Received XX July 2019; | Revised XX March 2020; | Accepted XX YY 2020

DOI: xxx/xxxx

RESEARCH ARTICLE

Efficient POSIX Submatch Extraction on NFA

Angelo Borsotti[1] | Ulya Trofimovich[2]

[1]Email: angelo.borsotti@mail.polimi.it
[2]Email: skvadrik@gmail.com

Summary

In this paper we study the performance of NFA-based POSIX submatch extraction algorithms. We propose an algorithm that combines Laurikari tagged NFA and extended Okui-Suzuki disambiguation. The algorithm works in worst-case $O(n\,m^2\,t)$ time and $O(m^2)$ space (including preprocessing), where n is the length of input, m is the size of the regular expression with bounded repetition expanded and t is the number of capturing groups and subexpressions that contain them. On real-world benchmarks our algorithm performs close to the $O(n\,m\,t)$ complexity of leftmost-greedy matching, although on artificial benchmarks it can be significantly slower. We propose a lazy version of the algorithm that runs much faster, but requires $O(n\,m^2)$ space. We show that the Kuklewicz algorithm is slower in practice, and the backward matching algorithm proposed by Cox is incorrect.

KEYWORDS:
Regular Expressions, Parsing, Submatch Extraction, Finite-State Automata, POSIX

1 | INTRODUCTION

Regular expressions (RE) are a convenient notation for describing regular languages. RE syntax varies among different specifications and standards. Most of the variations allow a more succinct representation, but do not increase the expressive power of RE. In this paper we do not consider such trivial extensions, with the exception of bounded repetition, which requires non-trivial changes in the algorithms and proofs. We also do not consider extensions that bring RE beyond regular languages, such as backreferences. The recognition problem for RE can be solved in $O(n\,m)$ time and $O(m)$ space, where n is the size of input and m is the size of RE (e.g. by simulation of a Thompson's NFA [1 2]). The parsing problem is more difficult because it has to deal with ambiguity. Ambiguity means that there are multiple ways to parse the input. The preferred way is usually defined by a disambiguation policy. There are two widely used policies: the Perl leftmost-greedy policy and the POSIX longest-match policy. The Perl policy is defined in terms of RE structure; it admits a simple and efficient implementation. The POSIX policy, on the other hand, is defined in terms of the structure of parse results; it is difficult to implement compared to the Perl policy. In this paper we focus on the POSIX policy. The problem of submatch extraction is a special case of parsing: it has the same worst-case complexity, but in practice specialized submatch extraction algorithms are faster than generic full parsing algorithms.

We study NFA-based approaches to the problem. Namely, we consider the algorithms proposed by Okui and Suzuki[3], Kuklewicz[5] and Cox[6]. Our experiments show that in general Okui-Suzuki approach is the most efficient one. We combine it with the insights and useful techniques from other approaches and suggest a few algorithmic and practical improvements of our own. The proposed algorithm is thoroughly formalized. We are aware of an alternative approach based on Brzozowski derivatives proposed by Sulzmann and Lu[7], but in our experience this approach is slower in practice and has worse algorithmic complexity (discussed in more detail below). Both NFA-based and derivative-based approach can be used to construct DFA with POSIX

semantics[8 9 11]. The resulting DFA are very fast, because the disambiguation is done at determinization time and there is no run-time overhead. However, determinization is not always viable due to its exponential worst-case complexity. The algorithm we propose in this paper is based on NFA, but it can also be used as a basis for DFA construction.

We give an overview of existing algorithms, including some that are incorrect, but interesting from historical perspective. The list is by no means exhaustive, but in our experience other approaches produce incorrect results or require memory proportional to the length of input (e.g. the Glibc implementation[22]).

Laurikari, 2001 (incorrect). Laurikari described an algorithm based on TNFA — ϵ-NFA with tagged transitions[4]. He represented each submatch group with a pair of *tags* (opening and closing). Disambiguation is based on minimizing the value of opening tags and maximizing the value of closing tags, where different tags have priority according to the POSIX subexpression hierarchy. The algorithm gives incorrect results for REs with iteration subexpressions, such as `(a|aa)*` on a string `aa` (it minimizes the value of the opening tag on the last iteration, and fails to take into account the preceding iterations). The reported time complexity is $O(n\,m\,c\,t\,log(t))$, where n is the length of input, m is TNFA size, c is the time for comparing tag values and t is the number of tags. Space complexity is $O(m\,t)$. Notably, Laurikari used the idea of topological order to avoid worst-case exponential time of ϵ-closure construction, although his closure algorithm is not optimal[11].

Kuklewicz, 2007. Kuklewicz fixed Laurikari algorithm by introducing *orbit* tags for iteration subexpressions. He gave only an informal description[5], but the algorithm was later formalized in[11]. It works in the same way as the Laurikari algorithm, except that the comparison of orbit tags takes into account the history of all iterations, not just the most recent one. The key idea that allows the algorithm to execute in bounded memory is to compress orbit tag histories in a matrix of size $t \times m$, where m is the size of TNFA and t is the number of tags. t-Th row of the matrix represents the ordering of closure states with respect to t-th tag. The matrix is updated at each step using the continuations of tag histories. The algorithm requires $O(m\,t)$ space and $O(n\,m\,t\,(m + t\,log(m))$ time, where n is the length of input (we assume a worst-case optimal $O(m^2\,t)$ algorithm for ϵ-closure construction, and an $O(m\,log(m)\,t^2)$ matrix update, because for t tags m states are sorted with an $O(t)$ comparison function). Kuklewicz disambiguation is combined with Laurikari determinization in construction of TDFA[11].

Cox, 2009 (incorrect). Cox came up with the idea of backward POSIX matching[6], which is based on the observation that reversing the longest-match rule simplifies the handling of iteration subexpressions: instead of maximizing submatch from the first to the last iteration, one needs to maximize the iterations in reverse order. This means that the disambiguation is always based on the most recent iteration, removing the need to remember all previous iterations (except for the backwards-first, i.e. the last one, which contains submatch result). The algorithm tracks two pairs of offsets per each submatch group: the *active* pair (used for disambiguation) and the *result* pair. It gives incorrect results under two conditions: (1) ambiguous matches have equal offsets on some iteration, and (2) disambiguation happens too late, when the active offsets have already been updated and the difference between ambiguous matches is erased. We found that such situations may occur for two reasons. First, the ϵ-closure algorithm may compare ambiguous paths *after* their join point, when both paths have a common suffix with tagged transitions. This is the case with the Cox prototype implementation[6]; for example, it gives incorrect results for `(aa|a)*` and string `aaaaa`. Most of such failures can be repaired by exploring states in topological order, but a topological order does not exist in the presence of ϵ-loops. The second reason is bounded repetition: ambiguous paths may not have an intermediate join point at all. For example, in the case of `(aaaa|aaa|a){3,4}` and string `aaaaaaaaaa` we have matches `(aaaa)(aaaa)(a)(a)` and `(aaaa)(aaa)(aaa)` with a different number of iterations. Assuming that the bounded repetition is unrolled by chaining three sub-automata for `(aaaa|aaa|a)` and an optional fourth one, by the time ambiguous paths meet both have active offsets `(0,4)`. Despite the flaw, Cox algorithm is interesting: if somehow the delayed comparison problem was fixed, it would work. The algorithm requires $O(m\,t)$ memory and $O(n\,m^2\,t)$ time (assuming a worst-case optimal closure algorithm), where n is the length of input, m it the size of RE and t is the number of submatch groups and subexpressions that contain them.

Okui and Suzuki, 2013. Okui and Suzuki view the disambiguation problem from the point of comparison of parse trees[3]. Ambiguous trees have the same frontier of leaf symbols, but their branching structure is different. Each subtree corresponds to a subexpression. The *norm* of a subtree is the number of alphabet symbols in it (a.k.a. submatch length). The longest match corresponds to a tree in which the norm of each subtree in leftmost in-order traversal is maximized. The clever idea of Okui and Suzuki is to relate the norm of subtrees to their *height* (the distance from the root). Namely, if we walk through the leaves of two ambiguous trees, tracking the height of each complete subtree, then at some step the heights will diverge: subtree with a smaller norm will already be complete, but the one with a greater norm will not. The height of subtrees is easy to track by

attributing it to parentheses and encoding in the automaton transitions. Okui and Suzuki use PAT — ϵ-free position automaton with transitions labeled by sequences of parentheses. Disambiguation is based on comparing parentheses along ambiguous PAT paths. Similar to Kuklewicz, Okui and Suzuki avoid recording full-length paths by pre-comparing them at each step and storing the comparison results in a pair of matrices indexed by PAT states. The authors report complexity $O(n(m^2 + c))$, where n is the input length, m is the number of occurrences of the most frequent symbol in RE and c is the number of submatch groups and repetition operators. Memory requirement is $O(m^2)$. However, these estimates leave out precomputation of the precedence relation and the construction of PAT, which may grow exponential in the size of RE: the transformation from RE to PAT is ambiguity-preserving, which means that there may be multiple transitions with different labels between a pair of PAT states. For example, for RE `((a*)|(a*)){k}` there are 2^k different ways to match the empty string, and consequently 2^k transitions between the initial and the final states of PAT. Although such RE are unlikely in practice, RE engines should be able to handle them in reasonable time. Okui-Suzuki disambiguation is combined with Berry-Sethi construction in parsing DFA[9].

Sulzmann and Lu, 2013. Sulzmann and Lu based their algorithm on Brzozowski derivatives[7] (correctness proof is given by Ausaf, Dyckhoff and Urban[10]). The algorithm unfolds a RE into a sequence of derivatives and then folds it back into a parse tree. Each derivative is obtained from the previous one by consuming an input symbol in left-to-right order, and each tree is built from the next tree by injecting a symbol in reversed right-to-left order. In practice, Sulzmann and Lu fuse backward and forward passes, which allows to avoid potentially unbounded memory usage on keeping all intermediate derivatives. The algorithm is elegant in that it does not require explicit disambiguation: by the definition of derivative, parse trees are ordered by the longest-match criterion. Time and space complexity is not entirely clear. In[7] Sulzmann and Lu consider the size of RE as a constant. In[8] they give more precise estimates: $O(2^m\, t)$ space and $O(n\, log(2^m)\, 2^m\, t^2)$ time, where m is the size of RE, n is the length of input and t the number of submatch groups (the authors do not differentiate between m and t). However, this estimate assumes worst-case $O(2^m)$ derivative size and on-the-fly DFA construction. The authors also mention a better $O(m^2)$ theoretical bound for derivative size. If we adopt this bound and exclude DFA construction, we get $O(m^2\, t)$ memory requirement and $O(n\, m^2\, t^2)$ time, which seems reasonably close to (but worse than) NFA-based approaches.

Our contributions are the following:

- We extend Okui-Suzuki algorithm on partially ordered parse trees, which greatly reduces the disambiguation overhead.
- We extend Okui-Suzuki algorithm on the case of bounded repetition.
- We use Laurikari TNFA instead of Okui-Suzuki PAT, which avoids the potential blowout of PAT size.
- We introduce *negative tags* that allow us to handle no-match and match cases uniformly.
- We consider ϵ-closure construction as a shortest-path problem and use the Goldberg-Radzik algorithm.
- We give a fast $O(m^2)$ algorithm for updating the precedence matrix (where m is the size of the ϵ-closure).
- We provide a way to output either parse trees or POSIX-style offsets.
- We give a lazy version of the algorithm that is faster, but uses memory proportional to the size of input.
- We provide a C++ implementation of the described algorithms.

The rest of this paper is arranged as follows. In section 2 we present the main idea and the skeleton of our algorithm. In section 3 we provide theoretical foundations for the rest of the paper. After that, we go into specific details: section 4 is concerned with ϵ-closure construction, section 5 discusses data structures used to represent TNFA paths, section 6 discusses possible output formats (parse trees or POSIX-style offsets), section 7 gives the core disambiguation algorithms, section 8 presents a lazy version of the algorithm, and section 9 gives specific TNFA construction. The remaining sections 10, 11 and 12 contain complexity analysis, benchmarks, conclusions and directions for future work.

2 | SKELETON OF THE ALGORITHM

Our algorithm is based on four cornerstone concepts: regular expressions, parse trees, parenthesized expressions and tagged NFA. Following Okui and Suzuki[3], we give the interpretation of regular expressions as sets of parse trees and define POSIX disambiguation semantics in terms of order on parse trees. This definition reflects the POSIX standard, but it is too high-level to be used in practice. From parse trees we go to their linearized representation — parenthesized expressions. We define an

order on parenthesized expressions and show its equivalence to the order on parse trees. The latter definition of order is more low-level and can be converted to an efficient disambiguation algorithm. Finally, we construct TNFA and map parenthesized expressions to its paths, which allows us to compare ambiguous paths using the algorithm for parenthesized expressions. In this section we give the four basic definitions and the skeleton of the algorithm. In the following sections we formalize the relation between different representations and fill in all the details.

Definition 1. *Regular expressions (RE)* over finite alphabet Σ, denoted $\mathcal{R}_\Sigma$:

1. Empty RE ϵ and unit RE a (where $a \in \Sigma$) are in $\mathcal{R}_\Sigma$.
2. If $e_1, e_2 \in \mathcal{R}_\Sigma$, then alternative $e_1|e_2$, concatenation $e_1 \cdot e_2$, repetition $e_1^{n,m}$ (where $0 \le n \le m \le \infty$), and submatch group (e_1) are in $\mathcal{R}_\Sigma$. (Convention: $e_1 \cdot e_2$ may be shortened as $e_1 e_2$, and $e^{n,n}$ may be shortened as e^n.)

Definition 2. *Parse trees (PT)* over finite alphabet Σ, denoted $\mathcal{T}_\Sigma$:

1. Nil tree $\perp^i$, empty tree ϵ^i and unit tree a^i (where $a \in \Sigma$ and $i \in \mathbb{Z}$) are in $\mathcal{T}_\Sigma$.
2. If $t_1, \dots, t_n \in \mathcal{T}_\Sigma$ (where $n \ge 1$, and $i \in \mathbb{Z}$), then $T^i(t_1, \dots, t_n)$ is in $\mathcal{T}_\Sigma$.

Definition 3. *Parenthesized expressions (PE)* over finite alphabet Σ, denoted $\mathcal{P}_\Sigma$:

1. Nil expression $\langle\rangle$, empty expression ϵ and unit expression a (where $a \in \Sigma$) are in $\mathcal{P}_\Sigma$.
2. If $\alpha, \beta \in \mathcal{P}_\Sigma$, then $\alpha\beta$ and $\langle\alpha\rangle$ are in $\mathcal{P}_\Sigma$.

Definition 4. *Tagged Nondeterministic Finite Automaton (TNFA)* is a structure $(\Sigma, Q, M, \Delta, q_0, q_f)$, where:

Σ is a finite set of symbols (*alphabet*)
Q is a finite set of *states*
M is a function that maps *tags* $m \in \mathbb{Z}$ to tuples (S, N),
  where $S \subset \mathbb{Z}$ is a set of submatch groups, and $N \subset \mathbb{Z}$ is a set of nested tags
$\Delta = \Delta^\Sigma \sqcup \Delta^\epsilon$ is the *transition* relation, consisting of two parts:
  Δ^Σ contains untagged transitions on symbols of the form (q_1, a, ϵ, q_2)
  Δ^ϵ contains optionally tagged ϵ-transitions with priority of the form (q_1, n, m, q_2)
    where $q_1, q_2 \in Q$, $a \in \Sigma$, $n \in \mathbb{Z}$ is *priority*, $m \in \mathbb{Z} \cup \{\epsilon\}$ is an optional tag
$q_0 \in Q$ is the *initial* state
$q_f \in Q$ is the *final* state

Our definition of RE is extended with submatch operator and generalized repetition. This is not just syntactic sugar: in POSIX `(a)(a)` is semantically different from `(a){2}`, and `(a)` in not the same as `a`. Parse trees have a special *nil-tree* constructor and an upper index, which allows us to distinguish between submatch and non-submatch subtrees. Mirroring parse trees, parenthesized expressions have a *nil-parenthesis*. TNFA is in essence a nondeterministic finite-state transducer which reads symbolic strings and transduces them to sequences of *tags* — integer numbers that denote opening and closing parentheses of submatch groups. Tags on transitions can be negative, which represents the absence of match and corresponds to the nil-parenthesis $\langle\rangle$ and the nil-tree $\perp$. Transition priorities are used to impose specific order of TNFA traversal.

