Disambiguation in Regular Expression Matching via Position Automata with Augmented Transitions*

Satoshi Okui[1] and Taro Suzuki[2]

[1] Department of Computer Science, Chubu University, Japan
okui@cs.chubu.ac.jp
[2] School of Computer Science and Engineering, The University of Aizu, Japan
taro@u-aizu.ac.jp

Abstract. This paper offers a new efficient regular expression matching algorithm which follows the POSIX-type leftmost-longest rule. The algorithm basically emulates the subset construction without backtracking, so that its computational cost even in the worst case does not explode exponentially; the time complexity of the algorithm is $O(mn(n+c))$, where m is the length of a given input string, n the number of occurrences of the most frequently used letter in a given regular expression and c the number of subexpressions to be used for capturing substrings. A formalization of the leftmost-longest semantics by using parse trees is also discussed.

1 Introduction

Disambiguation in regular expression matching is crucial for many applications such as string replacement in document processing. POSIX 1003.2 standard requires that the ambiguity of regular expressions to be resolved by following the *leftmost-longest* rule.

For implementations that rely on backtrack, strictly following the leftmost-longest rule forces loss of efficiency; the greedy, or the first match, policy [6] is easier for such systems to follow. For this problem, some recent implementations, e.g., [9,8], take approaches based on automata, realising the leftmost-longest semantics efficiently.

This paper presents a new regular expression matching algorithm which follows the leftmost-longest semantics. The time complexity of our algorithm is $O(mn(n+c))$ where m is the length of a given input string, n the number of occurrences of the most frequently used letter in a given regular expression and c the number of subexpressions to be used for *capturing* portions of the input string.

Our discussion for achieving that algorithm proceeds in three steps as follows. First, in Section 2, we formalize the leftmost-longest rule based on *parse trees*. A parse tree for a given input string represents a way of accepting that string; e.g., which alternative is selected or how many times each iteration is performed, and so forth. Imposing a priority order on parse trees results in a straightforward interpretation of the leftmost-longest rule, which serves as a basis for our later discussion. We restrict ourselves to consider *canonical* parse trees for avoiding the matches which is unacceptable in the

* This work is supported by Japan Society for Promotion of Science, Basic Research (C) No.22500019.

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 231–240, 2011.

leftmost-longest semantics. Next, Section 3 introduces a slight extension of traditional *position automata* [10,7] in order to enumerate canonical parse trees. In Section 4, we introduce the regular expression matching algorithm, which basically performs subset construction at runtime. This algorithm incrementally compares, in each step, the priority of any two paths to eliminate unnecessary ones. Finally, Section 5 is devoted to a brief comparison with other studies.

Due to limitations of space, correctness issues of our algorithm are not discussed here. These are found in an extended version of this paper (`http://www.scl.cs.chubu.ac.jp/reg2010/`).

2 Formalizing the Leftmost-Longest Semantics

A *regular expression* is an expression of the form: $r ::= 1 \mid a \mid r{\cdot}r \mid r{+}r \mid r*$ where 1 is the null string, a a letter in the alphabet Σ and $r_1{\cdot}r_2$ the concatenation of r_1 and r_2. Our definition does not include the regular expression for the empty language. We think of the regular expressions as abstract syntax trees (AST). For strings accepted by a regular expression r, we consider the set $\mathtt{PT}(r)$ of *parse trees*:

$$\mathtt{PT}(1) = \{1\} \quad \mathtt{PT}(a) = \{a\} \quad \mathtt{PT}(r_1{\cdot}r_2) = \{t_1{\cdot}t_2 \mid t_i \in \mathtt{PT}(r_i), i = 1, 2\}$$
$$\mathtt{PT}(r_1{+}r_2) = \{\mathtt{L}(t) \mid t \in \mathtt{PT}(r_1)\} \cup \{\mathtt{R}(t) \mid t \in \mathtt{PT}(r_2)\}$$
$$\mathtt{PT}(r*) = \{\mathtt{I}(t_1, \ldots, t_n) \mid t_1, \ldots, t_n \in \mathtt{PT}(r), n \geq 0\}$$

Note that parse trees are *unranked* in the sense that an `I`-node has an arbitrary number of children (possibly none). Parse trees for strings should not be confused with the AST of the regular expression; basically, parse trees are obtained from the AST of a regular expression by horizontally expanding children of $*$-nodes and choosing the left or right alternatives for +-nodes.

A *position* (also called a *path string*) is a sequence of natural numbers. The root position is the empty string and denoted as Λ. The child of the `R`-node has a position ending with 2 rather than 1. For other kind of nodes, the first sibling always has a position ending with 1, the next sibling has a position ending with 2 and so forth. The length of a position p (as string) is denoted by $|p|$. The subterm of t at position p is denoted as $t|_p$. We write $p_1 \lessdot p_2$ iff p_1 precedes p_2 in lexicographic order.

