Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science 389 (2007) 307–317
Theoretical Computer Science
www.elsevier.com/locate/tcs

Prefix-free regular languages and pattern matching☆

Yo-Sub Han[a,*], Yajun Wang[b], Derick Wood[b]

[a] *Intelligence and Interaction Research Center, Korea Institute of Science and Technology, P.O. Box 131, Cheongnyang, Seoul, Republic of Korea*
[b] *Department of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong*

Received 5 July 2006; received in revised form 1 October 2007; accepted 6 October 2007

Communicated by M. Crochemore

Abstract

We explore the regular-expression matching problem with respect to prefix-freeness of the pattern. We prove that a prefix-free regular expression gives only a 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.

Keywords: String pattern matching; Regular-expression matching; Prefix-free regular languages; Pruned prefix-free languages

1. Introduction

In 1968, Thompson [13] introduced what became a classical automaton construction, the Thompson construction. It was used to find all matching strings in 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 [7] and for expression automata [8].

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 [6]

☆ A preliminary version appeared in *Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching*, CPM 2005. Part of this research was carried out while Han was in HKUST.

* Corresponding author. Tel.: +82 2 958 5608; fax: +82 2 958 5649.
E-mail addresses: emmous@kist.re.kr (Y.-S. Han), yalding@cs.ust.hk (Y. Wang), dwood@cs.ust.hk (D. Wood).

0304-3975/$ - see front matter
doi:10.1016/j.tcs.2007.10.017

presented an algorithm to find all end positions of matching substrings of T with respect to $L(E)$ in $O(mn)$ time using $O(m)$ space. Myers et al. [12] 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 [5] considered an interesting problem, the *shortest-match substring search*: Given a finite-state automaton (FA) A and a text T, identify all substrings of T that are accepted by A and also give an *infix-free set*. They showed that there are at most n matching substrings in T and they suggested an $O(kmn)$ 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 [5] 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. [12] and Clarke and Cormack [5]. 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 (NFA) or by a regular expression. If L is represented by a deterministic finite-state automaton (DFA), then L is prefix-free if and only if there are no out-transitions from any final state in the given automaton [8].

In Section 2, we define some basic notions. Then, in Section 3, we 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 [6]. 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. We also consider the problem of computing prefix-free regular languages from NFAs, which are not prefix-free, based on the structural properties of FAs.

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 $X^R = \{x^R \mid x \in X\}$.

An FA A is specified by a tuple $(Q, \Sigma, \delta, s, F)$, where Q is a finite set of states, Σ is an input alphabet, $\delta \subseteq Q \times \Sigma \times Q$ 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 an FA 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.

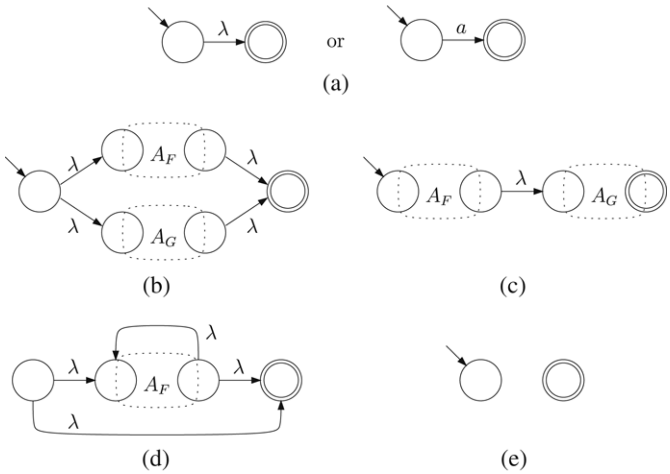

Fig. 1. The Thompson construction. Let E, F and G denote regular expressions and A_F and A_G denote the corresponding Thompson automata of F and G, respectively. (a) $E = \lambda + a$, (b) $E = F + G$, (c) $E = F \cdot G$, (d) $E = F^*$ and (e) E is empty.

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 [4, 11] 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 [13]. We give an inductive construction of Thompson automata in Fig. 1. From the construction, we observe the following properties.

Observation 2. *In a Thompson automaton,*

(1) *a state q has at most two in-transitions and at most two out-transitions.*
(2) *if q has an out-transition (q, a, r) and $a \in \Sigma$, then the target state r has at most two out-transitions and its out-transitions are always null-transitions.*

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 (EM) defined in Fig. 2. Note that ExpressionMatching was already considered by Crochemore and Hancart [6], which is a modified version of Aho's algorithm [1].