We use the following notation in the algorithms throughout the paper (with possible subscripts and diacritics):

- Integer numbers, indexes and tags are denoted with i, j, k, l, m, n.
- Strings over Σ are denoted with w and $a_1 \dots a_n$.
- REs are denoted with e.
- PTs are denoted with t, s, r.
- PEs and PE fragments are denoted with α, β, γ, δ.
- TNFA is denoted with F, and its states are denoted with q.
- U is the *tag path tree* — a data structure that stores tag sequences along TNFA paths. Individual paths are represented with *tree indices* — integer numbers, denoted with u. Index zero is the empty path.
- P is the *precedence matrix* — a square matrix indexed by TNFA states, with elements in the $\{-1, 0, 1\}$ set, denoted with p. It is used for disambiguation and corresponds to the Okui-Suzuki D-matrix. The value of a matrix cell $P[q_1][q_2]$ equals -1 if the path to q_1 precedes the path to q_2, 0 if they are equal, or 1 otherwise.

- H is the *height matrix* — a square integer matrix indexed by TNFA states, with elements denoted with h. It contains auxiliary data used in the computation of P-matrix, and corresponds to the Okui-Suzuki B-matrix.
- Partial match results (parse trees or POSIX-style offsets) are denoted with d.
- C is a set of *configurations* (q_2, q_1, u, d), where q_2 is a unique *target* state, q_1 is the *origin* state (an index in the H and P matrices), u is the tagged path from q_1 to q_2 (an index in the U-tree), and d is the partial match result.
- We use either subscript (as in a_i) or square brackets (as in $P[q_1][q_2]$) to denote the elements of sequences or matrices.
- We use the dot notation (as in $U[n].pred$) to access the elements of tuples and structures.
- We avoid global state and explicitly pass parameters to functions.

Below is the skeleton of the matching algorithm:

```
1  match(F = (Σ, Q, M, Δ, q_0, q_f), a_1 ... a_n)
2      H, P : square integer matrices of size |Q|
3      U = empty_path_tree()
4      C = {(q_0, ⊥, 0, initial_result(M))}
5      for i = 1, n do
6          C = closure(F, C, U, H, P)
7          C = update_result(T, C, U, i, a_i)
8          (H, P) = update_precedence(F, C, U, H, P)
9          C = {(q_2, q_1, 0, d) | (q_1, _, _, d) ∈ C ∧ (q_1, a_i, ϵ, q_2) ∈ Δ}
10         if C = ∅ then
11             return ⊥
12     C = closure(F, C, U, H, P)
13     if ∃(q, _, u, d) ∈ C | q = q_f then
14         return final_result(M, U, u, d, n)
15     return ⊥
```

ALGORITHM 1: TNFA simulation on a string.

The *match* algorithm takes a TNFA F and a string $a_1 \dots a_n$ as input. During the initialization step it creates an empty U-tree and allocates H and P matrices (initialization of their elements is not needed, as the initial values are not used). The initial configuration set C contains a single configuration that consists of the initial TNFA state q_0, undefined origin state $\bot$, empty tagged path and initial match result. The algorithm loops over the input characters a_i until either all characters are matched, or the configuration set C becomes empty, indicating a match failure. At each step the algorithm constructs ϵ-closure of the current configuration set, extending the tagged paths in the U-tree, updates the partial match result, re-computes the precedence information in the H and P matrices, and steps on TNFA transitions labeled with the current input symbol. Finally, if all input symbols have been matched and the final configuration set contains a configuration with the final state q_f, the algorithm terminates successfully and returns the final match result. Otherwise it returns a failure.

We intentionally leave some parts of the algorithm undefined in this section. They will be addressed in detail in subsequent sections, after we present the formal foundations of our algorithm in section 3:

- Function *closure* constructs ϵ-closure of a configuration set C (section 4). Essentially, it builds a shortest path from the states in the current set, following ϵ-transitions in TNFA and comparing paths by the POSIX criterion.
- Functions *empty_path_tree*, *extend_path* and *unroll_path* are used with the U-tree (section 5).
- Functions *initial_result*, *update_result* and *final_result* update match results (section 6).
- Function *update_precedence* computes the H and P matrices by performing a pairwise comparison of all configurations in C (section 7). It is the key part of the algorithm that allows to compress precedence information about arbitrary long tagged paths in constant space. If the paths originating from the current configurations join at some future step, the *closure* function will compare them using the information in H and P. If, on the other hand, the paths do not join, then the comparison performed by *update_precedence* is redundant. Unfortunately, we cannot avoid such redundant comparisons, as we do not know in advance which configurations will spawn ambiguous paths. It is possible to use on-demand comparison instead of *update_precedence* (section 8), but that requires keeping arbitrary long tagged paths in memory.

3 | FORMALIZATION

In this section we establish the relation between all intermediate representations. For brevity all proofs are moved to the appendix. First of all, we rewrite REs in a form that separates submatch information from RE structure: instead of using parentheses to denote submatch groups, we store submatch information for each subexpression in the form of an *implicit submatch index* and a set of *explicit submatch indices*. Explicit indices enumerate submatch groups in RE: an empty set $\emptyset$ means that the subexpression is not a submatch group, and a set with multiple indices means that the subexpression is enclosed in multiple submatch groups, as in $((e))$. Implicit indices enumerate all subexpressions that affect disambiguation, in top-down and left-to-right order (this includes submatch groups and subexpressions that contain nested or sibling submatch groups). The above representation reflects the POSIX standard, which states that submatch extraction applies only to parenthesized subexpressions, but the disambiguation rules apply to all subexpressions regardless of parentheses.

Definition 5. *Indexed regular expressions (IRE)* over finite alphabet Σ, denoted $\mathcal{IR}_\Sigma$:

1. Empty IRE (i, J, ϵ) and unit IRE (i, J, α) are in $\mathcal{IR}_\Sigma$, where $\alpha \in \Sigma$, $i \in \mathbb{Z}$ is an *implicit submatch index* and $J \subseteq \mathbb{Z}$ is the set of *explicit submatch indices*.
2. If $e_1, e_2 \in \mathcal{IR}_\Sigma$, $i \in \mathbb{Z}$ and $J \subseteq \mathbb{Z}$, then alternative $(i, J, e_1 \mid e_2)$, concatenation $(i, J, e_1 \cdot e_2)$ and repetition $(i, J, e_1^{n,m})$ are in $\mathcal{IR}_\Sigma$, where $0 \le n \le m \le \infty$.

Function $IRE : \mathcal{R}_\Sigma \to \mathcal{IR}_\Sigma$ transforms RE into IRE. It is defined via a composition of two functions, *mark*() that transforms RE into IRE with implicit indices in the boolean range $\{0, 1\}$, and *enum*() that substitutes boolean indices with consecutive numbers. A step-by-step example of constructing an IRE from a RE is given on figure 1 .

$$
\begin{aligned}
&mark : \mathbb{Z} \times \mathcal{R}_\Sigma \to \mathbb{Z} \times \mathcal{IR}_\Sigma \\
&mark(j, e) \mid_{e \in \{\epsilon, \alpha\}} = (j, (0, \emptyset, e)) \\
&mark(j, e_1 \circ e_2) \mid_{\circ \in \{|, \cdot\}} = (j_2, (i, \emptyset, (i, J_1, e_1') \circ (i, J_2, e_2'))) \\
&\quad \text{where } (j_1, (i_1, J_1, e_1')) = mark(j, e_1) \\
&\qquad (j_2, (i_2, J_2, e_2')) = mark(j_1, e_2) \\
&\qquad i = i_1 \vee i_2 \\
&mark(j, e^{n,m}) = (j_1, (i, 0, (i, J, e')^{n,m})) \\
&\quad \text{where } (j_1, (i, J, e')) = mark(j, e) \\
&mark(j, (e)) = (j_1, (1, J \cup \{j\}, e')) \\
&\quad \text{where } (j_1, (i, J, e')) = mark(j + 1, e)
\end{aligned}
\qquad
\begin{aligned}
&enum : \mathbb{Z} \times \mathcal{IR}_\Sigma \to \mathbb{Z} \times \mathcal{IR}_\Sigma \\
&enum(i', (i, J, e)) \mid_{e \in \{\epsilon, \alpha\}} = (i' + i, (i' \times i, J, e)) \\
&enum(i', (i, J, e_1 \circ e_2)) \mid_{\circ \in \{|, \cdot\}} = (i_2, (i' \times i, J, e_1' \circ e_2')) \\
&\quad \text{where } (i_1, e_1') = enum(i' + i, e_1) \\
&\qquad (i_2, e_2') = enum(i_1, e_2) \\
&enum(i', (i, J, e^{n,m})) = (i_1, (i' \times i, J, e_1^{n,m})) \\
&\quad \text{where } (i_1, e_1) = enum(i' + i, e) \\
&\\
&IRE(e) = e' \\
&\quad \text{where } (\_, e') = enum(1, mark(1, e))
\end{aligned}
$$

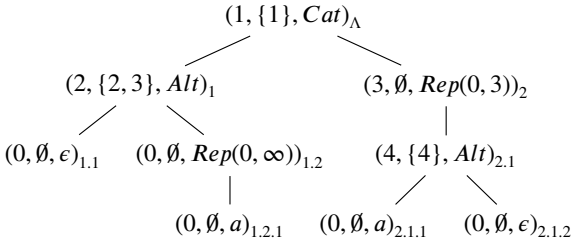
The relation between REs and PTs is given by the operator $PT : \mathcal{IR}_\Sigma \to 2^{\mathcal{T}_\Sigma}$. Each IRE denotes a set of PTs. We write $str(t)$ to denote the string formed by concatenation of all alphabet symbols in the left-to-right traversal of t, and $PT(e, w)$ denotes the set $\left\{t \in PT(e) \mid str(t) = w\right\}$ of all PTs for IRE e and a string w.

$$
\begin{aligned}
PT\big((i, \_, \epsilon)\big) &= \{\epsilon^i\} \\
PT\big((i, \_, \alpha)\big) &= \{\alpha^i\} \\
PT\big((i, \_, (i_1, J_1, e_1) \mid (i_2, J_2, e_2))\big) &= \left\{T^i(t, \perp^{i_2}) \mid t \in PT\big((i_1, J_1, e_1)\big)\right\} \cup \left\{T^i(\perp^{i_1}, t) \mid t \in PT\big((i_2, J_2, e_2)\big)\right\} \\
PT\big((i, \_, (i_1, J_1, e_1) \cdot (i_2, J_2, e_2))\big) &= \left\{T^i(t_1, t_2) \mid t_1 \in PT\big((i_1, J_1, e_1)\big), t_2 \in PT\big((i_2, J_2, e_2)\big)\right\} \\
PT\big((i, \_, (i_1, J_1, e_1)^{n,m})\big) \mid_{n=0} &= \left\{T^i(t_1, \dots, t_m) \mid t_k \in PT\big((i_1, J_1, e_1)\big), 1 \le k \le m\right\} \cup \left\{T^i(\perp^{i_1})\right\} \\
PT\big((i, \_, (i_1, J_1, e_1)^{n,m})\big) \mid_{n>0} &= \left\{T^i(t_n, \dots, t_m) \mid t_k \in PT\big((i_1, J_1, e_1)\big), n \le k \le m\right\}
\end{aligned}
$$

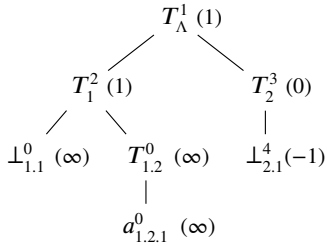
Following Okui and Suzuki, we assign *positions* to the nodes of IRE and PT. The root position is Λ, and position of the i-th subtree of a tree with position x is $x.i$ (we shorten $\|t\|_\Lambda$ as $\|t\|$). The *length* of position x, denoted $|x|$, is defined as 0 for Λ and $|x| + 1$ for $x.i$. The subtree of a tree t at position x is denoted $t|_x$. Position x is a *prefix* of position y iff $y = x.x'$ for some x', and a *proper prefix* if additionally $x \neq y$. Position x is a *sibling* of position y iff $y = z.i, x = z.j$ for some z and $i, j \in \mathbb{N}$. Positions are ordered lexicographically. The set of all positions of a tree t is denoted $Pos(t)$. Additionally, we define a set of *submatch positions* as $Sub(t) = \left\{x \in Pos(t) \mid \exists i \neq 0, s : t|_x = s^i\right\}$ — a subset of $Pos(t)$ that contains positions of subtrees with nonzero implicit submatch index. Intuitively, this is the set of positions important for disambiguation: in the case of ambiguity we do not need to consider the full trees, just the relevant parts of them. PTs have two definitions of norm, one for Pos and one for Sub, which we call *p-norm* and *s-norm* respectively.

$$
\begin{aligned}
&mark(1,\ (((\epsilon|a^{0,\infty}))(\epsilon|a)^{0,3})) = \lceil \\
&\quad mark(2,\ ((\epsilon|a^{0,\infty}))(\epsilon|a)^{0,\infty}) = \lceil \\
&\quad\quad mark(2,\ ((\epsilon|a^{0,\infty}))) = \lceil \\
&\quad\quad\quad mark(3,\ (\epsilon|a^{0,\infty})) = \lceil \\
&\quad\quad\quad\quad mark(4,\ \epsilon|a^{0,\infty}) = \lceil \\
&\quad\quad\quad\quad\quad mark(4,\ \epsilon) = (4,\ (0,\emptyset,\epsilon)) \\
&\quad\quad\quad\quad\quad mark(4,\ a^{0,\infty}) = \lceil \\
&\quad\quad\quad\quad\quad\quad mark(4,\ a) = (4,\ (0,\emptyset,a)) \\
&\quad\quad\quad\quad\quad \rfloor = (4,\ (0,\emptyset,(0,\emptyset,a)^{0,\infty})) \\
&\quad\quad\quad\quad \rfloor = (4,\ (0,\emptyset,(0,\emptyset,\epsilon) \mid (0,\emptyset,(0,\emptyset,a)^{0,\infty}))) \\
&\quad\quad\quad \rfloor = (4,\ (1,\{3\},(0,\emptyset,\epsilon) \mid (0,\emptyset,(0,\emptyset,a)^{0,\infty}))) \\
&\quad\quad \rfloor = (4,\ (1,\{2,3\},(0,\emptyset,\epsilon) \mid (0,\emptyset,(0,\emptyset,a)^{0,\infty}))) \\
&\quad\quad mark(4,\ (\epsilon|a)^{0,3}) = \lceil \\
&\quad\quad\quad mark(4,\ (\epsilon|a)) = \lceil \\
&\quad\quad\quad\quad mark(5,\ \epsilon|a) = \lceil \\
&\quad\quad\quad\quad\quad mark(5,\ \epsilon) = (5,\ (0,\emptyset,\epsilon)) \\
&\quad\quad\quad\quad\quad mark(5,\ a) = (5,\ (0,\emptyset,a)) \\
&\quad\quad\quad\quad \rfloor = (5,\ (0,\emptyset,(0,\emptyset,\epsilon) \mid (0,\emptyset,a))) \\
&\quad\quad\quad \rfloor = (5,\ (1,\{4\},(0,\emptyset,\epsilon) \mid (0,\emptyset,a))) \\
&\quad\quad \rfloor = (5,\ (1,\emptyset,(1,\{4\},(0,\emptyset,\epsilon) \mid (0,\emptyset,a))^{0,3})) \\
&\quad \rfloor = (5,\ (1,\emptyset,(1,\{2,3\},(0,\emptyset,\epsilon) \mid (0,\emptyset,(0,\emptyset,a)^{0,\infty})) \\
&\qquad\qquad \cdot (1,\emptyset,(1,\{4\},(0,\emptyset,\epsilon) \mid (0,\emptyset,a))^{0,3}))) \\
&\rfloor = (5,\ (1,\{1\},(1,\{2,3\},(0,\emptyset,\epsilon) \mid (0,\emptyset,(0,\emptyset,a)^{0,\infty})) \\
&\qquad\qquad \cdot (1,\emptyset,(1,\{4\},(0,\emptyset,\epsilon) \mid (0,\emptyset,a))^{0,3})))
\end{aligned}
$$