Let t be a parse tree. The *norm* of t at $p \in \mathtt{Pos}(t)$, written $\| t \|_p$, is the number of letters (of Σ) in $t|_p$ (Note: we do not count 1-nodes). For p not in $\mathtt{Pos}(t)$, we define $\| t \|_p = -1$. The subscript of $\| t \|_\Lambda$ is omitted.

According to the POSIX specification [1], "*a subexpression repeated by '*' shall not match a null expression unless this is the only match for the repetition.*" For example, for a regular expression, $a{**}$, $\mathtt{I}(\mathtt{I}())$ satisfies this requirement while $\mathtt{I}(\mathtt{I}(), \mathtt{I}())$, $\mathtt{I}(\mathtt{I}(), a)$ or $\mathtt{I}(a, \mathtt{I}())$ does not. This leads us to the notion of *canonical* parse tree:

Definition 1. *(canonical parse tree) A parse tree t is called* canonical *if any subterm of t such as $\mathtt{I}(t_1, \ldots, t_n)$, $n \geq 2$, satisfies $\| t_i \| > 0$ for any $1 \leq i \leq n$.*

Obviously, a leaf of a parse tree is either 1 or a letter in Σ. Reading those letters in Σ from left to right, we obtain the string *derived* from that parse tree. We write $\mathtt{CPT}(r, w)$ for the set of canonical parse trees deriving w.

The specification [1] also describes that "*consistent with the whole match being the longest of the leftmost matches, each subpattern, from left to right, shall match the longest possible string*." This leads the following definition of priority:

Definition 2. *For any* $t_1, t_2 \in \mathtt{CPT}(r)$*, we say* t_1 is prior to t_2*, written* $t_1 \lessdot t_2$*, if the following conditions are satisfied for some* $p \in \mathtt{Pos}(t_1) \cup \mathtt{Pos}(t_2)$*:*

1. $\| t_1 \|_p > \| t_2 \|_p$
2. $\| t_1 \|_q = \| t_2 \|_q$ *for any position* $q \in \mathtt{Pos}(t_1) \cup \mathtt{Pos}(t_2)$ *such that* $q \lessdot p$.

Recall that we define $\| t \|_p = -1$ for $p \notin \mathtt{Pos}(t)$. That corresponds to the requirement [1]: "*a null string shall be considered to be longer than no match at all*." The specification also states that "*the search for a matching sequence starts at the beginning of a string and stops when the first sequence matching the expression is found*" and that "*if the pattern permits a variable number of matching characters and thus there is more than one such sequence starting at that point, the longest such sequence is matched*," which we formalize as follows.

Given a regular expression r and a string w, we define the set $\mathtt{PC}(r, w)$ of *parse configurations* for r and w as $\{\langle u, t, v\rangle \mid t \in \mathtt{CPT}(r, w'), uw'v = w\}$. For $\langle u_1, t_1, v_1\rangle$ and $\langle u_2, t_2, v_2\rangle$ in $\mathtt{PC}(r, w)$, we write $\langle u_1, t_1, v_1\rangle \lessdot \langle u_2, t_2, v_2\rangle$ if either $|u_1| < |u_2|$ or else $|u_1| = |u_2|$ and $t_1 \lessdot t_2$.

Theorem 1. $\mathtt{PC}(r, w)$ *is a finite and strict total order set.*

The finiteness ensures that $\mathtt{PC}(r, w)$ has the least (that is, the most prior) parse configuration, which we think of as representing the leftmost-longest matching. Notice that the least element may not exist for arbitrary (non-canonical) parse trees; e.g., for $\mathtt{1}*$ and the empty string, we would have an infinite decreasing sequence: $\mathtt{I}() \gtrdot \mathtt{I}(\mathtt{1}) \gtrdot \mathtt{I}(\mathtt{11}) \gtrdot \ldots$.

3 Enumerating Parse Trees via Position Automata

3.1 Correctly Nested Parenthesis Expressions

A *parenthesis expression* means a sequence consisting of parentheses, each of which is indexed by a position, or letters in Σ. A parenthesis expression α is called *correctly nested* if all parentheses in α are correctly nested. For each t in $\mathtt{CPT}(r)$, we assign, as its string representation, a correctly nested parenthesis expression $\Phi(t, p)$:

$$\Phi(1, p) = (_p\ )_p \qquad \Phi(\mathtt{R}(t), p) = (_p \Phi(t, p.2)\ )_p$$
$$\Phi(a, p) = (_p a\ )_p \quad (a \in \Sigma) \qquad \Phi(t_1 \cdot t_2, p) = (_p \Phi(t_1, p.1)\ \Phi(t_2, p.2)\ )_p$$
$$\Phi(\mathtt{L}(t), p) = (_p \Phi(t, p.1)\ )_p \qquad \Phi(\mathtt{I}(t_1, \ldots, t_n), p) = (_p \Phi(t_1, p.1) \ldots \Phi(t_n, p.1)\ )_p$$

For a parse configuration $\langle u, t, v\rangle$, we define $\Phi(\langle u, t, v\rangle) = u\ \Phi(t, \Lambda)\ v$. Note that the indexes in $\Phi(t, \Lambda)$ do not come from the parse tree t but from the AST of the underlying regular expression.

