Prefix-Free Regular-Expression Matching*

Yo-Sub Han, Yajun Wang, and Derick Wood

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

Abstract. We explore the regular-expression matching problem with respect to prefix-freeness of the pattern. We show that the prefix-free regular expression gives only linear number of matching substrings in the size of a given text. Based on this observation, we propose an efficient algorithm for the prefix-free regular-expression matching problem. Furthermore, we suggest an algorithm to determine whether or not a given regular language is prefix-free.

1 Introduction

In 1968, Thompson [11] introduced what became a classical automaton construction, the Thompson construction. It was used to find all matching strings from a text with respect to a given regular expression in the unix editor, `ed`. Subsequently, Aho [1] investigated the regular-expression matching problem as an extension of the keyword pattern matching problem [2], where the set of keywords is represented by a regular expression. Regular-expression matching has been adopted in many applications such as `grep`, `vi`, `emacs` and `perl`. For instance, with `grep`, we search for the last position of a matching string since the command outputs the line that contains the matched string.

Prefix-freeness is fundamental in coding theory; for example, Huffman codes are prefix-free sets. The advantage of prefix-free codes is that we can decode a given encoded string deterministically. Since codes are languages and prefix-free codes are a proper subfamily of codes, prefix-free regular languages are a proper subfamily of regular languages. Prefix-free regular languages have already been used to define *determinism* for generalized automata [6] and for expression automata [7].

The regular-expression matching problem has been well studied in the literature. Given a regular expression E and a text T, Aho [1] showed that we can determine whether or not there is a substring of T that is in $L(E)$ in $O(mn)$ time using $O(m)$ space, where m is the size of E and n is the size of T. Recently, Crochemore and Hancart [5] presented an algorithm to find all end positions

* Han and Wood were supported under the Research Grants Council of Hong Kong Competitive Earmarked Research Grant HKUST6197/01E and Wang was supported under the Research Grants Council of Hong Kong Competitive Earmarked Research Grant HKUST6206/02E.

A. Apostolico, M. Crochemore, and K. Park (Eds.): CPM 2005, LNCS 3537, pp. 298–309, 2005.

of matching substrings of T with respect to $L(E)$ in $O(mn)$ time using $O(m)$ space. Myers et al. [10] solved the problem of identifying start positions and end positions of all matching substrings of T that belong to $L(E)$ in $O(mn \log n)$ time using $O(m \log n)$ space. Clarke and Cormack [4] considered an interesting problem, the *shortest-match substring search*. Given a finite-state automaton A and a text T, identify all substrings of T that are accepted by A and also form an *infix-free set*. They showed that there are at most n matching substrings in T and they suggested an $O(kmn)$ worst-case running time algorithm using $O(m)$ space, where k is the maximum number of out-transitions from a state in A, m is the number of states and n is the size of T. (If we assume that A is a Thompson automaton, then $k = 2$.) In the regular-expression matching problem, there are a quadratic number of matching substrings of a given text in the worst-case. On the other hand, Clarke and Cormack [4] hinted that if an input regular expression is infix-free, then there are at most a linear number of matching substrings and it ensures a faster running time. Since the family of prefix-free regular languages is a proper subfamily of regular languages and a proper superfamily of infix-free regular languages, it is natural to investigate the prefix-free regular-expression matching problem. As far as we are aware, there does not appear to have been any prior consolidated effort to study the prefix-free regular-expression matching problem.

We want to find all $(start, end)$ positions of matching substrings; similar to the work of Myers et al. [10] and Clarke and Cormack [4]. We reexamine the regular-expression matching problem with this requirement and investigate the prefix-free regular-expression matching problem. Moreover, we suggest an algorithm to determine whether or not a given regular language L is prefix-free, where L is described by a nondeterministic finite-state automaton or by a regular expression. If L is represented by a deterministic finite-state automaton, then L is prefix-free if and only if there are no out-transitions from any final state in the given automaton [7].

