

Efficient Submatch Extraction for Practical Regular Expressions

Stuart Haber¹, William Horne^{1,*}, Pratyusa Manadhata¹, Miranda Mowbray²,
and Prasad Rao¹

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 $O(nlc)$ 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.