|$. A language over $\Sigma$ is any subset of $\Sigma^*$. The symbol $\emptyset$ denotes the empty language and the symbol $\lambda$ denotes the null string. Given a string $x$ from a set $X$, let $x^R$ be the reversal of $x$, in which case $X^R = \{x^R \mid x \in X\}$.

A finite-state automaton (FA) $A$ is specified by a tuple $(Q, \Sigma, \delta, s, F)$, where $Q$ is a finite set of states, $\Sigma$ 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. Let $|Q|$ be the number of states in $Q$ and $|\delta|$ be the number of transitions in $\delta$. Then, the size $|A|$ of $A$ is $|Q| + |\delta|$. Given a transition $\delta(p, a) = q$, 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 say that $A$ is *non-returning* if the start state of $A$ does not have any in-transitions and $A$ is *non-exiting* if all final states of $A$ do not have any out-transitions. In the following, we always assume that $A$ has only *useful* states; that is, each state of $A$ appears in some path from the start state to some final state.

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 $A_{\overrightarrow{q}}$.

Definition 1 is a list of codes that we use to define subfamilies of regular languages in the following sections.

**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 $y = \lambda$.*

- *a* intercode *of index k (or a k-intercode) if* $L^{k+1} \cap \Sigma^+ L^k \Sigma^+ = \emptyset$.
- *a* hypercode *if, for all distinct strings* $x, y \in \Sigma^*$, $x \in L$ *and* $y \in L$ *imply that x and y are not subsequences of each other.*

A regular language $L$ is prefix-free if $L$ is a prefix-free set. We say that an FA $A$ is prefix-free if $L(A)$ is prefix-free. We can establish similar notions for the other code sets in Definition 1.

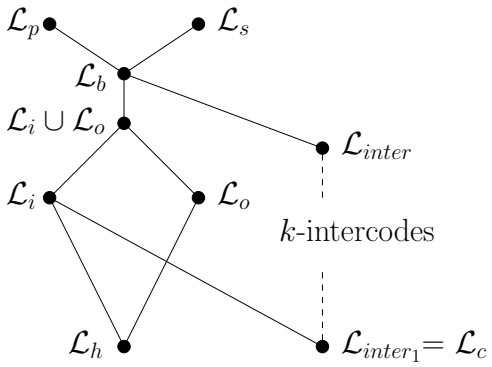


Figure 1: The families of languages defined by code properties in Definition 1. Solid lines indicate proper inclusions and a dotted line denotes a proper hierarchy. The diagram does not, in general, indicate intersections or unions. The full diagram with more code families can be found in Jürgensen and Konstantinidis [24].

For all unexplained notions related to formal languages, refer to the textbooks [21, 30]. For more details on coding theory, refer to Berstel and Perrin [3] or Jürgensen and Konstantinidis [24].

# 3 State-pair graphs

FAs are the basic model used to represent regular languages in many applications. FAs are essentially labeled directed graphs and each path from a start state to a final state spells out an accepted string. There are two well-known families of FAs in the literature: the Thompson automata [29] and the position automata [13, 26]. One advantage of using such families of FAs is that these automata preserve the structural properties of corresponding regular expressions. Caron and Ziadi [5] studied the structural properties of the position automata and Giammarresi et al. [12] examined the structural properties of the Thompson automata.

On the other hand, if we manipulate FAs, then these FAs easily lose certain structural properties; for example, if we catenate a position automaton and a Thompson automaton, then the resulting automaton does not preserve either the position automaton properties or the Thompson automaton properties. Nevertheless, one property remains unchanged in FAs: a path from a start state to a final state spells out an accepted string. The use of state-pair graphs relies on this fact. Applications of state-pair graphs have been already investigated earlier by Berstel and Perrin [3], where this notion is called the square of an automaton.

We first recall the definition of a state-pair graph and its complexity from Han et al. [17]. Given an FA $A = (Q, \Sigma, \delta, s, F)$, we assign a unique number for each state from 1 to $m$, where 1 denotes the start state and $m = |Q|$. If $A$ has a single final state, then we assume that $m$ denotes the final state.