In Section 2, we define some basic notation. We then, in Section 3, present an algorithm to identify all matching substrings of T with respect to a regular expression E based on the algorithm by Crochemore and Hancart [5]. The worst-case running time for the algorithm is $O(mn^2)$ using $O(m)$ space, where m is the size of E and n is the size of T. We also study the infix-free regular expression matching problem motivated by the shortest-match substring search problem. In Section 4, we examine the prefix-free regular-expression matching problem and propose an $O(mn)$ worst-case running time algorithm using $O(m)$ space. It implies that if E is prefix-free, then we can improve the total running time for the matching problem. In Section 5, we present a polynomial-time algorithm to determine whether or not a given regular language is prefix-free.

2 Preliminaries

Let Σ denote a finite alphabet of characters and Σ^* denote the set of all strings over Σ. A language over Σ is any subset of Σ^*. The character $\emptyset$ denotes the

empty language and the character λ denotes the null string. Given two strings x and y in Σ^*, x is said to be a *prefix* of y if there is a string w such that $xw = y$. Given a set X of strings over Σ, X is *prefix-free* if no string in X is a prefix of any other string in X. Given a string x, let x^R be the reversal of x, in which case $X^R = \{x^R \mid x \in X\}$.

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

We define a (regular) language L to be prefix-free if L is a prefix-free set. A regular expression E is prefix-free if $L(E)$ is prefix-free. In a similar way, we define suffix-free regular languages and regular expressions. We define L to be *infix-free* if, for all distinct strings x and y in L, x is not a substring of y and y is not a substring of x. Then, a regular expression E is infix-free if $L(E)$ is infix-free. The size $|E|$ of a regular expression E is the total number of character appearances.

3 Regular-Expression Matching

The regular-expression matching problem is an extension of the pattern matching problem, for which a pattern is given as a regular expression E. If $L(E)$ consists of a single string, then the problem is the string matching problem [3, 9] and if $L(E)$ is a finite language, then we obtain the multiple keyword matching problem [2].

Definition 1. *Given a regular expression E and a text $T = w_1 w_2 \cdots w_n$, the regular-expression matching problem is to identify all matching substrings of T that belong to $L(E)$.*

We answer the regular-expression matching problem by using Thompson automata [11]. We give the inductive construction of Thompson automata in Fig. 1. Note that a state q in a Thompson automaton has at most two in-transitions and at most two out-transitions. Furthermore, if q has a transition (q, a, r) and $a \in \Sigma$, then state r has at most two out-transitions that are null.

Given a regular expression E over Σ, we prepend Σ^* to E; thus, allowing matching to begin at any position in T. We construct the Thompson automaton A for $\Sigma^* E$ and process T using ExpressionMatching defined in Fig. 2.

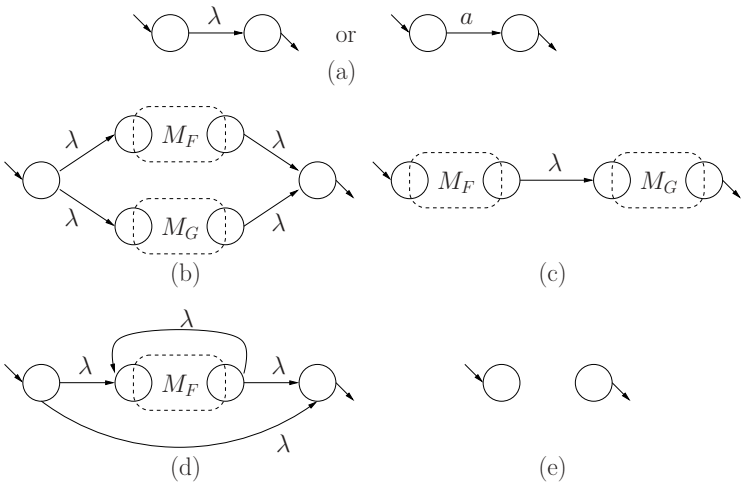

Fig. 1. The Thompson construction. Let E, F and G be regular expressions and M_F and M_G be the corresponding Thompson automata for F and G, respectively. (a) $E = a + \lambda$, (b) $E = F + G$, (c) $E = F \cdot G$, (d) $E = F^*$ and (e) $E = \emptyset$.

ExpressionMatching (A, T)

$Q = null(\{s\})$
if $f \in Q$ **then output** λ
for j=1 **to** n
  $Q = null(goto(Q, w_j))$
  if $f \in Q$ **then output** j

Fig. 2. A regular-expression matching procedure for a given Thompson automaton $A = (Q, \Sigma, \delta, s, f)$ and a text $T = w_1 \cdots w_n$. The procedure reports all the end positions of matching substrings of T.