ExpressionMatching (A, T)

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

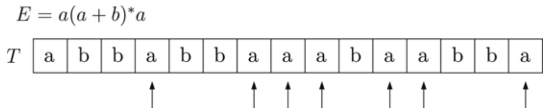

Fig. 3. An example of finding all end positions of T for a given regular expression E using EM. EM reports seven end positions indicated by "↑". There are, however, 28 matching substrings of T with respect to E and some matching substrings end at the same position.

EM in Fig. 2 has two sub-functions: $null(X)$ and $goto(X, w_j)$. The function $null(X)$ computes all states in A that can be reached from a state in the set X of states by null-transitions. We use depth-first traversal to compute $null(X)$ 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 $null(X)$ step takes $O(m)$ time in the worst-case, where m is the size of A. Now $goto(X, w_j)$ gives all states that can be reached from a state in X by a transition with w_j, the current input character. We only have to check whether a state in X has an out-transition with w_j on it since the target state of the current state can have only null out-transitions by Observation 2. Therefore, the $goto(X, w_j)$ step takes $O(m)$ time. 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.

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 $T = abaaabababa \cdots aba$. 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 3. *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 [5]. 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 4. *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.*

Proof. A brief description of an algorithm for Theorem 4 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 in T^R with respect to A' using EM. Since EM reads T character by character from left to right, we can keep P in ascending order without running an additional sorting procedure. We now have P and P^R that are sorted in ascending order.

Since $L(E)$ is infix-free, no matching substring can be nested within another matching substring. Otherwise, it violates infix-freeness. Therefore, once we have P^R and P, 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 with $O(m)$ space. □

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

[1] This is a special case of Lemma 5 in Section 4 since an infix-free language is also a prefix-free language.

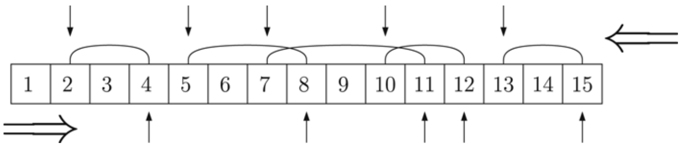

Fig. 4. An example of an infix-free regular-expression matching. The upper arrows indicate P^R and the lower arrows indicate P. We output (2, 4), (5, 8), (7, 11), (10, 12) and (13, 15).

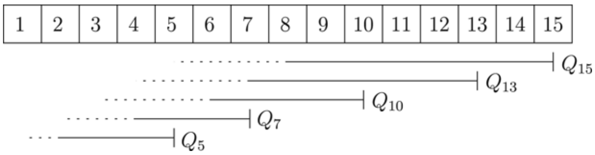

Fig. 5. Once we find the set P of all end positions, then we read T^R and maintain sets of reachable states for P in EM. For example, we have $\mathcal{Q}_{15}$, $\mathcal{Q}_{13}$ and $\mathcal{Q}_{10}$ when reading w_8 of T^R.

4. The prefix-free regular-expression matching problem

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

Lemma 5. *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. □

Before we design an efficient algorithm for the prefix-free regular-expression matching problem, we explore an implication of Lemma 5. Given a regular expression pattern E and a text T, there can be at most n^2 matching substrings in T with respect to E in the worst-case. For example, $E = (a + b)^*$ and $T = abbaabaaba \cdots baba$ 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. There are two well-known *linearizing restrictions*: The *longest-match* rule, which is a generalization of the leftmost longest-match rule of IEEE POSIX [10] and the *shortest-match substring search* rule of Clarke and Cormack [5]. These two previous rules [5,10] define what to output from a given text and a pattern. Thus they give the different results for the same text and the pattern. On the other hand, Lemma 5 shows that if we use a prefix-free pattern, then we can always guarantee a linear number of matching substrings. In other words, we can achieve the linearizing restrictions by using prefix-free patterns. Furthermore, it would be an interesting task to characterize the family of patterns that guarantees the linear number of matching substrings, which would be a superset of the family of prefix-free patterns.

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 6. 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 6 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.

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 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 Q_j and need to maintain another set Q_{j-1} of reachable states for position $j-1$ in P. For example, we may have sets Q_{15}, Q_{13} and 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 the prefix-freeness of E.