Φ is not injective in general; e.g., both $\Phi(\mathtt{I}(a), p)$ and $\Phi(\mathtt{L}(a), p)$ are $(_p (_{p.1} a)_{p.1})_p$. It is, however, injective if the domain is restricted to each $\mathtt{PC}(r, w)$. We refer to the image of $\mathtt{PC}(r, w)$ by Φ, i.e., $\{\Phi(c) \mid c \in \mathtt{PC}(r, w)\}$, as $\mathtt{PC}_\Phi(r, w)$.

Any expression in $\mathtt{PC}_\Phi(r)$ has a bounded nesting depth, which does not exceed the height of (the AST of) r indeed. This means that $\mathtt{PC}_\Phi(r)$ is a regular language, thereby, recognizable (or equivalently, enumerable) by an automaton. We will construct such automata in the next section.

Let α be an arbitrary parenthesis expression that is not necessarily in the range of the function Φ. Parenthesis expressions are sometimes "packetized" with regarding the occurrences of letters as separators. For any parenthesis expression α, which is always written in the form $\beta_0 a_1 \beta_1 a_2 \dots \beta_{n-1} a_n \beta_n$ where $a_1 \dots a_n$ are letters and $\beta_0 \dots \beta_n$ possibly empty sequences of parentheses, we call β_i $(0 \le i \le n)$ the i-th *frame* of α. Let $\alpha_0, \dots, \alpha_n$ and $\beta_0, \dots, \beta_n$ be the sequences of frames of α and β respectively. If α_k and β_k make the first distinction (that is, k is the greatest index such that $\alpha_i = \beta_i$ for any $0 \le i < k$), then the index k is called the *fork* of α and β.

3.2 Position NFAs with Augmented Transitions

A *position automaton with augmented transitions* (PAT, in short) is a 6-tuple which consists of (1) a finite set Σ of letters, (2) a finite set Q of *states*, (3) a finite set T of *tags*, (4) a subset Δ of $Q \times \Sigma \times Q \times T^*$, (5) an initial state $\mathtt{q}_\wedge$ in Q and (6) a final state $\mathtt{q}_\$$ in Q. For $a \in \Sigma$, Q_a denotes $\{q \in Q \mid \langle p, a, q, \tau\rangle \text{ for some } p \text{ and } \tau\}$. We call an element of Δ a *transition*.

For an input string $a_0 \dots a_{n-1}$, a sequence $q_0 \tau_0 \dots \tau_{n-1} q_n$ where $q_0 = \mathtt{q}_\wedge$ is called a *path* if for any $0 \le i < n$ we have $\langle q_i, a_i, q_{i+1}, \tau_i\rangle \in \Delta$; $q_i \tau_i q_{i+1}$ $(0 \le i < n)$ is called the i-th step of the path. For a PAT $\langle \Sigma, Q, T, \Delta, \mathtt{q}_\wedge, \mathtt{q}_\$\rangle$ and an input string w, a *configuration* is a triple $\langle u, p, \alpha\rangle$ where u is a suffix of w, p a state in Q, and α a sequence of tags. For a transition $\delta = \langle p_1, a, p_2, \tau\rangle$ in Δ, we write $\langle u_1, p_1, \alpha_1\rangle \vdash_\delta \langle u_2, p_2, \alpha_2\rangle$ if $u_1 = a u_2$ and $\alpha_2 = \alpha_1 \tau a$. The *initial* configuration is $\langle w, \mathtt{q}_\wedge, \varepsilon\rangle$ (where ε denotes the empty sequence), while a *finial* configuration is of the form $\langle \varepsilon, \mathtt{q}_\$, \alpha\rangle$ for some α. We say a PAT M *accepts* an input string w, *yielding* a sequence α if there exists a sequence of configurations: $\langle w, \mathtt{q}_\wedge, \varepsilon\rangle \vdash \cdots \vdash \langle \varepsilon, \mathtt{q}_\$, \alpha\rangle$.

It might help to see a PAT as a sequential transducer. However, a PAT emits parenthesis expressions only for the sake of deciding priority of paths. Hence, a PAT has more similarity with Lurikari's *NFA with tagged transitions* (TNFA) [9] although the formulations are rather different; in a PAT, a transition is augmented with a sequence of tags, while TNFA allows at most one tag for each transition. Another difference is that a PAT is ε-transition free; indeed, a PAT is based on a position NFA, while a TNFA primary assumes a Thompson NFA as its basis, so that it has ε-transitions.