Note that ExpressionMatching was already considered by Crochemore and Hancart [5], which is a modified version of Aho's algorithm [1].

ExpressionMatching (EM) in Fig. 2 has two sub-functions: $null(Q)$ and $goto(Q, w_j)$. The function $null(Q)$ computes all states in A that can be reached from a state in the set Q of states by null transitions. We use depth-first traversal to compute $null(Q)$ since A is essentially a graph. We traverse A using only null transitions. If we reach a state q that has already been visited by another null transition, then we stop exploring from q. Therefore, each state in A is visited at most twice since a state in a Thompson automaton has at most two in-transitions. Thus, the $null(Q)$ step takes $O(m)$ time in the worst-case, where m is the size of A. Now $goto(Q, w_j)$ gives all states that can be reached from a state in Q by a transition with w_j, the current input character. We only have to check whether a state in Q has an out-transition with w_j on it since the tar-

get state of the current state can have only null out-transitions. Therefore, the $goto(Q, w_j)$ step takes $O(|Q|)$ time, which is $O(m)$ in the worst-case. Overall, EM runs in $O(mn)$ worst-case time using $O(m)$ space.

Note that EM reports all the last positions of matching substrings of T with respect to A. It is, in some applications like `grep`, sufficient to have the end positions of matching substrings. However, if we want to report exact positions of matching strings, then we have to read T from right to left for each end position to find the corresponding start positions. For example, we need seven reverse scans of T to find all matching substrings in Fig. 3.

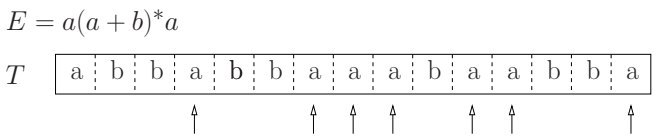

Fig. 3. An example of finding all end positions of T for a given regular expression E using EM. EM reports seven end positions indicated by "↑". There are, however, 28 matching substrings of T with respect to E and some matching substrings end at the same position.

We construct the Thompson automaton A' for E^R to find the start positions that correspond to the end positions we have already computed. For each end position j in T, we process $w_j \cdots w_2 w_1$ with respect to A' using EM to identify all corresponding start positions for j. In the worst-case, there are $O(n)$ end positions for matching substrings and we have to read T^R for each end position to find all corresponding start positions. A worst-case example is when $E = (a + b)^*$ and $T = abaaababababa \cdots aba$. Total running time for the regular-expression matching problem is $O(mn) + O(mn) \cdot O(n) = O(mn^2)$; that is (search all end positions) + [(find all corresponding start positions for each end position) × (the number of end positions)], using $O(m)$ space in the worst-case.

Theorem 1. *Given a regular expression E and a text T, we can identify all matching substrings of T that belong to $L(E)$ in $O(mn^2)$ worst-case time using $O(m)$ space, where m is the size of E and n is the size of T.*

Before we tackle the prefix-free regular-expression matching problem, we consider the simpler case of E being infix-free. Note that this problem is similar to, yet different from, the shortest-match substring search by Clarke and Cormack [4]. They were interested in reporting all matching substrings that form an infix-free set for a given (normal) regular expression and we are interested in the case when a given regular expression is strictly infix-free.

Theorem 2. *Given an infix-free regular expression E and a text T, we can identify all matching substrings of T that belong to $L(E)$ in $O(mn)$ worst-case time using $O(m)$ space, where m is the size of E and n is the size of T.*

A brief description of the algorithm for Theorem 2 is as follows: First, we find all end positions $P = \{p_1, p_2, \ldots, p_k\}$ of matching substrings in T using EM, where k is the number of matching substrings in T. Note that $k \leq n$ since $L(E)$ is infix-free[1]. Then, we construct the Thompson automaton A' for $\Sigma^* E^R$ and find all the end positions $P^R = \{q_1, q_2, \ldots, q_k\}$ of substrings of T^R with respect to A' using EM. Note that P^R also has k positions. We assume that both P and P^R are sorted in ascending order.

