Efficient Submatch Extraction for Practical Regular Expressions

Stuart Haber[1], William Horne[1,⋆], Pratyusa Manadhata[1], Miranda Mowbray[2], and Prasad Rao[1]

[1] HP Labs Princeton, 5 Vaughn Drive, Suite 301, Princeton, NJ 08540, USA
[2] HP Labs Bristol, Long Down Ave, Stoke Gifford, Bristol BS34 8QT, UK

Abstract. A *capturing group* is a syntax used in modern regular expression implementations to specify a subexpression of a regular expression. Given a string that matches the regular expression, *submatch extraction* is the process of extracting the substrings corresponding to those subexpressions. *Greedy* and *reluctant* closures are variants on the standard closure operator that impact how submatches are extracted. The state of the art and practice in submatch extraction are automata based approaches and backtracking algorithms. In theory, the number of states in an automata-based approach can be exponential in n, the size of the regular expression, and the running time of backtracking algorithms can be exponential in ℓ, the length of the string. In this paper, we present an $O(\ell c)$ runtime automata based algorithm for extracting submatches from a string that matches a regular expression, where $c > 0$ is the number of capturing groups. The previous fastest automata based algorithm was $O(n\ell c)$. Both our approach and the previous fastest one require worst-case exponential compile time. But in practice, the worst case behavior rarely occurs, so achieving a practical speed-up against state-of-the-art methods is of significant interest. Our experimental results show that, for a large set of regular expressions used in practice, our algorithm is approximately twice as fast as Java's backtracking based regular expression library and approximately twenty times faster than the RE2 regular expression engine.

1 Introduction

Regular expressions (REs) are a succinct method to formally represent sets of strings over an alphabet. Given an RE and a string, the RE *matches* the string if the string belongs to the set described by the RE. Many RE implementations also support *search*, i.e. finding the first substring of an input string that matches the RE. In this paper we only address matching, which has practical applications in network security, bioinformatics, and other areas.

Most textbooks on compiler design and related topics (e.g. [4]) describe REs from a theoretical perspective, but omit additional features including *capturing*

⋆ Corresponding author.

groups and *reluctant closure*, that are supported in practical implementations of REs, such as PCRE [3], Java [7], and RE2 [2].

A *capturing group* is a syntax used to specify a subexpression. Given a string that matches the regular expression, *submatch extraction* is the process of extracting the substrings corresponding to those subexpressions. This feature enables regular expressions to be used as parsers. Parentheses are commonly used to indicate the beginning and end of a capturing group. For example, the RE $(.*) = (.*)$ could be used to parse key-value pairs. (Here, the meta-character '.' matches any character in the alphabet, so that $.*$ matches any string.)

The reluctant closure operator, denoted $*?$, appears in both Java and PCRE, and is widely used in practice. This operator is a variant of the standard *greedy closure* operator for REs, denoted $*$, with different submatching behavior: where other rules do not apply, shorter submatches to a subexpression $E*?$ take priority over longer ones, whereas for $E*$ the reverse is true. For example, consider matching the string $a = b = c$ first against $(.*?) = (.*)$, and then against $(.*) = (.*?)$. In the first case the capturing groups match a and $b = c$, respectively, while in the second case the submatches are $a = b$ and c.

If the two closure operators in this example are both greedy or both reluctant, then it is ambiguous which of these two assignments of submatches should be reported by a matching algorithm. There are no formal standards that specify precedence rules for such cases. We aimed for consistency with Java's implementation, which we verified with extensive testing.

Though REs are widely studied, the problem of efficiently implementing submatch extraction has not received much attention. The state of the art includes backtracking and automata based approaches. Java, PCRE, Perl, Python, Ruby, and many other tools implement submatch extraction using recursive backtracking, where an input string may be scanned multiple times before a match is found. Pike implemented the first automata based submatch extraction algorithm in the `sam` text editor [8] based on Thompson's algorithm [10] for RE matching, which converts the RE to a nondeterministic finite automaton (NFA). RE2 uses a combination of deterministic finite automata (DFAs) and NFAs to improve the time efficiency of submatch extraction [2]. RE2 uses DFAs to locate a RE's overall match location in an input string and then uses NFA-based matching on the overall match to extract submatches. Laurikari [5,6] studied ways to implement submatch extraction using a DFA. Both Pike's and Laurikari's implementations require worst-case exponential time to construct the DFA. Once the DFA has been constructed these implementations run in $O(n\ell c)$ time, where n is the number of states in the NFA corresponding to a RE, ℓ is the length of the string being matched, and $c > 0$ is the number of capturing groups.

