The Practical Efficiency of Regular Expression Membership Algorithms

Justin Gray
(Supervised by Dr. Konstantinidis)

Formal Languages

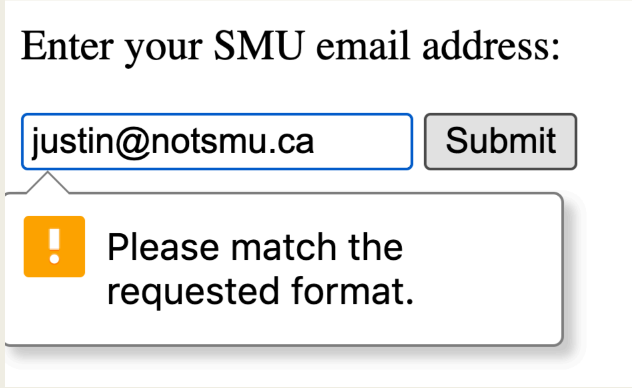
- Alphabet set Σ of symbols
  {0, 1} or {a, b, c, d, …, z} or …
- Word string made from these symbols
  0101011, jbc, …
  ε for the empty word “”
- Language set of words
  {ε, 0, 1, 00, 01, 10, 11, …}

- “Regular” languages can be described by regular expressions

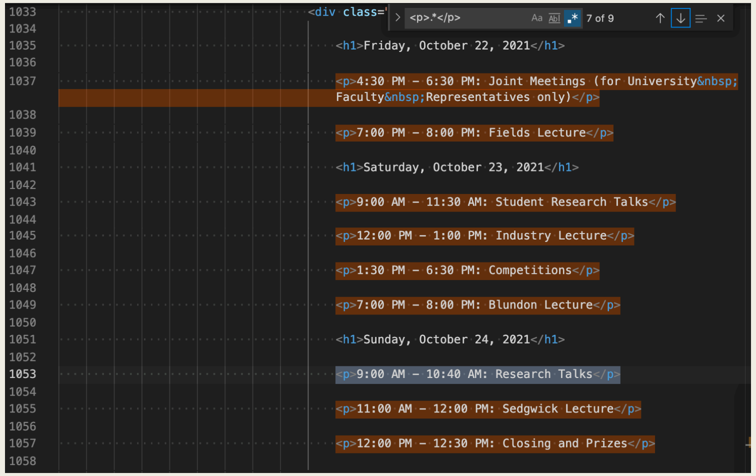
Applications of Regular Expressions

Searching

Validating



Extracting



- DNA searching
- Online quiz fill-in-the-blank correctness
- Filtering shell output (grep)

AND… Anywhere things are encoded as text.

Mathematical Regular Expressions

- Concatenation
  $L(ab) = \{ab\}$
  “a” followed by a “b”

- Disjunction
  $L(a + b) = \{a, b\}$
  “a” or “b”

- Star
  $L(a^*) = \{\varepsilon, a, aa, aaa, \ldots\}$
  0 or more “a”s

Some Compatible UNIX Extensions

- Wild dot matches any single symbol
  L(a.a) = {aσa : for any $\sigma \in \Sigma$}
  {aaa, a a, aba, a0a} $\subset$ L(a.a)

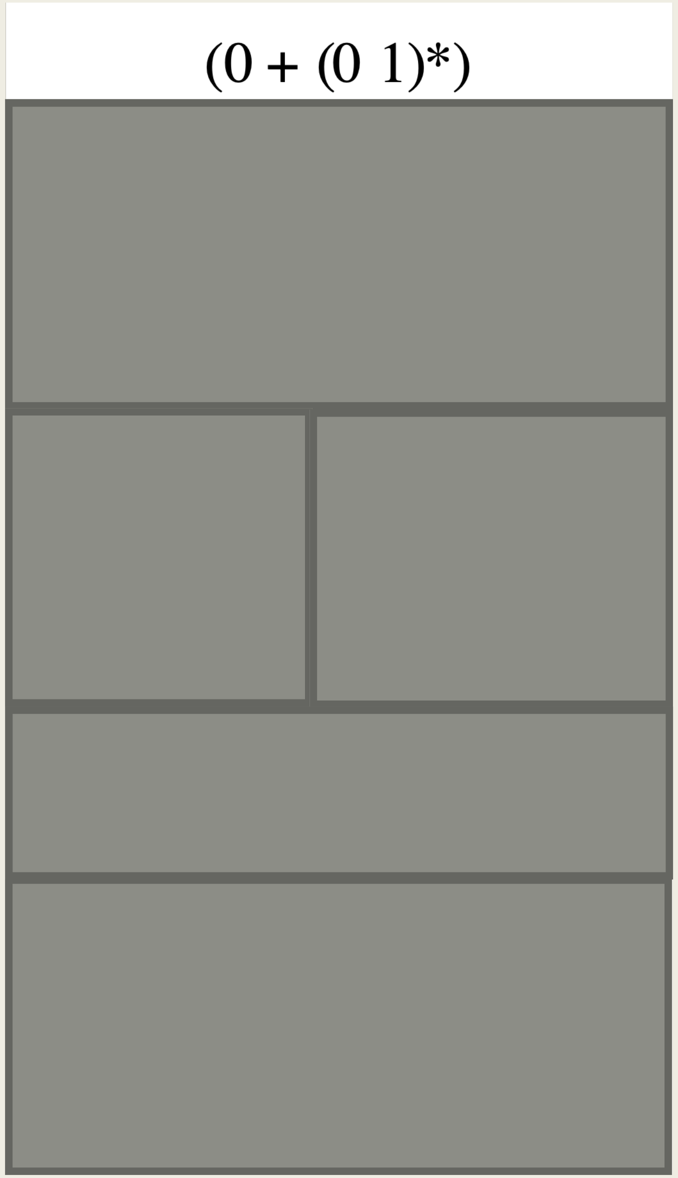
- Character Classes
  L([abc0-9]) = {a, b, c, 0, 1, 2, 3, …, 9}
  L([^a]) = {$\sigma \in \Sigma \setminus \{a\}$}