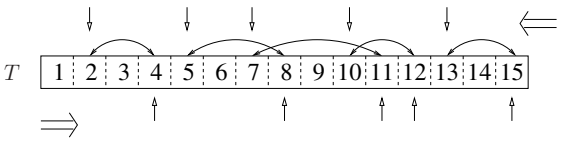

Fig. 4. An example of infix-free regular-expression matching. The upper arrows indicate P^R and the lower arrows indicate P. We output (2,4), (5,8), (7, 11), (10,12) and (13,15).

Since $L(E)$ is infix-free, no matching substring can be nested within any other matching substring. Therefore, once we have P^R and P, then we output (q_i, p_i) for $1 \leq i \leq k$, where $q_i \in P^R$ and $p_i \in P$. Fig. 4 illustrates this step when $P^R = \{2, 5, 7, 10, 13\}$ and $P = \{4, 8, 11, 12, 15\}$. Since we run EM twice to compute P and P^R and the output step from P and P^R takes only linear time in the size of P, which is $O(n)$ in the worst-case, the total complexity is $O(mn)$ time using $O(m)$ space.

Since all infix-free (regular) languages are prefix-free (regular) languages it is natural to investigate more general case, the prefix-free regular-expression matching problem.

4 The Prefix-Free Regular-Expression Matching Problem

We now consider the regular-expression matching problem for prefix-free regular expressions.

Lemma 1. *Given a prefix-free regular expression E and a text T, there are at most n matching substrings that belong to $L(E)$, where n is the size of T.*

Proof. Assume that the number of matching substrings is greater than n. Then, by the pigeonhole principle, there must be two distinct substrings s_1 and s_2 that start from the same position in T. We assume without loss of generality that s_1 is shorter than s_2, which, in turn, implies that s_1 is a prefix of s_2 — a contradiction. Therefore, there are at most n matching substrings. □

[1] This is a special case of Lemma 1 in Section 4 since an infix-free language is also a prefix-free language.

We design an algorithm for the prefix-free regular-expression matching problem. First, we find all end positions of matching substrings of $T = w_1 \cdots w_n$ using EM with respect to E. Let $P = \{p_1, p_2, \ldots, p_k\}$ be the set of end positions of matching substrings, where $k \leq n$ is the number of matching substrings. Then, we need to search for the corresponding start position of each end position in P. We construct the Thompson automaton $A' = (Q, \Sigma, \delta', s', f')$ for E^R and scan $T^R = w_n \cdots w_1$ starting from the last position p_k in P. Note that E^R is suffix-free.

Definition 2. *Given a position $j \in P$ and a current input position i in T^R in EM, where $i < j$, we define $\mathcal{Q}_j$ to be the set of states such that there is a path from s' to each state in $\mathcal{Q}_j$ that spells out the substring $w_j w_{j-1} \cdots w_i$ of T^R in A'.*

The notion of a set of reachable states in Definition 2 is not new. We already used it in EM in Fig. 2 implicitly. We now maintain sets of reachable states in A' for all end positions in P.

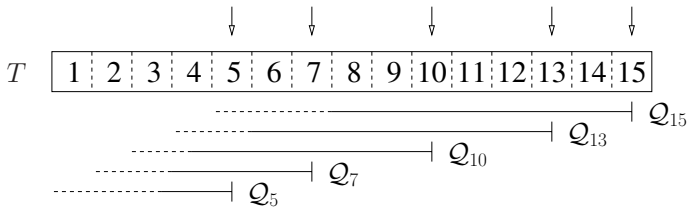

Fig. 5. Once we find the set P of all end positions, then we read T^R and maintain sets of reachable states for P in EM. For example, we have $\mathcal{Q}_{15}$, $\mathcal{Q}_{13}$ and $\mathcal{Q}_{10}$ when reading w_8 of T^R.

