Overlap-Free Regular Languages

Yo-Sub Han[1,⋆] and Derick Wood[2]

[1] System Technology Division, Korea Institute of Science and Technology, P.O. BOX 131, Cheongryang, Seoul, Korea
emmous@kist.re.kr
[2] Department of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong SAR
dwood@cs.ust.hk

Abstract. We define a language to be overlap-free if any two distinct strings in the language do not overlap with each other. We observe that overlap-free languages are a proper subfamily of infix-free languages and also a proper subfamily of comma-free languages. Based on these observations, we design a polynomial-time algorithm that determines overlap-freeness of a regular language. We consider two cases: A language is specified by a nondeterministic finite-state automaton and a language is described by a regular expression. Furthermore, we examine the prime overlap-free decomposition of overlap-free regular languages and show that the prime overlap-free decomposition is not unique.

1 Introduction

Regular languages are popular in many applications such as editors, programming languages and software systems in general. People often use regular expressions for searching in text editors or for `UNIX` command; for example, `vi`, `emacs` and `grep`. Moreover, regular expression searching is also used in pattern matching.

The pattern matching problem is to find all matching substrings of a text T with respect to a pattern L. If L is a regular language given by a regular expression, then the problem becomes the regular-expression matching problem. Many researchers have investigated various regular-expression matching problems [1, 3, 7, 18]. One question in regular-expression matching is how many matching substrings are in T. Given a regular expression E and a text T, there can be at most n^2 matching substrings in T with respect to $L(E)$, where n is the size of T. For example, $E = (a + b)^*$ and $T = aabababa \cdots abaa$ over the alphabet $\{a, b\}$. These matching substrings often overlap and nest with each other. To avoid this situation, researchers restrict the search to find and report only a linear subset of the matching substrings. We call it *linearizing restriction*. There are two well-known linearizing restrictions in the literature: The *longest match* rule, which is a generalization of the *leftmost longest match* rule of IEEE POSIX [14] and the *shortest-match substring search* rule of Clarke and Cormack [3]. These two rules have different semantics and, therefore, identify different matching

⋆ The author was supported by KIST Tangible Space Initiative Grant 2E19020.

D.Z. Chen and D.T. Lee (Eds.): COCOON 2006, LNCS 4112, pp. 469–478, 2006.

substrings for same pattern and text in general. On the other hand, Han and Wood [10] showed that if the pattern language is infix-free, then both rules give the same output. Furthermore, they proposed another linearizing restriction, *leftmost non-overlapping match* rule that only reports non-overlapping matching substrings of T. This new rule leads us to define a new subfamily of regular languages, *overlap-free regular languages*. We define a language L to be overlap-free if any two strings in L do not overlap with each other. (We give a formal definition in Section 3.) If we use an overlap-free regular language as pattern, it guarantees that all matching substrings of a text do not overlap with each other and, therefore, ensures a linear number of matching substrings.

As a continuation of our investigations of subfamilies of regular languages, it is natural to examine overlap-free regular languages and the prime overlap-free decomposition problem since overlap-free regular languages are a proper subfamily of regular languages. Our goal is to design an efficient algorithm that determines overlap-freeness of a given regular language and to study the prime overlap-free decomposition and its uniqueness.

We define some basic notions in Section 2. In Section 3, we define overlap-free languages and design an efficient algorithm that determines overlap-freeness of a given regular language L based on the structural properties of L. Then, in Section 4, we demonstrate that an overlap-free regular language does not have a unique prime overlap-free decomposition. We also develop an algorithm for computing a prime overlap-free decomposition from a minimal deterministic finite-state automaton (DFA) of an overlap-free regular language.

