BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2023
http://excel.fit.vutbr.cz

Matching Regexes with Back-References Using Register Set Automata

Jan Vašák*

Abstract
Regexes with back-references are currently usually matched using backtracking algorithms, which can be very inefficient. This work reports on an implementation of matching regexes using register set automata, a computational model that allows deterministic representation of regexes with back-references.

*xvasak01@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Back-references are a common and useful extension of regexes. They, however, cannot be matched using traditional algorithms based on finite automata (FAs), as FAs are not expressive enough to describe back-references. Extensions of FAs, such as register automata (RAs), that are powerful enough to describe regexes with back-references are not consistently determinisable.

Given these limitations, state-of-the-art regex matchers that support regexes with back-references use backtracking algorithms to match them [1, 2]. However, usage of these algorithms can lead to the so-called catastrophic backtracking, which can cause massive slowdown in matching the regex. This can, e.g., leave web applications making use of such regexes vulnerable to a regular expression denial of service (ReDoS) attack (in this scenario, the attacker would maliciously input a string that would cause catastrophic backtracking, using up resources and making the web unresponsive to other users).

Register set automata (RsAs), presented in [3], are a computational model that can be used to determinise a large class of RAs, making them useful for matching regexes with back-references that can be converted to RAs belonging to the determinisable class. Thus, a regex matcher could use an automata-based algorithm to match this class of regexes.

We show that given certain (unfortunate) combinations of a regex and an input, even a prototype RsA-based regex matcher (implemented in Python) can outperform highly optimized regex matchers that use backtracking algorithms.

2. Preliminaries

Register Automata. Register automata extend finite automata by adding a finite set of *registers*. Each register can store a symbol from the input string, or it can be empty. On a transition, each register is either assigned a value of the current input symbol, a value stored in a register, or it is emptied. Transitions also have two *guard* sets $g^{=}$ and $g^{\neq}$. The automaton can only move over the transition if the current input symbol is equal to the value stored in every register in $g^{=}$ and is not equal to the value stored in any register in $g^{\neq}$. We also allow sets of symbols on transitions instead of singular symbols for convenience in regex matching. An example of an RA can be seen in Figure 1.

Register Set Automata. Register set automata are the same as RAs, except the registers are replaced by *set-registers*. Set-registers (often simply referred to as registers within the context of RsAs) can hold a set of symbols from the input string. They are updated with a set which can contain registers and the input symbol. Guards of an RsA transition $(g^{\in}, g^{\notin})$ are semantically the same as RA guards, except that membership is tested instead of equality. Figure 2 shows an example of an RsA.

Determinisation of RAs. A large class of RAs can be determinised into *deterministic* RsAs (DRsAs) using an algorithm presented in [3].

3. Implementation

A prototype regex matcher using an algorithm based on RsAs was implemented in Python.

RA Construction. A simple regular expression parser was implemented, creating an abstract syntax tree representing the input regex. An example of such a syntax tree with a corresponding regex is shown in Figure 3. An RA is constructed using this syntax tree by creating a sub-automaton for each sub-regex and joining them together. An example of an RA created this way is shown in Figure 4.

Determinisation and Matching An adaptation of the determinisation algorithm was created to be better suited for regex matching. The created RA needs to first be preprocessed into a form accepted by our implementation of the determinisation algorithm. See an example of such an RA in Figure 5. If the created RA falls into the determinisable class of RAs, an equivalent DRsA is created. See Figure 6 for an example of a DRsA output by the algorithm. Words are then matched to the regex by simply *running* them on the created DRsA.

4. Experiments

To show the advantages of an automata-based algorithm for regexes with back-references, a particularly hard combination of a regex and an input was chosen. The regex being

$$\texttt{/\^{}.*(.).*\textbackslash 1.*;.*;.*\textbackslash 1\$/},$$

and the input words being `'a;a;a;a'`, `'a;;a;a;a'`, ..., `'a;`^{994}`a;a;a'`. The regex and the input words were then measured on our prototype implementation and version 3.6 of the pattern matching tool `grep` [4]. Graphs of the results are shown in Figures 7 and 8. They show that the time-to-match grew exponentially for `grep`, while our implementation's time-to-match remained almost constant (for the 1000 character long word, `grep` took around 30 minutes, while our prototype took around 0.16 seconds).

5. Conclusion and Future Work

It was shown that creating RAs from regexes and determinising them into DRsAs is possible and practical, especially for longer inputs. Furthermore, the RsA-based algorithm for regex-matching will outperform backtracking algorithms for certain types of regexes. Importantly, these regexes cause extreme slowdown in backtracking algorithms, which is an issue automata-based algorithms do not have.

In the future, we will analyze regexes that are actually used in practice to see how many are potentially problematic for backtracking algorithms, and how many of those are determinisable into DRsAs. Also, an efficient implementation (as opposed to the prototype implemented for this report) for manipulating RsAs (and by extension for matching regexes) is planned.

Acknowledgements

I would like to thank my supervisor, Ing. Ondřej Lengál Ph.D, for providing excellent guidance and feedback during my project practice that this submission is based on.

References

[1] Free Software Foundation, Inc. *GNU grep 3.8 manual*, September 2022.

[2] Philip Hazel. *Perl-compatible Regular Expressions (revised API: PCRE2)*, August 2021.

[3] Sabína Gulčíková and Ondřej Lengál. Register set automata (technical report), 2022.

[4] Free Software Foundation, Inc. Gnu grep 3.6. online.