**Definition 2.** *Given an FA $A = (Q, \Sigma, \delta, s, F)$, we define the state-pair graph $G_A = (V_G, E_G)$ of $A$, where $V_G$ is a set of nodes and $E_G$ is a set of labeled edges, as follows:*

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

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

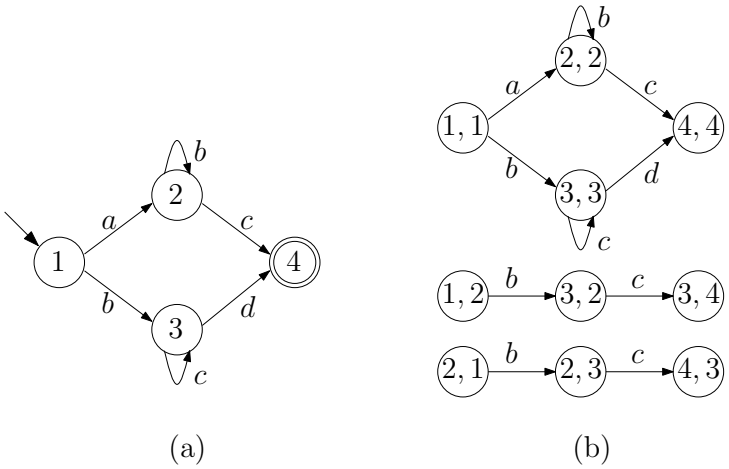


Figure 2: (a) is an FA $A$ and (b) is the corresponding state-pair graph $G_A$. We omit all nodes without transitions in $G_A$.

For FAs with $\lambda$-transitions, we use $a \in \Sigma \cup \{\lambda\}$ for computing $E_G$ of $G_A$. The crucial property of $G_A$ is that if there is a string $w$ spelled out by two distinct paths

in $A$, for example, one path is from $i$ to $p$ and the other path is from $j$ to $q$, then, there is a path from $(i, j)$ to $(p, q)$ in $G_A$ that also spells out the same string $w$. The complexity of $G_A$ is as follows:

Since we compute all pairs of states from $A$,

$$|V_G| = |Q|^2. \tag{1}$$

Let $|\delta_i|$ be the number of out-transitions from state $i$ 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,

$$|E_G| = \sum_{i,j=1}^{m} |\delta_i| \times |\delta_j| \leq |\delta|^2. \tag{2}$$

Therefore, by (1) and (2), $G_A$ has at most $|Q|^2$ nodes and $|\delta|^2$ edges.

**Proposition 3.** *Given an FA $A = (Q, \Sigma, \delta, s, F)$ and its state-pair graph $G_A$, $|G_A| \leq |Q|^2 + |\delta|^2$. Namely, $|G_A| = O(|A|^2)$.*

Note that the construction of $G_A$ does not require an input FA $A$ to be deterministic. In the following sections, we present recent results using state-pair graphs for determining subfamilies of regular languages.

# 4 Prefix-free, suffix-free and infix-free regular languages

We first examine three well-known codes. Prefix-freeness and suffix-freeness are symmetric. A language $L$ is prefix-free if and only if $L^R$ is suffix-free. Infix-freeness is stronger than both prefix-freeness and suffix-freeness as shown in Fig. 1; if $L$ is infix-free, then $L$ is always prefix-free and suffix-free.

## 4.1 Prefix-free and suffix-free regular languages

Prefix-freeness has already been used in the literature. Prefix-freeness defines Huffman codes [22] and *determinism* for generalized automata [10] and for expression automata [19]. Recently, Han et al. [16] considered prefix-free regular expressions as patterns in text searching and designed an efficient algorithm for the prefix-free regular-expression matching problem based on prefix-freeness.

If a given FA is deterministic, then it is easy to verify the prefix-freeness of $L(A)$; $L(A)$ is prefix-free if and only if $A$ is non-exiting [3, 19]. On the other hand, if $A$ is nondeterministic, then the condition that $A$ is non-exiting is only necessary but not sufficient. Thus, we have to check whether or not $L(A) \cap L(A)\Sigma^+ = \emptyset$. Let us

examine what $L(A) \cap L(A)\Sigma^+ = \emptyset$ implies in state-pair graphs. If $L(A) \cap L(A)\Sigma^+ \neq \emptyset$, then there are two distinct strings $w_1 = xy$ and $w_2 = x$ for some strings $x$ and $y$, where $x, y \neq \lambda$. Since $w_1$ and $w_2$ have a common prefix $x$, there is a path from $(1, 1)$ to $(m, j)$ such that $j \neq m$. Based on this observation, Han et al. [16] established the following result.