Let r be a regular expression, and p a position of (the AST of) r. The PAT $M(r, p)$ for r with respect to p is recursively constructed according to the structure of r as follows:

1. $M(1, p)$ is $\langle \Sigma, \{\mathtt{q}_\wedge, \mathtt{q}_\$\}, \{(_p,)_p\}, \{\langle \mathtt{q}_\wedge, a, \mathtt{q}_\$, (_p)_p\rangle \mid a \in \Sigma\}, \mathtt{q}_\wedge, \mathtt{q}_\$\rangle$.
2. For $a \in \Sigma$, $M(a, p)$ is $\langle \Sigma, \{\mathtt{q}_\wedge, p, \mathtt{q}_\$\}, \{(_p,)_p\}, \Delta, \mathtt{q}_\wedge, \mathtt{q}_\$\rangle$ where Δ consists of the transitions $\langle \mathtt{q}_\wedge, a, p, (_p\rangle$ and $\langle p, b, \mathtt{q}_\$,)_p\rangle$ for any $b \in \Sigma$.
3. If $M(r_i, p.i)$ is $\langle \Sigma, Q_i, T_i, \Delta_i, \mathtt{q}_\wedge, \mathtt{q}_\$\rangle$ for $i = 1, 2$, then $M(r_1{+}r_2, p)$ is

$$\langle \Sigma, Q_1 \cup Q_2, T_1 \cup T_2 \cup \{(_p,)_p\}, [\Delta_1 \cup \Delta_2]_p, \mathtt{q}_\wedge, \mathtt{q}_\$\rangle.$$

4. If $M(r_i, p.i)$ is $\langle \Sigma, Q_i, T_i, \Delta_i, \mathtt{q}_\wedge, \mathtt{q}_\$ \rangle$ for $i = 1, 2$, then $M(r_1 \cdot r_2, p)$ is

$$\langle \Sigma, Q_1 \cup Q_2, T_1 \cup T_2 \cup \{(_p,)_p\}, [\Delta_1 \cdot \Delta_2]_p, \mathtt{q}_\wedge, \mathtt{q}_\$ \rangle.$$

5. If $M(r, p.1)$ is $\langle \Sigma, Q, T, \Delta, \mathtt{q}_\wedge, \mathtt{q}_\$ \rangle$, then $M(r*, p)$ is $\langle \Sigma, Q, T \cup \{(_p,)_p\}, [\Delta*]_p, \mathtt{q}_\wedge, \mathtt{q}_\$ \rangle$.

Finally, for $M(r, \Lambda) = \langle \Sigma, Q, T, \Delta, \mathtt{q}_\wedge, \mathtt{q}_\$ \rangle$, we define the PAT $M(r)$ for a regular expression r as $\langle \Sigma, Q, T, \Delta \cup \Delta_0, \mathtt{q}_\wedge, \mathtt{q}_\$ \rangle$ where Δ_0 consists of the transitions $\langle \mathtt{q}_\wedge, a, \mathtt{q}_\wedge, \varepsilon \rangle$ for any $a \in \Sigma$ and $\langle \mathtt{q}_\$, a, \mathtt{q}_\$, \varepsilon \rangle$ for any $a \in \Sigma$.

In the above construction, $[\Delta]_p$, $\Delta_1 \cdot \Delta_2$ and $\Delta*$ are respectively given as follows:

$$\begin{aligned}
[\Delta]_p &= \{\langle q_1, a, q_2, \tau \rangle \in \Delta \mid q_1 \neq \mathtt{q}_\wedge, q_2 \neq \mathtt{q}_\$\} \cup \{\langle \mathtt{q}_\wedge, a, \mathtt{q}_\$, (_p\tau)_p \rangle \mid \langle \mathtt{q}_\wedge, a, \mathtt{q}_\$, \tau \rangle \in \Delta\} \\
&\quad \cup \{\langle \mathtt{q}_\wedge, a, q, \ (_p\tau \rangle \mid \langle \mathtt{q}_\wedge, a, q, \tau \rangle \in \Delta, q \neq \mathtt{q}_\$\} \\
&\quad \cup \{\langle q, a, \mathtt{q}_\$, \ \tau)_p \rangle \mid \langle q, a, \mathtt{q}_\$, \tau \rangle \in \Delta, q \neq \mathtt{q}_\wedge\} \\
\Delta_1 \cdot \Delta_2 &= \{\langle q_1, a, q_2, \tau \rangle \in \Delta_1 \mid q_2 \neq \mathtt{q}_\$\} \cup \{\langle q_1, a, q_2, \tau \rangle \in \Delta_2 \mid q_1 \neq \mathtt{q}_\wedge\} \\
&\quad \cup \{\langle q_1, a, q_2, \tau_1\tau_2 \rangle \mid \langle q_1, _, \mathtt{q}_\$, \tau_1 \rangle \in \Delta_1, \langle \mathtt{q}_\wedge, a, q_2, \tau_2 \rangle \in \Delta_2\} \\
\Delta* &= \Delta \cup \{\langle \mathtt{q}_\wedge, a, \mathtt{q}_\$, \varepsilon \rangle \mid a \in \Sigma\} \\
&\quad \cup \{\langle q_1, a, q_2, \tau_1\tau_2 \rangle \mid \langle q_1, _, \mathtt{q}_\$, \tau_1 \rangle \in \Delta, \langle \mathtt{q}_\wedge, a, q_2, \tau_2 \rangle \in \Delta, q_1 \neq \mathtt{q}_\wedge, q_2 \neq \mathtt{q}_\$\}
\end{aligned}$$