2 Preliminaries

Let Σ denote a finite alphabet of characters and Σ^* denote the set of all strings over Σ. A language over Σ is any subset of Σ^*. The character $\emptyset$ denotes the empty language and the character λ denotes the null string. A finite-state automaton (FA) A is specified by a tuple $(Q, \Sigma, \delta, s, F)$, where Q is a finite set of states, Σ is an input alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is a (finite) set of transitions, $s \in Q$ is the start state and $F \subseteq Q$ is a set of final states. Let $|Q|$ be the number of states in Q and $|\delta|$ be the number of transitions in δ. Then, the size $|A|$ of A is $|Q| + |\delta|$. Given a transition (p, a, q) in δ, where $p, q \in Q$ and $a \in \Sigma$, we say that p has an *out-transition* and q has an *in-transition*. Furthermore, p is a *source state* of q and q is a *target state* of p. A string x over Σ is accepted by A if there is a labeled path from s to a state in F such that this path spells out the string x. Thus, the language $L(A)$ of an FA A is the set of all strings that are spelled out by paths from s to a final state in F. We say that A is *non-returning* if the start state of A does not have any in-transitions and A is *non-exiting* if the final state of A does not have any out-transitions. We assume that A has only *useful* states; that is, each state of A appears on some path from the start state to some final state.

Given two strings x and y over Σ, x is a *prefix* of y if there exists $z \in \Sigma^*$ such that $xz = y$ and x is a *suffix* of y if there exists $z \in \Sigma^*$ such that $zx = y$.

Furthermore, x is said to be a *substring* or an *infix* of y if there are two strings u and v such that $uxv = y$. Given a set X of strings over, X is *infix-free* if no string in X is an infix of any other string in X. Similarly, X is *prefix-free* if no string in X is a prefix of any other string in X.

3 Overlap-Free Regular Languages

Given two strings x and y, we say that x and y overlap with each other if either a suffix of x is a prefix of y or a suffix of y is a prefix of x. For example, $x = abcd$ and $y = cdee$ overlap.

Definition 1. *Given a (regular) language L, we define L to be* overlap-free *if any two distinct strings in L do not overlap with each other.*

Since we examine overlap of strings, we can think of the derivative operation [2]. The *derivative* $x\backslash L$ of a language L with respect to a string x is the language $\{y \mid xy \in L\}$.

Proposition 1. *If a language L is overlap-free, then $x\backslash L \cup L$ is prefix-free for any string x.*

Let us examine the relationship with other families of languages. By Definition 1, overlap-free languages are a proper subfamily of infix-free languages. Golomb et al. [6] introduced comma-free languages: A language L is comma-free if $LL \cap \Sigma^+ L \Sigma^+ = \emptyset$. Comma-free languages are also a proper subfamily of infix-free languages [15]. We compare these two subfamilies of infix-free languages and establish the following result:

Proposition 2. *Overlap-free languages are a proper subfamily of comma-free languages.*

A regular language is represented by an FA or described by a regular expression. Thus, we define a regular expression E to be overlap-free if $L(E)$ is overlap-free and an FA A to be overlap-free if $L(A)$ is overlap-free.

We now investigate the decision problem of overlap-freeness of a regular language. Given a language L, L is prefix-free if and only if $L \cap L\Sigma^+ \neq \emptyset$ [15]. If L is a regular language, then we can check the emptiness of $L \cap L\Sigma^+$ in polynomial time. Thus, if we can find a proper string x, then we can use Proposition 1 for deciding overlap-freeness of L. However, we do not know which string is proper unless we check the emptiness of $(x\backslash L \cup L) \cap (x\backslash L \cup L)\Sigma^+$ and certainly it is undesirable to try all possible strings over Σ. Recently, Han et al. [8] introduced state-pair graphs and proposed an algorithm for determining infix-freeness of a regular language L based on the structural properties of L. Based on state-pair graphs, we design algorithms that determine overlap-freeness of a regular language. Since an overlap-free language must be infix-free, we assume that a given language L is infix-free. Note that we can check infix-freeness of L in quadratic

time in the size of the representation of L [8]; if L is not infix-free, then L is not overlap-free.

