International Journal of Foundations of Computer Science
Vol. 24, No. 5 (2013) 679–687
© World Scientific Publishing Company
DOI: 10.1142/S0129054113500238

AN IMPROVED PREFIX-FREE REGULAR-EXPRESSION MATCHING

YO-SUB HAN
Department of Computer Science, Yonsei University
Seoul 120-794, Republic of Korea
emmous@cs.yonsei.ac.kr

Received 8 May 2012
Accepted 28 January 2013
Communicated by Kai Salomaa

We revisit the regular-expression matching problem with respect to prefix-freeness of the pattern. It is known that a prefix-free pattern gives only a linear number of matching substrings in the size of an input text. We improve the previous algorithm and suggest an efficient algorithm that finds all pairs (start, end) of start and end positions of all matching substrings with a single scan of the input when the pattern is a prefix-free regular expression.

Keywords: String pattern matching; regular-expression matching; prefix-free regular expressions.

1. Introduction

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, 10] and if $L(E)$ is a finite language, then we obtain the multiple keyword matching problem [2]. For an infinite regular pattern, 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 a pattern regular expression E and n is the size of an input text T. Crochemore and Hancart [6] presented an algorithm, which is a modified version of Aho's algorithm, to find all end positions of matching substrings in $O(mn)$ time using $O(m)$ space. Myers *et al.* [11] solved the problem of identifying start positions and end positions of all matching substrings in $O(mn \log n)$ time using $O(m \log n)$ space. Regular-expression matching has been adopted in many applications such as `Google RE2`, `grep`, `vi`, `emacs` and `perl`.

In the regular-expression matching problem, there are a quadratic number of matching substrings with respect to the size of an input in the worst-case. On the other hand, Han [7] studied pattern matching rules that guarantee a linear number of matchings. 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. Recently, Han *et al.* [8] observed that the prefix-free pattern also has at most a linear number of matching substrings and designed efficient algorithms that identify all pairs (start, end) of start and end positions of matching substrings for prefix-free regular-expression patterns and infix-free regular-expression patterns. Their algorithms read the input text twice. We revisit the prefix-free pattern matching algorithm by Han *et al.* [8] and develop an algorithm that finds all pairs (start, end) of all matching substrings with a single scan of the input text. Note that it is crucial to reduce the number of scans in pattern matching especially when an input size is very large such as biodata or protein data. Although we do not have a specific application for the prefix-free regular-expression matching, the new algorithm should be useful in some applications using large size of inputs. This is the motivation of our research.

In Sec. 2, we define some basic notions. We then, in Sec. 3, briefly describe how we can solve the regular-expression matching problem using the Thompson construction [12]. We design an improved prefix-free regular-expression matching algorithm that needs only single scan of the input text in Sec. 4 and conclude the paper in Sec. 5.

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

The symbol $\emptyset$ denotes the empty language and the character λ denotes the empty string. A finite-state automaton (FA) A is specified by a tuple $(Q, \Sigma, \delta, s, F)$, where Q is a finite set of states, Σ is an input alphabet, $\delta : Q \times \Sigma \to 2^Q$ 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, we use f instead of $\{f\}$ for simplicity. The size $|A|$ of A is the number of states and the number of transitions. For a transition $q \in \delta(p, a)$ in A, we say that p has an *out-transition* and q has an *in-transition*. 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. 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. 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. The size $|E|$ of E is the number of symbol appearances in E over Σ.

For complete background knowledge in automata theory, the reader may refer to textbooks [9, 13].