We process T^R from the last position in P with respect to A' using EM. If $\mathcal{Q}_j$, for some position $j \in P, 1 \leq j \leq n$, contains the final state f' of A' when reading w_i of T^R, where $i < j$, then we output the matching substring position (i, j) and continue to read the remaining input of T^R. Since each end position in P has exactly one corresponding start position, we can delete $\mathcal{Q}_j$ from our data structure after identifying a matching substring. However, we may meet another end position $j-1$ before finding the start position for $\mathcal{Q}_j$ and need to maintain another set $\mathcal{Q}_{j-1}$ of reachable states for position $j-1$ in P. For example, we may have sets $\mathcal{Q}_{15}$, $\mathcal{Q}_{13}$ and $\mathcal{Q}_{10}$ when we are reading w_8 of T^R in Fig. 5. We have to maintain k sets of reachable states and update k sets simultaneously while reading each character for T^R in the worst-case. As proved in Section 3, the size of each set of reachable states can be $O(m)$ in the worst-case. Therefore, we need $O(kmn)$ time and $O(km)$ space to answer the prefix-free regular-expression matching problem, which is $O(mn^2)$ time and $O(mn)$ space in the worst-case. We now show that we can reduce the complexity to $O(mn)$ time and $O(m)$ space because of prefix-freeness of E.

Lemma 2. *If a state r in A' is reached from two different states p and q, where $p \in \mathcal{Q}_i$ and $q \in \mathcal{Q}_j$, when reading a character w_h in EM, where $h \leq i < j$, then both paths from p and q via r cannot reach f' by reading any prefix of the remaining input in EM.*

Proof. Note that it is not possible that one path reaches f' while the other path does not since both paths must share the same path after reading w_h and arriving at r. Assume that both paths reach f' after reading some prefix $w_{h-1} \cdots w_g$ of the remaining input from r, where $g < h$. It implies that both strings $w_i \cdots w_h \cdots w_g$ and $w_j \cdots w_h \cdots w_g$ belong to $L(E^R)$. Observe that $w_i \cdots w_g$ is a suffix of $w_j \cdots w_g$. It contradicts the suffix-freeness of E^R. Therefore, if r is reached by two states from different sets of reachable states, then both paths from p and q via r cannot reach f' by reading any prefix of the remaining input in EM. □

Lemma 2 demonstrates that if a state r in A' is reached from two different sets of reachable states when reading a character w_h in EM, then r should not belong to the both sets since both paths cannot reach the final state by reading any prefix of the remaining input. Therefore, each state in A' appears in at most one reachable set and any two sets of reachable states are disjoint from each other as a result of reading a character in T^R. Since any state r in a Thompson automaton has at most two in-transitions, r can be visited at most twice in EM and we need at most $O(m)$ time to update all sets of reachable states simultaneously at each step to read a character in EM. Note that we use only $O(m)$ space.

Theorem 3. *Given a prefix-free regular expression E and a text T, we can identify all matching substrings of T that belong to $L(E)$ in $O(mn)$ worst-case time using $O(m)$ space, where $m = |E|$ and $n = |T|$.*

5 Prefix-Free Regular Languages

A regular language is represented by a finite-state automaton or described by a regular expression. We present algorithms to determine whether or not a given regular language L is prefix-free based either on finite-state automata or on regular expressions. Note that if a finite-state automaton A is deterministic, then $L(A)$ is prefix-free if and only if A is non-exiting.

We first consider the representation of a regular language L by a nondeterministic finite-state automaton (NFA) A. If A has any out-transitions from a final state, then we immediately know that $L(A)$ is not prefix-free; A must be non-exiting to be prefix-free. If A is non-exiting and has several final states, then all final states are equivalent and, therefore, merged into a single final state.

Given an NFA $A = (Q, \Sigma, \delta, s, f)$, we assign a unique number for each state from 1 to m, where m is the number of states in Q. Assume 1 denotes s and m denotes f. We use q_i, for $1 \leq i \leq m$, to denote the corresponding state in A. If $L(A)$ is not prefix-free, then there are two strings s_1 and s_2 accepted by A and s_1

is a prefix of s_2. It implies that there are two distinct paths in A that spell out s_1 and s_2 and these two paths spell out the same prefix s_1. For example, in Fig. 6, two paths for $s_1 = abcbb$ and $s_2 = abcbbab$ are different although they have the same subpath for ab in common. If the path for s_1 is a subpath of the path for s_2, then it implies that there is another final state that has an out-transition. This contradicts that A is non-exiting.

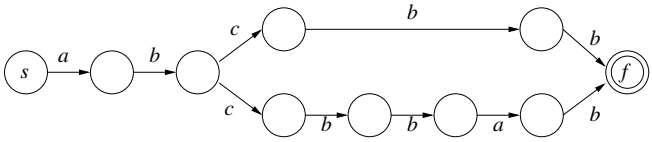

Fig. 6. Two distinct paths for *abcbb* and *abcbbab*.