First, we consider when a language is given by an FA. 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 is the number of states in A.

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 edges, as follows:*

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

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 complexity of the state-pair graph $G_A = (V_G, E_G)$ for an FA $A = (Q, \Sigma, \delta, s, F)$ is as follows:

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

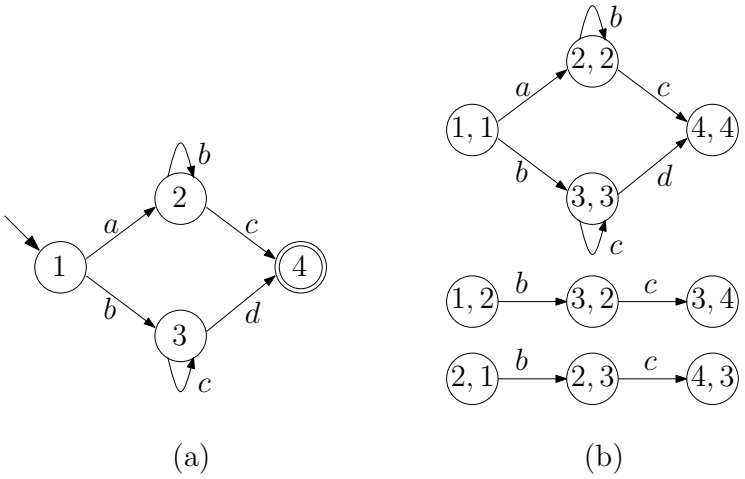

Fig. 1. (a) is an FA A for $L(ab^*c + bc^*d)$ and (b) is the corresponding state-pair graph G_A. We omit all nodes without transitions in G_A. Note that $L(A)$ is not overlap-free.

Fig. 1 illustrates the state-pair graph for a given FA A. Note that the language $L(A) = L(ab^*c + bc^*d)$ in Fig. 1 is not overlap-free since abc and bcd overlap, and the overlapped string bc appears on the path from $(1, 2)$ to $(3, 4)$ in G_A.

Since we assume that $L(A)$ is infix-free, a final state of A has no out-transitions and the start state has no in-transitions. Namely, A is non-returning and non-exiting. Therefore, if A has more than one final state, then all final states can be merged into a single final state since they are equivalent. From now on, we assume that a given FA is non-returning and non-exiting and has only one final state.

Theorem 1. *Given an FA $A = (Q, \Sigma, \delta, s, f)$, $L(A)$ is overlap-free if and only if the state-pair graph G_A for A has no path from $(1, i)$ to (j, m), where $i \neq m$ and $j \neq 1$, and 1 denotes the start state and m denotes the final state.*

We can identify such a path in Theorem 1 in linear time in the size of G_A using Depth-First Search (DFS) [4]. Thus, we obtain the following result from Proposition 3 and Theorem 1:

Theorem 2. *Given an FA $A = (Q, \Sigma, \delta, s, f)$, we can determine whether or not $L(A)$ is overlap-free in $O(|Q|^2 + |\delta|^2)$ worst-case time.*

Since $O(|\delta|) = O(|Q|^2)$ in the worst-case for NFAs, the runtime is $O(|Q|^4)$ in the worst-case. On the other hand, if a regular language is given by a regular expression E, then we can construct an FA for E that improves the worst-case running time. Since the complexity of state-pair graphs is closely related to the number of states and the number of transitions of input FAs, we use an FA construction that gives fewer states and transitions. One possibility is the Thompson construction [18].