Fig. 1 shows the PAT $M(\texttt{(ab+a*)*})$ obtained by our construction, where the symbol □ stands for arbitrary letters in Σ; that is, a transition with □ actually represents several transitions obtained by replacing □ with a letter in Σ.

Notice that the PATs constructed above requires a *look-ahead* symbol. We assume that Σ includes an extra symbol \$ for indicating the "end of string," and that any string given to a PAT $M(r)$ only has a trailing \$.

Let $\mathrm{PE}(M, w) = \{\alpha \mid \ M \text{ accepts } w\$ \text{ yielding } \alpha\$ \}$. The following theorem states that a PAT is capable of exactly enumerating any, and only canonical parse configurations:

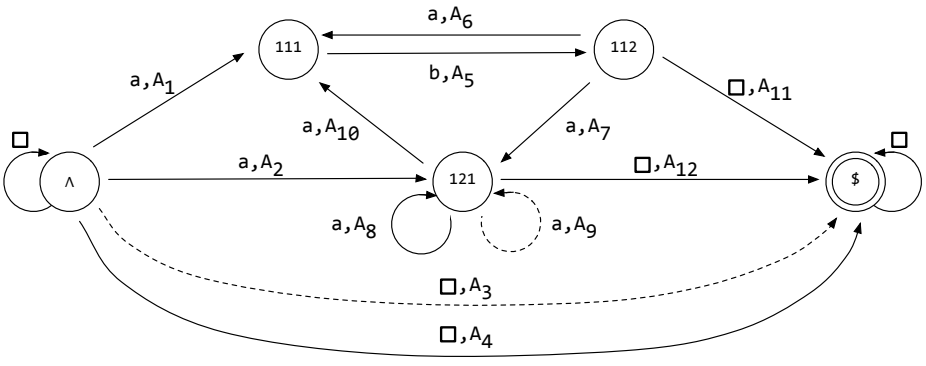
Theorem 2. $\mathrm{PC}_\Phi(r, w) = \mathrm{PE}(M(r), w)$.

4 Developing a Matching Algorithm

4.1 Basic Idea for Choosing the Most Prior Path

As mentioned before, our matching algorithm basically emulates the subset construction on the fly. The only but crucial difference is that we need to choose, in each step, the most prior one when paths converge on the same state. To spell out how to do this, consider *imaginary* stacks, one for each path. Along each path, opening parentheses are pushed on the stack in order of occurrence, and are eventually removed when the corresponding closing parentheses are encountered.

Consider two paths and the first step at which they are branching. At that moment, the stacks for these paths store exactly the same content, which, since opening parentheses occur in order of priority, corresponds to the closing parentheses that contribute to decide which path is prior to the other. To distinguish this content, we prepare a *bottom* pointer for each stack, which initially designates the top of the content (equivalently, the



A_1: (∧,(1,(11,(111
A_2: (∧,(1,(12,(121
A_3: (∧,)∧
A_4: (∧,(1,(12,)12,)1,)∧
A_5:)111,(112
A_6:)112,)11,)1,(1,(11,(111
A_7:)112,)11,)1,(1,(12,(121
A_8:)121,(121
A_9:)121,)12,)1,(1,(12,(121
A_{10}:)121,)12,)1,(1,(11,(111
A_{11}:)112,)11,)1,)∧
A_{12}:)121,)12,)1,)∧

Fig. 1. The position automata with augmented transitions generated from the regular expression (a·b+a*)*

bottom of the forthcoming parentheses) and decreases when the corresponding closing parentheses are found. Comparing the bottom pointers at each step allows us to know exactly when the corresponding closing parentheses appear.

Actually, the particular content of each stack does not matter since we already know that parentheses are correctly nested; what's really important is the minimum record of each bottom pointer that have ever achieved within the steps from the beginning to the current step. Moreover, since the index of a parenthesis indicates the value of the stack pointer when it is pushed, we no longer need the stack pointers.

This consideration allows us to develop a rather simple way of comparing the priority of paths only based on operations of the bottom pointers, without concerning each of particular parentheses at runtime. This idea is formalized in Section 4.2 below.

4.2 Formalization

First, we define the *height* of an opening parenthesis $(_p$ as $|p| + 1$ while the height of a closing parenthesis $)_p$ as p. Intuitively, the height of a parenthesis is the value of the (imaginary) stack pointer we discussed above.