We introduce the *state-pair graph* to capture the situation when two distinct paths in A spell out s_1 and s_2 and s_1 is a prefix of s_2.

Definition 3. *Given a finite-state automaton $A = (Q, \Sigma, \delta, s, f)$, we define the state-pair graph $G_A = (V, E)$, where V is a set of nodes and E is a set of edges, as follows:*

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

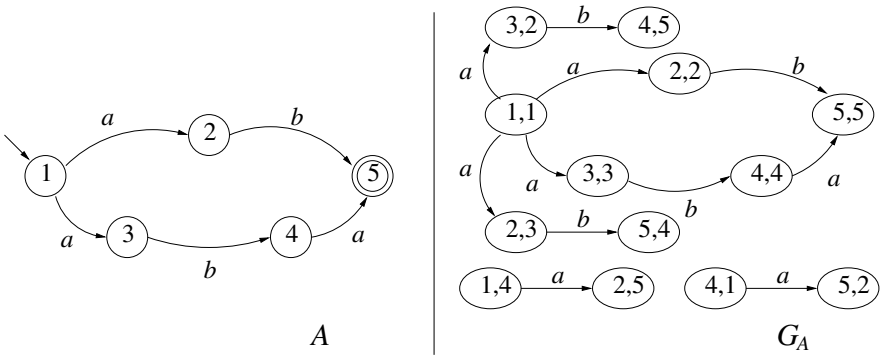

Fig. 7. An example of a state-pair graph G_A for a given finite-state automaton A. We omit all nodes that have no out-transitions in G_A.

Fig. 7 illustrates the state-pair graph for a given finite-state automaton A; $L(A) = \{ab, aba\}$ is not prefix-free because the prefix ab appears on the path from $(1, 1)$ to $(5, 4)$ in G_A.

Theorem 4. *Given a finite-state automaton A, $L(A)$ 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.*

Proof. $\Longrightarrow$ Assume that there is a path from $(1,1)$ to (m,j) that spells out a string x in G_A. Then, by the definition of state-pair graphs, there should be two distinct paths, one of which is from q_1 to q_m and the other is from q_1 to q_j in A, where $q_m = f$ and $q_j \neq f$. Note that both paths spell out x in A. Since A has only useful states, state q_j must have an out-transition (q_j, z_1, q_k), where $z_1 \in \Sigma$. Then, there is a transition sequence $(q_j, z_1, q_k), (q_k, z_2, q_{k+1}), \ldots, (q_{k+l-2}, z_l, q_m)$, for some $l \geq 1$, such that $z_1 \cdots z_l = z$. In other words, A accepts both x and xz — a contradiction. Therefore, if $L(A)$ is prefix-free, then there is no path from $(1,1)$ to (m,j) in G_A.

$\Longleftarrow$ Assume that $L(A)$ is not prefix-free. Then, there are two strings x and y and x is a prefix of y in $L(A)$. Since A is non-exiting, there should be two distinct paths that spell out x and y in A. Since x is a prefix of y, these two paths in A make a path from $(1,1)$ to (m,j), where $j \neq m$ in G_A — a contradiction. Thus, if there is no path from $(1,1)$ to (m,j) for any $j \neq m$ in G_A, then $L(A)$ is prefix-free. □

Let us consider the complexity of the state-pair graph $G_A = (V,E)$ for a given finite-state automaton $A = (Q, \Sigma, \delta, s, f)$. It is clear that $V = |Q|^2$ from Definition 3. Let δ_i denote the set of out-transitions from state q_i in A. Then, $|\delta| = \sum_{i=1}^{m} |\delta_i|$, where $m = |Q|$. Since a node (i,j) in G_A can have at most $|\delta_i| \times |\delta_j|$ out-transitions, $|E| = \sum_{i,j=1}^{m} |\delta_i| \times |\delta_j| \leq |\delta|^2$. Therefore, the complexity of G_A is $|Q|^2$ nodes and $|\delta|^2$ edges.

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

if A is not non-exiting
  then return no
Construct $G_A = (V,E)$ 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

Fig. 8. A prefix-freeness checking algorithm for a given automaton.