Given a regular expression E, the Thompson construction takes $O(|E|)$ time and the resulting Thompson automaton has $O(|E|)$ states and $O(|E|)$ transitions [13]; namely, $O(|Q|) = O(|\delta|) = O(|E|)$. Even though Thompson automata are a subfamily of NFAs, they define all regular languages. Therefore, we can use Thompson automata to determine overlap-freeness of a regular language. Since Thompson automata allow null-transitions, we include the null-transition case to construct the edges for state-pair graphs as follows:

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

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

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

Furthermore, we can use state-pair graphs for determining comma-freeness of regular languages. A regular language L is comma-free if and only if $LL \cap \Sigma^+ L \Sigma^+ = \emptyset$. Because of the assumption that a given FA A is infix-free, (otherwise, $L(A)$ is not comma-free.) A has a single final state that has no out-transitions. Using this structural property, we construct an FA A' for LL by catenating two As; see Fig. 2 for an example.

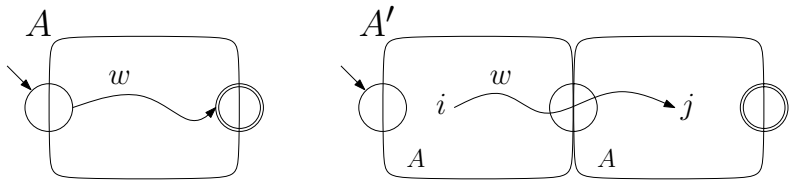

Fig. 2. Given an FA $A = (Q, \Sigma, \delta, s, f)$, we construct A' by merging the final state of one A and the start state of the other A. If $L(A)$ is not comma-free, then there exist two paths, one is from $A' = AA$ and the other is from A, and both path spell out the same string w.

Now we construct the state-pair graph for $L(A)$. The construction of state-pair graph for the comma-free case is slightly different from the state-pair graph in Definition 2. Given an FA $A = (Q, \Sigma, \delta, s, f)$, let $A' = (Q', \Sigma, \delta', s', f')$ be the catenation of two As; namely, $L(A') = L(A)L(A)$. The state-pair graph $G_A = (V_G, E_G)$ for the comma-free case is defined as follows:

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

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

A subfamily of languages with certain properties is often closed under catenation. For example, prefix-free languages, bifix-free languages, infix-free languages and outfix-free languages are all closed under catenation, respectively [8, 9, 11]. Now we characterize the family of overlap-free (regular) languages in terms of closure properties.

Theorem 5. *The family of overlap-free (regular) languages is closed under intersection but not under catenation, union, complement or star.*

4 Prime Overlap-Free Regular Languages and Decomposition

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

Mateescu et al. [16, 17] showed that the primality of regular languages is decidable and the prime decomposition of a regular language is not unique. Czyzowicz et al. [5] showed that for a given prefix-free regular language L, the prime

prefix-free decomposition is unique and the decomposition can be computed in $O(m)$ worst-case time, where m is the size of the minimal DFA for L. 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. On the other hand, the prime outfix-free decomposition of outfix-free regular languages is unique [11]. We investigate prime overlap-free regular languages and decomposition.

4.1 Prime Overlap-Free Regular Languages

Definition 3. *We define a regular language L to be a* prime *overlap-free language if $L \neq L_1 \cdot L_2$, for any overlap-free regular languages L_1 and L_2.*

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

Definition 4. *We define a state b in a DFA A to be a* bridge state *if the following conditions hold:*

1. *State b is neither a start nor a final state.*
2. *For any string $w \in L(A)$, its path in A must pass through b only once.*
3. *State b is not in any cycles in A.*
4. *$L(A_1)$ and $L(A_2)$ are overlap-free.*

Given an overlap-free DFA $A = (Q, \Sigma, \delta, s, f)$ with a bridge state $b \in Q$, we can partition A into two subautomata A_1 and A_2 as follows: $A_1 = (Q_1, \Sigma, \delta_1, s, b)$ and $A_2 = (Q_2, \Sigma, \delta_2, b, f)$, where Q_1 is a set of states that appear on some path from s and b in A, δ_1 is a set of transitions that appear on some path from s and b in A, $Q_2 = Q \setminus Q_1 \cup \{b\}$ and $\delta_2 = \delta \setminus \delta_1$. See Fig. 3 for an example.