**Theorem 4.** *Given an FA $A = (Q, \Sigma, \delta, s, f)$, $L(A)$ is prefix-free if and only if there is no path from $(1, 1)$ to $(m, j)$, for any $j \neq m$, in $G_A$, where 1 denotes the start state and $m = |Q|$ denotes the final state.*

In Theorem 4, we implicitly assume that $A$ has a single final state. This assumption is valid since a prefix-free FA must be non-exiting and, thus, all final state are equivalent and mergible into a single state. Using Theorem 4, they proposed a prefix-freeness checking algorithm for an FA.

---

**Prefix-Freeness**($A = (Q, \Sigma, \delta, s, f)$)

**if** $A$ is not non-exiting
  **then** return no
Construct $G_A = (V_G, E_G)$ from $A$

DFS($(1, 1)$) in $G_A$
**if** we meet a node $(m, j)$ for some $j$, $j \neq m$
  **then** return no

return yes

---

Figure 3: A prefix-freeness checking algorithm.

The sub-function DFS($(1, 1)$) in Prefix-Freeness (PF) in Fig. 3 is a depth-first search that starts at node $(1, 1)$ in $G_A$. The construction $G_A = (V, E)$ from $A$ takes $O(|Q|^2 + |\delta|^2)$ time and DFS takes $O(|V| + |E|)$ time. Therefore, the total running time for PF is $O(|Q|^2 + |\delta|^2)$. For details on DFS, refer to the textbook [8]. Note that PF takes an input as FAs. Thus, if a regular language is given by a regular expression $E$, then we can construct the Thompson automaton $A_E$ for $E$ since $|A_E| = O(|E|)$ [21, 29]. Now PF guarantees the following result.

**Theorem 5.** *Given an FA $A$, we can determine the prefix-freeness of $L(A)$ in $O(|A|^2)$ worst-case time.*

Next, we consider suffix-freeness. Since $L$ is prefix-free if and only if $L^R$ is suffix-free by definition, we can establish the following statement from Theorems 4 and 5.

**Theorem 6.** *Given an FA $A$, $L(A)$ is suffix-free if and only if there is no path from $(1, i)$ to $(m, m)$, for any $i \neq 1$, in $G_A$. Moreover, we can determine the suffix-freeness of $L(A)$ in $O(|A|^2)$ worst-case time.*

To be precise for Theorem 6, we have to transform an FA with multiple final states into an FA with a single final state before computing its state-pair graph: We introduce a new final state $f'$ and make $f'$ to be a target state of all final states by a $\lambda$-transition. Then, we change all final states except for $f'$ to non-final states.

A language $L$ is bifix-free if and only if $L$ is prefix-free and suffix-free. Since we can verify prefix-freeness and suffix-freeness in quadratic time, we can also decide bifix-freeness in the same runtime using state-pair graphs.

**Proposition 7.** *Given an FA $A$, we can determine the bifix-freeness of $L(A)$ in $O(|A|^2)$ worst-case time using its state-pair graph.*

## 4.2 Infix-free regular languages

We turn to infix-freeness. Infix-free languages have been used in text searching [7, 16] and computing forbidden words [2, 9]. Ito et al. [23] showed that it is decidable whether or not a given regular language is infix-free and recently, Béal et al. [2] proposed a polynomial-time algorithm that determines infix-freeness for DFAs. We review the infix-freeness decision algorithm for general FAs based on state-pair graphs designed by Han et al. [17].

Given an FA $A$, if $L(A)$ is not infix-free, then there are two distinct strings $w_1$ and $w_2$ accepted by $A$ and $w_2$ is an infix of $w_1$. This implies that there are two distinct paths in $A$ that spell out $w_1$ and $w_2$, respectively, and the path for $w_1$ has a subpath that spells out $w_2$.