Our algorithm is suitable for settings where the automaton is compiled once and matched many times against different input strings. This scenario is common, for example, in security applications such as intrusion detection systems and event processing systems which rely heavily on REs and require high-speed matching of input strings.

In this paper, we present an $O(\ell c)$ runtime automata based algorithm for RE matching and submatch extraction. The time complexity of the runtime operation for our algorithm does not depend on n, but our algorithm may require $O(2^n)$ compile time and storage space in the worst case.

However, the asymptotic analysis of these algorithms is deceiving. Backtracking and automata-based approaches almost never have the worst-case behavior on REs that are used in practice. Thus, achieving a practical speed-up against state-of-the-art methods is of significant interest. Our experimental results show that, for a large set of regular expressions used in practice, our algorithm is approximately twice as fast as Java's backtracking based regular expression library and approximately twenty times faster than RE2.

2 Valid Submatch

For the purposes of this paper, the syntax of REs with capturing groups and reluctant closures on an alphabet Σ is

$$E ::= \epsilon \mid a \mid EE \mid E\texttt{|}E \mid E\texttt{*} \mid E\texttt{*?} \mid \texttt{(}E\texttt{)}$$

where a stands for an element of Σ, and ϵ is the empty string. The notation $\texttt{(}E\texttt{)}$ indicates a capturing group. If X, Y are sets of strings we use XY to denote concatenation, i.e. $XY = \{xy : x \in X, y \in Y\}$, and $X|Y$ to denote the union of sets X and Y. If β is a string and B a set of symbols we use $\beta|_B$ to denote the string in B^* obtained by deleting from β all elements that are not in B.

Grouping terms is optional when the order of operations is clear. Specifically, capturing groups have a higher priority than greedy and reluctant closures, which have a higher priority than concatenation, which has a higher priority than union.

We use indices to identify capturing groups within a RE. Given a RE E containing c capturing groups, we assign indices $1, 2, \ldots c$ to each capturing group in the order of their left parentheses as E is read from left to right. We use the notation $idx(E)$ to refer to the resulting *indexed RE*. For example, if $E = ((a)*|b)(ab|b)$ then $idx(E) = ((a)_2*|b)_1(ab|b)_3$.

We introduce the set of symbols $T = \{S_t, E_t : 1 \le t \le c\}$, referred to as *tags*, which will be used to encode the start and end of capturing groups.

The language $L(F)$ for an indexed RE $F = idx(E)$ is a subset of $(\Sigma \cup T)^*$, defined by $L(\epsilon) = \{\epsilon\}$, $L(a) = \{a\}$, $L(F_1F_2) = L(F_1) \cdot L(F_2)$, $L(F_1|F_2) = L(F_1) \cup L(F_2)$, $L(F*) = L(F*?) = L(F)*$, and $L((F)_t) = \{S_t \alpha E_t : \alpha \in L(F)\}$, where $()_t$ denotes a capturing group with index t.

Definition 1. *A* valid assignment of submatches *for RE E and input string α is a map* SUB : $\{1, \ldots, c\} \to \Sigma^* \cup \{\text{NULL}\}$ *such that there exists $\beta \in L(idx(E))$ satisfying: (i) $\beta|_\Sigma = \alpha$; (ii) if S_t occurs in β then* SUB$(t) = \beta_t|_\Sigma$*, where β_t is the substring of β between the last occurrence of S_t and the last occurrence of E_t; (iii) if S_t does not occur in β then* SUB(t) = NULL. □

For example, given the RE $E = ((a)*|b)(ab|b)$ and an input string $aaab$, the assignment sub with SUB$(1) = aaa$, SUB$(2) = a$, and SUB$(3) = b$ is valid for $\beta = S_1S_2aE_2S_2aE_2S_2aE_2E_1S_3bE_3$.

If $\alpha \in \Sigma^*$ we say that α *matches* E if and only if $\alpha = \beta|_\Sigma$ for some $\beta \in L(idx(E))$. Note that for a RE without capturing groups, this coincides with the standard definition of the set of strings matching the expression.

Given a RE containing capturing groups and an input string, the task of a submatch extraction algorithm is to report a valid assignment of submatches if there is one, and to report that the string does not match if there is not.

3 Description of the Algorithm

For this entire section we fix a RE E, and show how to compile E into two deterministic automata, denoted M_3 and M_4, that will be used to match a string. This is done in the preprocessing stage. For the matching and extraction operations, we use M_3 to determine whether the input string matches E, and if it does, we use M_4 to determine what submatches to report.