Note that the second requirement in Definition 4 ensures that the decomposition of $L(A)$ is $L(A_1) \cdot L(A_2)$ and the third requirement is from the property that overlap-free FAs must be non-returning and non-exiting.

Theorem 6. *An overlap-free regular language L is prime if and only if the minimal DFA A for L does not have any bridge states.*

We tackle the decomposition problem based on FA partitioning using bridge states. Note that Czyzowicz et al. [5] demonstrated the use of FA partitioning for the prefix-free decomposition and Han and Wood [12] proposed an efficient algorithm that computes shorter regular expressions from FAs based on FA partitioning. In many applications, FAs become more and more complicated and the size of FAs is too large to fit into main memory. Therefore, FA decomposition is necessary and FA partitioning is one approach for solving this problem.

4.2 Prime Decomposition of Overlap-Free Regular Languages

The prime decomposition for an overlap-free regular language L is to represent L as a catenation of prime overlap-free regular languages. If L is prime, then L

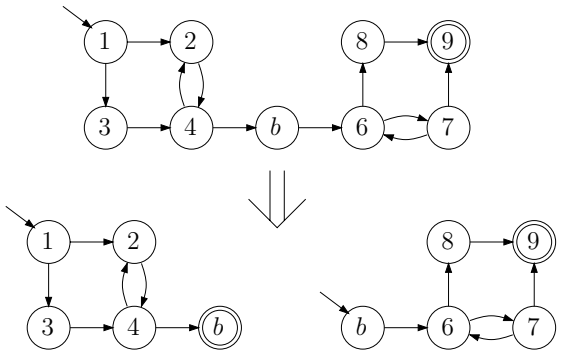

Fig. 3. An example of the partitioning of an FA at a bridge state b

itself is a prime decomposition. Thus, given an overlap-free regular language L, we, first, determine whether or not L is prime. If L is not prime, then there should be some bridge state(s) and we decompose L using the bridge state(s). Let A_1 and A_2 be two subautomata partitioned at a bridge state for L. If both $L(A_1)$ and $L(A_2)$ are prime, then a prime decomposition of L is $L(A_1) \cdot L(A_2)$. Otherwise, we repeat the preceding procedure for a non-prime language.

Let B denote a set of bridge states for a given minimal DFA A. The number of states in B is at most m, where m is the number of states in A. Note that once we partition A at $b \in B$ into A_1 and A_2, then only the states in $B \setminus \{b\}$ can be bridge states in A_1 and A_2. (It is not necessary for all remaining states to be bridge states as demonstrated in Fig. 4.) Therefore, we can determine the primality of $L(A)$ by checking whether or not A has bridge states. Moreover, we can compute a prime decomposition of $L(A)$ using these bridge states. Since there are at most m bridge states in A, we can compute a prime decomposition of $L(A)$ after a finite number of decompositions at bridge states.

Note that the first three requirements in Definition 4 are based on the structural properties of A. We call a state that satisfies the first three requirements a *candidate bridge state*. We first compute all candidate bridge states and, then we determine whether or not each candidate bridge state satisfies the fourth requirement in Definition 4.

Proposition 4 (Han et al. [8]). *Given a minimal DFA $A = (Q, \Sigma, \delta, s, f)$, we can identify all candidate bridge states in $O(|Q| + |\delta|)$ worst-case time.*

Let $\mathcal{C}_B$ denote a set of candidate bridge states that we compute from an overlap-free DFA A based on Proposition 4. Then, for each state $b_i \in \mathcal{C}_B$, we check whether or not two subautomata A_1 and A_2 partitioned at b_i are overlap-free. If both A_1 and A_2 are overlap-free, then L is not prime and, thus, we decompose L into $L(A_1) \cdot L(A_2)$ and continue to check and decompose for each A_1 and A_2, respectively, using the remaining states in $\mathcal{C}_B \setminus \{b_i\}$.

