Journal of Universal Computer Science, vol. 16, no. 5 (2010), 715-728
submitted:22/6/09, accepted: 12/2/10, appeared: 1/3/10 © J.UCS

On the Linear Number of Matching Substrings

Yo-Sub Han[1]
(Department of Computer Science, Yonsei University
Seoul 120-749, Republic of Korea
emmous@cs.yonsei.ac.kr)

Abstract: We study the number of matching substrings in the pattern matching problem. In general, there can be a quadratic number of matching substrings in the size of a given text. The linearizing restriction enables to find at most a linear number of matching substrings. We first explore two well-known linearizing restriction rules, the *longest-match* rule and the *shortest-match substring search* rule, and show that both rules give the same result when a pattern is an infix-free set even though they have different semantics. Then, we introduce a new linearizing restriction, the *leftmost non-overlapping match* rule that is suitable for find-and-replace operations in text searching, and propose an efficient algorithm for the new rule when a pattern is described by a regular expression. We also examine the problem of obtaining the maximal number of non-overlapping matching substrings.
Key Words: string pattern matching, regular expression searching, linearizing restriction, Thompson automata
Category: F.2, F.4.3

1 Introduction

People use regular expressions in many applications such as editors, programming languages and software systems in general. In vi or emacs, we use regular expressions for searching patterns in an editor and in UNIX command grep, we use regular expressions for searching patterns in text documents. Regular expressions are often used for describing a pattern in the pattern matching problem. If a pattern language L consists of a single string, then we have the string matching problem [Boyer and Moore(1977), Knuth et al.(1977)]. If L is a finite set of strings, then we have the multiple keyword matching problem [Aho and Corasick(1975)]. If L is a regular language given by a regular expression, then we have the regular-expression matching problem.

Researchers have investigated various regular-expression matching problems. Thompson [Thompson(1968)] presented the first regular expression matching algorithm for his UNIX editor, ed. Aho [Aho(1990)] suggested an algorithm that determines whether or not a text T has a matching substring with respect to a given regular expression pattern E in $O(mn)$ time using $O(m)$ space, where m is the size of E and n is the size of T. Later, Crochemore and Hancart [Crochemore and Hancart(1997)] extended Aho's algorithm to find all end

[1] The research is supported by the IT R&D program of MKE/IITA 2008-S-024-01.

positions of matching substrings of T with the same runtime and space complexity. Note that the back reference is very useful to describe a pattern and is often used in practice. For instance, we can use '$\backslash m$' to denote the mth matching substring in the pattern in emacs or perl. The back reference gives more expressive power; it can describe a pattern that is not regular [Câmpeanu et al.(2003)] and its membership problem is NP-complete [Aho(1990)].

We only consider the regular-expression pattern matching problem. It is, in applications such as grep, sufficient to obtain the end positions of matching substrings to report lines containing the matched substrings. However, we often need to find both the start positions and the end positions of matching substrings to replace or delete the matched strings. Myers et al. [Myers et al.(1998)] solved the problem of identifying start positions and end positions of matching substrings of T with respect to E in $O(mn \log n)$ time using $O(m \log n)$ space based on the four Russian technique [Aho et al.(1974)]. Recently, Han et al. [Han et al.(2007)] proposed another algorithm that runs in $O(mn^2)$ time using $O(m)$ space based on the algorithm of Crochemore and Hancart [Crochemore and Hancart(1997)].

Given a regular expression pattern E and a text T, there can be at most $\frac{n(n+1)}{2}$ matching substrings of T that belong to $L(E)$. For example, $E = (a+b)^+$ and $T = abbaabaaba \cdots baba$, where $|T| = n$. 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 suggested by IEEE POSIX [IEEE(1993)] and the *shortest-match substring search* rule suggested by Clarke and Cormack [Clarke and Cormack(1997)]. These two rules have different semantics and, therefore, may identify different matching substrings for same E and T.

In Section 2, we define some basic notions. We revisit the two linearizing restrictions in the literature and examine the relationship between them in Section 3. We observe that the two rules allow overlapping strings, which is not suitable for some applications, and we propose a new linearizing restriction, the *leftmost non-overlapping match* rule in Section 4. The new rule does not allow overlapping strings and guarantees a linear number of matching substrings. We demonstrate that the new rule is suitable for find-and-replace operations in text searching. Then, we apply the rule to the regular-expression matching problem and develop an algorithm for the problem in Section 5. The algorithm is based on the Thompson automata and it is easy to implement as similar algorithms [Aho(1990), Crochemore and Hancart(1997)]. We also investigate the problem of obtaining the maximal number of non-overlapping substrings.