- Optional
  L(a?) = {ε, a}

Regular Expression Trees

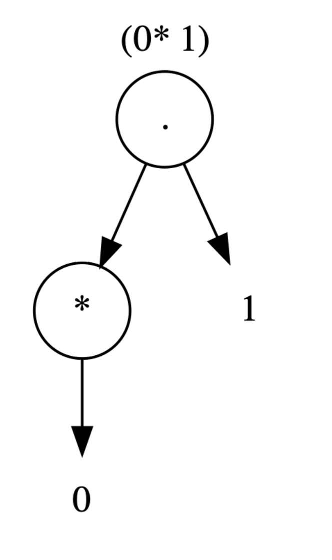
Consider the expression (0 + (0 1)*)

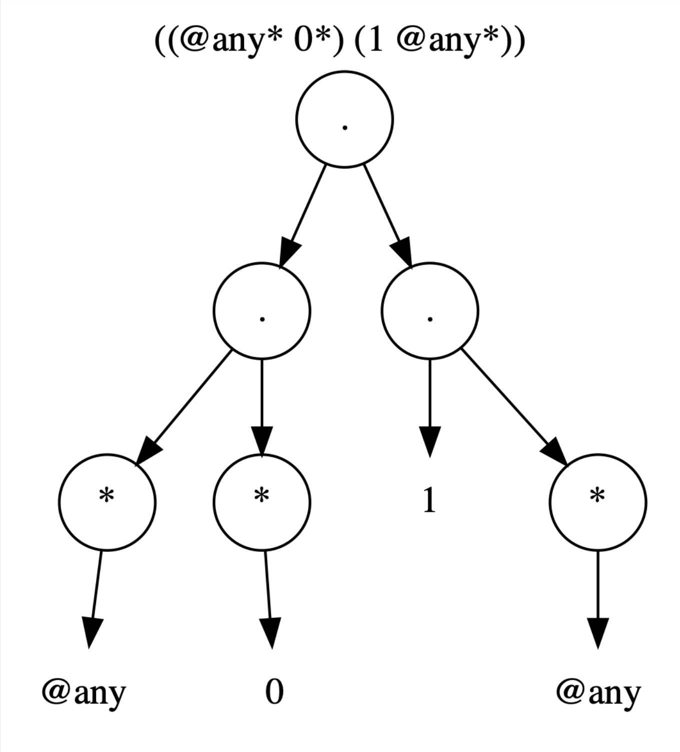
- Disjunction: (0 + (0 1)*)
- “0” is a symbol
- Star: (0 1)*
- Concatenation: (0 1)
- “0” is a symbol
- “1” is a symbol



Partial Matching

- UNIX regular expressions generally match words if any substring of that word is accepted. (I.e., α will accept xwy if w∈L(α))
- Mathematical regular expressions require full matches
- We can make our regular expressions partially match by adding ".*" appropriately
  (this generally means at the beginning and end of leaves)





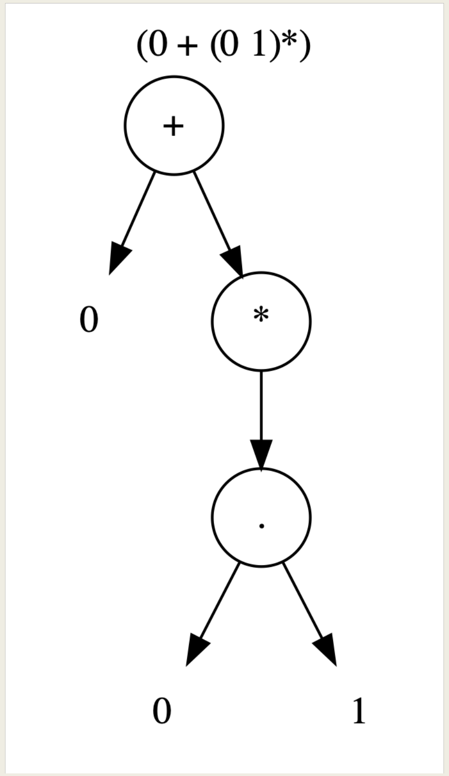
The Membership Problem

- Given a word w and a regular expression α, $w \in L(\alpha)$?
- 3 Approaches:
  - (Partial) Derivatives
  - Backtracking
  - Automata

- If you were designing a regular expression library today, which algorithm should you follow?