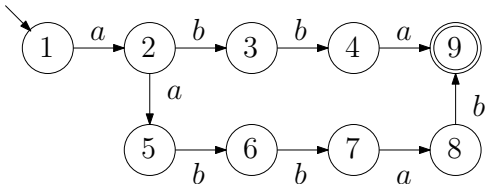


Figure 4: Two strings *abba* and *aabbab* are spelled out by two distinct paths.

In Fig. 4, for example, the FA accepts $w_1 = aabbab$ and $w_2 = abba$ and the subpath $q_2 \to q_5 \to q_6 \to q_7 \to q_8$ of the path for $w_1$ also spells out $w_2$. We can identify such strings in $L(A)$ from its state-pair graph since both strings have a common substring.

**Theorem 8.** *Given an FA $A = (Q, \Sigma, \delta, s, f)$, $L(A)$ is infix-free if and only if there is no path from $(1, i)$ to $(m, j)$ apart from $(1, 1)$ to $(m, m)$, where $1 \leq i \leq m$, $1 \leq j \leq m$, 1 denotes the start state and $m = |Q|$ denotes the final state. Moreover, we can determine the infix-freeness of $L(A)$ in $O(|A|^2)$ worst-case time.*

The proof for Theorem 8 can be found in Han et al. [17].

# 5 Intercodes

While comma-free languages have not been studied to the extent of prefix-free languages in the literature, the comma-free property was already introduced in 1958 [14]. Furthermore, Shyr and Yu [27] introduced *intercodes*, as a generalization of comma-free codes, see also Yu [31]. Comma-free codes are the intercodes of index one. Jürgensen et al. [25] have studied the decidability of the intercode property.

Note that if an index $k$ is given, then we can fairly easily check whether or not $L$ is an intercode of index $k$. However, if no index is given, then the problem is not as straightforward. Jürgensen et al. [25] established that it is decidable whether or not a given regular language is an intercode (of any index). There the complexity of the decision algorithm is not discussed explicitly, but it is easy to verify that an algorithm derived from the construction of the decidability proof is not a polynomial-time algorithm in the general case where the input language is specified by an NFA.

Recently, Han et al. [15] designed an algorithm that determines whether or not a given regular language $L$ is an intercode (of any index) using state-pair graphs. The algorithm runs in polynomial time for both DFAs and NFAs. Besides having better time complexity, the algorithm is conceptually easier to understand and implement compared with the algorithm derived from Jürgensen et al. [25].

Note that intercode (regular) languages are a proper subfamily of bifix-free (regular) languages as shown in Fig. 1. Thus, if a given FA $A$ is not bifix-free, then we immediately know that $L(A)$ is not an intercode. Therefore, we can assume that a given FA $A$ is bifix-free and this guarantees that

1. $A$ is non-returning and non-exiting.

2. $A$ has a single final state.

From such an FA $A = (Q, \Sigma, \delta, s, f)$, we can construct an FA $A^2$ for the language $L(A)L(A)$ by merging $f$ of the first copy of $A$ and $s$ of the second copy of $A$. The FA $A^2$ has $2|Q| - 1$ states and $2|\delta|$ transitions; namely, $|A^2| < 2|A|$. We can repeat this procedure to construct an FA for the catenation of several $A$'s. We use

$A^k$ to denote the FA for the catenation of $k$ copies of $A$ and $A_i$ to denote the $i$th component $A$ of $A^k$, for $1 \leq i \leq k$. We use $(i, j)$ in $A^k$ to denote the state $i$ in $A_j$ of $A^k$.

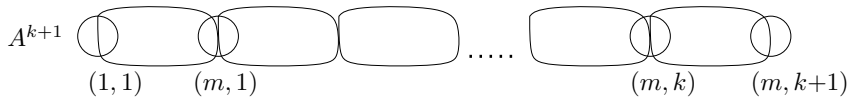


Figure 5: An example of an FA for the catenations of $k$+1 $A$s.

The following results have been proved in Han et al. [15].

**Lemma 9.** *Given an FA $A$ and an index $k$, we can determine whether or not $L(A)$ is a $k$-intercode in $k^2 \cdot O(|A|^2)$ worst-case time.*

**Corollary 10.** *Given an FA $A$, we can determine the comma-freeness of $L(A)$ in $O(|A|^2)$ worst-case time since a $k$-intercode for $k = 1$ is a comma-free code.*

Han et al. [15] also tackled the case when $k$ is unknown. Instead of trying all possible indices, which is very inefficient, they discovered that if $L(A)$ is not a $k$-intercode for a certain constant $k$, then $L(A)$ is not an intercode for any index.