3. Regular-Expression Matching

The regular-expression matching problem is formally defined as follows:

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 pairs* (start, end) *of start and end positions of all matching substrings of T that belong to $L(E)$.*

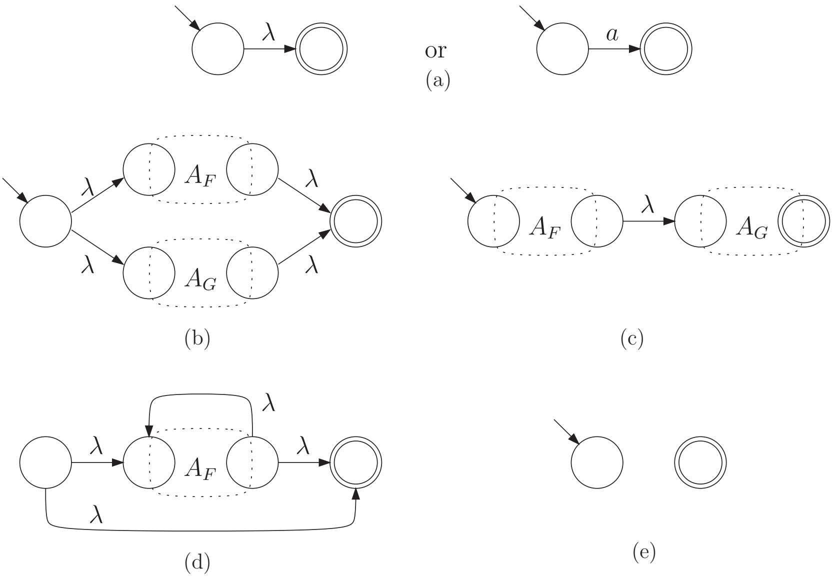


Fig. 1. The Thompson construction. Let E, F and G denote regular expressions and A_F and A_G denote the corresponding Thompson automata of F and G, respectively. (a) $E = \lambda + a$, (b) $E = F + G$, (c) $E = F \cdot G$, (d) $E = F^*$ and (e) E is empty.

For the regular-expression matching problem, we use the Thompson construction [12] for E to process T. Figure 1 is the inductive Thompson construction. Given E over Σ, we first prepend Σ^* to E; thus, allowing matching to begin at any position in an input T. We construct the Thompson automaton $A_E = (Q, \Sigma, \delta, s, f)$ for $\Sigma^* E$, where Q is the set of states, Σ is an input alphabet, δ is a set of transitions, $s \in Q$ is the start state and $f \in Q$ is the final state. Then we use ExpressionMatching (EM) described in Fig. 2 to process T.