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 over Σ, we define x to be a *prefix* of y if there exists $z \in \Sigma^*$ such that $xz = y$ and x to be a *suffix* of y if there exists $z \in \Sigma^*$ such that $zx = y$. Furthermore, we say that x is a *substring* or an *infix* of y if there are two strings u and v such that $uxv = y$. Given a string $x = x_1 \cdots x_n$, $|x|$ is the number of characters in x and $x(i, j) = x_i x_{i+1} \cdots x_j$ is the substring of x from position i to position j, where $i \leq j$. Given a set X of strings over Σ, we define X to be *infix-free* if no string in X is an infix of any other string in X. Given a string x, let x^R be the reversal of x, in which case $X^R = \{x^R \mid x \in X\}$. We define a (regular) language L to be infix-free if L is an infix-free set. A regular expression E is infix-free if $L(E)$ is infix-free. We can define prefix-free and suffix-free regular expressions and languages in a similar way.

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, $\delta : Q \times \Sigma \to 2^Q$ is a transition function, $s \in Q$ is the start state and $F \subseteq Q$ is a set of final states. If F consists of a single state f, then we use f instead of $\{f\}$ for simplicity. Let $|Q|$ be the number of states in Q and $|\delta|$ be the number of transitions in δ. Then, the size of A is $|A| = |Q| + |\delta|$. 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 assume that A has only *useful* states; that is, each state appears on some path from the start state to some final state.

A pattern is essentially a language. Given a pattern L and a text T, we define a string x to be a *matching substring* of T with respect to L if x is a substring of T and $x \in L$. The pattern matching problem is to identify all matching substrings of T with respect to a given pattern L. If L is represented by a regular expression E, then it is the regular-expression matching problem. If E is prefix-free, then it is the prefix-free regular-expression matching problem.

For complete background knowledge in automata theory, the reader may refer to Wood [Wood(1987)].

3 Linearizing Restrictions

In the pattern matching problem for a text T, matching substrings of T often overlap with or nest with other matching substrings. Moreover, in the worst-case, there are a quadratic number of matching substrings of T. To avoid these situations, researchers have designed methods to find a linear subset of the matching substrings while preserving specified properties for each matching string. We call such methods *linearizing restrictions*. There are two well-known linearizing restrictions in the matching problem.

3.1 Longest-match Rule

We first examine the leftmost longest match rule defined in the IEEE POSIX Standard [IEEE(1993)] as follows:

> *"The search is performed as if all possible suffixes of the string were tested for a prefix matching the pattern; the longest suffix containing a matching prefix is chosen, and the longest possible matching prefix of the chosen suffix is identified as the matching sequence."*

The rule reports the matching substring whose start position is leftmost and if there are several matching substrings at the same start position, then the longest string is identified. Since it is simple and easy to implement, the rule has been adopted in many tools such as regex, perl and tcl/tk. This rule reports at most one matching string.

The longest-match rule is a generalization of the leftmost longest match rule that performs a general search instead of identifying a single match string. The longest-match rule is defined as follows: Given a text T and a pattern L, we search for the longest matching prefix with respect to L from position i in T, for $1 \leq i \leq n$, where n is the size of T. Since there can be at most one longest matching prefix at each position, there are at most n matching substrings. Namely, the longest-match rule guarantees a linear number of matching substrings in the size of T and, therefore, is a linearizing restriction.

We consider the longest-match rule for the regular-expression matching problem. Given a regular expression E and a text T, we can find all start positions of matching substrings of T in $O(mn)$ time using $O(m)$ space based on the algorithm of Aho [Aho(1990)], where $m = |E|$ and $n = |T|$. Once we have all start positions, we search for the longest matching substring starting from each start position. For each state position, it takes $O(mn)$ time to find the longest matching substring. Therefore, we can solve the regular-expression matching problem using the longest-match rule in $O(kmn)$ time and $O(m)$ space, where k is the number of output matching substrings. Note that we can improve the running time by using the algorithm of Myers [Myers(1992)] with additional space.