For any sequences α and β of parentheses, $\alpha \sqcap \beta$ denotes the longest common prefix of α and β. For any sequence α of parentheses and a prefix α' of α, $\alpha \backslash \alpha'$ denotes the remaining sequence obtained by removing α' from α. In case $\alpha \backslash (\alpha \sqcap \beta)$ is non-empty, we denote the first element of $\alpha \backslash (\alpha \sqcap \beta)$ as α / β.

Algorithm 1. Match a string w against a regular expression r

$K := \{\mathtt{q}_\wedge\}$
$P[q][t] := -1$ for any state q and tag t in the set T_{cap} of tags for captured subexpressions
for all $n := 0, \ldots, |w|$ **do**
  Read the next letter a in $w\$$ (left to right)
  $trans := \mathtt{effective\_transitions}(a)$ // *Choose transitions available for this step*
  $\mathtt{proceed\_one\_step}(n, trans)$ // *Update D, P, K and B accordingly*
  if K contains the final state $\mathtt{q}_\$$ **then**
    From K drop all q such that $P[q][\langle_\Lambda] > P[\mathtt{q}_\$][\langle_\Lambda]$ or $P[q][\langle_\Lambda] = -1$
    break if K contains only $\mathtt{q}_\$$
  end if
end for
if K contains the final state $\mathtt{q}_\$$ **then**
  Report *SUCCESS* and output $P[\mathtt{q}_\$][t]$ for each $t \in T_{\text{cap}}$
else
  Report *FAILURE*
end if

Algorithm 2. $\mathtt{effective\_transitions}(a)$

$trans := \emptyset$
for all state $q \in Q_a$ such that $\mathtt{tag}(p, q) \neq \bot$ for some $p \in K$ **do**
  $K' := K \backslash \{p\}$
  for all state $p' \in K'$ such that $\mathtt{tag}(p', q) \neq \bot$ **do**
    $\langle \rho, \rho' \rangle := B[p][p']$
    $\rho := \min\{\rho, \mathtt{minsp}(\mathtt{tag}(p, q))\}$; $\rho' := \min\{\rho', \mathtt{minsp}(\mathtt{tag}(p', q))\}$;
    $p := p'$ **if** $\rho < \rho'$ or $\rho = \rho'$ and $D[p'][p] = 1$
  end for
  Add $\langle p, q, \mathtt{tag}(p, q) \rangle$ to *trans*
end for
return trans

Let α and β be parenthesis expressions whose frames are $\alpha_0, \ldots, \alpha_n$ and $\beta_0, \ldots, \beta_n$ respectively. Suppose that k is the fork (i.e., the index of the first different frames) of α and β. We define the *trace* of α with respect to β, written $\mathtt{tr}_\beta(\alpha)$, as a sequence $\rho_0, \ldots, \rho_n$ of integers as follows:

$$\rho_i = \begin{cases} -1 & (i < k) \\ \min\{\mathtt{lastsp}(\alpha_k \sqcap \beta_k), \mathtt{minsp}(\alpha_k \backslash (\alpha_k \sqcap \beta_k))\} & \\ \min\{\rho_{i-1}, \mathtt{minsp}(\alpha_i)\} & (i > k) \end{cases}$$

where $\mathtt{lastsp}(\gamma)$ denotes the height of the last (the rightmost) parenthesis in γ or else 0 if γ is empty, while $\mathtt{minsp}(\gamma)$ is the minimal height of the parentheses in γ or else 0 if γ is empty. Intuitively, The i-th value ρ_i of $\mathtt{tr}_\beta(\alpha)$ ($i > k$ where k is the fork) means the minimal record of the bottom pointer for α within the steps from the fork to the i-th step. The negative number -1 just means that the bottom pointer is not yet set. We denote the initial value $\min\{\mathtt{lastsp}(\alpha \sqcap \beta), \mathtt{minsp}(\alpha \backslash (\alpha \sqcap \beta))\}$ as $\mathtt{bp0}_\beta(\alpha)$.

Algorithm 3. `proceed_one_step`$(n, trans)$

1: $K := \{q \mid \langle \_, q, \_ \rangle \in trans\}$
2: $D' := D$; $P' := P$; $B' := B$
3: **for all** $\langle p, q, \alpha \rangle, \langle p', q', \alpha' \rangle \in trans$ such that $q \lessdot q'$ **do**
4: **if** $p = p'$ **then**
5: $D[q][q'] := 1$ **if** $\alpha \sqsubset \alpha'$; $D[q][q'] := -1$ **if** $\alpha' \sqsubset \alpha$
6: $\rho := \mathtt{bp0}_{\alpha'}(\alpha)$; $\rho' := \mathtt{bp0}_{\alpha}(\alpha')$
7: **else**
8: $D[q][q'] := D'[p][p']$
9: $\langle \rho, \rho' \rangle := B'[p][p']$
10: $\rho := \min\{\rho, \mathtt{minsp}(\alpha)\}$; $\rho' := \min\{\rho', \mathtt{minsp}(\alpha')\}$;
11: **end if**
12: $D[q][q'] := 1$ **if** $\rho > \rho'$; $D[q][q'] := -1$ **if** $\rho < \rho'$; $D[q'][q] := -D[q][q']$
13: $B[q][q'] := \langle \rho, \rho' \rangle$; $B[q'][q] := \langle \rho', \rho \rangle$
14: **end for**
15: **for all** $\langle p, q, \alpha \rangle \in trans$ **do**
16: $P[q] := P'[p]$
17: $P[q][t] := n$ **for all** $t \in T_{\mathrm{cap}} \cap \alpha$
18: **end for**