The algorithm 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. The function $goto(X, w_j)$ gives all states that can be reached from a state in X by a transition with w_j, the current input character. The $goto(X, w_j)$ function is computed efficiently using two stacks and a bit vector indexed by states. One stack is used to store Q, the current set of states, and the

```
ExpressionMatching $(A_E, T)$
$X = null(\{s\})$
if $f \in X$ then output $0$
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 Thompson automaton A_E and a text $T = w_1 \cdots w_n$. The procedure reports all the end positions of matching substrings of T.

other stack to determine the next set of states. Since each state has at most two out-transitions, each state on the first stack can add at most two new states to the second stack. The bit vector is used to quickly determine whether or not a state is already on the second stack so that we do not add it twice. The algorithm runs in $O(mn)$ using $O(m)$ space [1, 6].

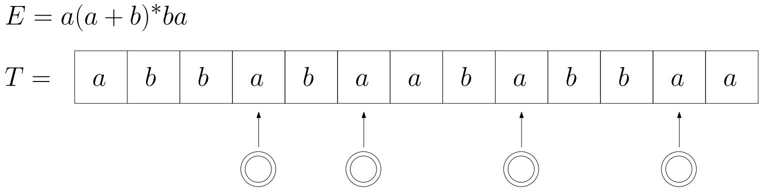


Fig. 3. An example of running the algorithm in Fig. 2. The matching substring pairs are (1,4), (1,6), (4,6), (1,9), (4,9), (6,9), (7,9), (1, 12), (4, 12), (6, 12), (7, 12), (9, 12).

Once we have identified all end positions of matching substrings, we can find the corresponding start positions of each end position using T^R and E^R. Since there are at most n possible end positions, we can thus solve the regular-expression matching problem in $O(mn^2)$ time using $O(m)$ space.

4. Prefix-Free Regular-Expression Matching

Recently Han *et al.* [8] considered the case when the pattern regular expression E is prefix-free; namely, for any two distinct strings $x, y \in L(E)$, x is not a prefix of y. They noticed that if E is prefix-free, then there are at most n matching substrings of T, where $n = |T|$. Based on this observation, they proposed an $O(mn)$ runtime algorithm using $O(m)$ space, which reads T twice. For large size of inputs such as protein data or DNA data, it is desirable to read the input as fewer as possible. This motivates us to investigate a prefix-free regular-expression matching algorithm that reads T only once.

Given a prefix-free regular expression E and an input $T = w_1 w_2 \cdots w_n$, we consider all positions w_i, for $1 \leq i \leq n$, of T as possible end positions of matching substrings and look for start positions of matching substrings. Since we search for start positions, we read T backward and report a matching pair when we find a start position. For the backward processing, we construct the Thompson automaton A_{E^R} of E^R, the reversal of E.

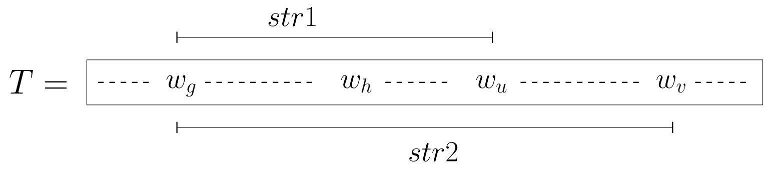
```
        PrefixPatternMatching ($A_{E^R} = (Q, \Sigma, \delta, s, f), T$)
1:  $X_n = null(\{s\})$   // initialize $X_n$ for the last character position $w_n$ of $T$

2:  for $j = n$ to 1
3:    for $i = n$ to $j$
4:      $X_i = goto(X_i, w_j)$
5:      if $f \in X_i$ then output $(j, i)$
6:    for each state $q \in Q$ do Set $q_{in} = 0$
7:    $null_p(j)$
8:    if $f_{in} \neq -1, 0$ then output $(j, f_{in})$

---- module $null_p(j)$ ----------------------------
9:    for $i = n$ to $j$
10:     for each state $q \in X_i$
11:       for each null-reachable state $q'$ from $q$
12:         if $q'_{in} = 0$ then Set $q'_{in} = i$
13:         elsif $q'_{in} = i$ then continue
14:         elsif $q'_{in} \neq i$ then Set $q'_{in} = -1$
// line 9–14; We mark from which state of $X_i$ that a state $q$ is reached by $\lambda$-transitions.
//           This step can be performed by DFS in $O(m)$ time.

15:   for each null-reachable state $q'$ from $s$
16:     if $q'_{in} = 0$ then Set $q'_{in} = j-1$
17:     elsif $q'_{in} = j-1$ then continue
18:     elsif $q'_{in} \neq j-1$ then Set $q'_{in} = -1$
// line 15–18; We compute all null-reachable states from $s$ to create the next set $X_{j-1}$
//           of reachable states for the input character position $w_{j-1}$.
//           This step can be done in $O(m)$ by DFS.

19:   for each state $q \in Q$
20:     if $q_{in} = -1$ then
21:       for all null-reachable states $q'$ from $q$
22:         Set $q'_{in} = -1$
// line 19–22; We mark all null-reachable states from two different sets.
//           All such states can be identified in $O(m)$ time by DFS.