3.2 Shortest-match Substring Search Rule

Clarke and Cormack [Clarke and Cormack(1997)] proposed a different linearizing restriction, the shortest-match substring search rule:

"Locate the set of shortest nonnested (but possible overlapping) strings that each match the pattern."

We can rephrase the rule as follows: Given a text T and a pattern L, identify all matching substrings of T whose infixes are not matching substrings; thus, the resulting set of matching substrings by this rule is an infix-free set. They demonstrated that the shortest-match substring search rule is appropriate for searching structured text such as SGML and XML.

Clarke and Cormack [Clarke and Cormack(1997)] showed that there are at most linear number of matching substrings in the size of T. Furthermore, they considered the case when a pattern is described by an FA A. Let l be the maximal number of out-transitions from a state in A, m be the number of states in A and n be the size of T. They proposed an $O(lmn)$ worst-case running time algorithm using $O(m)$ space. If we use the Thompson automata, which are often used in the regular-expression matching problem, then the running time is $O(mn)$ since l is at most 2 in the Thompson automata [Giammarresi et al.(2004), Thompson(1968)] Although the rule is simple and straightforward, the idea of this linearizing restriction is shown to be very useful in various cases.

3.3 Comparison of Two Linearizing Restrictions

Both the longest-match rule and the shortest-match substring search rule ensure that the number of matching substrings is linear in the size of T. However, the two rules have different semantics and, thus, may give different results for the same text and the same pattern. For example, if $T = abc$ and the pattern $L = \{a, abc\}$, then the longest-match rule outputs abc whereas the shortest-match substring search rule outputs a. Notice that both rules determine what to report for given an arbitrary text T and an arbitrary pattern L; namely, there are no restrictions on the pattern and on the text. On the other hand, Han et al. [Han et al.(2007)] investigated the case that a certain pattern L can have at most linear number of matching substrings for any input text[2]. Then, they observed that if L is prefix-free, then there can be at most n matching substrings of T because of the prefix-freeness of L and designed an efficient algorithm when L is given by a prefix-free regular expression. Hence, the following observation is immediate.

Proposition 1. *If L is prefix-free or suffix-free, then there are at most n matching substrings of T with respect to L, where n is the size of a given text T.*

[2] We cannot have an input text that guarantees a linear number of matching substrings in general; for instance, for a pattern Σ^+, any input text has more than a linear number of matching substrings.

Proposition 1 demonstrates that we can apply the linearizing restriction for patterns to obtain a linear number of matching substrings. Moreover, we may have the same matching substrings for the two different rules. This leads us to examine the linearizing restriction on patterns that can bridge the semantic difference between the longest-match rule and the shortest-match substring search rule.

Proposition 2. *Given a pattern L and a text T, if L is infix-free, then the longest-match rule and the shortest-match substring search rule give the same result. However, the converse does not hold.*

Proof. Assume that a set $S = \{s_1, \ldots, s_k\}$ is the set of matching substrings of T with respect to L, where k is the number of the matching substrings. Let n be the size of T. Since L is infix-free, there are at most n matching substrings and therefore $k \leq n$ [Han et al.(2007)]. By the definition of matching substrings, $s_i \in S$, for $1 \leq i \leq k$, must belong to L; it implies that S is a subset of L and, therefore, S is also infix-free. Thus, S is the output of the shortest-match substring search rule. Note that all strings in S start from different positions in T. (If any two strings s_i and s_j, for $1 \leq i \neq j \leq k$, start from the same position, then the shorter string must be a prefix of the longer string—a contradiction.) Since each string in S starts from a different position, all strings in S are identified as matching substrings by the longest-match rule. Therefore, S is the output of both rules.

We demonstrate that the converse does not hold with the following counter example; $T = abccbb$ and $L = \{abc, cc, ccc\}$. Both rules output abc, cc but L is not infix-free. □

Theorem 2 shows that we can eliminate the semantic difference between two rules by choosing an infix-free pattern. Moreover, if we know that a given pattern is infix-free, then an algorithm for one rule can be used for the other rule. For example, if a given pattern is an infix-free regular language, then we can use the algorithm of Clarke and Cormack [Clarke and Cormack(1997)] for the regular-expression matching problem with the longest-match rule. In additions, we can use an infix-free regular-expression matching algorithm [Han et al.(2007)] for both linearizing restriction rules; the algorithm reads T only twice using the Thompson automata.