To understand the need for two automata, consider the RE $(.*)a|(.*)b$. If a procedure is to match a given string against this expression and at the same time to decide whether to match the string to the first or the second capturing group, then clearly the procedure must look ahead to the end of the string. A finite automaton, whose only operating memory is carried by the state it is in, requires other means in order to perform this "look ahead". We achieve this by converting our RE into the DFA M_3 that we run backwards on the input string, accepting or rejecting it as a match; while doing so, we journal the states this DFA goes through. This journaled sequence of states is used as the input to the second automaton, M_4, which has been constructed using tagging information from our RE (encoded by the symbols $S_i, E_i, +, -$). This automaton outputs the appropriate tagging information, which in turn is used to report submatches.

In the preprocessing stage we first construct two other automata, M_1 and M_2, and then use these to derive M_3 and M_4, as described below.

3.1 The Automaton M_1

The automaton M_1 encodes $idx(E)$ as a automaton. We use separate transitions with labels S_t and E_t to indicate the start and end of a capturing group with index t, in addition to transitions labeled with alphabet characters to consume an input character, and transitions labeled with $+$ and $-$ to indicate submatching priorities.

The automaton M_1 is described by the tuple $(Q_1, \Sigma_1, \Delta_1, s_1, f_1)$, where Q_1 is a set of states identified by the integers in the set $\{1, 2 \ldots f\}$, Σ_1 is the alphabet $\Sigma \cup \{+, -\} \cup T$, where $+$ and $-$ are two special alphabet characters that will be described below, Δ_1 is a transition function, $s_1 = 1$ is the start state and $f_1 = f$ is the unique final state. Δ_1 is built using structural induction on $idx(E)$ following the rules illustrated by the diagrams in Fig. 1. The initial state is marked with $>$ and the final state with a double circle. A dashed arrow with label F or G is used as shorthand for the diagram corresponding to the indexed

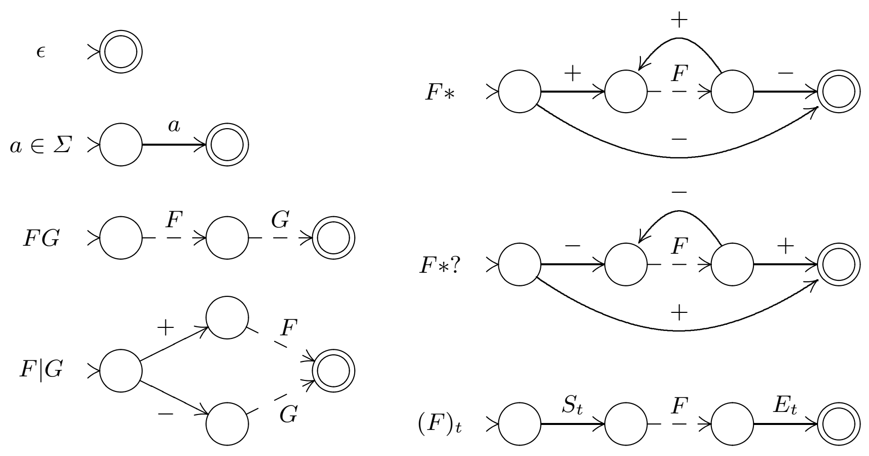

Fig. 1. Rules for the construction of M_1

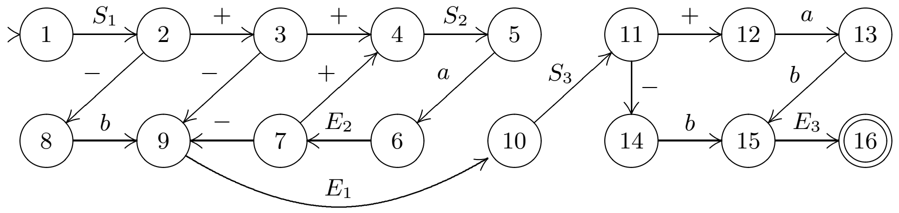

Fig. 2. The DFA M_1 for $((a)*|b)(ab|b)$

expression F or G. For example, the automaton M_1 for $((a)*|b)(ab|b)$ is shown in Fig. 2.

If x is any directed path in M_1, when it is considered as a directed graph, we write $ls(x)$ for its label sequence. For example in Fig. 2 $ls(8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 14) = bE_1S_3-$.