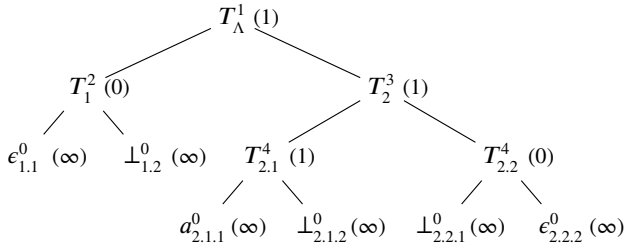
$$
\begin{aligned}
&enum(1,\ (1,\{1\},(1,\{2,3\},(0,\emptyset,\epsilon) \mid (0,\emptyset,(0,\emptyset,a)^{0,\infty})) \\
&\qquad\qquad \cdot (1,\emptyset,(1,\{4\},(0,\emptyset,\epsilon) \mid (0,\emptyset,a))^{0,\infty}))) = \lceil \\
&\quad enum(2,\ (1,\{2,3\},(0,\emptyset,\epsilon) \mid (0,\emptyset,(0,\emptyset,a)^{0,\infty}))) = \lceil \\
&\quad\quad enum(3,\ (0,\emptyset,\epsilon)) = (3,\ (0,\emptyset,\epsilon)) \\
&\quad\quad enum(3,\ (0,\emptyset,(0,\emptyset,a)^{0,\infty})) = \lceil \\
&\quad\quad\quad enum(3,\ (0,\emptyset,a)) = (3,\ (0,\emptyset,a)) \\
&\quad\quad \rfloor = (3,\ (0,\emptyset,(0,\emptyset,a)^{0,\infty})) \\
&\quad \rfloor = (3,\ (2,\{2,3\},(0,\emptyset,\epsilon) \mid (0,\emptyset,(0,\emptyset,a)^{0,\infty}))) \\
&\quad enum(3,\ (1,\emptyset,(1,\{4\},(0,\emptyset,\epsilon) \mid (0,\emptyset,a))^{0,\infty})) = \lceil \\
&\quad\quad enum(4,\ (1,\{4\},(0,\emptyset,\epsilon) \mid (0,\emptyset,a))) = \lceil \\
&\quad\quad\quad enum(5,\ (0,\emptyset,\epsilon)) = (5,\ (0,\emptyset,\epsilon)) \\
&\quad\quad\quad enum(5,\ (0,\emptyset,a)) = (5,\ (0,\emptyset,a)) \\
&\quad\quad \rfloor = (5,\ (4,\{4\},(0,\emptyset,\epsilon) \mid (0,\emptyset,a))) \\
&\quad \rfloor = (5,\ (3,\emptyset,(4,\{4\},(0,\emptyset,\epsilon) \mid (0,\emptyset,a))^{0,\infty})) \\
&\rfloor = (5,\ (1,\{1\},(2,\{2,3\},(0,\emptyset,\epsilon) \mid (0,\emptyset,(0,\emptyset,a)^{0,\infty})) \\
&\qquad\qquad \cdot (3,\emptyset,(4,\{4\},(0,\emptyset,\epsilon) \mid (0,\emptyset,a))^{0,\infty})))
\end{aligned}
$$



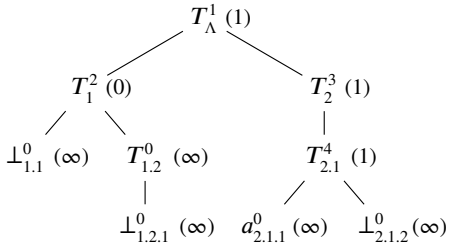
$$
\begin{aligned}
&IRE\big((((\epsilon|a^{0,\infty}))(a|\epsilon)^{0,3})\big) = \\
&\qquad (1,\{1\},(2,\{2,3\},(0,\emptyset,\epsilon) \mid (0,\emptyset,(0,\emptyset,a)^{0,\infty})) \\
&\qquad\quad \cdot (3,\emptyset,(4,\{4\},(0,\emptyset,a) \mid (0,\emptyset,\epsilon))^{0,3}))
\end{aligned}
$$



$$s\ =\ T^1(T^2(\bot^0, T^0(a^0)), T^3(\bot^4))$$



$$r\ =\ T^1(T^2(\epsilon^0, \bot^0), T^3(T^4(a^0, \bot^0), T^4(\bot^0, \epsilon^0)))$$



$$t\ =\ T^1(T^2(\bot^0, T^0(\bot^0)), T^3(T^4(a^0, \bot^0)))$$

FIGURE 1 IRE for RE $(((\epsilon|a^{0,\infty}))(a|\epsilon)^{0,3})$ and examples of PTs for string a with S-norm in parentheses.

Definition 6. The *p-norm* and *s-norm* of a PT t at position x are:

$$\|t\|_x^{pos} = \begin{cases} -1 & \text{if } x \in Pos(t) \text{ and } t|_x = \perp^i \\ |str(t|_x)| & \text{if } x \in Pos(t) \text{ and } t|_x \neq \perp^i \\ \infty & \text{if } x \notin Pos(t) \end{cases} \qquad \|t\|_x^{sub} = \begin{cases} -1 & \text{if } x \in Sub(t) \text{ and } t|_x = \perp^i \\ |str(t|_x)| & \text{if } x \in Sub(t) \text{ and } t|_x \neq \perp^i \\ \infty & \text{if } x \notin Sub(t) \end{cases}$$

Generally, the norm of a subtree means the number of alphabet symbols in its leaves, with two exceptions. First, for nil subtrees the norm is -1: intuitively, they have the lowest "ranking" among all possible subtrees. Second, for nonexistent subtrees (those with positions not in $Pos(t)$) the norm is infinite. This may seem counter-intuitive at first, but it makes sense in the presence of REs with empty repetitions. According to POSIX, optional empty repetitions are not allowed, and our definition reflects this: if a tree s has a subtree $s|_x$ corresponding to an empty repetition, and another tree t has no subtree at position x, then the infinite norm $\|t\|_x$ "outranks" $\|s\|_x$. We define two orders on PTs:

Definition 7 (P-order on PTs)**.** Given parse trees $t, s \in PT(e, w)$ for some IRE e and string w, we say that $t <_x s$ w.r.t. *decision position* x iff $\|t\|_x^{pos} > \|s\|_x^{pos}$ and $\|t\|_y^{pos} = \|s\|_y^{pos}\ \forall y < x$. We say that $t < s$ iff $t <_x s$ for some x.

Definition 8 (S-order on PTs)**.** Given parse trees $t, s \in PT(e, w)$ for some IRE e and string w, we say that $t \prec_x s$ w.r.t. *decision position* x iff $\|t\|_x^{sub} > \|s\|_x^{sub}$ and $\|t\|_y^{sub} = \|s\|_y^{sub}\ \forall y < x$. We say that $t \prec s$ iff $t \prec_x s$ for some x.

Definition 9. PTs t and s are *incomparable*, denoted $t \sim s$, iff neither $t \prec s$, nor $s \prec t$.

Theorem 1. P-order $<$ is a strict total order on $PT(e, w)$ for any IRE e and string w.

Theorem 2. S-order $\prec$ is a strict weak order on $PT(e, w)$ for any IRE e and string w.

The following theorem 3 establishes an important relation between P-order and S-order. P-order is total, and there is a unique $<$-minimal tree t_{min}. S-order is partial, it partitions all trees into equivalence classes and there is a whole class of $\prec$-minimal trees T_{min} (such trees coincide in submatch positions, but differ in some non-submatch positions). Theorem 3 shows that $t_{min} \in T_{min}$. This means that P-order and S-order "agree" on the notion of minimal tree: we can continuously narrow down T_{min} until we are left with t_{min}. Note that this doesn't mean that P-order is an extension of S-order: the two orders may disagree. For example, consider trees t and r on figure1 : on one hand $t \prec_{2.2} r$, because $\|t\|_{2.2}^{sub} = \infty > 0 = \|r\|_{2.2}^{sub}$ and s-norms at all preceding submatch positions agree; on the other hand $r <_{1.1} t$, because $\|t\|_{1.1}^{pos} = -1 < 0 = \|r\|_{1.1}^{pos}$ and p-norms at all preceding positions agree.

Theorem 3. Let t_{min} be the $<$-minimal tree in $PT(e, w)$ for some IRE e and string w, and let T_{min} be the class of $\prec$-minimal trees in $PT(e, w)$. Then $t_{min} \in T_{min}$.

The following theorem 4 states that submatch refinement is a contiguous process: adding more parentheses in RE does not cardinally change existing submatch results, it only adds new details.

Theorem 4. Let e, e' be two REs, such that e' is constructed from e by adding a pair of parentheses around some subexpression, and let $\bar{e} = IRE(e)$ and $\bar{e}' = IRE(e')$. If T_{min}, T'_{min} are the classes of $\prec$-minimal trees in $PT(\bar{e}, w)$ and $PT(\bar{e}', w)$ respectively for some string w, then $T'_{min} \subseteq T_{min}$.

Following the idea of Okui and Suzuki, we go from comparison of parse trees to comparison of their linearized representation — parenthesized expressions. Parenthesis $\langle$ is opening, and parenthesis $\rangle$ is closing; the *nil*-parenthesis $\langle\rangle$ is both opening and closing. For convenience we sometimes annotate parentheses with *height*, which we define as the number of preceding opening parentheses (including this one) minus the number of preceding closing parentheses (including this one). Explicit height annotations allow us to consider PE fragments in isolation without losing the context of the whole expression. However, height is not a part of parenthesis itself, and it is not taken into account when comparing the elements of PEs. Function $\Phi : \mathbb{Z} \times \mathcal{T}_\Sigma \to \mathcal{P}_\Sigma$ transforms PT at the given height into a PE:

$$\Phi_h(t^i) = \begin{cases} str(t^i) & \text{if } i = 0 \\ \langle\rangle_h & \text{if } i \neq 0 \wedge t = \perp \\ \langle_{h+1}\rangle_h & \text{if } i \neq 0 \wedge t = \epsilon \\ \langle_{h+1} a\rangle_h & \text{if } i \neq 0 \wedge t = a \in \Sigma \\ \langle_{h+1} \Phi_{h+1}(t_1) \dots \Phi_{h+1}(t_n)\rangle_h & \text{if } i \neq 0 \wedge t = T(t_1, \dots, t_n) \end{cases}$$

For a given IRE e and string w the set of all PEs $\{\Phi_0(t) \mid t \in PT(e,w)\}$ is denoted $PE(e,w)$.

Definition 10 (Frame representation of PEs)**.** A given PE β can be represented as $\beta_0 a_1 \beta_1 \dots a_n \beta_n$, where β_i is the i-th *frame* — a possibly empty sequence of parentheses between subsequent alphabet symbols a_i and a_{i+1}, or the beginning and end of α.

Definition 11 (Comparable PE fragments)**.** PE fragments α and β are *comparable* if they are prefixes of strings in $PE(e,w)$ for some IRE e and string w and have the same number of frames.

Definition 12 (Fork)**.** The *fork* of comparable PE fragments α and β is the index of the first distinct pair of frames.

We use the following notation. For PE fragments α and β, $\alpha \sqcap \beta$ denotes their longest common prefix, and $\alpha \backslash \beta$ denotes the suffix of α after removing $\alpha \sqcap \beta$. For a PE fragment α, $first(\alpha)$ denotes the first parenthesis in α (or $\perp$ if α is empty or begins with an alphabet symbol), $lasth(\alpha)$ denotes the height of the last parenthesis in α (or ∞ if α is empty or begins with an alphabet symbol), and $minh(\alpha)$ denotes the minimal height of parenthesis in α (or ∞ if α is empty or begins with an alphabet symbol).

Definition 13 (Traces)**.** Let α, β be comparable PE prefixes, such that $\alpha = \alpha_0 a_1 \alpha_1 \dots a_n \alpha_n$, $\beta = \beta_0 a_1 \beta_1 \dots a_n \beta_n$ and k is the fork. We define $trace(\alpha, \beta)$ as the sequence $(h_0, \dots, h_n)$, where:

$$h_i = \begin{cases} -1 & \text{if } i < k \\ min(lasth(\alpha_i \sqcap \beta_i), minh(\alpha_i \backslash \beta_i)) & \text{if } i = k \\ min(h_{i-1}, minh(\alpha_i)) & \text{if } i > k \end{cases}$$

We write $traces(\alpha, \beta)$ to denote $\big(trace(\alpha, \beta), trace(\beta, \alpha)\big)$.

Definition 14. (Longest precedence.) Let α, β be comparable PE prefixes and $traces(\alpha, \beta) = \big((h_0, \dots, h_n), (h'_0, \dots, h'_n)\big)$. Then $\alpha \sqsubset \beta \Leftrightarrow \exists i \le n : \big(h_i > h'_i\big) \wedge \big(h_j = h'_j \ \forall j > i\big)$. If neither $\alpha \sqsubset \beta$, nor $\beta \sqsubset \alpha$, then α, β are *longest-equivalent*: $\alpha \sim \beta$ (note that in this case $h_i = h'_i$ for $1 \le i \le n$).

Definition 15. (Leftmost precedence.) Let α, β be comparable PE prefixes, and let $x = first(\alpha \backslash \beta)$, $y = first(\beta \backslash \alpha)$. Then $\alpha \subset \beta \Leftrightarrow x < y$, where the set of possible values of x and y is ordered as follows: $\perp <\rangle < \langle < \langle\rangle$.

Definition 16. (Longest-leftmost precedence.) Let α, β be comparable PE prefixes, then $\alpha < \beta \Leftrightarrow \big(\alpha \sqsubset \beta\big) \vee \big(\alpha \sim \beta \wedge \alpha \subset \beta\big)$.

Theorem 5. If $s, t \in PT(e,w)$ for some IRE e and string w, then $s \prec t \Leftrightarrow \Phi_h(s) < \Phi_h(t)\ \forall h$.