4 Leftmost Non-overlapping Match Rule

In the pattern matching, two matching substrings of a given text T may overlap with each other. Assume that we want to find matching substrings and delete them from T. Then, only one of two overlapping matching substrings should be identified. For example, if T = BEFOREIGN and the pattern L = {BEFORE, FOREIGN}, then both BEFORE and FOREIGN are matching substrings. However, if we delete BEFORE from T, then FOREIGN does not exist anymore. Similar situations can happen if we do modification or replacement for matching substrings. Therefore, if two matching substrings overlap, then we select the string that starts ahead of the other string. Sometimes one matching substring is nested in the other matching substring. Even in this case, we choose the string that has an earlier start position. For example, if T = AUTOPIAN and L = {TO, UTOPIA}, then UTOPIA is identified even though TO is in L and shorter than UTOPIA since UTOPIA starts ahead of TO in T. These two examples show that the previous two rules, the longest-match rule and the shortest-match substring search rule, are not suitable for such find-and-replace operations in text searching since both rules allow matching substrings to overlap. We suggest a new linearizing restriction that is suitable for find-and-replace operations by identifying only non-overlapping matching substrings.

Definition 3. We define the leftmost non-overlapping match rule as follows:

> Given a text T, we identify the leftmost matching substring. Then, we move to the next position of the matching substring in T and repeat the identification of the leftmost matching substring in the remaining text until we cannot find it anymore. For example, if two matching strings overlap, then we choose the string whose start position is ahead of the other string's start position and discard the other string; see (a) in Fig. 1. If there are more than two matching substrings that start from the same position, then we choose the shortest string among them; see (b) in Fig. 1.

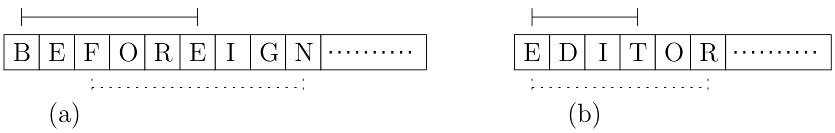

Figure 1: The figure illustrates the leftmost non-overlapping match rule. (a) When the pattern is {BEFORE, FOREIGN}; the rule chooses BEFORE. (b) When the pattern is {EDIT, EDITOR}; the rule chooses EDIT.

Let $\mathcal{PM}(L, T)$ denote the set of matching substrings of a given text T with respect to a given pattern L by the leftmost non-overlapping match rule. Let $|\mathcal{PM}(L, T)|$ be the number of matching strings in $\mathcal{PM}(L, T)$. For example, if

$T = abcbabb$ and $L = \{aa, ab, ba, bb\}$, then $\mathcal{PM}(L, T) = \{(1, 2), (4, 5), (6, 7)\}$ and $|\mathcal{PM}(L, T)| = 3$. Note that although the substring $T(5, 6) = ab$ is in L, it is not in $\mathcal{PM}(L, T)$ since it overlaps with another matching substring $T(4, 5)$. From the definition of the leftmost non-overlapping match rule, we obtain the following results.

Proposition 4. *The leftmost non-overlapping match rule ensures that the number of matching substrings of T is at most n, where n is the size of T. Namely, $|\mathcal{PM}(L, T)| \leq n$*

Proposition 5. *If two distinct matching pairs (u_1, v_1) and $(u_2, v_2) \in \mathcal{PM}(L, T)$, then either $v_1 < u_2$ or $v_2 < u_1$.*

Proposition 4 shows that we always have a linear number of matching substrings in the size of a given text by the leftmost non-overlapping match rule. Note that we do not require L to be a particular type of language such as a regular language or a context-free language. Similar to the longest-match rule or the shortest-match substring search rule, the leftmost non-overlapping match rule can be treated as a general principle for any text search application. Since regular expressions are often used for the matching problem, we study the regular-expression matching problem with the leftmost non-overlapping match rule in Section 5.

5 Regular-expression Matching Problem

We consider the regular-expression matching problem using the leftmost non-overlapping match rule. Before we present an algorithm for this problem, we explain an example. Assume that we are given a regular expression $E = a(a+b)^*c$ for the text in Fig. 2. Then, $\mathcal{PM}(L(E), T) = \{(1, 5), (8, 11), (12, 14)\}$.