23:   Empty all $X_i$
24:   for each state $q \in Q$
25:     if $q_{in} \neq 0, -1$ then Put $q_{in}$ in $X_{q_{in}}$
----------------------------------------------------
```

Fig. 4. A prefix-free regular-expression matching procedure (PEM).

Similar to the original regular-expression matching procedure in Fig. 2, the algorithm (named PEM) in Fig. 4 reads the input text T backward and maintains sets X_i of reachable states from each character position w_i of T and update X_i with respect to the current input character w_j and null transitions. Before we analyze the running time of PEM, we establish the following property that is crucial for calculating the runtime.

Lemma 2. *If a state r of a Thompson automaton $A_{E^R} = (Q, \Sigma, \delta, s, f)$ is reached from two different states p and q, where $p \in X_u$ and $q \in X_v$, when reading a character w_h in PEM, where $h \leq u < v$, then both paths from p and q via r cannot reach f by reading any prefix of the remaining input in PEM.*

Proof. Notice 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 $str1 = w_g \cdots w_h \cdots w_u$ and $str2 = w_g \cdots w_h \cdots w_v$ belong to $L(E)$.


$str1$
$T =$
----- w_g --------- w_h ------ w_u ---------- w_v -----
$str2$

We have the following three cases for w_u and w_v:

(1) Both u and v are end positions of the matching substrings: This implies that $str1, str2 \in L(E)$, which contradicts the prefix-freeness of $L(E)$ since $str1$ is a prefix of $str2$.
(2) One (say u) is an end position of a matching substring and the other (say v) is not: This immediately contradicts $str2 \in L(E)$.
(3) Both u and v are not end positions: This contradicts $str1$ and $str2$ are in $L(E)$.

Therefore both paths from p and q via r cannot reach f by reading any prefix of the remaining input. □

Lemma 2 shows that if a state r is reached from two different sets of reachable states when reading a character w_h in PEM, then r should not belong to both sets since both paths via r cannot reach the final state by reading any remaining input. Thus, each state of A_{E^R} appears in at most one reachable set in PEM. Based on this observation, we establish the following properties:

Lemma 3. *For any two sets X_u and X_v in PEM, for $1 \leq u < v \leq n$,*

$$X_u \cap X_v = \emptyset$$

and, thus,

$$\sum_{i=1}^{n} |X_i| \le m,$$

where m is the number of states in A_{E^R}.

Based on Lemma 2, we design the $null_p$ function as described in Fig. 4. Note that in line 9–14 in $null_p$, we explore all states of A_{E^R} that are reachable by λ-transitions from each set X_i of states. We can use any efficient graph exploring algorithms such as DFS (Depth-First Search) or BFS (Breadth-First Search). (See the textbook [5] for details on DFS or BFS.) There are three possible cases:

(1) If the state q' that is being visited currently is explored the first time, then we put a mark that q' is reached from a state in X_i (line 12).
(2) If q' is already visited from another state in X_i, then we continue to explore the remaining states via λ-transitions (line 13).
(3) Finally, if q' is already visited from a state that is not in X_i, then we put a mark that q' should not belong to any state sets due to Lemma 2 (line 14).

Since we can explore all states and mark the state status in $O(|A_{E^R}|) = O(m)$ time and the sum of all $|X_i|$ is at most m as shown in Lemma 3, the running time for the submodule in line 9–14 is $O(m)$.

We run a similar step for the next set X_{j-1} of reachable states for the next position $j-1$ of T in line 15–18. Here we only consider the start state s. This step requires for reading the next character w_{j-1} in the main module (line 3–7). Note that we also need to remove all states that are reachable by λ-transitions from a state q' with $q'_{in} = -1$ (in other words, q' is reached by two different state sets). For this step, we once again run DFS and mark such states as described in line 19–22 in PEM. It is easy to verify that both steps run in $O(m)$ time.

Finally, once we have finished to classify which state is reached from which state set, and which state is reached from two different state sets, then we recompute the state sets for the next input character in line 23–25. This requires $O(m)$ time. Therefore, the total running time of PEM is $O(mn)$.

Theorem 4. *Given a prefix-free regular expression E and a text T, we can find all pairs* (start, end) *of start and end positions of all matching substrings in $O(mn)$ time using $O(m)$ space with a single scan of T, where $m = |E|$ and $n = |T|$.*