The sub-function DFS$((1,1))$ in Prefix-Freeness (PF) in Fig. 8 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 in the worst-case and DFS takes $(|V| + |E|)$ time. Therefore, the total running time for PF is $O(|Q|^2 + |\delta|^2)$.

Theorem 5. *Given a finite-state automaton $A = (Q, \Sigma, \delta, s, f)$, we can determine whether or not $L(A)$ is prefix-free in $O(|Q|^2 + |\delta|^2)$ worst-case time using PF.*

Since $O(|\delta|) = O(|Q|^2)$ in the worst-case for NFAs, the running time of PF is $O(|Q|^4)$ in the worst-case. On the other hand, if a language is described by a regular expression, then we can choose a construction for finite-state automata that improves the worst-case running time. Since the complexity of the state-pair graph depends on the number of states and the number of transitions of a given automaton, we need a finite-state automata construction that results in fewer states and transitions. One possibility is to use the Thompson construction [11].

Given a regular expression E for L, the Thompson construction shown in Fig. 1 takes $O(|E|)$ time and the resulting Thompson automaton has $O(|E|)$ states and $O(|E|)$ transitions [8]; namely, $|Q| = |\delta| = O(|E|)$. Even though Thompson automata are a subfamily of NFAs, they define all regular languages. Therefore, we can use Thompson automata to determine prefix-freeness of a regular language given by a regular expression. Since Thompson automata have null transitions, we include the null transition case to construct the edges for a state-pair graph as follows:

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

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

Theorem 6. *Given a regular expression E, we can determine whether or not $L(E)$ is prefix-free in $O(|E|^2)$ worst-case time.*

Proof. We construct the Thompson automaton A_T for E. Hopcroft and Ullman [8] showed that the number of states in A_T is $O(|E|)$ and also the number of transitions, $|Q| = |\delta| = O(|E|)$. Thus, we construct the state-pair graph based on the new construction that includes null transitions and determine whether or not there is a path from $(1,1)$ to (m,j) for some $j \neq m$ in $O(|E|^2)$ time using PF. □

6 Conclusions

We have investigated the regular-expression, the infix-free regular-expression and the prefix-free regular-expression matching problems. We have shown that the regular-expression matching problem can be solved in $O(mn^2)$ time using $O(m)$ space based on the algorithm of Crochemore and Hancart [5]. Whereas, we observed that the infix-free regular-expression matching problem can be solved in $O(mn)$ time using $O(m)$ space. We have extended the matching problem for a more general case, the prefix-free regular-expression matching problem and proved that the prefix-free regular-expression matching problem can also be solved in $O(mn)$ worst-case time using $O(m)$ space.

Furthermore, we have shown that we can determine whether or not $L(A)$ is prefix-free for a given NFA $A = (Q, \Sigma, \delta, s, f)$ in $O(|Q|^2 + |\delta|^2)$ worst-case time

based on the state-pair graph defined in Section 5. Finally, if a language L is described by a regular expression E, then we can improve the running time to $O(|E|^2)$ using the Thompson construction [11].

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. A. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18:333–340, 1975.
3. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.
4. 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.
5. M. Crochemore and C. Hancart. Automata for matching patterns. In G. Rozenberg and A. Salomaa, editors, *Linear modeling: background and application*, volume 2 of *Handbook of Formal Languages*, 399–462. Springer-Verlag, 1997.
6. D. Giammarresi and R. Montalbano. Deterministic generalized automata. *Theoretical Computer Science*, 215:191–208, 1999.
7. Y.-S. Han and D. Wood. The generalization of generalized automata: Expression automata. In *Proceedings of CIAA'04*, 156–166. Springer-Verlag, 2004. Lecture Notes in Computer Science 3317.
8. J. Hopcroft and J. Ullman. *Formal Languages and Their Relationship to Automata*. Addison-Wesley, Reading, MA, 1969.
9. D. Knuth, J. Morris, Jr., and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.
10. E. W. Myers, P. Oliva, and K. S. Guimãraes. Reporting exact and approximate regular expression matches. In *Proceedings of CPM'98*, 91–103. Springer-Verlag, 1998. Lecture Notes in Computer Science 1448.
11. K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11:419–422, 1968.