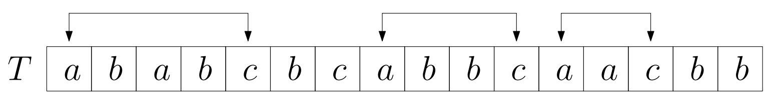

Figure 2: The output of $\mathcal{PM}(L(E), T)$, where $E = a(a + b)^*c$.

Note that $T(1, 5), T(8, 11)$ and $T(12, 14)$ are not the only matching substrings of T for $L(E)$. $T(3, 5) = abc$ and $T(13, 14) = ac$ are also in $L(E)$. Nevertheless, since both $T(3, 5)$ and $T(13, 14)$ overlap other matching substrings of T and they are not the leftmost matching substrings, the leftmost non-overlapping

---

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
rof

Figure 3: A regular-expression matching procedure for finding all the end positions of matching substrings of T for the pattern $L(A)$, where $A = (Q, \Sigma, \delta, s, f)$ is a Thompson automaton and $T = w_1 \cdots w_n$ is a text.

match rule does not identify them. For example, both $T(1, 5)$ and $T(3, 5)$ are in $L(E)$ but $T(1, 5)$ is selected since $T(1, 5)$ is the leftmost matching substring.

We show that the regular-expression matching problem with the leftmost non-overlapping match rule can be solved using a double scan of T based on the algorithm of Crochemore and Hancart [Crochemore and Hancart(1997)].

Proposition 6 [Crochemore and Hancart(1997)]. *Given a regular expression E and a text T, we can find all the end positions of matching substrings of T with respect to $L(E)$ in $O(mn)$ worst-case time with $O(m)$ space using ExpressionMatching in Fig. 3, where m is the size of E and n is the size of T.*

The algorithm ExpressionMatching (EM) in Fig. 3 is a modified version of Aho's algorithm [Aho(1990)] that determines whether or not a given text has a substring accepted by a given FA. EM has two sub-functions: 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 and the $goto(X, w_j)$ function gives all states that can be reached from a state in X by a transition with w_j, the current input character. The $null(X)$ function takes $O(m)$ time to identify all null transition reachable states and the $goto(X, w_j)$ function takes a constant time to compute the new set. For details of the algorithm, the sub-functions and the time complexity, the reader may look at the literatures [Aho(1990), Crochemore and Hancart(1997)].

Given a regular expression E and a text $T = w_1 \cdots w_n$, we first compute all start positions of matching substrings of T with respect to E. We prepend Σ^* to E^R; thus, allowing matching to begin at any position in T^R. We construct the Thompson automaton A for $\Sigma^* E^R$ and run ExpressionMatching (A, T^R). Therefore, we can find all start positions of matching substrings in $O(mn)$ time, where $m = |E|$ and $n = |T|$. For example, if we run EM on the text in Fig. 2, then we obtain the following positions as indicated by "↓" in Fig. 4.

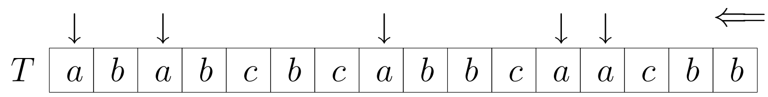

Figure 4: The output of a single scan of T^R with respect to $\Sigma^* E^R$ using EM, where $E = a(a+b)^*c$.

Let $P = \{p_1, \ldots, p_k\}$ be the set of the start positions of matching substrings that we have computed after the single scan of T^R, where k is the number of start positions of matching substrings and $p_i < p_j$ for $i < j$. Then, we read a character from p_i position of T to find the corresponding shortest matching string that belongs to $L(E)$. Once we find one matching substring $T(p_i, j)$, where $p_i < j$, we move to the next start position in P that is greater than j to avoid the overlapping. A full algorithm is given in Fig. 5.

ReverseEM (A, T, P)

$X = \{\ \}$, i = 1
for $j = p_i$ **to** n
    $X = null(goto(X, w_j))$
    if $f \in X$
        output (p_i, j)
        while $(p_i < j)$
            i = i + 1
        $j = p_i$
    fi
rof

Figure 5: A reverse-scan matching procedure for a given Thompson automaton $A = (Q, \Sigma, \delta, s, f)$ for E, a text $T = w_1 \cdots w_n$ and a set $P = \{p_1, \ldots, p_k\}$ of the start positions of matching substrings of T.