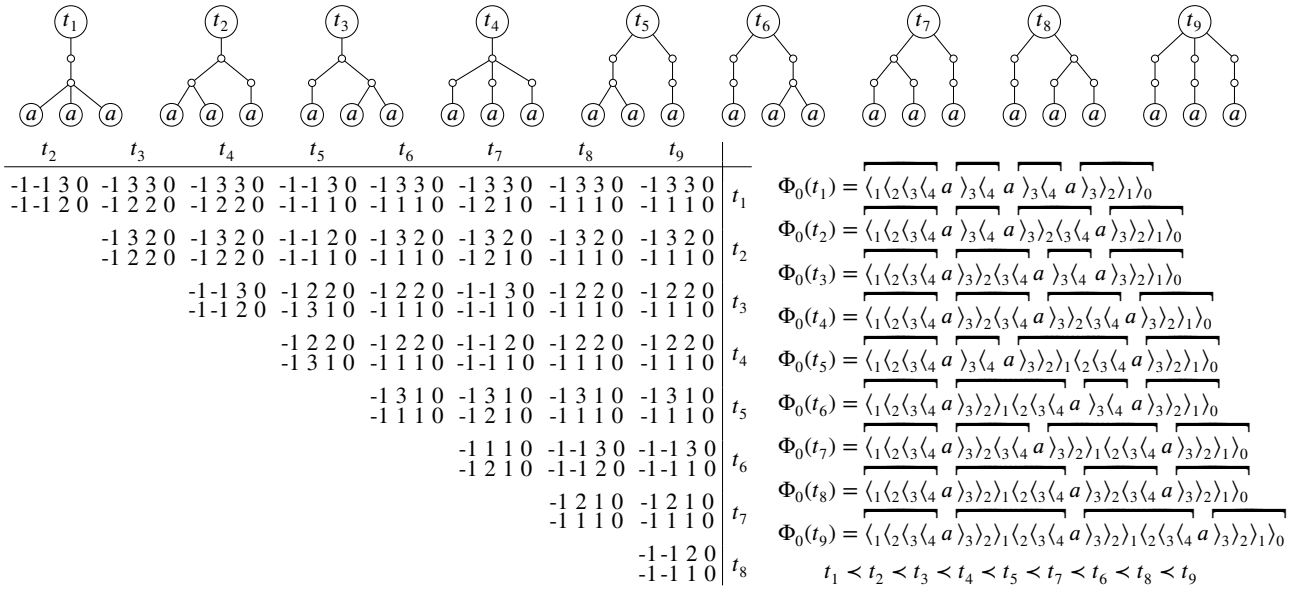
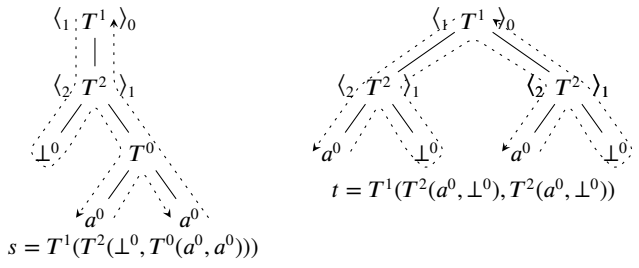


FIGURE 2 Example of pairwise frame-by-frame comparison of all PEs for RE $(((a)^{1,3})^{1,3})^{1,3}$ and string *aaa*. Table entry (t_i, t_j) contains $traces(\Phi_0(t_i), \Phi_0(t_j))$.



$\alpha = \Phi_0(s) = \overline{\langle_1\langle_2\ a\ \ a\ \rangle_1\rangle_0}$

$\beta = \Phi_0(t) = \overline{\langle_1\langle_2\ a\ \rangle_1\langle_2\ a\ \rangle_1\rangle_0}$

$traces(\alpha, \beta)$:

$$\begin{bmatrix} h_0 = h'_0 = -1 \\ h_1 = min(lasth(\langle_1\langle_2), minh(\epsilon)) = min(2, \infty) = 2 \\ h'_1 = min(lasth(\langle_1\langle_2), minh(\rangle_1\langle_2)) = min(2, 1) = 1 \\ h_2 = min(h_1, minh(\rangle_1\rangle_0)) = min(2, 0) = 0 \\ h'_2 = min(h'_1, minh(\rangle_1\rangle_0)) = min(1, 0) = 0 \end{bmatrix}$$

$\|s\|_1^{sub} = 2 > 1 = \|t\|_1^{sub} \wedge \|s\|_x^{sub} = \|t\|_x^{sub}\ \forall x < 1 \Rightarrow s \prec_1 t$ $\qquad h_1 > h'_1 \wedge h_2 = h'_2 \Rightarrow \alpha \sqsubset \beta \Rightarrow \alpha < \beta$

(a) – Rule 1: longest precedence, RE $(a|aa)^{0,\infty}$ and string aa.

$\alpha = \Phi_0(s) = \overline{\langle_1\langle_2\ a\ \rangle_1\langle\rangle_1\rangle_0}$

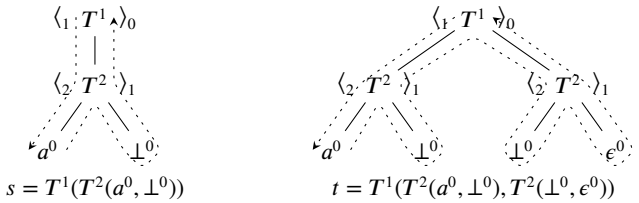
$\beta = \Phi_0(t) = \overline{\langle_1\langle\rangle_1\langle_2\ a\ \rangle_1\rangle_0}$

$traces(\alpha, \beta)$:

$$\begin{bmatrix} h_0 = min(lasth(\langle_1), minh(\langle_2)) = min(1, 2) = 1 \\ h'_0 = min(lasth(\langle_1), minh(\langle\rangle_1\langle_2)) = min(1, 1) = 1 \\ h_1 = min(h_0, minh(\rangle_1\langle\rangle_1\rangle_0)) = min(1, 0) = 0 \\ h'_1 = min(h'_0, minh(\rangle_1\rangle_0)) = min(1, 0) = 0 \end{bmatrix}$$

$\|s\|_1^{sub} = 1 > -1 = \|t\|_1^{sub} \wedge \|s\|_x^{sub} = \|t\|_x^{sub}\ \forall x < 1 \Rightarrow s \prec_1 t$ $\qquad h_i = h'_i \forall i \wedge first(\alpha \backslash \beta) = \langle < \langle\rangle = first(\beta \backslash \alpha) \Rightarrow \alpha \sim \beta \wedge \alpha \subset \beta \Rightarrow \alpha < \beta$

(b) – Rule 2: leftmost precedence, RE $(a)|(a)$ and string a.



$\alpha = \Phi_0(s) = \overline{\langle_1\langle_2\ a\ \rangle_1\rangle_0}$

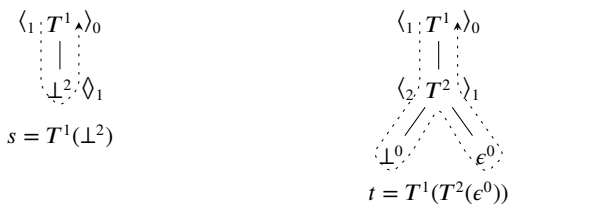
$\beta = \Phi_0(t) = \overline{\langle_1\langle_2\ a\ \rangle_1\langle_2\rangle_1\rangle_0}$

$traces(\alpha, \beta)$:

$$\begin{bmatrix} h_0 = h'_0 = -1 \\ h_1 = min(lasth(\rangle_1), minh(\rangle_0)) = min(1, 0) = 0 \\ h'_1 = min(lasth(\rangle_1), minh(\langle_2\rangle_1\rangle_0)) = min(1, 0) = 0 \end{bmatrix}$$

$\|s\|_2^{sub} = \infty > 0 = \|t\|_2^{sub} \wedge \|s\|_x^{sub} = \|t\|_x^{sub}\ \forall x < 2 \Rightarrow s \prec_2 t$ $\qquad h_i = h'_i \forall i \wedge first(\alpha \backslash \beta) = \rangle < \langle = first(\beta \backslash \alpha) \Rightarrow \alpha \sim \beta \wedge \alpha \subset \beta \Rightarrow \alpha < \beta$

(c) – Rule 3: no optional empty repetitions, RE $(a|\epsilon)^{0,\infty}$ and string a.



$\alpha = \Phi_0(s) = \overline{\langle_1\langle\rangle_1\rangle_0}$

$\beta = \Phi_0(t) = \overline{\langle_1\langle_2\rangle_1\rangle_0}$

$traces(\alpha, \beta)$:

$$\begin{bmatrix} h_0 = min(lasth(\langle_1), minh(\langle\rangle_1\rangle_0)) = min(1, 0) = 0 \\ h'_0 = min(lasth(\langle_1), minh(\langle_2\rangle_1\rangle_0)) = min(1, 0) = 0 \end{bmatrix}$$

$\|s\|_2^{sub} = 0 > -1 = \|s\|_2^{sub} \wedge \|s\|_x^{sub} = \|t\|_x^{sub}\ \forall x < 2 \Rightarrow t \prec_2 s$ $\qquad h_i = h'_i \forall i \wedge first(\alpha \backslash \beta) = \langle\rangle > \langle = first(\beta \backslash \alpha) \Rightarrow \alpha \sim \beta \wedge \beta \subset \alpha \Rightarrow \beta < \alpha$

(d) – Rule 4: empty match is better than no match, RE $(a|\epsilon)^{0,\infty}$ and string ϵ.

FIGURE 3 Examples: (a) – (d): four main rules of POSIX comparison.

Next, we go from comparison of PEs to comparison of TNFA paths. We extend the notion of order from PEs to paths.

Definition 17 (Path)**.** A *path* in TNFA $(\Sigma, Q, T, \Delta, q_0, q_f)$ is a sequence of transitions $\{(q_i, a_i, b_i, q_{i+1})\}_{i=1}^{n-1} \subseteq \Delta$, where $n \in \mathbb{N}$. Every path induces a string of alphabet symbols and a mixed string of alphabet symbols and tags that corresponds to a PE fragment: positive opening tags map to $\langle$, positive closing tags map to $\rangle$, and negative tags map to $\langle\rangle$. We write $q_1 \xrightarrow{w|\alpha} q_n$ to denote that a path from q_1 to q_n induces an alphabet string w and a PE fragment α.

Definition 18 (Path comparison)**.** A path $\pi_1 = q_1 \xrightarrow{w|\alpha} q_2$ is less than a path $\pi_2 = q_1 \xrightarrow{w|\beta} q_3$, denoted $\pi_1 < \pi_2$, if $\alpha < \beta$.

Definition 19 (Minimal path)**.** A path in TNFA for IRE e is *minimal* if it induces $\alpha = PE(t)$ for some minimal tree $t \in PT(e)$.

Definition 20 (Ambiguous paths)**.** Paths $q_1 \xrightarrow{w|\alpha} q_2$ and $q_1' \xrightarrow{w'|\beta} q_2'$ are *ambiguous* if $q_1 = q_1'$, $q_2 = q_2'$ and $w = w'$.

Definition 21 (Join point)**.** For paths $\pi_1 = q_1 \xrightarrow{w_1|\alpha} q \xrightarrow{w_2|\gamma} q_2$ and $\pi_2 = q_1 \xrightarrow{w_1|\beta} q \xrightarrow{w_3|\delta} q_3$ that have ambiguous prefixes $q_1 \xrightarrow{w_1|\alpha} q$ and $q_1 \xrightarrow{w_1|\beta} q$, state q is a *join point* (a state where ambiguous prefixes meet).

In order to justify our TNFA simulation algorithm, we need to show that PEs induced by TNFA paths can be compared incrementally (otherwise we would have to keep full-length PEs, which requires the amount of memory proportional to the length of input). Justification of incremental comparison consists of two parts: the following lemma 1 justifies comparison between frames, and lemmas 2, 3, 4 in section 4 justify comparison at join points inside of one frame (this is necessary as the number of paths in closure may be exponential in the number of states).

Lemma 1 (Frame-by-frame comparison of PEs)**.** If α, β are comparable PE prefixes, c is an alphabet symbol and γ is a single-frame PE fragment, then $\alpha < \beta$ implies $\alpha c \gamma < \beta c \gamma$.

4 | CLOSURE CONSTRUCTION

The problem of constructing ϵ-closure with POSIX disambiguation can be formulated as a shortest path problem on a directed graph with weighted arcs. In our case weight is not a number — it is the PE fragment induced by the path. We give two alternative implementations of *closure*(): *closure_gor*1() and *closure_gtop*(). The first algorithm, GOR1, is named after the well-known Goldberg-Radzik algorithm [15]. The second algorithm, GTOP, is an abbreviation of "global topological order". Both algorithms have the usual structure of shortest-path finding algorithms. Each algorithm starts with a set of initial configurations, an empty queue and an empty set of resulting configurations. Initial configurations are enqueued and the algorithm loops until the queue becomes empty. At each iteration it dequeues a configuration (q_2, q_1, u, d) and scans ϵ-transitions from state q. For a transition $(q_2, \_, \alpha, q_3)$ it constructs a new configuration (q_3, q_1, v, d) that combines u and α in an extended path v. If the resulting set already contains another configuration for state q_3, the algorithm chooses the configuration which has a better path from POSIX perspective. Otherwise it adds the new configuration to the resulting set. If the resulting set has been changed, the new configuration is enqueued for further scanning. Eventually all states in the ϵ-closure are explored, no improvements can be made, and the algorithm terminates.

The difference between GOR1 and GTOP is in the order they inspect configurations. Both algorithms are based on the idea of topological ordering. Unlike other shortest-path algorithms, their queuing discipline is based on graph structure, not on the distance estimates. This is crucial, because we do not have any distance estimates: paths can be compared, but there is no absolute "POSIX-ness" value that we can attribute to each path. GOR1 is described in [15]. It uses two stacks and makes a number of passes; each pass consists of a depth-first search on the admissible subgraph followed by a linear scan of states that are topologically ordered by depth-first search. The algorithm is one of the most efficient shortest-path algorithms [16]. n-Pass structure guarantees worst-case complexity $O(n\,m)$ of the Bellman-Ford algorithm, where n is the number of states and m is the number of transitions in the ϵ-closure (both can be approximated by TNFA size) [17]. GTOP is a simple algorithm that maintains one global priority queue (e.g. a binary heap) ordered by the topological index of states (for graphs with cycles, we assume reverse depth-first post-order). Since GTOP does not have the n-pass structure, its worst-case complexity is not clear. However, it is much simpler to implement and in practice it performs almost identically to GOR1 on graphs induced by TNFA ϵ-closures. On acyclic graphs, both GOR1 and GTOP have linear $O(n + m)$ complexity.

The general proof of correctness of shortest-path algorithms is out of the scope of this paper. However, we need to justify the application of these algorithms to our problem. In order to do that, we recall the framework for solving shortest-path algorithms based on *closed semirings* described in [12] (section 26.4) and show that our problem fits into this framework. A *semiring* is a structure $(\mathbb{K}, \oplus, \otimes, 0, 1)$, where $\mathbb{K}$ is a set, $\oplus{:}\mathbb{K} \times \mathbb{K} \to \mathbb{K}$ is an associative and commutative operation with identity element 0, $\otimes{:}\mathbb{K} \times \mathbb{K} \to \mathbb{K}$ is an associative operation with identity element 1, $\otimes$ distributes over $\oplus$ and 0 is annihilator for $\otimes$. Additionally, *closed* semiring requires that $\oplus$ is idempotent, any countable $\oplus$-sum of $\mathbb{K}$ elements is in $\mathbb{K}$, and associativity, commutativity,

```
1  closure_gor1(F=(Σ,Q,M,Δ,q0,qf),C,U,H,P)
2      X = (F,U,H,P
3      , topsort,linear // stacks of states
4      , result // configuration or ⊥ mapped to a state
5      , status // status of a state (one of OFF,TOP,LIN)
6      , active // a boolean indicating if a state needs rescan
7      , etrans // ε-transitions from a state, in priority order
8      , next // index of the active transition for a state
9      ) // context
10     result(q) ≡ ⊥
11     status(q) ≡ OFF
12     active(q) ≡ false
13     next(q) ≡ 1
14     for c = (_,q,_,_) ∈ C sorted by inverted prec() do
15         result(q) = c
16         push(topsort,q)
17     while topsort is not empty do
18         while topsort is not empty do
19             q = pop(topsort)
20             if status(q) ≠ LIN then
21                 status(q) = TOP
22                 push(topsort,q)
23                 if ¬scan(X,q,false) then
24                     status(q) = LIN
25                     pop(topsort)
26                     push(linear,q)
27         while linear is not empty do
28             q = pop(linear)
29             if active(q) then
30                 next(q) = 1
31                 active(q) = false
32                 scan(X,q,true)
33             status(q) = OFF
34     return prune(X)

35 scan(X,q,all)
36     any = false
37     while next(q) ≤ |etrans(q)| do
38         (q,ε,α,q') = etrans(q)[next(q)]
39         next(q) = next(q) + 1
40         (q'',q,u,d) = result(q)
41         c1 = result(q')
42         c2 = (q'',q',extend_path(U,u,α),d)
43         if c1=⊥ ∨ indegree(q')<2 ∨ less(X,c2,c1) then
44             result(q') = c2
45             if status(q) = OFF then
46                 any = true
47                 next(q') = 1
48                 push(topsort,q')
49                 if ¬all then break
50             else active(q') = 1
51     return any

52 closure_gtop(F=(Σ,Q,M,Δ,q0,qf),C,U,H,P)
53     X = (F,U,H,P
54     , queue // priority queue of states
55     , result // configuration or ⊥ mapped to a state
56     , status // status of a state (one of IN,OUT)
57     , etrans // ε-transitions from a state, in priority order
58     ) // context
59     result(q) ≡ ⊥
60     status(q) ≡ OUT
61     for c1 = (_,q,_,_) ∈ C do
62         c2 = result(q)
63         if c2=⊥ ∨ less(X,c1,c2) then
64             result(q) = c1
65             if status(q) ≠ IN then
66                 k = topological_index(q)
67                 insert_with_priority(queue,q,k)
68                 status(q) = IN
69     while queue is not empty do
70         q = extract_min(queue)
71         status(q) = OUT
72         for (q,ε,α,q') ∈ etrans(q) do
73             (q'',q,u,d) = result(q)
74             c1 = result(q')
75             c2 = (q'',q',extend_path(U,u,α),d)
76             if c1=⊥ ∨ indegree(q')<2 ∨ less(X,c2,c1) then
77                 result(q') = c2
78                 if status(q') ≠ IN then
79                     k = topological_index(q')
80                     insert_with_priority(queue,q',k)
81                     status(q') = IN
82     return prune(X)

83 prune(X)
84     return {c | c = (_,q,_,_) = result(q), q ∈ Q
85             ∧ (q = qf ∨ ∃(q',a,_,_) ∈ Δ | q = q' ∧ a ∈ Σ)}

86 less(X,c1,c2)
87     (_,_,l) = compare(c1,c2,U,H,P)
88     return l < 0

89 prec(X,c1,c2)
90     (q1,_,_,_) = c1
91     (q2,_,_,_) = c2
92     return P[q1][q2] < 0
```