Lemma 7. *If a state r in A' is reached from two different states p and q, where $p \in Q_i$ and $q \in Q_j$, 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 7 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 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 8. *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

5.1. Decision problem of prefix-freeness

A regular language is represented by an FA 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 FAs or on regular expressions. Note that if an FA 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 an 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 that 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.

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 9. Given an FA $A = (Q, \Sigma, \delta, s, f)$, 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:

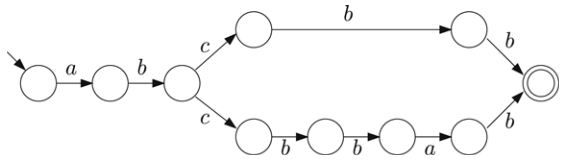

Fig. 6. Two distinct paths for *abcbb* and *abcbbab*.

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

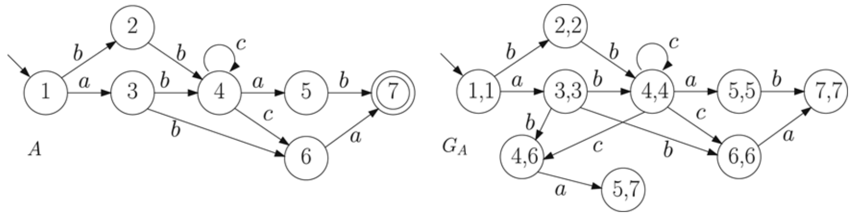

Fig. 7. An example of a state-pair graph G_A for a given FA A. We omit all nodes that are unreachable from node $(1, 1)$ in G_A.

Fig. 7 illustrates the state-pair graph for a given FA A; $L(A)$ is not prefix-free since A accepts both *aba* and *abab*. Note that the prefix *aba* appears on the path $(1, 1) \rightarrow (3, 3) \rightarrow (4, 6) \rightarrow (5, 7)$ in G_A.

Theorem 10. *Given an FA 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 FA $A = (Q, \Sigma, \delta, s, f)$. It is clear that $V = |Q|^2$ from Definition 9. 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.

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 11. *Given an FA* $A = (Q, \Sigma, \delta, s, F)$*, 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 FAs 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 an FA construction that results in fewer states and transitions. One possibility is to use the Thompson construction [13].

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 [9]; namely, $|Q| = |\delta| = O(|E|)$. Even though

```
Prefix-Freeness(A = (Q, Σ, δ, s, F))

if A is not non-exiting
    then return no
if |F| ≥ 2
    then merge all final states of F into a single final state
Construct G_A = (V, E) from A
DFS((1, 1)) in G_A
if we meet a node (m, j) for some j, j ≠ m
    then return no

return yes
```

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

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:

$V = \{(i, j) \mid q_i \text{ and } q_j \in Q\}$ 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 12. *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 [9] 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. □

5.2. *Pruned prefix-free regular languages*

Let us consider the problem for computing a prefix-free subset of a given regular language. There are two main methods for constructing prefix-free subsets of given languages. One is suggested by Yu [14].

Definition 13 (*Yu [14]*). Given a regular language L, we define

$\min(L) = \{w \in L \mid \text{there is no } x \in L \text{ such that } x \text{ is a prefix of } w, \text{ where } x \neq w\}$.

Note that if L is regular, then $\min(L)$ is also regular.

He also presented an algorithm to compute $\min(L)$ when L is given by a DFA. By definition, $\min(L)$ is a prefix-free subset of L. We call $\min(L)$ the *pruned prefix-free language* of L. The related method is that, given a language L, $L' = L \setminus L{\cdot}\Sigma^+$ is a prefix-free subset of L [3]. Observe that $\min(L) = L'$.

We now design an algorithm to compute the pruned prefix-free regular language from a given regular language based on state-pair graphs.

Proposition 14. *Given a regular language L, the pruned prefix-free language of L is unique.*

Proof. The proof is straightforward from Definition 13. □

The example in Fig. 7 shows a part of the state-pair graph for a given FA A, where each node is reachable from node (1,1) and $L(A) = L((bb + ab)c^*(ab + ca) + aba)$. Note that $L(A)$ is not prefix-free since A accepts aba ($1 \to 3 \to 6 \to 7$) and $abab$ ($1 \to 3 \to 4 \to 5 \to 7$). G_A identifies this common prefix aba that is spelled out by a path $(1, 1) \to (3, 3) \to (4, 6) \to (5, 7)$ as shown in G_A in Fig. 7, where $m = 7$. Note that there are sometimes more than one such path in G_A. For example, there is a path $(1, 1) \to (3, 3) \to (4, 4) \to (4, 6) \to (5, 7)$ that spells out $abca$ in G_A, which is a prefix of $abcab$, where A accepts both $abca$ and $abcab$.