Let $\pi : Q_1 \times Q_1 \rightarrow T^*$ be a mapping from a pair of states to a sequence of tags, to be used in the constructions below, defined as follows. For any two states $p, q \in Q_1$, consider a depth-first search of the graph of M_1, beginning at p and searching for q, using only transitions with labels from $T \cup \{+, -\}$. The construction rules for M_1 ensure that if there is any state with two different outgoing transitions, one will be labeled '+' and the other '−'. The search explores all states reachable via the transition labeled '+' before following the one labeled '−'. If this search succeeds, finding successful search path $\lambda(p, q)$, then $\pi(p, q) = ls(\lambda(p, q))|_T$ is the sequence of tags along this path. If it fails, then $\pi(p, q)$ is undefined. Note

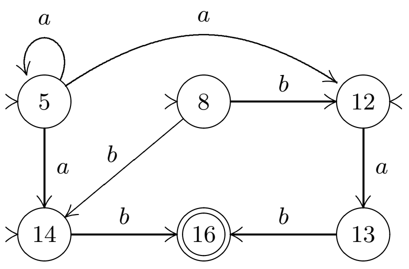

Fig. 3. The automaton M_2 for the RE $((a)*|b)(ab|b)$

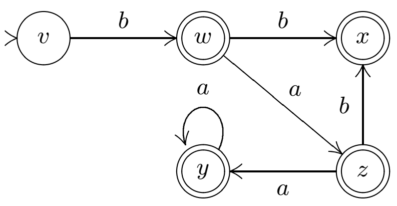

Fig. 4. The automaton M_3 for for the RE $((a)*|b)(ab|b)$, where $v = \{16\}$, $w = \{13, 14\}$, $x = \{8\}$, $y = \{5\}$, and $z = \{5, 12\}$

that $\pi(p, p)$ is defined to be the empty string, and that this description of the search uniquely specifies $\lambda(p, q)$, if it exists.

3.2 The Automaton M_2

Next, we convert M_1 into another automaton, the NFA M_2, described by the tuple $(Q_2, \Sigma, \Delta_2, S_2, f)$. The set Q_2 consists of the final state of M_1 together with any state in M_1 that has an outgoing transition labeled with a symbol in Σ, i.e.

$$Q_2 = \{f\} \cup \{q : \exists a \in \Sigma, p \in Q_1, (q, a, p) \in \Delta_1\}$$

If $p, q \in Q_2$ and $a \in \Sigma$, there is a transition $(p, a, q) \in \Delta_2$ if and only if there exists a state $r \in Q_1$ such that $(p, a, r) \in \Delta_1$ and $\pi(r, q)$ is defined. S_2 is a set of initial states, corresponding to those states, $p \in Q_2$, for which $\pi(1, p)$ is defined.

For example, the automaton M_2 for $((a)*|b)(ab|b)$ is shown in Fig. 3.

3.3 The Automaton M_3

Next, we convert M_2 into the DFA M_3, specified by the tuple $(Q_3, \Sigma, \Delta_3, \ s_3, F_3)$. The construction of M_3 from M_2 is a standard powerset construction of a DFA from a reversed NFA [9], modified in order to process the input string backwards. Specifically, each state in Q_3 corresponds to a subset of states in the powerset of Q_2. The initial state s_3 is $\{f\}$. We initialize Q_3 to $\{\{f\}\}$, and recursively add states r to Q_3 by constructing for each $a \in \Sigma$ the set

$$P(r, a) = \{p \in Q_2 : (p, a, q) \in \Delta_2 \text{ for some } q \in r\},$$

i.e. the set of states from which there is a transition labeled a to an element of r. If this set is not empty, it is added to Q_3 and the transition $(r, a, P(r, a))$ is added to Δ_3. We explore each previously unexplored state in Q_3 until there are no states in Q_3 left to explore. The set of final states in M_3, F_3, consists of any state q in Q_3 such that $q \cap S_2$ is not empty. The DFA M_3 for $((a)*|b)|(ab|b)$ is shown in Fig. 4.

3.4 The Automaton M_4

Next, we use M_1, M_2 and M_3 to construct another automaton, M_4, described by the tuple $(Q_4, \Sigma_4, \Delta_4, s_4)$. Q_4 is essentially M_2 with one extra state, where the input alphabet is $\Sigma_4 = Q_3$ instead of Σ, and some edges are deleted. Specifically, we introduce a new state labeled '0', which will be the start state of M_4, so that $Q_4 = Q_2 \cup \{0\}$. This is a DFA except that the transition function is a four-tuple, i.e. $\Delta_4 \subseteq Q_2 \times Q_3 \times Q_2 \times T^*$.