The Membership Problem: Partial Derivatives

- $\partial_\sigma(\sigma) = \{\varepsilon\}$ for $\sigma \in \Sigma$
- $\partial_\sigma(b) = \partial_\sigma(\varepsilon) = \{\,\}$ for $\sigma \neq b$
- $\partial_\sigma(\alpha + \beta) = \partial_\sigma(\alpha) \cup \partial_\sigma(\beta)$
- $\partial_\sigma(\alpha^*) = \partial_\sigma(\alpha)\alpha^*$
- $\partial_\sigma(\alpha\beta) = \begin{cases} \partial_\sigma(\alpha)\beta, & \text{if } \varepsilon \notin L(\alpha) \\ \partial_\sigma(\alpha)\beta \cup \partial_\sigma(\beta), & \text{if } \varepsilon \in L(\alpha) \end{cases}$



$0101 \in L(0+(01)^*)$?

Letters of 0101	Current PDs	Next PDs
0	$\{0+(01)^*\}$	$\partial_0(0+(01)^*) = \partial_0(0) \cup \partial_0((01)^*)$ $= \{\varepsilon\} \cup \partial_0(01)(01)^*$ $= \{\varepsilon\} \cup \{1\}(01)^*$ $= \{\varepsilon, 1(01)^*\}$
1	$\{\varepsilon, 1(01)^*\}$	$\partial_1(\varepsilon) = \{\,\}$ $\partial_1(1(01)^*) = \{(01)^*\}$
0	$\{\,\} \cup \{(01)^*\}$ $= \{(01)^*\}$	$\partial_0((01)^*) = \partial_0(01)(01)^*$ $= \{1\}(01)^*$ $= \{1(01)^*\}$
1	$\{1(01)^*\}$	$\partial_1(1(01)^*) = \{(01)^*\}$
	$\{(01)^*\}$	

$0101 \in L(0+(01)^*) \Leftrightarrow \varepsilon \in L((01)^*)$

$\Leftrightarrow$ Yes

The Membership Problem: Backtracking

- Implemented in almost every programming language
- Supports non-regular language operations (i.e., backreferences)
- Can work without constructing an automaton
- Usually very fast:

0101 ∈ L(0+(01)*)

0101 **X**

0101

0101

0101

0101 ✓

- 0101 can be processed through L(0+(01)*) in “5 steps”

The Membership Problem: Backtracking

Let α = (a + a)*

aab ∈ L(α)

aab
    aab ***X***
    aab ***X***
aab
    aab ***X***
    aab ***X***

Answer: No

aaab ∈ L(α)

aaab
    aaab
        aaab ***X***
        aaab **X**
    aaab
        aaab **X**
        aaab **X**
aaab
    aaab
        aaab ***X***
        aaab **X**
    aaab
        aaab **X**
        aaab **X**

Answer: No

aaaab ∈ L(α)

aaaab
    aaaab
        aaaab
            aaaab ***X***
            aaaab **X**
        aaaab
            aaaab **X**
            aaaab **X**
    aaaab
        aaaab
            aaaab ***X***
            aaaab **X**
        aaaab
            aaaab **X**
            aaaab **X**

aaaab
    aaaab
        aaaab
            aaaab ***X***
            aaaab **X**
        aaaab
            aaaab **X**
            aaaab **X**
    aaaab
        aaaab
            aaaab ***X***
            aaaab **X**
        aaaab
            aaaab **X**
            aaaab **X**

