Backreferences in practical regular expressions

Martin Hron   *Supervisor:* Ing. Ondřej Guth, Ph.D.

Faculty of Information Technology, Czech Technical University in Prague

FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Regular Expressions

- Compact description for sets of strings (regular languages in theory).
- Used for searching in text, input validation, vulnerability detection in networking, in tools for string manipulation (grep, awk, sed, text editors), and so on.
- Available and widely used in most modern programming languages.

Backreferences

- Regular expressions used in practice (called e.g. **regex**) differ from the theoretical ones and were over time extended with various new features.
- One of the major extensions are **backreferences**, which significantly increase expressive power. They allow to match repeated substrings in input.
- For example, the pattern '`(again) and \1`' will match string '`again and again`', and pattern '`([a-z])\1`' any double letters like '`ll`' or '`oo`'. In most implementations, the repeated part is marked using parentheses (and called capturing **group**), it is referenced using backslash and group index.
- Regular expressions with backreferences have **NP-complete** matching problem, meaning that practical implementations face exponential worst case execution time.
- Traditional implementations that support backreferences are usually based on recursive backtracking. When the exponential worst case occurs, it is called **catastrophic backtracking**.
- Existing implementations (like PCRE, C++ std::regex, Boost, etc.) exhibit catastrophic backtracking even on some patterns without backreferences. This has potential **security implications** and can lead to vulnerabilities like denial-of-service (DoS) attacks.

Memory Automata

- **Memory automaton (MFA)** is a model of computation for regex introduced in [1]. Recent research [2] based on MFA proposes techniques for regex matching in polynomial time under certain restrictions.
- In theory, the complexity upper bound for matching using memory automata is **polynomial** provided that the number of referenced groups is bounded by a constant. This is often the case in practice, some implementations even allow only up to e.g. 10 groups and more than few groups are rarely used.
- Main **goal** of the thesis was to **implement regex matching tool based on memory automata with support for backreferences**, which would hopefully achieve better complexity properties than existing regex implementations.

Contributions

- Regular expressions library based on memory automata was **implemented**.
- The theoretical memory automaton model and selected algorithms from [2] were adapted for practical regular expressions.
- An **extension** of the memory automaton model to handle **counting constraints** was proposed and implemented. Our model is illustrated in Figure 1.
- Counting constraints allow to repeat a subpattern number of times given by constant or in a form of range, like '`a{1,3}`' for matching 'a', 'aa' or 'aaa'.

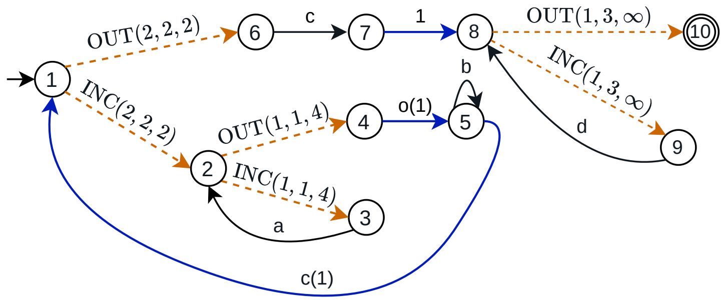

Fig. 1: Illustration of the proposed model to support counting constraints – an automaton with special transitions for handling backreferences and counting constraints. This example automaton accepts language of PCRE regex '`(?:a{1,4}(b*)){2}c\1d{3,}`'.

- Three **matching algorithms** were implemented, including a modification of recursive backtracking with mechanism to **prevent catastrophic backtracking**.
- Technique based on a property called **active variable degree** from [2] was implemented and we proposed and employed a **new algorithm** to compute this property.

Implementation

- Written in C++, released under the open source MIT licence and available at: `https://gitlab.com/hronmar/mfa-regex`
- Library and two command line tools (including a simple imitation of Unix **`grep`** tool).
- Supports practical PCRE–like syntax (similar to Perl regular expressions).
- Includes support for Unicode (UTF-8).

Evaluation and Results

- Our solution was compared with other existing implementations on various datasets, including a collection of regex extracted from production code.
- A dataset of "dangerous" (malicious) patterns and inputs, which are known to cause catastrophic backtracking was assembled and used for evaluation. In theory, our implementation should be **immune to catastrophic backtracking** on patterns with limited number of backreferences to different groups. Evaluation confirmed this, as our implementation did not exhibit exponential time on any of the tested inputs.
- Conversely, the other tested implementations that support backreferences each exhibited catastrophic backtracking on at least half of the tested malicious inputs.

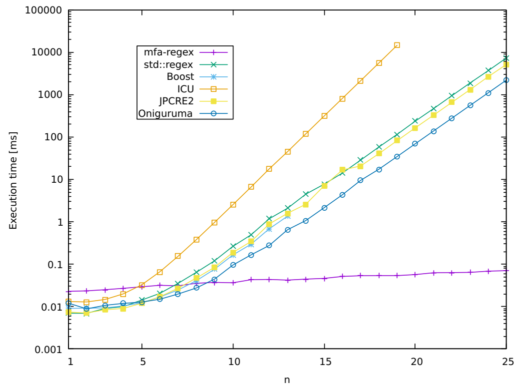

Fig. 2: Example of engines performance on a "dangerous" pattern '`(?:.*a)+`' for inputs $\mathtt{a}^n\mathtt{b}$.

- Overall (on "safe" inputs), performance of our implementation was slightly faster than that of C++ std::regex (tested on GCC 9.3), but it was significantly outperformed by other backtracking engines like PCRE 2.
- Still, the complexity upper bound makes matching based on memory automata an **interesting alternative to traditional implementations**.

References

[1] Markus L. Schmid. "Characterising REGEX languages by regular languages equipped with factor-referencing". In: *Information and Computation* 249 (2016), pp. 1–17. ISSN: 0890-5401. DOI: `10.1016/j.ic.2016.02.003`. URL: `http://www.sciencedirect.com/science/article/pii/S0890540116000109`.

[2] Markus L. Schmid. "Regular Expressions with Backreferences: Polynomial-Time Matching Techniques". In: *ArXiv e-prints* (2019). arXiv: `arXiv:1903.05896 [cs.FL]`. URL: `https://arxiv.org/abs/1903.05896`.