ALGORITHM 2: Closure algorithms GOR1 and GTOP. Context components are addressed without qualification in the subroutines, e.g. U rather than $X.U$. Functions *compare*() and *extend_path*() are defined in sections 7 and 5. The definition of *push*(), *pop*(), *insert_with_priority*(), *extract_min*(), *indegree*() and *topological_index*() is omitted for brevity.

distributivity and idempotence apply to countable $\oplus$-sums. Mohri generalizes this definition and notes that either left or right distributivity is sufficient[13]. In our case $\mathbb{K}$ is the set of closure paths without tagged ϵ-loops: the following lemma 2 and 3 show that, on one hand, paths with tagged ϵ-loops are not minimal, and on the other hand such paths are discarded by the algorithm, so they can be removed from consideration. Consequently $\mathbb{K}$ is finite. We have semiring $(\mathbb{K}, min, \cdot, \bot, \epsilon)$, where min is POSIX comparison of ambiguous paths, $\cdot$ is concatenation of paths at the join points (subject to restriction that paths do not contain tagged ϵ-loops and remain within TNFA bounds — concatenation of arbitrary paths is not in $\mathbb{K}$), $\bot$ corresponds to artificial infinitely long path, and ϵ is the empty path. It is easy to see that min is commutative and associative, $\bot$ is identity for min ($min(\pi, \bot) = min(\bot, \pi) = \pi$), $\cdot$ is associative, ϵ is identity for $\cdot$ ($\pi \cdot \epsilon = \epsilon \cdot \pi = \pi$), and $\bot$ is an annihilator for $\cdot$ ($\pi \cdot \bot = \bot \cdot \pi = \bot$). Right distributivity of $\cdot$ over min for paths with at most one ϵ-loop is given by lemma 4. Idempotence holds because $min(\pi, \pi) = \pi$. Since $\mathbb{K}$ is finite, the properties for $\oplus$-sums over countable subsets are satisfied.

Lemma 2. Minimal paths do not contain tagged ϵ-loops.

Lemma 3. GOR1 and GTOP discard paths with tagged ϵ-loops.

Lemma 4 (Right distributivity of comparison over concatenation for paths without tagged ϵ-loops)**.** Let $\pi_\alpha = q_0 \xrightarrow{u|\alpha} q_1$ and $\pi_\beta = q_0 \xrightarrow{u|\beta} q_1$ be ambiguous paths in TNFA f for IRE e, and let $\pi_\gamma = q_1 \xrightarrow{\epsilon|\gamma} q_2$ be their common ϵ-suffix, such that $\pi_\alpha\pi_\gamma$ and $\pi_\beta\pi_\gamma$ do not contain tagged ϵ-loops. If $\alpha < \beta$ then $\alpha\gamma < \beta\gamma$.

5 | TREE REPRESENTATION OF PATHS

In this section we specify the representation of path fragments in configurations and define tag path tree U and the associated functions. The U-tree is a *prefix tree* of tags induced by the shortest path tree that is constructed by $closure()$. Some care is necessary with TNFA construction in order to ensure the prefix property (subautomata for alternatives should not start with identical tagged transitions), but that is easy to accommodate and we give the details in section 9. The U-tree can be stored as an array of nodes $(pred, succ, tag)$ where $pred$ is the index of a predecessor node, $succ$ is an ordered set of successor indices, and tag is a positive or negative tag. Successor indices are only necessary if the advanced algorithm for $update\_precedence()$ is used (section 7), otherwise the $succ$ component can be omitted. Individual paths in the U-tree are addressed by integer indices of tree nodes (zero index corresponds to the empty path). It is important to use numeric indices rather than pointers to nodes because it allows the use of the "two-fingers" algorithm that find the fork of two paths (section 7). The prefix tree representation is space efficient (common path prefixes are not duplicated), and the operation of copying a path (e.g. from one configuration to another) is as cheap as copying an integer index. This representation was used by multiple researches, e.g. Laurikari mentions a *functional data structure*[4] and Karper describes it as the *flyweight pattern*[14].

```
1  empty_path_tree()
2      return an empty list of tuples (pred, succ, tag)
3  extend_path(U, n, t)
4      if t ≠ ϵ then
5          m = length(U) + 1
6          insert(U[n].succ, m)
7          append(U, (n, ∅, t))
8          return m
9      else return n
10 unroll_path(U, n)
11     s = ϵ
12     while n ≠ 0 do
13         append(s, U[n].tag)
14         n = U[n].pred
15     return reverse(s)
```

ALGORITHM 3: Operations on tag path tree. The definition of $insert()$, $append()$ and $reverse()$ is omitted for brevity.

6 | REPRESENTATION OF MATCH RESULTS

In this section we show two ways to construct match results: POSIX offsets and a parse tree.

In the case of POSIX offsets, the d-component of a configuration is an array of offsets for each tag. Offsets are updated incrementally at each step by scanning the corresponding path fragment and setting negative tags to -1 and positive tags to the

current step number. For negative tags, it is necessary to also set all nested tags to -1. Only the most recent value of each tag is needed, therefore the offset is updated at most once. At the end of match, the offset array is converted to the representation specified by the POSIX standard: an array *pmatch* of offset pairs (rm_so, rm_eo). Conversion takes into account the mapping of tags to submatch groups. Section 9 shows the construction of the mapping M from tags to their nested tags and submatch groups.

In the case of a parse tree, the d-component of a configuration is a tagged string that is accumulated at each step, and eventually converted to a parse tree at the end of match. The resulting parse tree is only partially structured: leaves that correspond to subexpressions with zero implicit submatch index contain "flattened" substring of alphabet symbols. It is possible to construct parse trees incrementally as well, but this is more complex and the partial trees may require even more space than tagged strings.

```
initial_result(M)
    d: integer array of size |dom(M)|
    return d

update_result(M, C, U, k, _)
    return {(q2, q1, u, apply_tags(M, U, u, d, k))
                | (q2, q1, u, d) ∈ C}

apply_tags(M, U, u, d, k)
    done(t) = false ∀t
    while u ≠ 0 do
        t = U[u].tag
        if t > 0 then
            if ¬done(t) then
                done(t) = true
                d[t] = k
        else
            for n ∈ M(−t).N do
                if ¬done(n) then
                    done(n) = true
                    d[n] = −1
        u = U[u].pred
    return d

final_result(M, U, u, d, k)
    d' = apply_tags(M, U, u, d, k)
    pmatch[0].rm_so = 0
    pmatch[0].rm_eo = k
    for i = 1, |dom(M)|/2 do
        for s ∈ M(2i).S do
            pmatch[s].rm_so = d'[2i − 1]
            pmatch[s].rm_eo = d'[2i]
    return pmatch
```

```
initial_result(_)
    return ϵ

update_result(_, C, U, _, a)
    return {(q2, q1, u, d · unroll_path(U, u) · a)
                | (q2, q1, u, d) ∈ C}

parse_tree(u, i)
    if u = (2i−1) · (2i) then
        return T^i(ϵ)
    if u = (1−2i) · … then
        return T^i(⊥)
    if u = (2i−1) · a1 … an · (2i) ∧ a1, … , an ∈ Σ then
        return T^i(a1, … , an)
    if u = (2i−1) · β1 … βm · (2i) ∧ β1 = 2j−1 ∈ T then
        n = 0, k = 1
        while k ≤ m do
            l = k
            while |β_{k+1}| > 2j do k = k + 1
            n = n + 1
            tn = parse_tree(βl … βk, j)
        return T^i(t1, … , tn)
    return ⊥ // ill-formed PE

final_result(_, U, u, d, _)
    return parse_tree(d · unroll_path(U, u), 1)
```

ALGORITHM 4: Construction of match results: POSIX offsets (on the left) and parse tree (on the right).

7 | DISAMBIGUATION PROCEDURES

In this section we define disambiguation procedures *compare*() and *update_precedence*(). The pseudocode follows definition 16 closely and relies on the prefix tree representation of paths given in section 6. In order to find fork of two paths in *compare*() we used the so-called "two-fingers" algorithm, which is based on the observation that parent index is always less than child index. Given two indices n_1 and n_2, we continuously set the greater index to its parent until the indices become equal, at which point we have either found fork or the root of U-tree. We track minimal height of each path along the way and memorize the pair of indices right after the fork — they are used to determine the leftmost path in case of equal heights.

```
compare(c1, c2, U, H, P)
    (_, q1, u1, _) = c1
    (_, q2, u2, _) = c2
    if q1 = q2 ∧ u1 = u2 then return (∞, ∞, 0)
    fork = (q1 = q2)
    if fork then h1 = h2 = ∞
    else h1 = H[q1][q2], h2 = H[q2][q1]
    u'1 = u'2 = ⊥
    while u1 ≠ u2 do
        if u1 > u2 then
            h1 = min(h1, height(U[u1].tag))
            u'1 = u1, u1 = U[u1].pred
        else
            h2 = min(h2, height(U[u2].tag))
            u'2 = u2, u2 = U[u2].pred
    if u1 ≠ ⊥ then
        h = height(U[u1].tag)
        h1 = min(h1, h), h2 = min(h2, h)
    if h1 > h2 then l = −1
    else if h1 < h2 then l = 1
    else if ¬fork then l = P[q1][q2]
    else l = leftprec(u'1, u'2, U)
    return (h1, h2, l)

leftprec(u1, u2, U)
    if u1 = u2 then return 0
    if u1 = ⊥ then return −1
    if u2 = ⊥ then return 1
    t1 = U[u1].tag, t2 = U[u2].tag
    if t1 < 0 then return 1
    if t2 < 0 then return −1
    if t1 mod 2 ≡ 0 then return −1
    if t2 mod 2 ≡ 0 then return 1
    return 0

update_precedence(F, C, U, H, P)
    for c1 = (q1, _, _, _) ∈ C do
        for c2 = (q2, _, _, _) ∈ C do
            (h1, h2, l) = compare(c1, c2, U, H, P)
            H'[q1][q2] = h1, P'[q1][q2] = l
            H'[q2][q1] = h2, P'[q2][q1] = −l
    return (H', P')
```

```
update_precedence(F, C, U, H, P)
    stack : empty stack of U-tree indices
    level : empty array of tuples (q', q, u, h)
    next : active successor for a given U-tree index
    next(u) ≡ 1
    i = 0
    push(stack, 0)
    while stack is not empty do
        u = pop(stack)
        if next(u) < k then
            push(stack, u)
            push(stack, U[u].succ[next(u)])
            next(u) = next(u) + 1
            continue
        h = height(U[u].tag), i1 = i
        for (q', q, u1, _) ∈ C | u1 = u do
            i = i + 1, level[i] = (q', q, ⊥, h)
        for j1 = i1 + 1, i do
            for j2 = j1, i do
                (q'1, q1, _, _) = level[j1]
                (q'2, q2, _, _) = level[j2]
                if u = 0 ∧ q1 ≠ q2 then
                    h1 = H[q1][q2], h2 = H[q2][q1]
                    l = P[q1][q2]
                else h1 = h2 = h, l = 0
                H'[q'1][q'2] = h1, P'[q'1][q'2] = l
                H'[q'2][q'1] = h2, P'[q'2][q'1] = −l
        for u' ∈ U[u].succ in reverse do
            i2 = i1
            while i2 > 0 ∧ level[i2].u = u' do i2 = i2 − 1
            for j1 = i2, i1 do
                level[j1].h = min(level[j1].h, h);
                for j2 = i1, i do
                    (q'1, q1, u1, h1) = level[j1]
                    (q'2, q2, u2, h2) = level[j2]
                    if u = 0 ∧ q1 ≠ q2 then
                        h1 = min(h1, H[q1][q2])
                        h2 = min(h2, H[q2][q1])
                    if h1 > h2 then l = −1
                    else if h1 < h2 then l = 1
                    else if q1 ≠ q2 then l = P[q1][q2]
                    else l = leftprec(u1, u2, U)
                    H'[q'1][q'2] = h1, P'[q'1][q'2] = l
                    H'[q'2][q'1] = h2, P'[q'2][q'1] = −l
                i1 = i2
            for j = i1, i do level[j].u = u
        return (H', P')
```

ALGORITHM 5: Disambiguation procedures. Function *height*() gives the height of a tag; its definition is omitted for brevity (it can be computed in one top-down pass over IRE when constructing the TNFA, see section 9).

We give two alternative algorithms for *update_precedence*(): a simple one with $O(m^2\,t)$ complexity (on the left) and a complex one with $O(m^2)$ complexity (on the right), where m is the number of states in the ϵ-closure and t is the number of tags. The worst case is demonstrated by RE $((a|\epsilon)^{0,k})^{0,\infty}$ where $k \in \mathbb{N}$, for which the simple algorithm takes $O(k^3)$ time and the complex

algorithm takes $O(k^2)$ time. The idea of the complex algorithm is to avoid repeated re-scanning of path prefixes in the U-tree. It makes one pass over the tree, constructing an array $level$ of items (q', q, u, h), where q' and q are target and origin states (as in configurations), u is the current tree index, and h is the current minimal height. One item is added per each closure configuration (q', q, u, d) when traversal reaches the tree node with index u. After a subtree has been traversed, the algorithm scans $level$ items added during traversal of this subtree (such items are distinguished by their u-component), sets their h-component to the minimum of h and the height of tag at the current node, and computes the new value of H and P matrices for each pair of target states in items from different branches. After that, u-component of all scanned items is downgraded to the tree index of the current node (erasing the difference between items from different branches).