We define the language specified by G_A as follows: we make node $(1, 1)$ the start state and node (j, m), for $j \neq m$, a final state. Then, G_A is an FA. Let $L(G_A)$ be the regular language defined from G_A. Note that if a string w is accepted by G_A, then it is also accepted by A. Furthermore, for such w, there must be a string that has w as a prefix in $L(A)$. Based on these observations, we obtain the following results.

Lemma 15. *Given an FA A and its state-pair graph G_A, where $L(A) \neq \emptyset$ and $L(A) \neq \{\lambda\}$,*

(1) $L(G_A) \subsetneq L(A)$.
(2) $L(G_A) = \emptyset$ *if and only if* $L(A)$ *is prefix-free.*

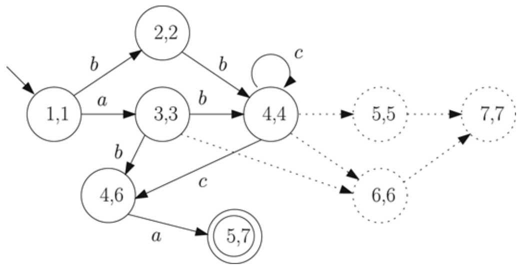

Fig. 9. An example of a regular language of G_A for the state-pair graph in Fig. 7. The dotted states are useless states.

Fig. 9 illustrates an example of $L(G_A)$. We now show how to compute the pruned prefix-free language of $L(A)$ using $L(G_A)$.

Theorem 16. *Given an FA A, the pruned prefix-free language of $L(A)$ is $L(A) \setminus (L(G_A)\cdot\Sigma^+)$, where G_A is the state-pair graph of A and + is the Kleene plus.*

Proof. Let L' denote $L(A) \setminus (L(G_A)\cdot\Sigma^+)$ and L_p denote the pruned prefix-free language of $L(A)$. We prove that $L' = L_p$. Note that L' is a subset of $L(A)$ by the definition.

(1) Let s be a string in L_p. It implies that s is in $L(A)$ and A accepts s. We only need to show that $s \notin (L(G_A)\cdot\Sigma^+)$ in order to prove that $s \in L'$. Assume that $s \in (L(G_A)\cdot\Sigma^+)$. It implies that a prefix $s'(\neq s)$ of s is spelled out by a path from $(1, 1)$ to (j, m), for $j \neq m$ and, thus, s' is also accepted by A. Since s' and s are both accepted by A, s cannot be in L_p — a contradiction. Therefore, if $s \in L_p$, then, $s \notin (L(G_A)\cdot\Sigma^+)$ and, thus, $s \in L'$.
(2) Let s be a string that is not in L_p. We want to prove that $s \notin L'$. If $s \notin L(A)$, then $s \notin L'$ since L' is a subset of $L(A)$. Let us consider the case when $s \in L(A)$. Assume that $s \in L'$. It means that $s \notin (L(G_A)\cdot\Sigma^+)$ and, therefore, none of prefixes of s can be accepted by A except itself. Then, by Definition 13, s must be in L_p — a contradiction. Therefore, if $s \notin L_p$, then $s \notin L'$.

Therefore, $L' = L_p$. □

The regular language of G_A in Fig. 9 is $L((bb + ab)c^*ca + aba)$ and, therefore, the pruned prefix-free language of $L(A)$ is

$$L(((bb + ab)c^*(ab + ca) + aba)) \setminus L((((bb + ab)c^*ca + aba)\Sigma^+)).$$

We extend Theorem 16 to other cases.

Given an FA $A = (Q, \Sigma, \delta, s, f)$, let A^R be $(Q, \Sigma, \delta^R, f, s)$ such that $(p, a, q) \in \delta^R$ if and only if $(q, a, p) \in \delta$, where p and $q \in Q$ and $a \in \Sigma$. Then, $L(A) = L(A^R)^R$. If $L(A)$ is prefix-free, then $L(A^R)$ is suffix-free. By Proposition 14, the pruned suffix-free language of $L(A)$ is also unique.