For example, if we run ReverseEM for the result in Fig. 4, where $P = \{1, 3, 8, 12, 13\}$, then the algorithm first outputs $(1, 5)$. The algorithm skips 3 in P since it makes an overlapping with the current output $(1, 5)$ and goes to 8 in P to avoid an overlapping. Fig. 6 illustrates this step.

ReverseEM is based on EM in Fig. 3 and the **while** loop in ReverseEM speeds up for finding the next matching substring by skipping inappropriate start

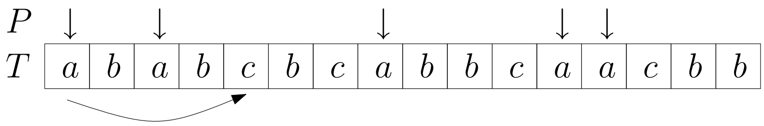

Figure 6: An example of ReverseEM to find corresponding end positions for a given set P according to the leftmost non-overlapping match rule, where $E = a(a+b)^*c$.

positions and ensures that the algorithm prohibits the overlapping matching substrings. Note that the **while** loop is executed at most k times in total even though it is inside the **for** loop. Therefore, the worst-case time complexity of ReverseEM is still $O(mn)$.

Theorem 7. *Given a pattern regular expression E and a text T, we can compute the set of matching substrings that conforms the leftmost non-overlapping match rule in $O(mn)$ worst-case time using $O(m)$ space, where m is the size of E and n is the size of T.*

Next, we show that our algorithm gives the correct matching substrings.

Theorem 8. *A pair (u, v) is recognized by ReverseEM if and only if $(u, v) \in \mathcal{PM}(L(E), T)$, where E is a given pattern regular expression and T is a given text.*

Proof. Assume that we have computed the set $P = \{p_1, \ldots, p_k\}$ of the start positions of matching substrings using EM in Fig. 3, where k is the number of start positions of matching substrings.

$\Longrightarrow$ If (u, v) is recognized by ReverseEM, then $T(u, v) \in L(E)$ and $u \in P$ since **output** in ReverseEM gives (p_i, j) and $p_i \in P$. It is clear that there is no matching substring $T(u, v')$, where $v' < v$, from the algorithm; namely, $T(u, v)$ is the shortest matching substring among all matching substrings that start from the same position u in T. Now assume that $T(u, v)$ overlaps with another matching substring $T(u', v')$ and $T(u, v)$ is not the leftmost matching substrings; hence, $u' < u < v'$. Then, when ReverseEM recognizes (u', v'), the value of j becomes v'. After the **output** (u', v'), ReverseEM executes the **while** loop to choose the next start position from P that is greater than the current position j. Since $u < j = v'$, u cannot be chosen as a start position because of the **while** loop. It implies that the algorithm skips the start position u and therefore (u, v) cannot be recognized by the algorithm—a contradiction; there cannot be a such matching substring $T(u', v')$ in T. Therefore, if (u, v) is recognized by ReverseEM, then $(u, v) \in \mathcal{PM}(L(E), T)$.

$\Longleftarrow$ Since $(u, v) \in \mathcal{PM}(L(E), T)$, $T(u, v)$ is the shortest matching substring from position u in T with respect to L and u must be in P. If u is p_1 in P, then it is clear that ReverseEM recognizes (u, v). Assume $u = p_i$, where $1 < i \leq k$. Now the only possible case that ReverseEM fails to recognize (u, v) is when u is skipped by the **while** in the algorithm; namely, $u < j$ for some j. It implies that there is an output (q', j), where $q' < u < j$ and $q' \in P$. It contradicts that $T(u, v)$ is the leftmost non-overlapping matching substring of T. Therefore, this situation is not possible and (u, v) must be recognized by ReverseEM. □

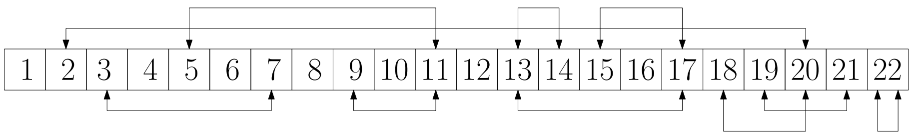

Figure 7: Assume that we have 10 matching substrings denoted by arrow lines. Then, the leftmost non-overlapping match rule gives a single matching substring out of 10 matching substrings.