8 | LAZY DISAMBIGUATION

Most of the disambiguation overhead in our algorithm comes from updating the H and P matrices at each step. It is all the more unfortunate since many comparisons performed by $update\_precedence()$ are useless — the compared paths may never meet. In fact, if the input is unambiguous, all comparisons are useless. A natural idea, therefore, is to compare paths only in case of a real ambiguity (when they meet in closure) and avoid the computation of precedence matrices altogether. We can do it with a few modifications to our original algorithm. First, we no longer need the H and P matrices and $update\_precedence()$ function. Instead, we introduce a cache Z that maps a pair of U-tree indices (u_1, u_2) to a triple of precedence values (h_1, h_2, l). The Z-cache stores the "useful" part of H and P matrices on multiple preceding steps. It is populated lazily during disambiguation and allows us to avoid re-computing the same values multiple times. Second, we need to modify the U-tree representation of paths in the following way: successor indices are no longer needed (they are only used in the advanced $update\_precedence()$ algorithm), and the U-tree nodes must be augmented with the current step and the origin state: $(pred, tag, step, orig)$. Third, instead of setting the u-component of configurations to zero at each step of the $match()$ algorithm in section 2, we need to set it to the u-component of the parent configurations, so that paths are accumulated rather than reset at each step. Fourth, we no longer need to call $update\_result()$ at each step — this can be done once at the end of match. The only part of the algorithm that requires non-trivial change is the $compare()$ function, which is defined below via co-recursive functions $compare1()$ that populates the Z-cache and $compare2()$ that recurses one frame back, or stops if the fork has been reached.

```
1  compare(c1, c2, U, Z)
2      (_, _, u1, _) = c1
3      (_, _, u2, _) = c2
4      return compare1(u1, u2, U, Z)

5  compare1(u1, u2, U, Z)
6      if Z(u1, u2) = ⊥ then
7          Z(u1, u2) = compare2(u1, u2, U, Z)
8      return Z(u1, u2)

9  compare2(u1, u2, U, Z)
10     if u1 = u2 then return (∞, ∞, 0)
11     h1 = h2 = ∞
12     q1 = U[u1].orig,  q2 = U[u2].orig
13     k1 = U[u1].step,  k2 = U[u2].step,  k = max(k1, k2)
14     fork = (q1 = q2) ∧ (k1 = k2)
15     u'1 = u'2 = ⊥
16     while u1 ≠ u2 ∧ (k1 ≥ k ∨ k2 ≥ k) do
17         if k1 ≥ k ∧ (u1 > u2 ∨ k2 < k) then
18             h1 = min(h1, height(U[u1].tag))
19             u'1 = u1,  u1 = U[u1].pred,  k1 = U[u1].step
20         else
21             h2 = min(h2, height(U[u2].tag))
22             u'2 = u2,  u2 = U[u2].pred,  k2 = U[u2].step
23     if ¬fork then
24         (h'1, h'2, l) = compare1(u1, u2, U, Z)
25         h1 = min(h1, h'1),  h2 = min(h2, h'2)
26     else if u1 ≠ ⊥ then
27         h = height(U[u1].tag)
28         h1 = min(h1, h),  h2 = min(h2, h)
29     if h1 > h2 then l = −1
30     else if h1 < h2 then l = 1
31     else if fork then l = leftprec(u'1, u'2, U)
32     return (h1, h2, l)
```

ALGORITHM 6: Lazy disambiguation procedures (the Z-cache is modified in-place).

The problem with this approach is that we need to keep full-length history of each active path: at the point of ambiguity we may need to look an arbitrary number of steps back in order to find the fork of ambiguous paths. This may be acceptable for small inputs (and memory footprint may even be smaller due to the reduction of precedence matrices), but it is infeasible for long or streaming inputs. A possible solution may be a hybrid approach that uses lazy disambiguation, but every k steps fully calculates the precedence matrices and "forgets" the path prefixes. Another possible solution is to keep both algorithms and switch between them depending on the length of input.

9 | TNFA CONSTRUCTION

TNFA construction is given by the function *tnfa*() that accepts IRE e and state q_f and returns TNFA for e with the final state q_f (algorithm 7). This precise construction is not necessary for the algorithms to work, but it has a number of important properties.

- Non-essential ϵ-transitions are removed, as they make closure algorithms GOR1 and GTOP slower.
- Bounded repetition $e^{n,m}$ is unrolled in a way that duplicates e exactly m times and factors out common path prefixes: subautomaton for $(k+1)$-th iteration is only reachable from subautomaton for k-th iteration. For example, $a^{2,5}$ is unrolled as $aa(\epsilon|a(\epsilon|a(\epsilon|a)))$, not as $aa(\epsilon|a|aa|aaa)$. This ensures that the tag tree build by ϵ-closure is a prefix tree.
- Priorities are assigned so as to make it more likely that depth-first traversal of the ϵ-closure finds short paths before long paths. This is an optimization that makes GOR1 much faster in specific cases with many ambiguous paths that are longest-equivalent and must be compared by the leftmost criterion. An example of such case is $(((\epsilon)^{0,k})^{0,k})^{0,k})$ for some large k. Because GOR1 has a depth-first component, it is sensitive to the order of transitions in TNFA. If it finds the shortest path early, then all other paths are just canceled at the first join point with the shortest path (because there is no improvement and further scanning is pointless). In the opposite case GOR1 finds long paths before short ones, and whenever it finds an improved (shorter) path, it has to schedule configurations for re-scan on the next pass. This causes GOR1 to make more passes and scan more configurations on each pass, which makes it significantly slower. Arguably this bias is a weakness of GOR1 — GTOP is more robust in this respect.
- When adding negative tags, we add a single transition for the topmost closing tag (it corresponds to the nil-parenthesis, which has the height of a closing parenthesis). Then we map this tag to the full range of its nested tags, including itself and the pair opening tag. An alternative approach is to add all nested negative tags as TNFA transitions and get rid of the mapping, but this may result in significant increase of TNFA size and major slowdown (we observed 2x slowdown on large tests with hundreds of submatch groups).
- Although for simplicity we use sets to represent nested tags and submatch groups for each tag (the M component of a TNFA), these sets contain consecutive numbers and in practice can be represented more efficiently as ranges.
- Passing the final state q_f in *tnfa*() function allows to link subautomata in a simple and efficient way. It allows to avoid tracking and patching of subautomaton transitions that go to the final state (when this final state needs to be changed).

See figure4 for an example of TNFA.

10 | COMPLEXITY ANALYSIS

Our algorithm consists of three steps: conversion of RE to IRE, construction of TNFA from IRE and simulation of TNFA on the input string. We discuss time and space complexity of each step in term of the following parameters: n — the length of input, m — the size of RE with counted repetition subexpressions expanded (each subexpression duplicated the number of times equal to the repetition counter), and t — the number of capturing groups and subexpressions that contain them.

The first step, conversion of RE to IRE, is given by the functions *mark*() and *enum*() from section 3. For each sub-RE, *mark*() constructs a corresponding sub-IRE, and *enum*() performs a linear visit of the IRE (which doesn't change its structure), therefore IRE size is $O(m)$. Each subexpression is processed twice (once by *mark*() and once by *enum*()) and processing takes $O(1)$ time, therefore total time is $O(m)$.

1 $tnfa(e, q_f)$

2 **if** $e = (0, \emptyset, \epsilon)$ **then**
3 **return** $(\Sigma, \{q_f\}, \emptyset, \emptyset, q_f, q_f)$

4 **else if** $e = (0, \emptyset, a) \mid_{a \in \Sigma}$ **then**
5 **return** $(\Sigma, \{q_0, q_f\}, \emptyset, \{(q_0, a, \epsilon, q_f)\}, q_0, q_f)$

6 **else if** $e = (0, \emptyset, e_1 \cdot e_2)$ **then**
7 $(\Sigma, Q_2, M_2, \Delta_2, q_2, q_f) = tnfa(e_2, q_f)$
8 $(\Sigma, Q_1, M_1, \Delta_1, q_1, q_2) = tnfa(e_1, q_2)$
9 **return** $(\Sigma, Q_1 \cup Q_2, M_1 \sqcup M_2, \Delta_1 \cup \Delta_2, q_1, q_f)$

10 **else if** $e = (0, \emptyset, e_1 \mid e_2)$ **then**
11 $(\Sigma, Q_2, M_2, \Delta_2, q_2, q_f) = tnfa(e_2, q_f)$
12 $(\Sigma, Q'_2, M_2, \Delta'_2, q'_2, q_f) = ntag(M_2, q_f)$
13 $(\Sigma, Q_1, M_1, \Delta_1, q_1, q'_2) = tnfa(e_2, q'_2)$
14 $(\Sigma, Q'_1, M_1, \Delta'_1, q'_1, q_2) = ntag(M_1, q_2)$
15 $Q = Q_1 \cup Q'_1 \cup Q_2 \cup Q'_2 \cup \{q_0\}$
16 $\Delta = \Delta_1 \cup \Delta'_1 \cup \Delta_2 \cup \Delta'_2 \cup \{(q_0, 1, \epsilon, q_1), (q_0, 2, \epsilon, q'_1)\}$
17 **return** $(\Sigma, Q, M_1 \sqcup M_2, \Delta, q_0, q_f)$

18 **else if** $e = (0, \emptyset, e_1^{n,m}) \mid_{1 < n \le m \le \infty}$ **then**
19 $(\Sigma, Q_1, M_1, \Delta_1, q_2, q_f) = tnfa((0, \emptyset, e_1^{n-1,m-1}), q_f)$
20 $(\Sigma, Q_2, M_2, \Delta_2, q_1, q_2) = tnfa(e_1, q_2)$
21 **return** $(\Sigma, Q_1 \cup Q_2, M_1 \sqcup M_2, \Delta_1 \cup \Delta_2, q_1, q_f)$

22 **else if** $e = (0, \emptyset, e_1^{0,m})$ **then**
23 $(\Sigma, Q_1, M_1, \Delta_1, q_1, q_f) = tnfa((0, \emptyset, e_1^{1,m}), q_f)$
24 $(\Sigma, Q'_1, M_1, \Delta'_1, q'_1, q_f) = ntag(M_1, q_f)$
25 $Q = Q_1 \cup Q'_1 \cup \{q_0\}$
26 $\Delta = \Delta_1 \cup \Delta'_1 \cup \{(q_0, 1, \epsilon, q_1), (q_0, 2, \epsilon, q'_1)\}$
27 **return** $(\Sigma, Q, M_1, \Delta, q_0, q_f)$

28 **else if** $e = (0, \emptyset, e_1^{1,\infty})$ **then**
29 $(\Sigma, Q_1, M_1, \Delta_1, q_0, q_1) = tnfa(e_1, \_)$
30 $Q = Q_1 \cup \{q_f\}$
31 $\Delta = \Delta_1 \cup \{(q_1, 1, \epsilon, q_0), (q_1, 2, \epsilon, q_f)\}$
32 **return** $(\Sigma, Q, M_1, \Delta, q_0, q_f)$

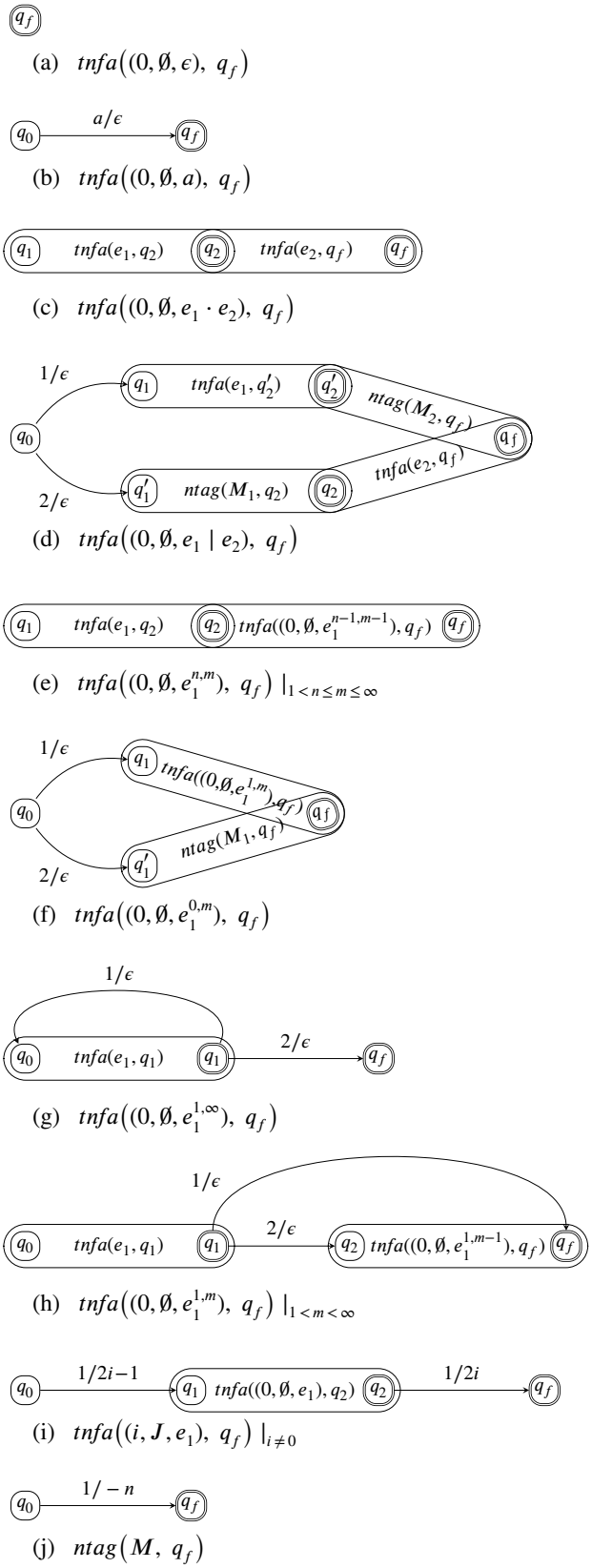
33 **else if** $e = (0, \emptyset, e_1^{1,1})$ **then return** $tnfa(e_1, q_f)$

34 **else if** $e = (0, \emptyset, e_1^{1,m}) \mid_{1 < m < \infty}$ **then**
35 $(\Sigma, Q_1, M_1, \Delta_1, q_1, q_f) = tnfa((0, \emptyset, e_1^{1,m-1}), q_f)$
36 $(\Sigma, Q_2, M_2, \Delta_2, q_0, q_2) = tnfa(e_1, q_1)$
37 $\Delta = \Delta_1 \cup \Delta_2 \cup \{(q_1, 1, \epsilon, q_f), (q_1, 2, \epsilon, q_2)\}$
38 **return** $(\Sigma, Q_1 \cup Q_2, M_1 \sqcup M_2, \Delta, q_0, q_f)$

39 **else if** $e = (i, J, e_1) \mid_{i \neq 0}$ **then**
40 $(\Sigma, Q_1, M_1, \Delta_1, q_1, q_2) = tnfa((0, \emptyset, e_1), q_2)$
41 $Q = Q_1 \cup \{q_0, q_f\}$
42 $N = dom(M_1) \sqcup \{2i-1, 2i\}$ // nested tags
43 $M(m) = \begin{cases} M_1(m) & \text{if } m \in dom(M_1) \\ (J, N) & \text{if } m \in \{2i-1, 2i\} \end{cases}$
44 $\Delta = \Delta_1 \cup \{(q_0, 1, 2i-1, q_1), (q_2, 1, 2i, q_f)\}$
45 **return** $(\Sigma, Q, M, \Delta, q_0, q_f)$

46 $ntag(M, q_f)$

// smallest closing tag represents all tags in M
47 $n = min\{m \in dom(M) \mid m \equiv 0 \ mod \ 2\}$
48 **return** $(\Sigma, \{q_0, q_f\}, M, \{(q_0, 1, -n, q_f)\}, q_0, q_f)$



ALGORITHM 7: TNFA construction. Notation $M = M_1 \sqcup M_2$ for functions with disjoint domains means that $M(m) = M_i(m)\ \forall m \in dom(M_i),\ 1 \le i \le 2$.

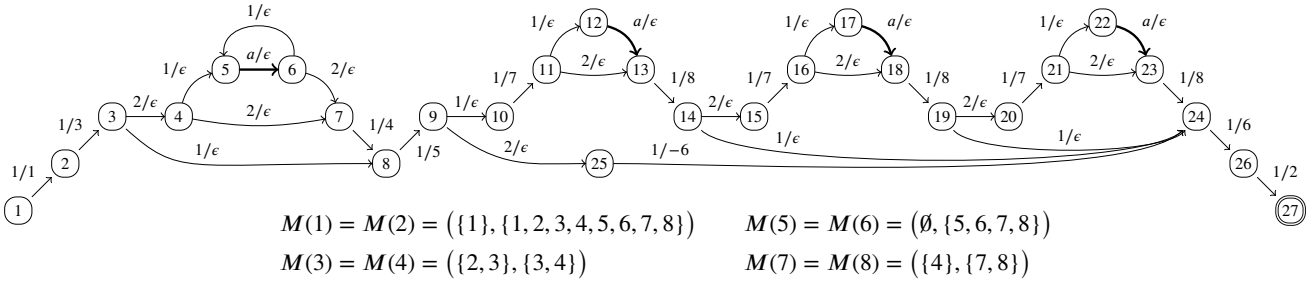


FIGURE 4 Example of TNFA for RE $(((\epsilon|a^{0,\infty}))(a|\epsilon)^{0,3})$.