Answer: No

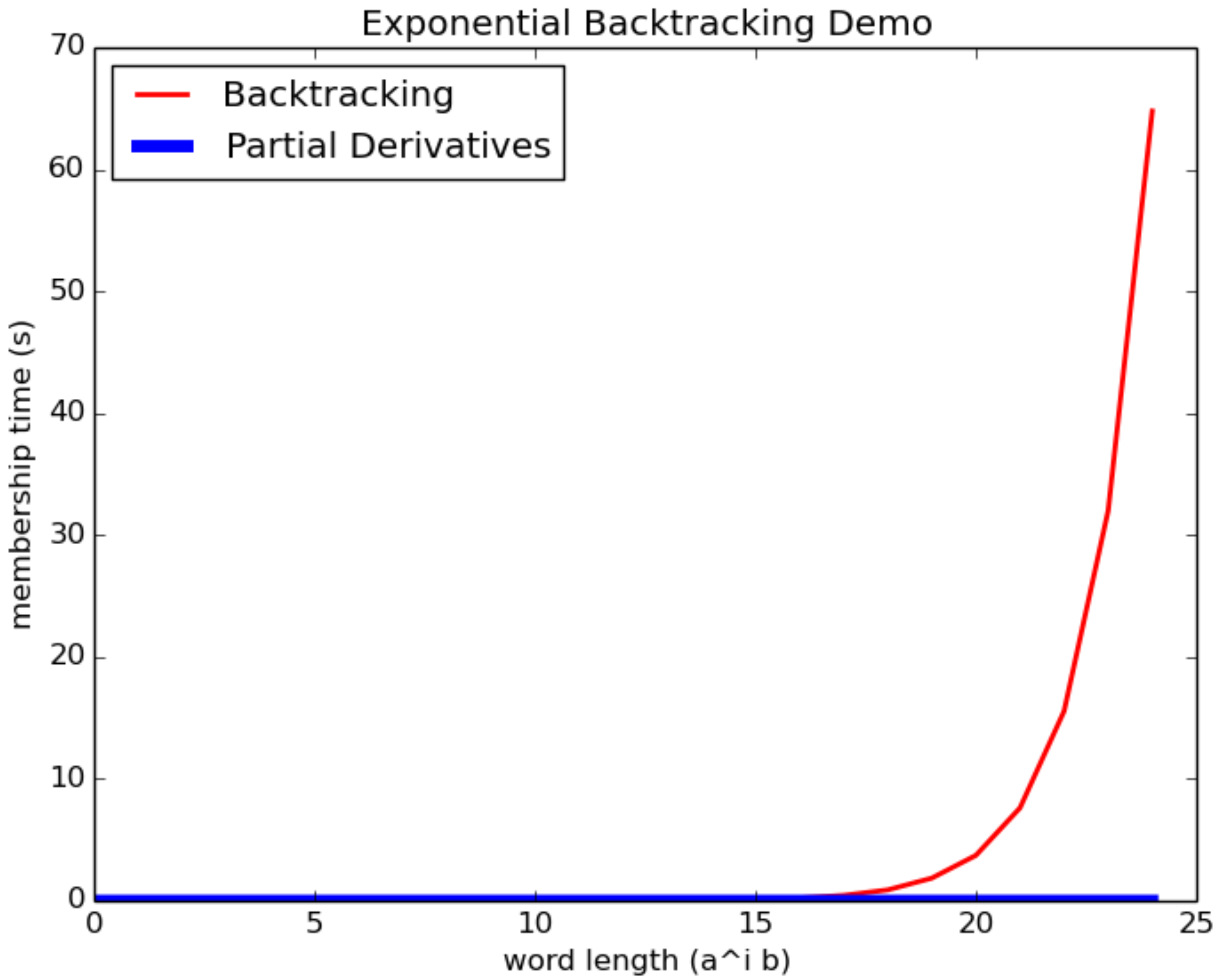


Figure 1: My implementation using the expression (a + a)*

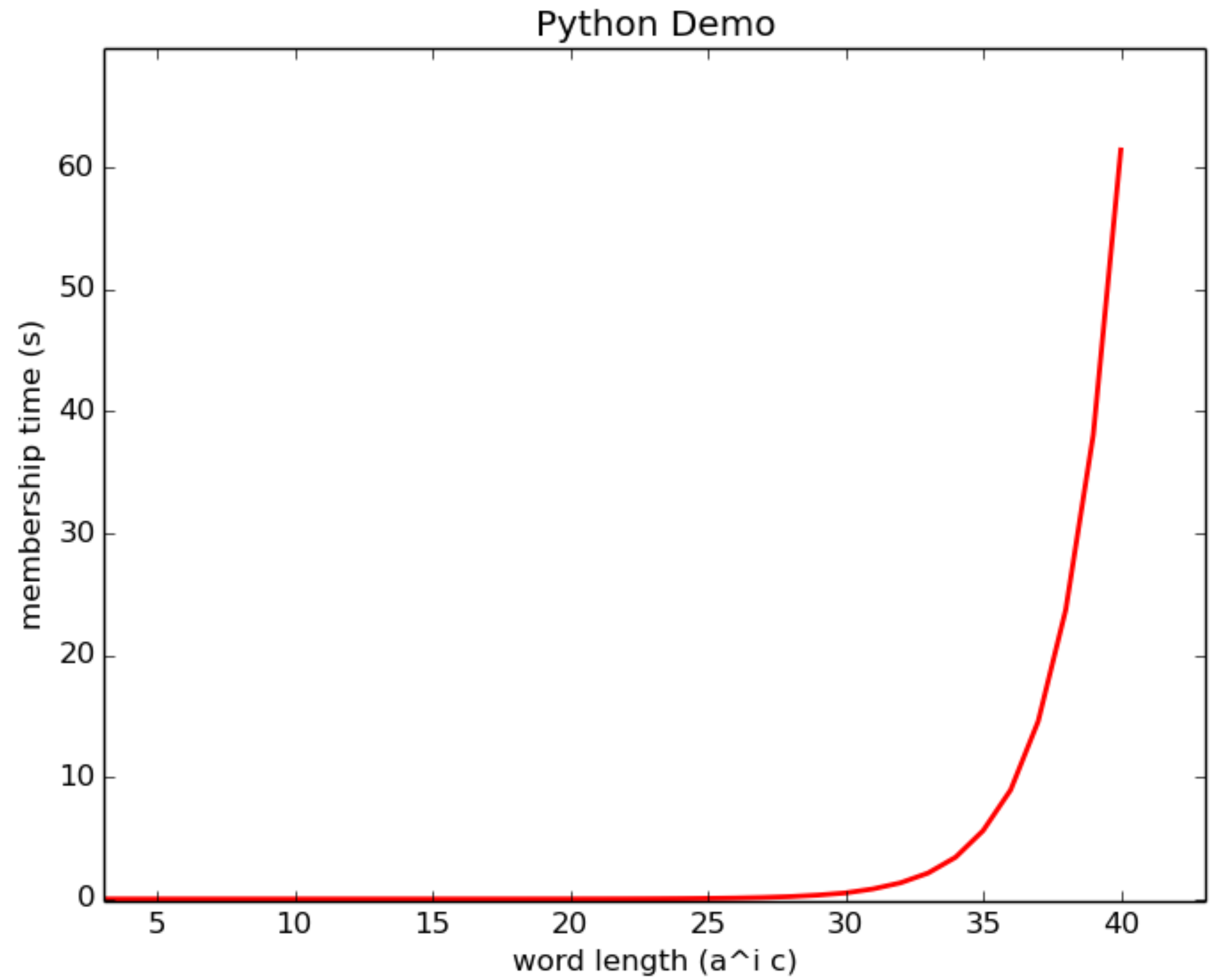
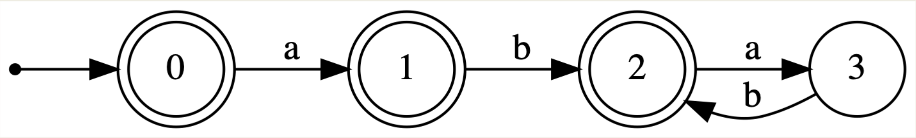