Now we consider the maximal number of non-overlapping matching substrings. The leftmost non-overlapping match rule does not guarantee the maximal number of non-overlapping matching substrings since it finds the leftmost matching substring. See Fig. 7 for an example. If we want to find the maximal number of non-overlapping matching substrings, first we identify all shortest matching substrings from each position of T. Next, we sort these matching substrings according to the end position and select the non-overlapping matching substrings from left to right in the greedy manner[3]. For example, in Fig. 7, this approach gives (3, 7), (9, 11), (13, 14), (15, 17), (18, 20), (22, 22). It is easy to prove that this approach guarantees the maximal number of non-overlapping matching substrings. Since we can find the shortest matching substring from each start position in $O(mn)$ time, we can find all shortest matching substrings in $O(mn^2)$ time. Let k be the number of such matching substrings. Then, we can find the maximal number of non-overlapping matching substrings in $O(k \log k)$ time using sorting and the greedy selection. Since $k \leq n$, we establish the following statement.

Theorem 9. *Given a pattern regular expression E and a text T, we can compute the maximal number of non-overlapping matching substrings of T in $O(mn^2)$ worst-case time using $O(m)$ space.*

[3] This procedure is similar to the optimal job scheduling algorithm that maximizes the number of compatible jobs [Kleinberg and Tardos(2005)].

6 Conclusions

We have investigated the linear number of matching substrings in the pattern matching problem. We have reexamined two linearizing restriction rules: The longest-match rule that is a generalization of the IEEE POSIX rule [IEEE(1993)] and the shortest-match substring search rule [Clarke and Cormack(1997)]. We have shown that the two rules give the same result when the given pattern is an infix-free language. Note that both rules have different semantics and give different outputs in general. Then, we have introduced a new linearizing restriction, the leftmost non-overlapping match rule, which should be useful for implementing find-and-replace operations in text searching. Furthermore, we have proposed an $O(mn)$ worst-case running time algorithm for the regular-expression matching problem using the new linearizing rule.

Acknowledgments

We wish to thank the referees for the careful reading of the paper and many valuable suggestions. As usual, however, we alone are responsible for any remaining sins of omission and commission.

References

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

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

[Aho et al.(1974)] Aho, A., Hopcroft, J., Ullman, J.: The Design and Analysis of Computer Algorithms; Addison-Wesley Publishing Company, 1974.

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

[Câmpeanu et al.(2003)] Câmpeanu, C., Salomaa, K., Yu, S.: "A formal study of practical regular expressions"; International Journal of Foundations of Computer Science; 14 (2003), 6, 1007–1018.

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

[Crochemore and Hancart(1997)] Crochemore, M., Hancart, C.: "Automata for matching patterns"; G. Rozenberg, A. Salomaa, eds., Linear modeling: background and application; volume 2 of Handbook of Formal Languages; 399–462; Springer-Verlag, 1997.

[Giammarresi et al.(2004)] Giammarresi, D., Ponty, J.-L., Wood, D., Ziadi, D.: "A characterization of Thompson digraphs"; Discrete Applied Mathematics; 134 (2004), 317–337.

[Han et al.(2007)] Han, Y.-S., Wang, Y., Wood, D.: "Prefix-free regular languages and pattern matching"; Theoretical Computer Science; 389 (2007), 1-2, 307–317.

[IEEE(1993)] IEEE: IEEE standard for information technology: Portable Operating System Interface (POSIX) : part 2, shell and utilities; IEEE Computer Society Press, 1993.

[Kleinberg and Tardos(2005)] Kleinberg, J., Tardos, E.: Algorithm Design; Addison-Wesley Longman Publishing Co., Inc., 2005.

[Knuth et al.(1977)] Knuth, D., Morris, Jr., J., Pratt, V.: "Fast pattern matching in strings"; SIAM Journal on Computing; 6 (1977), 323–350.

[Myers(1992)] Myers, E. W.: "A four Russians algorithm for regular expression pattern matching"; Journal of the ACM; 39 (1992), 2, 430–448.

[Myers et al.(1998)] Myers, E. W., Oliva, P., Guimãraes, K. S.: "Reporting exact and approximate regular expression matches"; Proceedings of CPM'98; Lecture Notes in Computer Science 1448; 91–103; 1998.

[Thompson(1968)] Thompson, K.: "Regular expression search algorithm"; Communications of the ACM; 11 (1968), 419–422.

[Wood(1987)] Wood, D.: Theory of Computation; John Wiley & Sons, Inc., New York, NY, 1987.