The second step, TNFA construction, is given by the *tnfa*() function (algorithm 7). At this step counted repetition is unrolled, so TNFA may be much larger than IRE. For each subexpressions of the form $(i, J, e^{k,l})$ automaton for e is duplicated exactly l times if $l \neq \infty$, or $max(1, k)$ times otherwise (at each step of recursion the counter is decremented and one copy of automaton is constructed). Since repetition counters are constants, unrolling increases the size of sub-TNFA by a constant factor. Other *tnfa*() clauses are in one-to-one correspondence with sub-IRE. Each clause adds only a constant number of states and transitions (including calls to *ntags*() and excluding recursive calls). Therefore TNFA contains $O(m)$ states and transitions. The size of mapping M is $O(t)$, which is $O(m)$. Therefore total TNFA size is $O(m)$. Time complexity is $O(m)$, because each clause takes $O(1)$ time (excluding recursive calls), and total $O(m)$ clauses are executed.

The third step, TNFA simulation, is given by algorithm 1. Initialization at lines 2-4 takes $O(1)$ time. Main loop at lines 5-11 is executed at most n times. The size of closure is bounded by TNFA size, which is $O(m)$ (typically closure is much smaller than that). Each iteration of the loop includes the following steps. At line 6 the call to *closure*() takes $O(m^2\,t)$ time if GOR1 from section 4 is used, because GOR1 makes $O(m^2)$ scans (closure contains $O(m)$ states and $O(m)$ transitions), and at each scan we may need to compare the tag sequences using *compare*() from section 7, which takes $O(t)$ time (there is $O(t)$ unique tags and we consider paths with at most one ϵ-loop, so the number of occurrences of each tag is bounded by the repetition counters, which amounts to a constant factor). At line 7 the call to *update_result*() takes $O(m\,t)$ time, because closure size is $O(m)$, and the length of the tag sequence is $O(t)$ as argued above. At line 8 the call to *update_precedence*() takes $O(m^2)$ time if the advanced algorithm from section 7 is used. At line 9 scanning the closure for reachable states takes $O(m)$ time, because closure size is $O(m)$. The sum of the above steps is $O(m^2\,t)$, so the total time of loop at lines 5-11 is $O(n\,m^2\,t)$. The final call to *closure*() at line 12 takes $O(m^2\,t)$, and *final_result*() at line 14 takes $O(m\,t)$. The total time taken by *match*() is therefore $O(n\,m^2\,t)$.

Space complexity of *match*() is as follows. The size of the closure is $O(m)$. Precedence matrices H and P take $O(m^2)$ space. Match results take $O(m\,t)$ space in case of POSIX-style offsets, because the number of parallel results is bounded by the closure size, and each result takes $O(t)$ space. In case of parse trees, match results take $O(m\,n)$ space, because each result accumulates all loop iterations. The size of the U-tree is $O(m^2)$ because GOR1 makes $O(m^2)$ scans and thus adds no more than $O(m^2)$ tags in the tree. The total space taken by *match*() is therefore $O(m^2)$ for POSIX-style offsets and $O(m(m+n))$ for parse trees.

For leftmost-greedy submatch extraction, space complexity is $O(m\,t)$ and time complexity is $O(n\,m\,t)$, because the ϵ-closure is constructed with a DFS in $O(m)$ time and has $O(m)$ size, and each closure configuration contains $O(t)$ submatch results that need to be updated.

11 | BENCHMARKS

In this section n, m, t have the same meaning as in section 10. We used two groups of benchmarks:

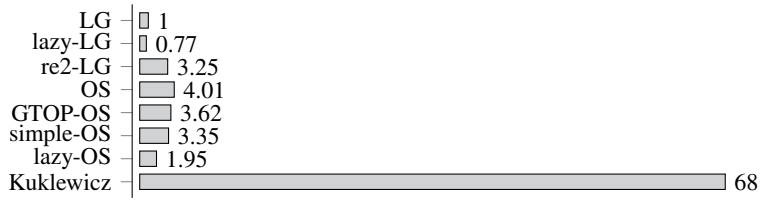
1. Real-world benchmarks (A1 – A12). These REs are designed to evaluate the algorithms in a natural setting. They describe practical concepts, such as URI or HTTP message headers, and range from very large (containing thousands of characters and hundreds of capturing groups) to small (containing under a hundred of characters and about five capturing groups). The input consists of short strings conforming to the RE grammar.

2. Artificial benchmarks (B1 – B12 and C1 – C12). These REs are designed to be highly ambiguous and reveal worst-case behavior and pathological cases for different algorithms. All REs in this group have an alphabet consisting of a single symbol `a`, and the input is a long string of `a`s (16K). There are two subgroups in this group:
   (a) REs with alternative. Benchmarks B1 – B6 contain REs of the form $(a^{k_1}|a^{k_2}|a^{k_3})^{0,\infty}$, where k_i are prime numbers. Benchmarks B7 – B12 contain their variations with more capturing groups: $(((a)^{k_1})|((a)^{k_2})|((a)^{k_3}))^{0,\infty}$. These REs show the difficulty of POSIX longest-match semantics, because the choice of alternatives varies indefinitely with the number of `a`s in the input string. For example, consider RE $(a^2|a^3|a^5)^{0,\infty}$: given an input string `a...a`, submatch on the last iteration depends on the length of input: it equals `aaaaa` for a $5n$-character string, `aa` for strings of length $5n-3$ and $5n-1$, and `aaa` for strings of length $5n-2$ and $5n+1$ ($n \in \mathbb{N}$). Variation continues indefinitely with a period of five characters. The variation period is longer for higher counter values.
   (b) REs with concatenation. These benchmarks contain REs of the form $(e^{0,k})^{0,\infty}$, where e is $(a|\epsilon)$ for C1 – C3, $(a^{0,\infty})$ for C4 – C6, a for C7 – C9 and (a) for C10 – C11. These REs have large ϵ-closure sizes.

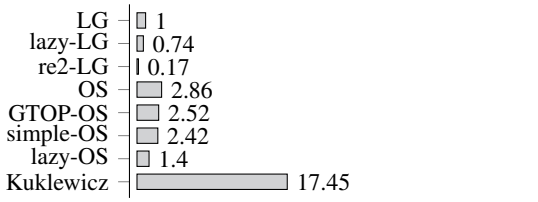
We benchmarked the following algorithms:

- LG: the basic leftmost-greedy implementation. It updates offsets at each step of TNFA simulation and runs in $O(n\,m\,t)$ time and $O(m\,t)$ space. This algorithm has no overhead on disambiguation and serves as a baseline for other algorithms.
- Lazy-LG: a "lazy" version of LG. It accumulates tag history of all simulation steps, updates offsets at the end of match rather than at each step, and runs in $O(n\,m)$ time and $O(n\,m)$ space. This algorithm reduces the overhead on incremental offset updates, but it requires memory proportional to the size of input.
- Re2-LG: a leftmost-greedy implementation by the Google RE2 library. This algorithm is used to relate our implementations to the real world: both RE2C and RE2 are written in C++, so they are comparable. Internally RE2 uses a bunch of different algorithms: simple cases are optimized, and the results vary considerably from one algorithm to another.
- OS: the main Okui-Suzuki implementation described in this paper. It uses GOR1 and the advanced *update_precedence*() algorithm. Disambiguation and offset updates are performed eagerly at each step of TNFA simulation and the algorithm runs in $O(n\,m^2\,t)$ time and $O(m^2)$ space.
- GTOP-OS: the same as OS, but uses GTOP instead of GOR1.
- Simple-OS: the same as OS, but uses the simple *update_precedence*() algorithm.
- Lazy-OS: the "lazy" version of OS described in section 8. It accumulates tag history of all simulation steps and disambiguates on demand, memoizing the results in cache. Offsets are updated at the end of match. The algorithm has the same $O(n\,m^2\,t)$ time complexity as OS (because the ϵ-closure construction takes $O(m^2\,t)$ time), and its space complexity is $O(n\,m^2)$ due to the need to store $O(m^2)$ tag path tree for each of the n steps. This algorithm reduces the overhead on incremental offset updates and precedence matrix computation, but it requires memory proportional to the size of input.
- Kuklewicz: the implementation of Kuklewicz algorithm[5] as described in[11]. The main difference with OS is that the algorithm splits the common tag history into individual tag histories and considers them separately. The algorithm runs in $O(n\,m\,t\,(m + t\,log(m))$ time and $O(m\,t)$ space.
- Backward: the backward-matching algorithm proposed by Cox[6]. We had to modify it substantially in order to support bounded repetition and fix bugs (described in the introduction). We also use a fast forward pass to find the matching string prefix before running the main backward pass. Naturally, these modifications incur some overhead. The resulting algorithm is not fully correct, but the errors are rare compared to the original algorithm (about 2.5% of our tests), and the test cases that trigger errors are more complex.

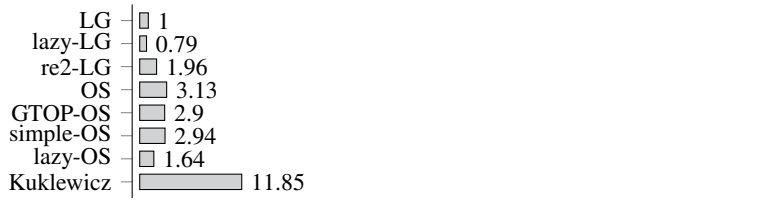
All algorithms except re2-LG are implemented in the open-source lexer generator RE2C[20] in the form of a regular expression library `libre2c`, which provides the usual POSIX API of `regcomp`, `regexec` and `regfree`. Individual algorithms are selected by passing non-standard flags to `regcomp`: for example, there is a flag that enables leftmost-greedy semantics instead of POSIX semantics, a flag that enables GTOP instead of GOR1, a flag that enables "lazy" mode, etc. Most of the flags can be combined. We selected a limited number of combinations for the benchmarks in order to show the impact of each individual feature. Correctness of our implementation has been tested on a subset of Glenn Fowler test suite[19] (we removed tests for backreferences and start/end anchors), extended by Kuklewicz and further extended by ourselves to some 500 tests.



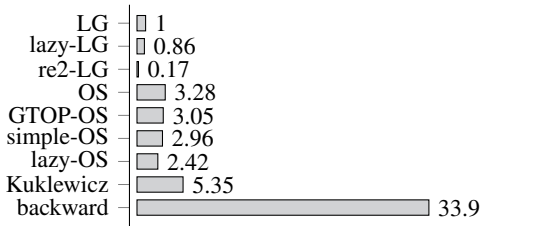
A1 RFC-7230 compliant HTTP parser (6204 symbols, 185 captures)



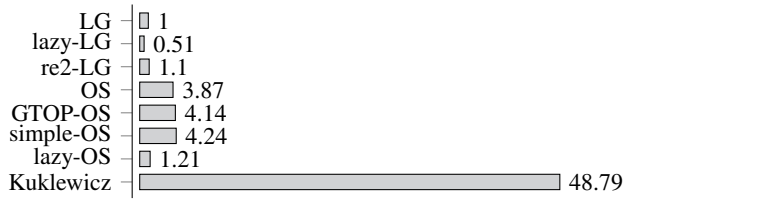
A2 simple HTTP parser (573 symbols, 33 captures)



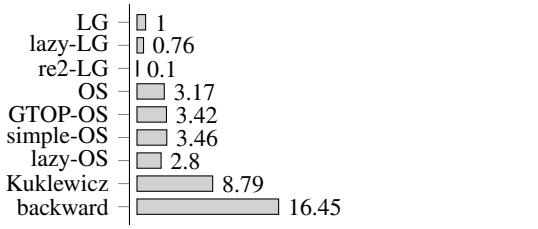
A3 RFC-3986 compliant URI parser (3149 symbols, 93 captures)



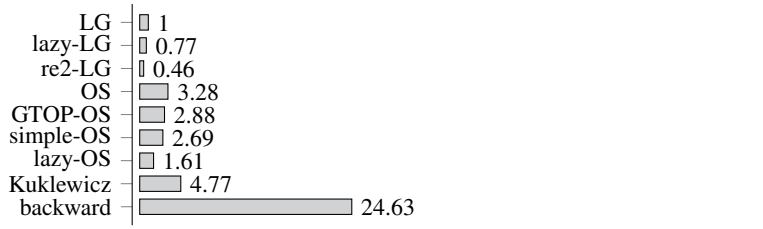
A4 simple URI parser (234 symbols, 14 captures)



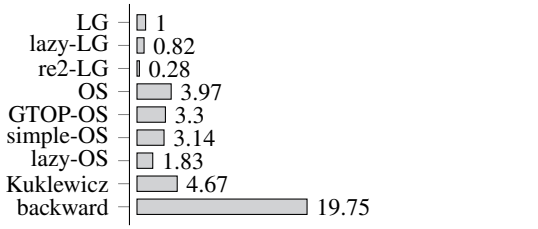
A5 RFC-7230 compliant IPv6 parser (2343 symbols, 62 captures)



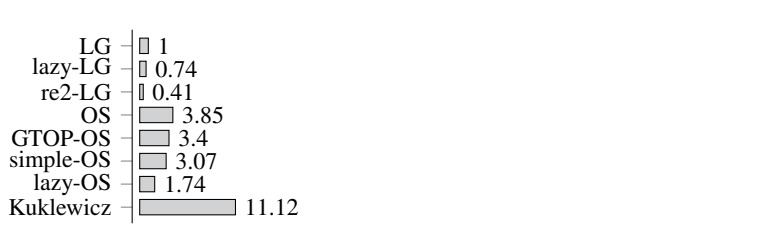
A6 simple IPv6 parser (93 symbols, 7 captures)



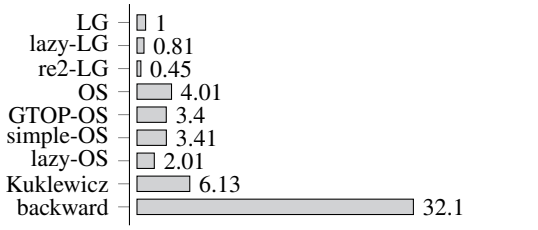
A7 IPv4 parser (235 symbols, 6 captures)



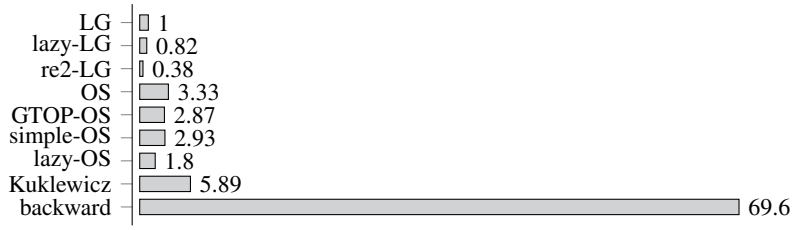
A8 simple IPv4 parser (57 symbols, 5 captures)



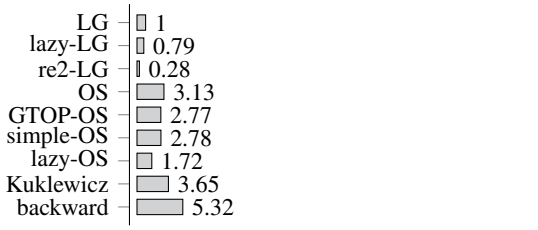
A9 date parser (224 symbols, 14 captures)



A10 simple date parser (81 symbols, 7 captures)



A11 Gentoo package atom (117 symbols, 14 captures)



A12 simple package atom (45 symbols, 5 captures)

FIGURE 5 Real-world benchmarks (A1 – A12).