Proof. It is easy to verify that a matching substring $w = w_i w_{i+1} \cdots w_j$ of T is identified by PEM. Since $w \in L(E)$, there exists an accepting path $q_0 (= s) \to q_1 \to \cdots \to q_{k-1} \to q_k (= f)$ for w^R in A_{E^R}. This implies that the set X_j of reachable states for position j in T should keep maintaining reachable states when reading each character from w_j to w_i in PEM. Moreover when the algorithm reads w_i, the final state f should be included in X_j and, thus, PEM reports (i, j), which is (start, end) of w.

Next we calculate the running time and space for the algorithm. Lemmas 2 and 3 guarantee that we need at most $O(m)$ time to update all sets of reachable states for each character w_i of T in PEM since a state in a Thompson automaton has at most two in-transitions and two out-transitions (See Fig. 1). In other words, at any step of reading a character from T in line 3–25 of PEM, the total size of the sets of reachable states is at most $O(m)$. Therefore, the total running time of PEM is $O(m) \times n = O(mn)$. Note that we use only $O(m)$ space for computation since $|A_{E^R}| = O(m)$. □

The proposed algorithm seems similar to the prefix-free regular-expression matching algorithm by Han *et al.* [8]. Yet, the main difference is the number of scans of T. Our algorithm considers all positions of T as candidates for end positions of matching substrings and searches for matching substrings; namely, the algorithm reads T **once**. On the other hand, the algorithm by Han *et al.* [8] first computes all end positions by reading T forward and finds corresponding start positions from the end positions by reading T backward; namely, their algorithm reads T **twice**.

Since the new approach considers all positions, it seems to require more processing time. However, our empirical study shows that if a position w_i is not a valid end position, then X_i becomes empty soon and PEM does not need to consider X_i anymore.

The algorithm PEM can be used in other subfamilies of regular-expression patterns. Given a suffix-free regular expression E, its reversal E^R is prefix-free. We also know that an infix-free regular expression is also prefix-free. Thus, we establish the following statement from Theorem 4.

Corollary 5. *Given a suffix-free regular expression or an infix-free regular expression as a pattern E and a text T, we can find all pairs (start, end) of start and end positions of all matching substrings in $O(mn)$ time using $O(m)$ space with a single scan of T, where $m = |E|$ and $n = |T|$.*

Note that the previous efficient algorithm for the infix-free regular-expression matching problem reads T twice [8] and PEM reads only once.

5. Conclusions

We have revisited the prefix-free regular-expression matching problem. Given a text T and a prefix-free regular expression E, Han *et al.* [8] suggested an $O(mn)$ algorithm for finding all pairs (start, end) of start and end positions of all matching substrings. This algorithm reads T twice. We have observed that we can reduce the number of scans of the input text if we know that the pattern is prefix-free. Based on this observation, we have designed an improved prefix-free regular-expression matching algorithm, which runs in $O(mn)$ but requires single scan of T. Note that $|T| \gg |E|$ in practice and, thus, it is desirable to read T as fewer as possible. Therefore, the new algorithm improves the previous algorithm.

The concluding remark is that we can use the same algorithm for an infix-free regular-expression pattern since an infix-free pattern is always prefix-free. Also it is straightforward to extend the algorithm for a suffix-free regular-expression pattern.

Acknowledgments

We wish to thank the referees for the care they put into reading the previous version of this manuscript. Their comments were invaluable in depth and detail, and the current version owes much to their efforts.

This research was supported by the Basic Science Research Program through NRF funded by MEST (2010-0009168, 2012R1A1A2044562).

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

[6] 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.

[7] Y.-S. Han. On the linear number of matching substrings. *Journal of Universal Computer Science*, 16:715–728, 2010.

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

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

[10] D. Knuth, J. Morris, Jr., and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.

[11] E. W. Myers, P. Oliva, and K. S. Guimãraes. Reporting exact and approximate regular expression matches. In *Proceedings of CPM′98*, Lecture Notes in Computer Science 1448, 91–103, 1998.

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

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