**Lemma 11.** *Given an FA $A$, $L(A)$ is not an intercode for any index $k$ if $L(A)$ is not a $(m+1)$-intercode, where $m$ is the number of states in $A$.*

Using Lemmas 9 and 11, the decision problem for intercode regular languages can be solved as follows:

**Theorem 12.** *Given an FA $A$, we can determine whether or not $L(A)$ is an intercode of index $k$, for some $k$, in $O(|A|^4)$ worst-case time.*

The family of intercode (regular) languages has a proper hierarchy as illustrated in Fig. 1. Thus, if a regular language $L$ is identified as a $k$-intercode, $L$ may be a $(k-1)$-intercode as well. This leads to a new problem that computes the smallest $k$ such that $L(A)$ is a $k$-intercode but not a $(k-1)$-intercode in polynomial time. Based on Theorem 12, Han et al. [15] suggested a polynomial-time algorithm that relies on the binary search approach.

**Theorem 13.** *Given an FA $A$, in $O(\log |Q| \cdot |A|^4)$ worst-case time, we can determine whether or not $L(A)$ is an intercode for some index $k > 0$, and if the answer is positive we can find the smallest index $l$ such that $L(A)$ is an $l$-intercode but not an $(l-1)$-intercode.*

# 6 Hypercodes and outfix-free regular languages

So far, all subfamilies of regular languages in the preceding sections can be infinite. We now consider two subfamilies of regular languages that are always finite; hypercodes and outfix-free regular languages.

## 6.1 Hypercodes

A set $X$ of strings is a hypercode if a string in $X$ is not a subsequence of any other string in $X$. Based on hypercodes, Head and Thierrin [20] derived properties of $0L$ languages. Hypercodes are a proper subfamily of outfix-free languages. Moreover, hypercodes are always finite [28]. Since hypercodes are finite, we can decide whether or not a given finite set of strings is a hypercode by comparing all pairs of strings in the set, although it is certainly undesirable to do so. We look at an efficient algorithm for the decision problem. Since an FA $A$ for a hypercode must be non-exiting and $L(A)$ must be finite, we assume that $A$ has a single final state and has no back transitions that make cycles.

Given an FA $A = (Q, \Sigma, \delta, s, f)$, we assign a unique number for each state in $A$ from 1 to $m$, where $m = |Q|$. We construct a new FA $A'$ from $A$ by duplicating $A$ and adding a self-loop with $\Sigma$ to all states. Namely, $A' = (Q, \Sigma, \delta', s, f)$, where

$$\delta' = \delta \cup \{(q, \Sigma, q) \mid q \in Q\}.$$

We introduce a new state-pair graphs from $A$ and $A'$ as follows:

**Definition 14.** *Given an FA $A = (Q, \Sigma, \delta, s, f)$ and 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:*

$$V_G = \{(i, j) \mid i \in Q_A \text{ and } j \in Q_{A'}\} \text{ and}$$

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

*where $Q_A$ denotes $Q$ of $A$ and $Q_{A'}$ denotes $Q$ of $A'$.*

Note that the only difference between the new state-pair graph and the previous state-pair graph in Section 3 is that the new graph is constructed from two similar yet different FAs whereas the previous graph is constructed from a single FA. The complexity of the new graph is same as before; $|G_A| = O(|A^2|)$.

Fig. 6 illustrates a state-pair graph constructed as in Definition 14. The language $L(A) = \{abc, ac\}$ is not a hypercode since $ac$ is a subsequence of $abc$ whose path is $(1, 1) \to (2, 4) \to (3, 4) \to (5, 5)$ in $G_A$. Then, state-pair graphs ensure the following result:

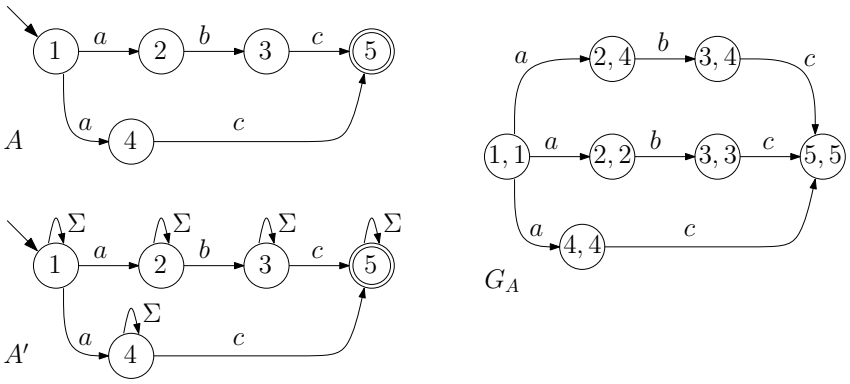


Figure 6: Given an FA $A$, we construct a new FA $A'$ for deciding whether or not $L(A)$ is a hypercode. $G_A$ is the corresponding state-pair graph. Note that $L(A) = \{abc, ac\}$ is not a hypercode since $ac$ is a subsequence of $abc$. We omit all nodes that do not appear in any path from $(1, 1)$ to $(5, 5)$ in $G_A$.

**Theorem 15.** *Given an FA $A = (Q, \Sigma, \delta, s, f)$, $L(A)$ is a hypercode, if and only if the state-pair graph $G_A$ for $A$ has no path $(i_1, j_1) \to (i_2, j_2) \to \cdots \to (i_k, j_k)$ that satisfies the following conditions:*

1. *$(i_1, j_1) = (1, 1)$ and $(i_k, j_k) = (m, m)$.*
2. *there exists at least one pair of two adjacent nodes $(i_u, j_u) \to (i_{u+1}, j_{u+1})$ such that $j_u = j_{u+1}$ for $1 \leq u < k$.*

Theorem 15 shows that given an FA $A$, we can check whether or not $L(A)$ is a hypercode in $O(|A|^2)$ worst-case time using its state-pair graph and DFS.

## 6.2 Outfix-free regular languages

We turn to 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 a common prefix $x$ and a common suffix $z$. Fig. 7 illustrates such $w_1$ and $w_2$.

Based on this property, Han and Wood [18] investigated the case when a finite language $L$ is given by a finite set of strings; $L = \{w_1, w_2, \ldots, w_n\}$. Note that a finite set of strings is often stored in a trie, which 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.

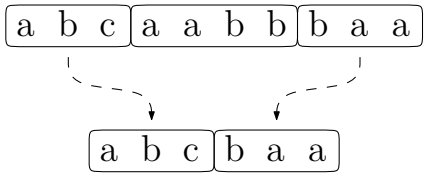


Figure 7: A graphical illustration of an outfix string; *abcbaa* is an outfix of *abcaabbbaa*.

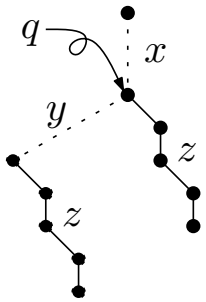


Figure 8: An example of a trie for strings $w_1 = xyz$ and $w_2 = xz$. Note that both paths end with the same subpath sequence in the trie because of a common suffix $z$.

Once we construct a trie for $L$, then two strings $w_1$ and $w_2$ share a common path from the root if they have a common prefix. See Fig. 8 for an example. Therefore, we only need to check whether or not a sub-trie of $T$ is suffix-free. If a sub-trie is not suffix-free, then $L$ is not outfix-free. By carefully analyzing this procedure, Han and Wood [18] designed an efficient decision algorithm and proved its correctness.

**Theorem 16.** *Given a finite set $L = \{w_1, w_2, \ldots, w_n\}$ of strings, we can determine whether or not $L$ 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 consider two stings $w_1 = xyz$ and $w_2 = xz$ in DFAs. If a DFA $A = (Q, \Sigma, \delta, s, f)$ accepts both $w_1$ and $w_2$, then there is a unique path from $s$ to a state $q$ that spells out $x$, which is a common prefix of $w_1$ and $w_2$. Then, $A_{\overrightarrow{q}}$ accepts $yz$ and $z$. This implies that $L_{\overrightarrow{q}}$ is not suffix-free.