Theorem 7. *Given a minimal DFA $A = (Q, \Sigma, \delta, s, f)$ for an overlap-free regular language, we can determine primality of $L(A)$ in $O(m^3)$ worst-case time*

and compute a prime decomposition for $L(A)$ in $O(m^4)$ worst-case time, where $m = |Q|$.

The algorithm for computing a prime decomposition for $L(A)$ in Theorem 7 looks similar to the algorithm for the infix-free regular language case studied by Han et al. [8]. However, there is one big difference between these two algorithms because of the different closure properties of two families: In fact, Han et al. [8] speeded up their algorithm by linear factor based on the fact that infix-free languages are closed under catenation whereas overlap-free languages are not closed as shown in Theorem 5.

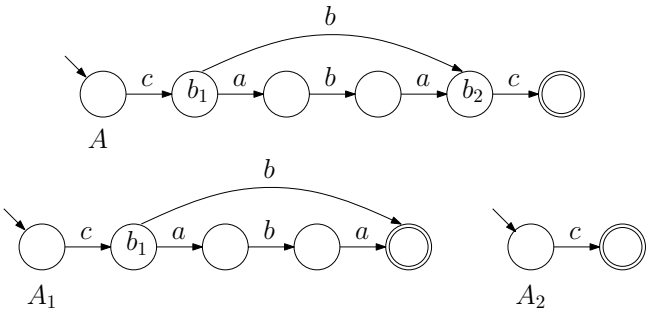

Fig. 4. States b_1 and b_2 are bridge states for A. However, once we decompose A at b_2, then b_1 is no longer a bridge state in A_1 since b_1 now violates the fourth requirement in Definition 4. Similarly, if we decompose A at b_1, then b_2 is not a bridge state.

We observe that a bridge state b_i of a minimal DFA A may not be a bridge state anymore if A is partitioned at a different bridge state b_j. See Fig. 4 for an example: It hints that the prime overlap-free decomposition might not be unique. Note that the prime prefix-free decomposition for a prefix-free regular language is unique [5] whereas the prime infix-free decomposition for an infix-free regular language is not unique [8]. Since overlap-free languages are a proper subfamily of prefix-free languages and a proper subfamily of infix-free languages, it is natural to examine the uniqueness of prime overlap-free decomposition. The following example demonstrates that the prime overlap-free decomposition is not unique.

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

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

References

[1] A. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science*, 255–300. The MIT Press, Cambridge, MA, 1990.

[2] J. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11:481–494, 1964.
[3] 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.
[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.
[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] S. Golomb, B. Gordon, and L. Welch. Comma-free codes. *The Canadian Journal of Mathematics*, 10:202–209, 1958.
[7] 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.
[8] 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.
[9] 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.
[10] Y.-S. Han and D. Wood. A new linearizing restriction in the pattern matching problem. In *Proceedings of FCT'05*, 552–562. Springer-Verlag, 2005. Lecture Notes in Computer Science 3623.
[11] Y.-S. Han and D. Wood. Outfix-free regular languages and prime outfix-free decomposition. In *Proceedings of ICTAC'05*, 96–109. Springer-Verlag, 2005. Lecture Notes in Computer Science 3722.
[12] Y.-S. Han and D. Wood. Shorter regular expressions from finite-state automata. In *Proceedings of CIAA'05*, 141–152. Springer-Verlag, 2005. Lecture Notes in Computer Science 3845.
[13] J. Hopcroft and J. Ullman. *Formal Languages and Their Relationship to Automata*. Addison-Wesley, Reading, MA, 1969.
[14] IEEE. *IEEE standard for information technology: Portable Operating System Interface (POSIX) : part 2, shell and utilities*. IEEE Computer Society Press, Sept. 1993.
[15] 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.
[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] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11:419–422, 1968.