For $\mathtt{tr}_{\alpha'}(\alpha) = \rho_0, \ldots, \rho_n$ and $\mathtt{tr}_{\alpha}(\alpha') = \rho'_0, \ldots, \rho'_n$, we write $\mathtt{tr}_{\alpha'}(\alpha) \lessdot \mathtt{tr}_{\alpha}(\alpha')$ if $\rho_i > \rho'_i$ for the least i such that $\rho_j = \rho'_j$ for any $j > i$. For frames α and α', we write $\alpha \sqsubset \alpha'$, if the following conditions, (1) and (2), hold for some position p; (1) there exists $\alpha/\alpha' = (_p$; (2) if there exists $\alpha'/\alpha = (_q$ for some q then $p \lessdot q$.

Definition 3. *Let α and β be parenthesis expressions. We say α is* prior to *β, written $\alpha \lessdot \beta$, if either $\mathtt{tr}_{\beta}(\alpha) \lessdot \mathtt{tr}_{\alpha}(\beta)$ or else $\mathtt{tr}_{\beta}(\alpha) = \mathtt{tr}_{\alpha}(\beta)$ and $\alpha' \sqsubset \beta'$ where α' and β' are the k-th frames of α and β respectively and k is the fork of α and β.*

The order we have just defined above is essentially the same as the priority order on parse configurations:

Theorem 3. *$\langle \mathtt{PC}_{\Phi}(r, w), \lessdot \rangle$ is an order set isomorphic to $\langle \mathtt{PC}(r, w), \lessdot \rangle$.*

4.3 Algorithm

Based on the above discussion, we provide a regular expression matching algorithm. Algorithm 1 is the main routine of the algorithm, which takes, apart from a PAT $M(r)$ built from a regular expression r, an input string $w\$$ then answers whether w matches r or not. If the matching succeeds, the algorithm tells us the positions of substrings captured by subexpressions.

Throughout the entire algorithm, a couple of global variables are maintained: K, B, D and P. K stores the set of current states. Let α and β be the paths getting to states p and q respectively. Then, $B[p][q]$ stores the minimal record of the bottom pointer for α and β. $D[p][q]$ remembers which path is currently prior to the other; this decision might be overridden later. $P[p][t]$, where t is a tag (i.e., a parenthesis), is the last step number in α at which t occurs.

The main routine invokes two other subroutines; Algorithm 2 takes a letter a of the input string then returns a set of transitions actually used for that step, pruning less prior, thus ineffective, transitions, while Algorithm 3 updates the values of the global variables for the next step of computation.

Algorithm 2 assumes that a given automaton has been preprocessed so that we have at most one transition $\langle p, q, a, \tau \rangle$ for any triple $\langle p, q, a \rangle$ such that $p, q \in Q$ and $a \in \Sigma$ by removing some (less prior) transitions. The following proposition, which immediately follows by Def. 3, justifies this:

Proposition 1. *Let α and β be parenthesis expressions whose frames are the same except for the k-th frames, say α' and β', for some k. We have $\alpha \lessdot \beta$ if either $\mathtt{bp0}_{\beta'}(\alpha') > \mathtt{bp0}_{\alpha'}(\beta')$ or else $\mathtt{bp0}_{\beta'}(\alpha') = \mathtt{bp0}_{\alpha'}(\beta')$ and $\alpha' \sqsubset \beta'$.*

For the automaton in Fig. 1, the prepossessing removes the transitions with A_3 and A_9 drawn in dashed lines. In Alg. 2, $\mathtt{tag}(p, q)$ denotes the sequence α of tags such that $\langle p, a, q, \alpha \rangle \in \Delta$ for some $a \in \Sigma$, or $\bot$ if such a transition does not exist (such α is determined uniquely if it exists).