**Proposition 17.** *Given a DFA $A = (Q, \Sigma, \delta, s, f)$, $L(A)$ is outfix-free if and only if $L_{\overrightarrow{q}}$ is suffix-free for all $q \in Q$.*

However, if a given FA is nondeterministic, then Proposition 17 does not hold anymore. Nevertheless, if an NFA $A$ is not outfix-free, then its state-pair graph has a similar property to the property used in Proposition 17. To make use of this property, Han and Wood [18] introduced a *state-pair DFA* for $A$ from its state-pair graph $G_A$ as follows:

**Definition 18.** *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:*

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

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

*Then, we define a new DFA $A'$ 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 $G_A$. We call $A'$ a* state-pair DFA.

We can decide the outfix-freeness of $L(A)$ using its state-pair DFA.

**Lemma 19.** *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 $A$.*

Han and Wood [18] gave a proof for Lemma 19 and proposed an algorithm that checks the outfix-freeness of $L(A)$ as follows; from $A$ and its state-pair DFA $A' = (Q', \Sigma, \delta', s', f')$, we 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 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

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

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

Since $O(|B|) = O(|A_{\overrightarrow{p}}| + |A_{\overrightarrow{q}}|) = O(|A|)$, this algorithm runs in polynomial time and gives the following result [18].

**Theorem 20.** *Given an FA $A = (Q, \Sigma, \delta, s, f)$, we can determine the outfix-freeness of $L(A)$ in $O(|A|^4)$ worst-case time.*

# 7 Conclusions

An FA $A$ for a regular language $L$ is more than just an acceptor for $L$; $A$ preserves the structural property of $L$. State-pair graphs were proposed to make use of these properties in FAs. We have surveyed a few number of decision algorithms using state-pair graphs for subfamilies of regular languages defined by code properties. We want to remark that all presented algorithms run in polynomial time. However, we do not know if there exist better time complexity algorithms for any subfamily of regular languages that has been considered. For instance, it is open if we can determine the prefix-freeness of $L(A)$ in subquadratic time.

# References

[1] A. Aho and J. Ullman. *The Theory of Parsing, Translation, and Compiling, Vol. I: Parsing*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1972.

[2] 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.

[3] J. Berstel and D. Perrin. *Theory of Codes*. Academic Press, Inc., 1985.

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

[5] P. Caron and D. Ziadi. Characterization of Glushkov automata. *Theoretical Computer Science*, 233(1–2):75–90, 2000.

[6] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956.

[7] 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.

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

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

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

[11] D. Giammarresi, R. Montalbano, and D. Wood. Block-deterministic regular languages. In *Proceedings of ICTCS'01*, Lecture Notes in Computer Science 2202, 184–196, 2001.

[12] D. Giammarresi, J.-L. Ponty, D. Wood, and D. Ziadi. A characterization of Thompson digraphs. *Discrete Applied Mathematics*, 134:317–337, 2004.

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

[14] S. Golomb, B. Gordon, and L. Welch. Comma-free codes. *The Canadian Journal of Mathematics*, 10:202–209, 1958.

[15] Y.-S. Han, K. Salomaa, and D. Wood. Intercode regular languages. *Fundamenta Informaticae*, 76(1-2):113–128, 2007.

[16] Y.-S. Han, Y. Wang, and D. Wood. Prefix-free regular-expression matching. In *Proceedings of CPM'05*, Lecture Notes in Computer Science 3537, 298–309, 2005.

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

[18] Y.-S. Han and D. Wood. Outfix-free regular languages and prime outfix-free decomposition. *Fundamenta Informaticae*. To appear.

[19] 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.

[20] T. Head and G. Thierrin. Hypercodes in deterministic and slender $0L$ languages. *Information and Control*, 45(3):251–262, 1980.

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

[22] D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40:1098–1101, 1952.

[23] 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.

[24] 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.

[25] H. Jürgensen, K. Salomaa, and S. Yu. Decidability of the intercode property. *Elektronische Informationsverarbeitung und Kybernetik*, 29(6):375–380, 1993.

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

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

[28] H.-J. Shyr and G. Thierrin. Hypercodes. *Information and Control*, 24(1):45–54, 1974.

[29] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11:419–422, 1968.

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

[31] S. Yu. A characterization of intercodes. *International Journal of Computer Mathematics*, 36:39–45, 1990.