Figure 2: Python’s re module and the expression ((a + aa)* b) [1]

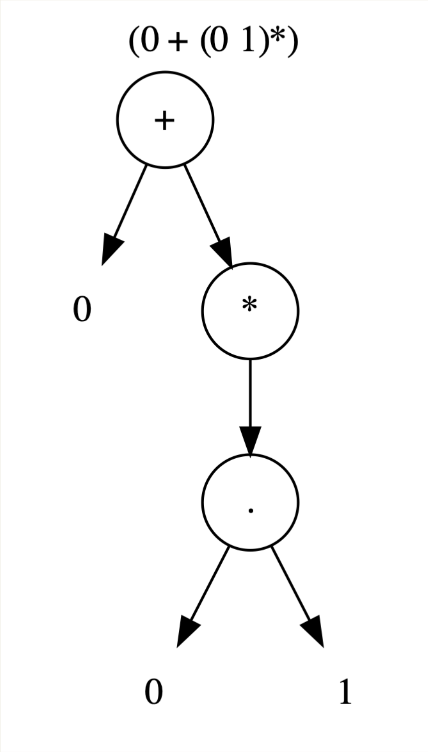
The Membership Problem: Automata

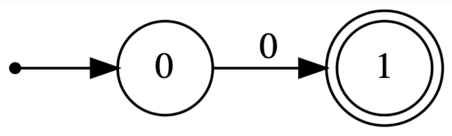
- Automata are state machines
- Input: word
- Output: YES if the entire word can be processed leading to any final state, NO otherwise
- Final states are double circled
- Symbols of the input word are consumed at each transition

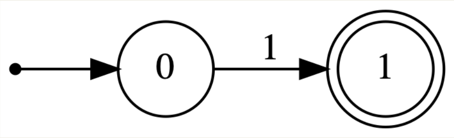


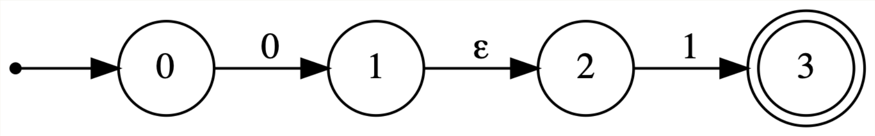
$(\{0\}, abab)$
$\vdash (\{1\}, bab)$
$\vdash (\{2\}, ab)$
$\vdash (\{3\}, b)$
$\vdash (\{2\}, \varepsilon)$
2 is final - accept