Our algorithm answers the latest positions of captured substrings[1] consistent with those given by the most prior parenthesis expression:

Theorem 4. *Let α be the least parenthesis expression in $\mathtt{PC}_{\Phi}(r, w)$ and $\alpha_0, \ldots, \alpha_n$ its frames. When the algorithm terminates, $P[\mathtt{q}_\$][t]$ gives the greatest i ($\leq n$) such that α_i contains t for any $t \in T_{\mathrm{cap}}$ that occur in α; $P[\mathtt{q}_\$][t] = -1$ for the other $t \in T_{\mathrm{cap}}$.*

We now consider the runtime cost of our algorithm. We need not count the computational cost for $\mathtt{bp}(\alpha)$, $\mathtt{bp0}_{\beta}(\alpha)$ and $\alpha \sqsubset \beta$ because they can computed, in advance, at compile time (before a particular input string is given). The number of transitions to be examined in $\mathtt{effective\_transitions}(a)$ does not exceed $|K| \cdot |Q_a|$ and we have $|K| \leq |Q_{a'}|$ for the letter a' processed in the previous iteration in Alg. 1. Since $|Q_a|$ (resp. $|Q_{a'}|$) is the number of occurrences of the letter a (resp. a') in the given regular expression plus 2, the time complexity of Alg. 2 is $O(n^2)$, where n is the number of occurrences of the most frequently used letter in the given regular expression and, likewise, we obtain $O(n(n + c))$ for Alg. 3, where c is the number of subexpressions with which we want to capture substrings. Therefore, for the length m of a input string, the time complexity of the whole algorithm is given as follows:

Theorem 5. *The time complexity of the algorithm at runtime is $O(mn(n + c))$.*

5 Related Work

The idea of realizing regular expression matching by emulating subset construction at runtime goes back to early days; see [2] for the history. For the idea of using tags in automata we are indebted to the study [9], in which Laurikari have introduced NFA with

[1] In [5], capturing within a previous iteration should be reset once; e.g, the matching *aba* to $(a(b)*)*$ should result in $P[\mathtt{q}_\$][(_{121}] = P[\mathtt{q}_\$][)_{121}] = -1$, while our algorithm currently reports $P[\mathtt{q}_\$][(_{121}] = 1$, $P[\mathtt{q}_\$][)_{121}] = 2$. This subtle gap could be resolved without affecting the order of the computational cost by replacing P with more appropriate data structures: e.g., trees.

tagged transitions (TNFA), the basis of TRE library, in order to formulate the priority of paths. Unfortunately, it is somewhat incompatible with today's interpretation [5] as for the treatment of repetition.

Dubé and Feeley have given a way of generating the entire set of parse trees from regular expressions [4] by using a grammatical representation. Frisch and Cardelli have also considered an ordering on parse trees [6], which is completely different from ours since they focus on the greedy, or first match, semantics. Their notion of "non-problematic value" is similar to, but stronger than, our canonical parse trees. They also focus on a problem of ε-transition loop, which does not occur in our case since we are based on position automata. Vansummeren [11] have also given a stable theoretical framework for the leftmost-longest matching, although capturing inside repetitions is not considered.

An implementation taking a similar approach to ours is Kuklewicz's Haskell TDFA library [8]. Although it is also based on position automata, the idea of using *orbit tags* for the comparison of paths is completely different from our approach. Another similar one is Google's new regular expression library called RE2 [3] which has come out just before we finish the preparation of this paper. Unlike ours, RE2 follows the greedy semantics rather than the leftmost-longest semantics. TRE, TDFA, RE2 and our algorithm are all based on automata, so that, while their scope is limited to regular expressions without *back references*, they all enable of avoiding the computational explosion.

Acknowledgments. The authors are grateful to anonymous reviewers for valuable comments.

References

1. The Open Group Base Specification Issue 6 IEEE Std 1003.1 2004 Edition (2004), http://www.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html
2. Cox, R.: Regular Expression Matching Can Be Simple and Fast (2007), http://swtch.com/~rsc/regexp/regexp1.html
3. Cox, R.: Regular Expression Matching in the Wild (2010), http://swtch.com/~rsc/regexp/regexp3.html
4. Dubé, D., Feeley, M.: Efficiently Building a Parse Tree from a Regular Expression. Acta Infomatica 37(2), 121–144 (2000)
5. Fowler, G.: An Iterpretation of the POSIX Regex Standard (2003), http://www2.research.att.com/~gsf/testregex/re-interpretation.html
6. Frisch, A., Cardelli, L.: Greedy Regular Expression Matching. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 618–629. Springer, Heidelberg (2004)
7. Glushkov, V.M.: The Abstract Theory of Automata. Russian Mathematical Surveys 16(5), 1–53 (1961)
8. Kuklewicz, C.: Regular Expressions: Bounded Space Proposal (2007), http://www.haskell.org/haskellwiki/Regular_expressions/Bounded_space_proposal
9. Laurikari, V.: Efficient Submatch Addressing for Regular Expressions. Master's thesis, Helsinki University of Technology (2001)
10. McNaughton, R., Yamada, H.: Regular Expressions and State Graphs for Automata. IEEE Transactions on Electronic Computers 9, 39–47 (1960)
11. Vansummeren, S.: Type Inference for Unique Pattern Matching. ACM Transactions on Programming Languages and Systems 28(3), 389–428 (2006)