The definition of M_4 uses a partial ordering on label sequences of paths in M_1 that corresponds to the priorities for submatches. The intuition for M_4 is that a transition (p, Q, q, τ) of M_4 exists if, among all the paths in M_1 that have start state p, end state in Q, first label in Σ and no other labels in Σ, the path with the highest-priority label sequence ends at state q and has label sequence $a\tau$ for some alphabet symbol $a \in \Sigma$. The output τ encodes the capturing groups that are entered and left as this path is followed; during the runtime operation, this information will be used to determine the submatch that should be reported for each capturing group.

Let $\prec$ be the lexicographic partial ordering on Σ_1^* that is induced by the relation $\{(a,a) : a \in \Sigma_1\} \cup \{(-,+)\}$ on Σ_1. For example, if a, b, c are different elements of Σ, then $a \prec a\text{-}{+}b \prec a{+}c$, but $ab \not\prec ac$ and $ac \not\prec ab$. Finally, we define Δ_4, the transition function for M_4, as follows. Let (p, Q, q, τ) be in Δ_4 iff there exist $p, r \in Q_2$, $Q \in Q_3$, $q \in Q$, $a \in \Sigma$, such that $(p, a, r) \in \Delta_1$, $\pi(r, q)$ is defined, and

$$\tau = \pi(r,q) = (\max_{\prec}\{ls(\lambda(r,q')) \ : \ q' \in Q\}) \mid_T .$$

Similarly, let $(0, Q, q, \tau)$ be in Δ_4 iff there exist $Q \in Q_3$, $q \in Q$ such that $\pi(1, q)$ is defined, and

$$\tau = \pi(1,q) = (\max_{\prec}\{ls(\lambda(1,q')) \ : \ q' \in P\}) \mid_T .$$

For space considerations, we omit from this paper the proof that the maximal elements used in these definitions exist, and are unique. Continuing with our running example, the automaton M_4 for $((a){*}|b)(ab|b)$ is shown in Fig. 5.

3.5 Runtime Operation

We extract submatches for a string $a_1 \ldots a_\ell \in \Sigma^*$ in runtime in three steps:

1. We process the string $a_\ell a_{\ell-1} \ldots a_1$ using M_3. As it is processed, we journal the states $q_\ell, q_{\ell-1}, \ldots$ visited during the processing, where q_ℓ is $\{f\}$, the initial state of M_3. If the processing terminates before the whole input string has been processed (i.e. because we hit a "dead state" of M_3), or terminates with $q_0 \notin F_3$, we report that the string does not match and stop. This step runs in $O(\ell)$ time.
2. If we did not stop in the previous step, we run M_4 on input $q_0, q_1, \ldots q_\ell$, using an additional data structure along the way in order to discover the

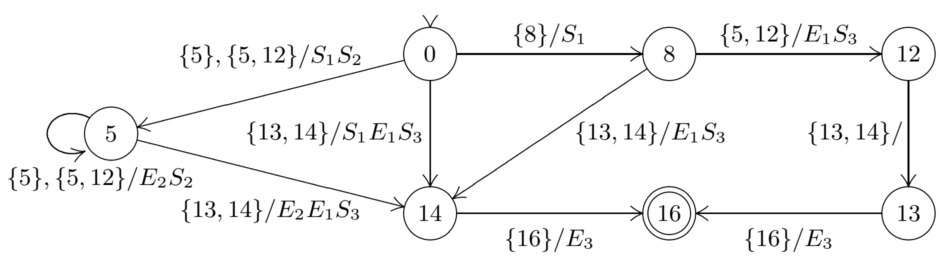

Fig. 5. The automaton M_4 for the RE $((a)*|b)(ab|b)$. Transitions are labeled with a slash separating inputs from outputs.

submatch values for each capturing group. The data structure consists of an array of length $2c$, indexed by elements of T, all initialized to NULL. While processing the ith transition, namely $(q_i, P, q_{i+1}, \tau) \in \Delta_4$, for each tag in $\tau \in T^*$ we write i in the array entry corresponding to the tag, overwriting the current entry. This step runs in $O(\ell c)$ time.

3. We use the resulting array to read off the submatches from the input string, as follows. If the array entries for the tags S_j and E_j are s_j and e_j, respectively, then the submatch for capturing group j is $a_{s_j+1} \ldots a_{e_j}$. If the array entries S_j and E_j are NULL, then there is no submatch for the jth capturing group. This step runs in $O(\ell c)$ time.