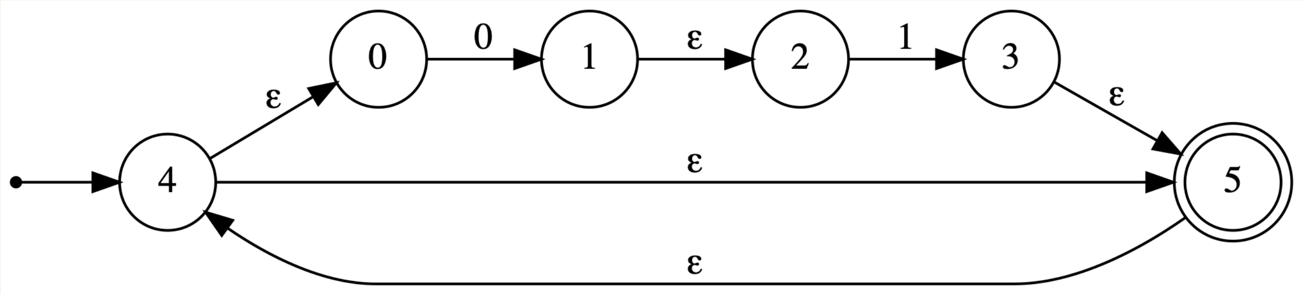
The Membership Problem: Automata

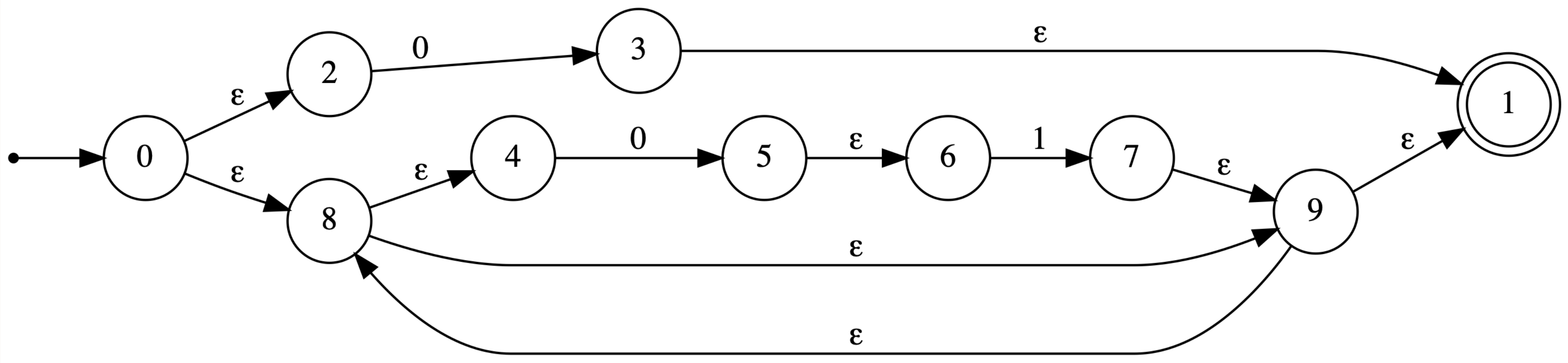












Collecting Practical Regular Expressions

https://grep.app indexes GitHub source code

Extracted 31,887 lines of code containing partial regular expressions from public GitHub repositories

12,023 of these lines have regular expressions satisfying:

- Completeness
- No errors
- No backreferences, lookaheads, boundary assertions
- Distinct when formatted as equivalent unambiguous mathematical expressions

Test Words

Accepting Words

- Pairwise-inspired generation [2]
- Enumeration

- Words are inserted into source code to simulate partial matching

Rejecting Words

- Distorting accepted words (adding, swapping, and removing characters)
- Comparing against a static list of word inputs

The average expression is tested against 1,980 accepting words and 4,468 rejecting words.

Only 726/12,023 regular expressions have been fully tested. Each regular expression is tested multiple times to ensure accuracy.

Results

- Regular expressions are grouped by string length
  - This is how they are most commonly defined

- The time measurement consists of:
  - Time to construct the appropriate object (parsing from a string into the tree, plus any automaton construction time)
  - Plus the time to evaluate the average test word