Proposition 17. *Given an FA $A = (Q, \Sigma, \delta, s, f)$, where A is non-returning, the pruned suffix-free language L_s of $L(A)$ is the reversal of the pruned prefix-free language of $L(A^R)$. Namely, $L_s = (L(A^R) \setminus (L(G_{A^R}) \cdot \Sigma^+))^R$.*

A language is bifix-free if and only if it is prefix-free and suffix-free. We obtain the following result for the pruned bifix-free language of $L(A)$.

Theorem 18. *Given an FA $A = (Q, \Sigma, \delta, s, f)$, where A is non-returning and non-exiting, the pruned bifix-free language L_b and the pruned infix-free language L_i of $L(A)$ are as follows:*

$$L_b = \{L(A) \setminus (L(G_A) \cdot \Sigma^+)\} \cap \{L(A^R) \setminus (L(G_{A^R}) \cdot \Sigma^+)\}^R$$

and

$$L_i = \{L(A) \setminus (\Sigma^+ \cdot L(G_A) \cdot \Sigma^+)\}.$$

Proof. Two conditions, non-returning and non-exiting, are necessary conditions for A to be bifix-free or infix-free. The proof is the combination of Theorem 16 and Corollary 17. The uniqueness of L_b and L_i can be proved by an argument similar to the proof of Proposition 14. □

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 [6]. 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 state-pair graphs. 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 [13].

We have also revisited the pruned prefix-free language and have proposed an algorithm for computing the pruned prefix-free language of a given NFA based on the structural properties of its state-pair graph.

Acknowledgements

We wish to thank the referee for the careful reading of the paper and the many valuable suggestions. Han was supported by the Research Grants Council of Hong Kong Competitive Earmarked Research Grant HKUST6197/01E and the KIST Tangible Space Initiative Grant. Wang was supported by the Research Grants Council of Hong Kong Competitive Earmarked Research Grant HKUST6206/02E. Wood was supported by the Research Grants Council of Hong Kong Competitive Earmarked Research Grant HKUST6197/01E.

References

[1] A. Aho, Algorithms for finding patterns in strings, in: J. van Leeuwen (Ed.), Algorithms and Complexity, in: Handbook of Theoretical Computer Science, vol. A, The MIT Press, Cambridge, MA, 1990, pp. 255–300.

[2] A. Aho, M. Corasick, Efficient string matching: An aid to bibliographic search, Communications of the ACM 18 (1975) 333–340.

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

[4] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Communications of the ACM 20 (10) (1977) 762–772.

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

[6] M. Crochemore, C. Hancart, Automata for matching patterns, in: G. Rozenberg, A. Salomaa (Eds.), Linear Modeling: Background and Application, in: Handbook of Formal Languages, vol. 2, Springer-Verlag, 1997, pp. 399–462.

[7] D. Giammarresi, R. Montalbano, Deterministic generalized automata, Theoretical Computer Science 215 (1999) 191–208.

[8] Y.-S. Han, D. Wood, The generalization of generalized automata: Expression automata, International Journal of Foundations of Computer Science 16 (3) (2005) 499–510.

[9] J. Hopcroft, J. Ullman, Formal Languages and Their Relationship to Automata, Addison-Wesley, Reading, MA, 1969.

[10] IEEE, IEEE Standard for Information Technology: Portable Operating System Interface (POSIX): Part 2, Shell and Utilities, IEEE Computer Society Press, Sept. 1993.
[11] D. Knuth, J. Morris Jr., V. Pratt, Fast pattern matching in strings, SIAM Journal on Computing 6 (1977) 323–350.
[12] E.W. Myers, P. Oliva, K.S. Guimãraes, Reporting exact and approximate regular expression matches, in: Proceedings of CPM'98, in: Lecture Notes in Computer Science, vol. 1448, Springer-Verlag, 1998, pp. 91–103.
[13] K. Thompson, Regular expression search algorithm, Communications of the ACM 11 (1968) 419–422.
[14] S. Yu, Regular languages, in: G. Rozenberg, A. Salomaa (Eds.), Word, Language, Grammar, in: Handbook of Formal Languages, vol. 1, Springer-Verlag, 1997, pp. 41–110.