The first two steps together are called *matching*; the third step is called *extraction*. For example, consider processing the input string $aaab$ for the RE $((a)*|b)|(ab|b)$. In step 1, we process the string $baaa$ with M_3. The states visited are $\{16\}$, $\{13, 14\}, \{5, 12\}, \{5\}, \{5\}$ (see Fig. 4). In step 2, we run automaton M_4 with input $\{5\}, \{5\}, \{5, 12\}, \{13, 14\}, \{16\}$, writing entries in the array with each transition (see Fig. 5). The resulting array reads

$$[S_1, E_1, \quad S_2, E_2, \quad S_3, E_3] = [0, 3, \quad 2, 3, \quad 3, 4].$$

In step 3, we read from the array that the three capturing groups have respective submatches aaa, a, and b.

To see that the $O(\ell c)$ complexity bound in step 2 gives the worst-case runtime for our algorithm, suppose that $E = [a(F_1)(F_2)...(F_c)]^*$ (using square brackets to denote a non-capturing group) with $a, b \in \Sigma$, $F_1 = b|\epsilon$, $F_2 = bb|\epsilon$, $F_3 = bbb|\epsilon \ldots$, and $a_i = a$ for all $1 \leq i \leq \ell$. Then for $1 \leq i \leq \ell$, the string output by M_4 when processing q_i in the second step of the operation is $S_1E_1S_2E_2 \ldots S_cE_c$, and so in this case the operation updates $2\ell c$ array elements.

Note that in our analysis, we assume that we can read or write the index of one of the states of our automata in constant time and space. In practice, we never run our algorithm for a RE whose automaton is exponentially large.

3.6 Correctness

Here we prove the theorem, which shows the correctness of our algorithm.

Theorem 2. *Suppose that the input string $\alpha = a_1 \dots a_n$ matches the RE E. Then our algorithm reports a valid assignment of submatches for E and α.*

Proof. We will prove the theorem by constructing a string $\gamma \in L(E)$, and then showing that the assignment of submatches for E and α satisfies the three properties in Definition 1 for a valid assignment, with β equal to γ.

The first step is to show that the operation of our algorithm does not terminate before all of $q_0, q_1, \dots q_\ell$ have been processed by M_4. Since α matches E, it is accepted by M_2, and by Rabin and Scott's result relating languages of automata [9] this implies that the reverse of α is accepted by M_3; thus, the first step of the runtime operation does not terminate early, and ends at state q_0 in F_3. By definition of F_3, there is some $j \in q_0$ such that $\pi(1, j)$ is defined. Therefore $(0, q_0, j_0, \tau_0) \in \Delta_4$ for some $j_0 \in q_0$ and $\tau_0 = \pi(1, j_0) \in T^*$. So the processing of $q_0, q_1 \dots q_\ell$ by M_4 does not terminate before q_0 has been processed.

Suppose inductively that $1 \leq i \leq \ell$, the processing of $q_0 \dots q_\ell$ by M_4 does not terminate before q_{i-1} has been processed, and that $j_{i-1} \in q_{i-1}$, where j_{i-1} is the state of M_4 reached just after q_{i-1} has been processed. Now, q_i is the i^{th} state of Q_3 visited when $a_\ell a_{\ell-1} \dots a_1$ is processed by M_3, and so it is the set of elements of Q_2 from which there is a path in M_2 from q to f with label sequence $a_{i+1} \dots a_\ell$. Therefore, $(j_{i-1}, a_i, j) \in \Delta_2$ for some $j \in q_i$. By the definition of M_2, there is some $k_i \in Q_1$ such that $e_i = (j_{i-1}, a_i, k_i) \in \Delta_1$ and $\pi(k_i, j)$ is defined. By the construction of M_4, it follows that there is some $j_i \in q_i$ and $\tau_i = \pi(k_i, j_i) \in T^*$ such that $(j_{i-1}, q_i, j_i, \tau_i) \in \Delta_4$. This shows that for $1 \leq i \leq \ell$, q_i is processed in step 2 of the operation, as required.

Note that $j_\ell \in q_\ell = \{f\}$, so $j_\ell = f$. Let y be the path in M_1 from 1 to f obtained by concatenating $\lambda(1, j_0)$, e_1, $\lambda(k_1, j_1)$, $\dots$ e_ℓ, $\lambda(k_\ell, j_\ell)$. Now we can define γ: it is $ls(y)|_{\Sigma \cup T}$. Note that it is equal to the concatenation of τ_0, a_1, τ_1, $\dots$ a_ℓ, τ_ℓ.