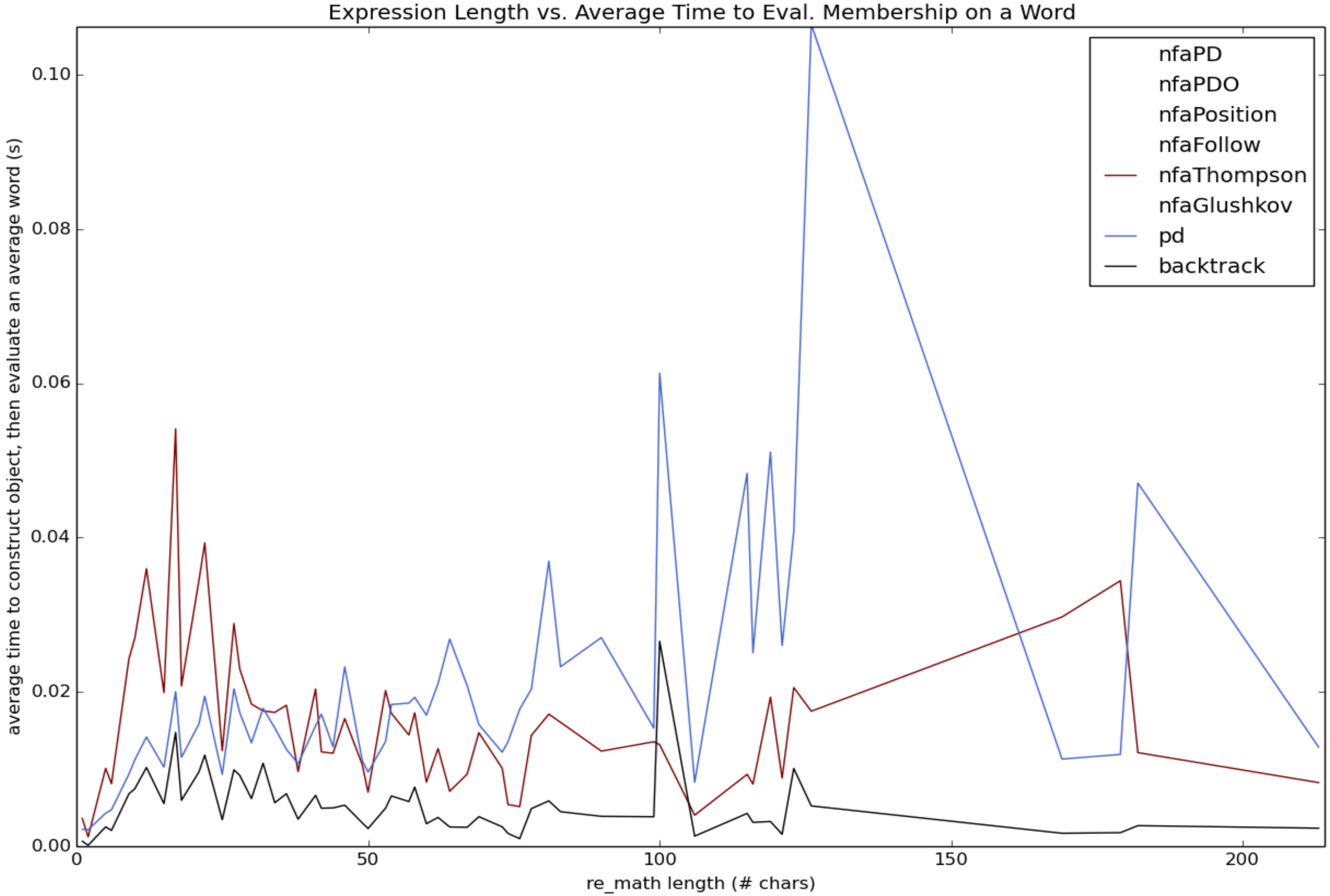


Figure 3: partial derivatives, vs. Thompson, vs. backtrack

$|pd(\alpha)| \leq |\alpha|_\Sigma$

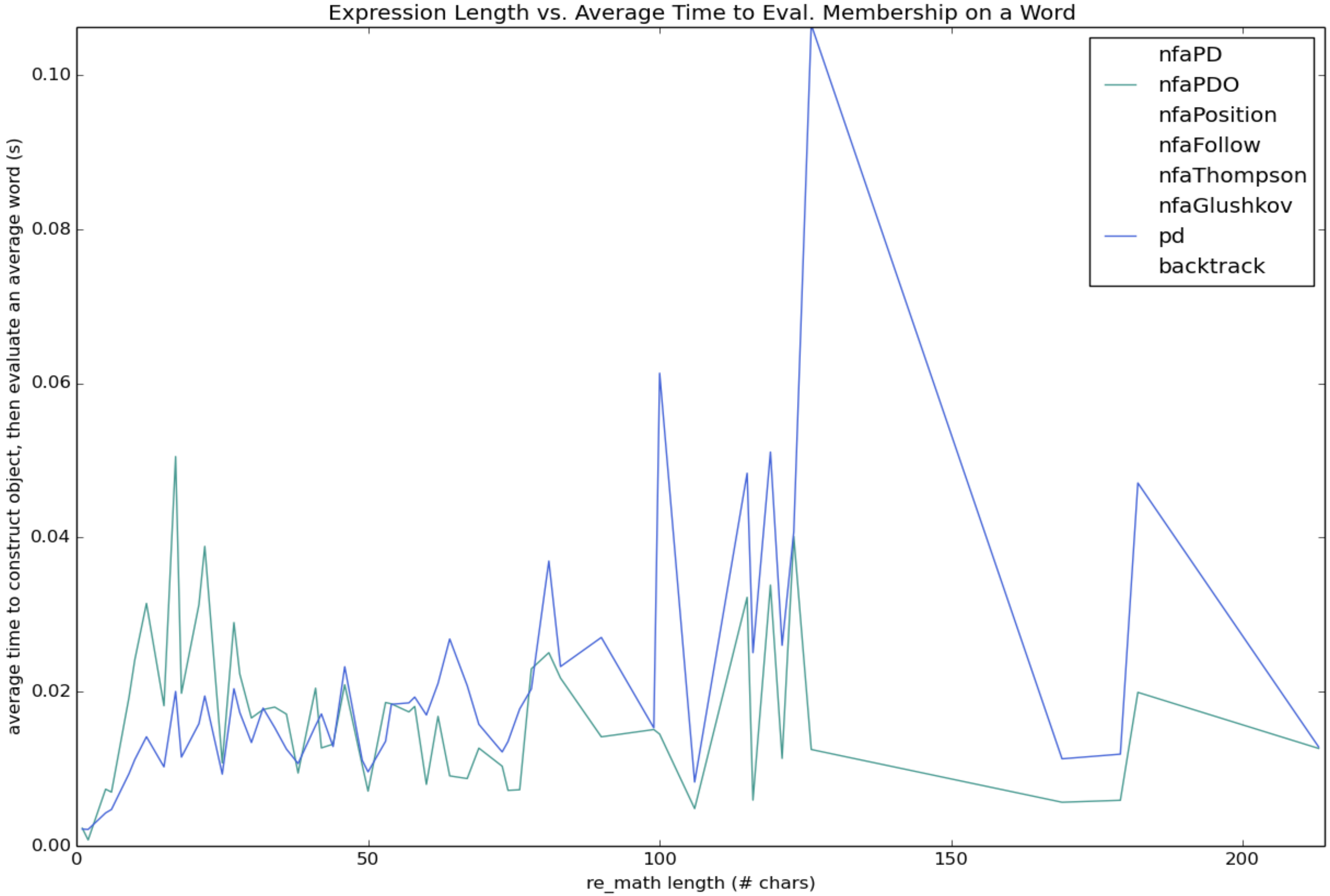


Figure 4: partial derivatives vs. partial derivative automaton

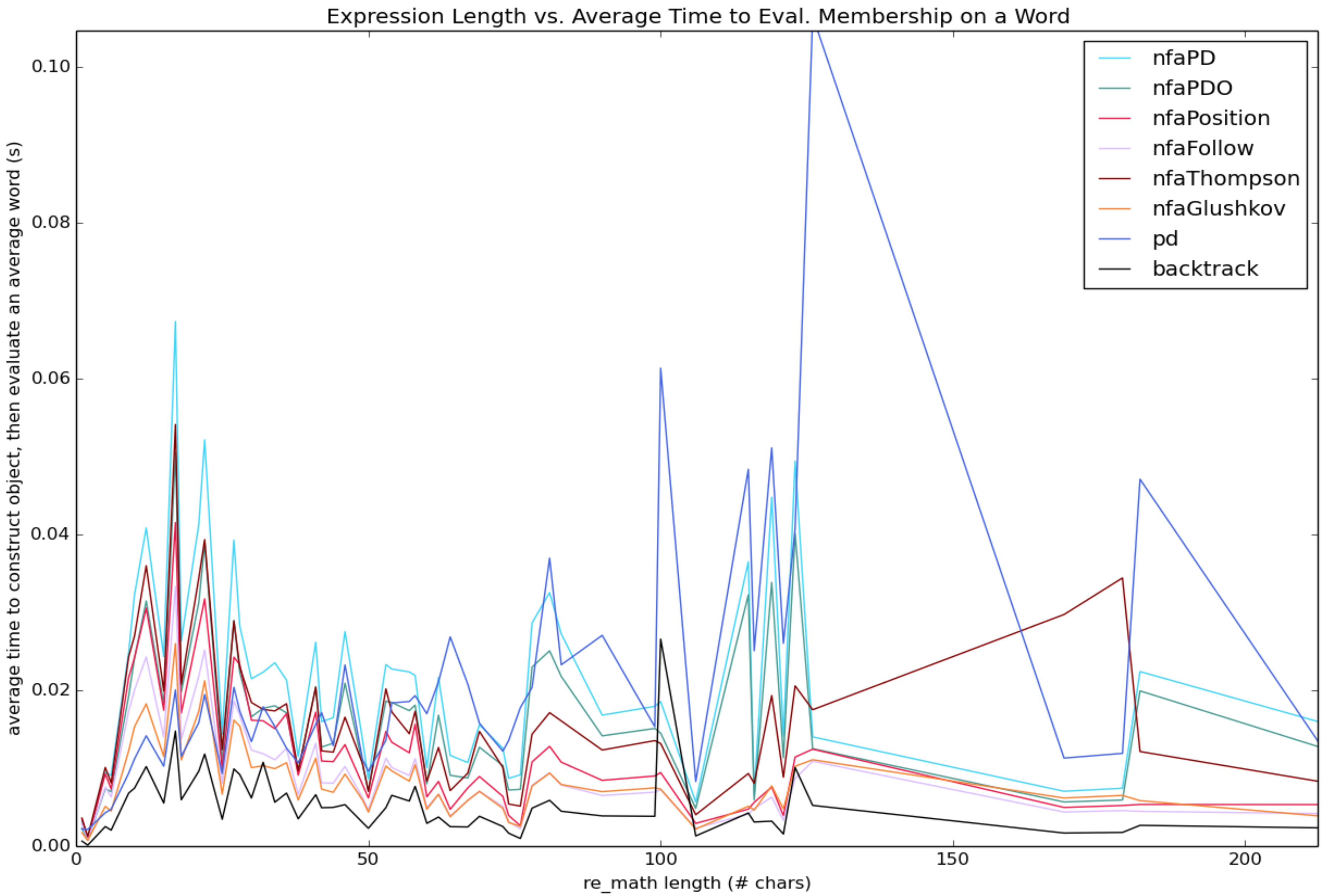


Figure 5: all algorithms

Further Work

- We hope to test these algorithms on randomly generated regular expressions (this has been done, but not with these UNIX extensions)
- Partial derivative optimizations when evaluating equality of large regular expression trees

Thank you!

1. To Dr. Konstantinidis for supervising this research and making his expertise available

2. For your attention

3. Questions?

The source code of this project is made publicly available at https://github.com/just1ngray/SMUHon-Practical-RE-Membership-Algs

Reference List

1. R. Sedgewick, K. Wayne, *Algorithms,* 4^{th} ed. Available: https://algs4.cs.princeton.edu/54regexp/
2. L. Zheng, S. Ma, Y. Wang, and G. Lin, “String Generation for Testing Regular Expressions,” *The Computer Journal*, vol. 63, no. 1, Jan., pp. 41-65, 2020.