It is straightforward to prove by induction on the size of E that $L(E) = \{\beta|_{\Sigma \cup T} : M_1 \text{ accepts } \beta\}$. The automaton M_1 accepts $ls(y)$, so $\gamma \in L(E)$.

We will now show that the assignment of submatches reported by the operation satisfies the three criteria for a valid assignment, with β equal to the string γ. Property (i) holds because $\gamma|_\Sigma = a_1 \dots a_\ell = \alpha$.

For property (ii), observe that for $0 \leq i \leq \ell$, when q_i is processed in step 2 of the operation, i is written in the array entry for each tag in τ_i. Thus if the array entries for S_j and E_j at the end of step 2 are s_j and e_j respectively, then the last occurrence of S_j in γ lies before a_1 if $s_j = 0$, between a_{s_j} and a_{s_j+1} if $0 < s_j < \ell$ or after a_ℓ if $s_j = \ell$, and similarly for e_j and the last occurrence of E_j in γ. Property (ii) follows.

For property (iii), observe that it follows from the definition of $L(E)$ that if S_t occurs in a string in $L(E)$, then E_t must also occur in the string. Suppose that S_t does not occur in γ. Then neither does E_t, and so S_t, E_t do not occur in any of $\tau_0, \dots \tau_\ell$. So at the end of step 2 the array entries for S_t, E_t are NULL, and step 3 reports that there is no match to capturing group t, as required.

Table 1. Results for DHCP and Snort log experiments (times in microseconds)

RE	Our Algorithm			Java			RE2	
	compile	match	extract	compile	match	extract	compile	match+extract
DHCP	16,993	47,685	19,809	44	119,852	21,423	32	1,153,118
Snort	24,761	64,484	804	14	138,138	1,666	22	1,414,001

4 Evaluation

Our first set of experiments deals with parsing logs for Microsoft DHCP logs and for Snort, which is an open source intrusion detection system. DHCP logs are a simple comma-separated format with exactly eight fields. To parse such a log record, the following RE is used,

```
([^,]*),([^,]*),([^,]*),([^,]*),([^,]*),([^,]*),([^,]*),([^,]*)
```

where the syntax `[^,]` means any character except a comma.

For Snort logs, we must extract the source and destination IP addresses and the source and destination ports (if they exist). The RE we used is

```
.*? (\d+\.\d+\.\d+\.\d+)(:\d+)? -> (\d+\.\d+\.\d+\.\d+)(:\d+)? .*
```

where the metacharacter `\d` represents any numeral, the operator `+` is a variation on closure that requires at least one instance of its operand, and the operator `?` means exactly zero or one instances of its operand.

All experiments were performed on a workstation with twelve 2.67GHz cores and 6GB of RAM. Our algorithm was implemented in Java, whereas RE2 is implemented in C++.

We ran an experiment where we matched 100,000 lines of DHCP log files and 25,741 lines of Snort log files against the regular expressions and experimentally evaluated it in comparison to Java and RE2, as shown in Table 1. Since RE2 matches and extracts in a single operation, these are grouped in the table. Clearly, our algorithm is much slower in the compilation phase than either Java or RE2. But as we have discussed in the introduction, we are willing to incur a significant penalty in the compilation phase, since for the problems we are interested in, we perform compilation once, but matching and extraction many times for each RE. When we amortize the compilation time over the matching and extraction time with multiple strings, our algorithm can actually outperform Java and RE2. This result is surprising given that RE2 has previously been reported to be faster than other `C/C++` based RE engines [1].

Next, we ran several experiments evaluating our algorithm for performance, storage usage, and correctness on a set of REs that are included as part of a commercial Security Information and Event Management (SIEM) system that uses 16,805 unique REs. Of these REs, 7732 (46.0%) have no capturing groups, and thus can be matched with an ordinary DFA; 7596 (45.2%) can be implemented with our two pass algorithm; 1396 of these REs (8.3%) can be implemented more

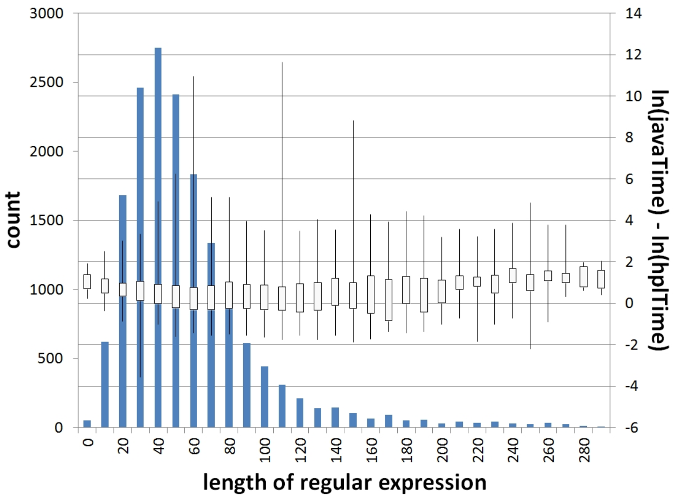

Fig. 6. Performance as a function of RE length

efficiently using a variation on our algorithm that only requires one pass (the details of that algorithm are beyond the scope of this paper); 51 (0.3%) caused the size of M_3 in our algorithm to grow beyond 4096 states, at which point we declared failure; 20 (0.1%) have syntactic features which we have not yet implemented in our algorithm, but which we believe will not impact performance.

For the first four categories of REs, we synthetically generated 1,000 matching strings for each of the REs. We then measured the time to match those strings using both our algorithm and Java. The results (broken down by RE length) are shown in Fig. 6. The blue bars are a histogram of the number of REs that have a length in each bin range. The count of the number of REs is shown on the left-hand y-axis. We then measure the performance as the log of the time taken by Java minus the log of the time taken by our algorithm in order to show speedups in both directions symmetrically. We then plot those results as a box plot showing the first, second, third and fourth quartiles for each bin. The min and max performance are the end of the line segments, while the range between the second and third quartile is shown in the box. The range of performance values are shown on the right-hand y-axis. As can be seen in the figure, our algorithm is faster than Java most of the time, regardless of the length of the RE, often significantly faster. On average, we are 2.3 times faster than Java.

This was a computationally intensive test which took over 8 hours on our workstation at 580,694 matches per second: we performed 1,000 tests on 1,000 strings for 16,724 REs. We omitted the performance comparisons against RE2 because RE2 would simply take too long.

Theoretically, the number of states in a DFA built using a powerset construction could be exponential in the size of the RE. However, as described above, less than 0.3% of the REs exhibited such behavior. In fact, for 99% of the DFAs built with a powerset construction, the ratio of the number of states to the length of the RE string was less than 5.25. For approximately 58% of the REs, the DFA actually had fewer states than the length of the RE string. The average RE needed 28 kBs for the transition tables and other associated data structures.

We were able to measure the memory usage of our own algorithm, but accurately measuring the data structures associated with regular expressions is infeasible in third party software. Regardless, backtracking algorithms generally just need enough space to store the regular expression itself, which is essentially negligible.

Although we have proved that our algorithm is guaranteed to generate a valid assignment of submatches, we are particularly interested in showing that our algorithm generates the same submatches as Java since there may be multiple valid assignments of submatches for a given RE. We synthetically generated 15 matching and non-matching strings for each RE in the first four categories. The submatches extracted by our approach and Java were identical.

5 Summary

In this paper, we introduced a new algorithm for converting REs to automata that handles submatch extraction and reluctant closures. Our experimental results show that our algorithm is approximately twice as fast as Java's backtracking based regular expression library and approximately twenty times faster than RE2 on real-world REs used for several problems involving processing of security event logs, including a comprehensive test of the algorithm against a database of 16,724 REs used by a commercial SIEM system.

References

1. Benchmark of Regex Libraries (July 2010), `http://lh3lh3.users.sourceforge.net/reb.shtml`
2. RE2 (January 2012), `http://code.google.com/p/re2/`
3. PCRE (2011), `http://www.pcre.org/`
4. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation. Addison-Wesley (2003)
5. Laurikari, V.: NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In: Proc. of the 7th Int. Symp. on String Processing and Information Retrieval, pp. 181–187 (2000)
6. Laurikari, V.: Efficient submatch addressing for regular expressions. Master's thesis, Helsinki University of Technology (2001)
7. Nourie, D., McCloskey, M.: Regular Expressions and the Java Programming Language (2010), `http://java.sun.com/developer/technicalArticles/releases/1.4regex`
8. Pike, R.: The Text Editor sam. Softw. Pract. Exper. 17, 813–845 (1987)
9. Rabin, M.O., Scott, D.: Finite automata and their decision problems. IBM J. Research and Development 3(2) (April 1959), doi:10.1147/rd.32.0114
10. Thompson, K.: Programming techniques: Regular expression search algorithm. Comm. ACM 11, 419–422 (1968)