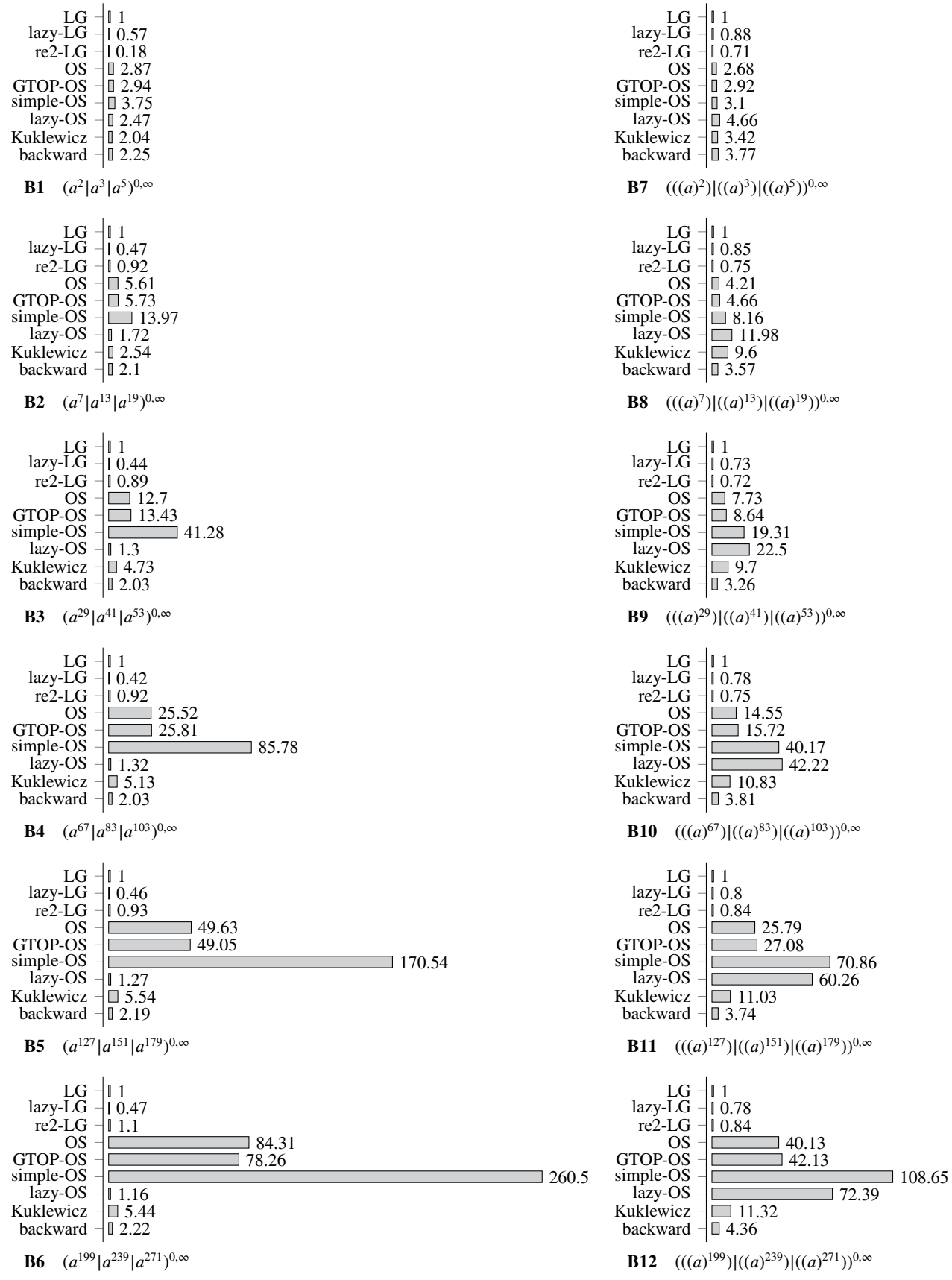


FIGURE 6 Artificial benchmarks (B1 — B12).
Highly ambiguous REs with alternative on 16K string of as.

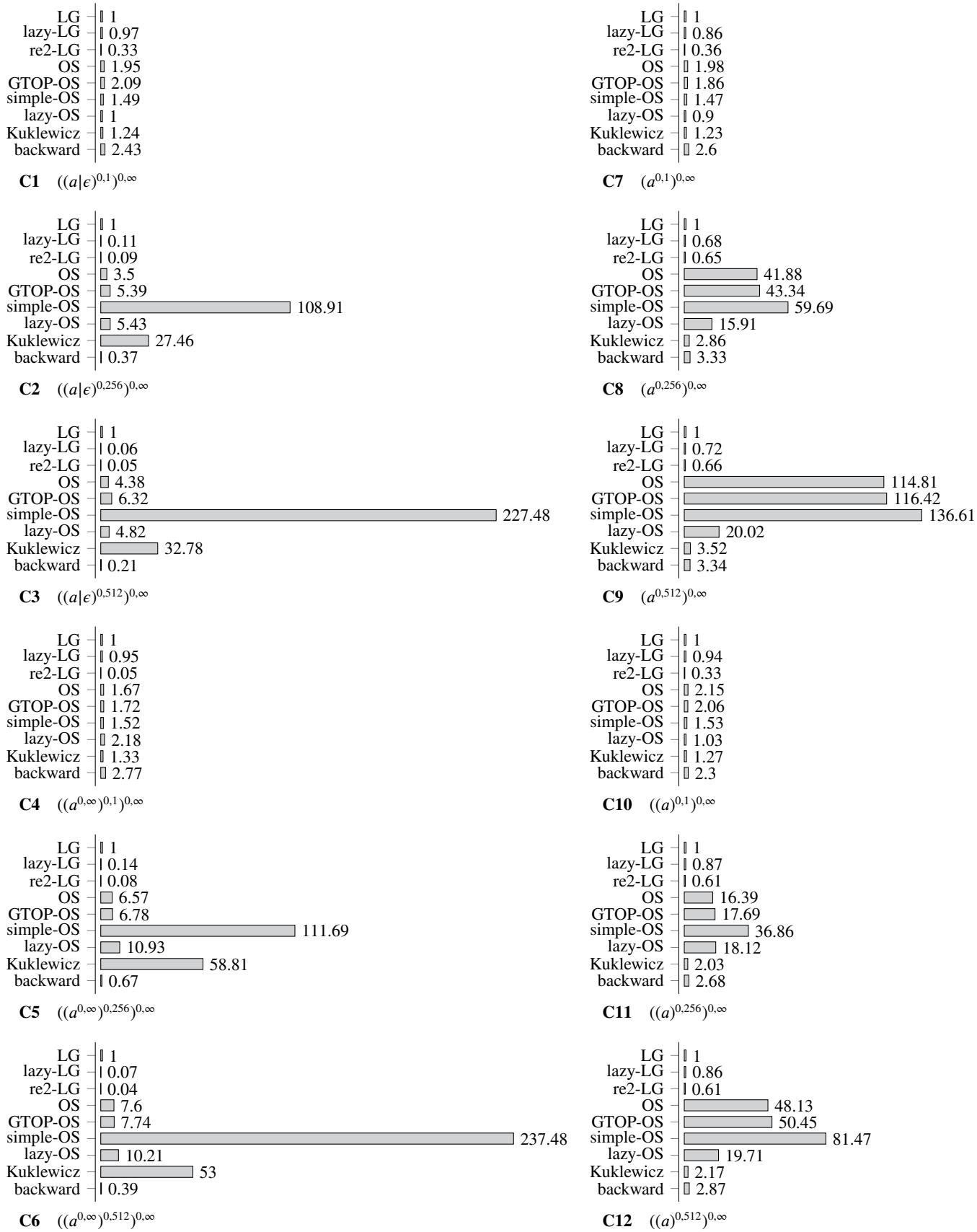


FIGURE 7 Artificial benchmarks (C1 — C12).
Highly ambiguous REs with concatenation on 16K string of `a`s.

In order to present benchmark results in a meaningful way, we show the time of each algorithm relative to LG. This allows us to show the net overhead on POSIX disambiguation. We measured the speed of each algorithm in characters per second and divided it by the speed of LG (therefore LG time is always equal to 1). We ran benchmarks on Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz machine with 12 cores (we ran in single-threaded mode), 32 GB of RAM, 32K L1d cache, 32K L1i cache, 256K L2 cache, 9216K L3 cache, running Gentoo Linux. We used RE2 version 2019-12-01 and RE2C from Git[20] at commit `662b45ef468713f59e667200aacd922f29580c4a`.

Memory consumption. We used Massif[23] to estimate memory usage. The algorithms can be partitioned in two groups: those that execute in bounded memory (depending on the size of RE, but not on the length of input), and those that require memory proportional to the length of input. The first group contains LG, re2-LG, OS, GTOP-OS, simple-OS, Kuklewicz and backward: these algorithms stay under 50M for all benchmarks, and usually the memory is pre-allocated at the beginning (e.g. for precedence matrices). The second group contains lazy-LG and lazy-OS: memory consumption of these algorithms grows with the length of input and reaches peak values for C6 (over 700M for lazy-LG and over 4G for lazy-OS). Both algorithms keep closure trees for all steps in memory, but the size of the trees is linear in the size of RE for lazy-LG (which uses a simple DFS), and worst-case quadratic for lazy-OS (which uses GOR1). Lazy-OS also allocates a large disambiguation cache. Our cache implementation is a red-black tree from the standard C++ library, which incurs a lot of small heap allocations and deallocations and is not optimal. However, it suffices to show the general problem.

Run-time performance. On real-world RE all algorithms except Kuklewicz and backward perform quite well. Lazy-OS is the fastest among POSIX implementations, and only ~2x slower than LG. Other OS algorithms (OS, GTOP-OS and simple-OS) are 3-4x slower than LG, with OS being generally a bit slower due to a more complex implementation. Kuklewicz algorithm is noticeably slower on large RE, and the backward algorithm is so slow that it timed out on some of the benchmarks. On artificial benchmarks the results are quite different. Of all POSIX implementations, only the backward algorithm performs reasonably well. Other algorithms show performance degradation on some of the benchmarks, and simple-OS is particularly slow. In general, we can observe the following tendencies:

- OS degrades with increased closure size, as shown by B1 – B6, B7 – B12, C7 – C9 and C10 – C12. This is caused by an $O(m^2)$ precedence matrix update algorithm.
- Simple-OS degrades with increased closure size, and much faster than OS, as shown by C1 – C3 and C4 – C6. (it is ~50x slower than OS on C3). This is caused by an $O(m^2\,t)$ precedence matrix update algorithm.
- Kuklewicz algorithm degrades with increased closure size, although not as fast as OS (as shown by B7 – B12, C1 – C3 and C4 – C6) and with increased number of tags (compare B1 – B6 with fewer captures to B7 – B12 with more captures), in particular orbit tags (compare C1 – C3 with an innermost alternative to C4 – C6 with an innermost repetition). On REs with many capturing groups Kuklewicz algorithm can get very slow (A1, A5). This is caused by an $O(m\,log(m)\,t^2)$ precedence matrix update algorithm.
- Backward algorithm degrades with increased number of tags (as shown by A1 – A12, and also observable on B7 – B12 compared to B1 – B6). This is caused by the necessity to copy offset arrays between closure configurations: instead of using a prefix tree to store tag histories during closure construction, the algorithm stores an array of offsets (a pair of offsets per tag) in each configuration. GOR1 is asymptotically faster than DFS with backtracking used by the original implementation[6], but it increases the amount of copying (DFS order of graph traversal allows to update and restore a single offset array in-place). Even with DFS copying is a bottleneck for REs with many capturing groups.
- Lazy-OS degrades with increased input length, increased closure size and ambiguity in RE (it performs well on B1 – B6, but much worse on B7 – B12 which differ only in capturing parentheses). This is caused by the need to keep all closure trees in memory and allocate a large disambiguation cache. In highly ambiguous cases the lazy algorithm does all the work of a non-lazy algorithm, but with the additional overhead on cache lookups/insertions and accumulation of data from the previous steps.
- GOR1, despite having $O(m^2\,t)$ worst-case complexity, in practice performs closer to $O(m\,t)$ and adds less overhead than updating the precedence matrix (compare B1 – B6 for OS and lazy-OS, which also uses GOR1, but no precedence matrix).
- GOR1 and GTOP performance is similar. GTOP is usually slightly faster on real-world RE (probably because the closure graphs are simple and GOR1 code is more complex). We failed to find a pathological case for GTOP; this may mean that we didn't look hard enough, or that TNFA closures have some special properties that make GTOP efficient.

- LG and lazy-LG performance is close to re2-LG, faster on large RE, but slower on small RE. This is explained by the fact that RE2 uses an optimized bitmap-based implementation for small RE.

12 | CONCLUSIONS AND FUTURE WORK

As the benchmarks show, none of the POSIX matching algorithms we considered is ideal, and for each algorithm there are use cases where it performs better than other algorithms. On the whole, though, we can make the following observation: lazy-OS is generally the fastest algorithm, and OS is generally the most robust one (it has the best worst-case time and space complexity). Lazy-OS can only be used for short inputs due to its unbounded memory consumption, and OS can be used for long inputs, as it executes in bounded memory. It seems that the best algorithm would use a combination of lazy-OS and OS. Note that using different algorithms for different use cases is what existing libraries like RE2 already do.

It is still an open question to us whether it is possible to combine the elegance of the derivative-based approach to POSIX disambiguation with the practical efficiency of NFA-based methods. The derivative-based approach constructs the match results in such order that the longest-leftmost result is always first. We experimented with recursive descent parsers that embrace the same ordering idea and constructed a prototype implementation.

It would be interesting to apply our approach to automata with counters instead of unrolling bounded repetition.

ACKNOWLEDGMENTS

Whatever I understand in mathematics I owe to my parents Vladimir and Elina, my teachers Tatyana Leonidovna and Demian Vladimirovich, and the Belarusian State University. This work would not exist without the inspiration and help of my husband Sergei. And finally, cheers to the RE2C users! *Ulya Trofimovich*

References

1. Ken Thompson, *Programming Techniques: Regular expression search algorithm*, Communications of the ACM, vol. 11 (6), 1968.
2. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, *Compilers: Principles, Techniques, & Tools* (chapter 3.7.3: *Efficiency of NFA Simulation*), 2nd edition, Boston, MA, USA: Pearson Addison-Wesley, ISBN 0-321-48681-1, pp. 157-159.
3. Satoshi Okui and Taro Suzuki, *Disambiguation in Regular Expression Matching via Position Automata with Augmented Transitions*, International Conference on Implementation and Application of Automata, pp. 231-240, Springer, Berlin, Heidelberg, 2013.
4. Ville Laurikari, *Efficient submatch addressing for regular expressions*, Helsinki University of Technology, 2001.
5. Chris Kuklewicz, *Regular expressions/bounded space proposal*, http://wiki.haskell.org/index.php?title=Regular_expressions/Bounded_space_proposal&oldid=11475 2007.
6. Russ Cox, backward POSIX matching algorithm (source code), https://swtch.com/~rsc/regexp/nfa-posix.y.txt 2009.
7. Martin Sulzmann, Kenny Zhuo Ming Lu, *POSIX Regular Expression Parsing with Derivatives*, International Symposium on Functional and Logic Programming, pp. 203-220, Springer, Cham, 2014.
8. Martin Sulzmann, Kenny Zhuo Ming Lu, *Correct and Efficient POSIX Submatch Extraction with Regular Expression Derivatives*, https://www.home.hs-karlsruhe.de/~suma0002/publications/posix-derivatives.pdf, 2013.
9. Angelo Borsotti1, Luca Breveglieri, Stefano Crespi Reghizzi, Angelo Morzenti, *From Ambiguous Regular Expressions to Deterministic Parsing Automata*, International Conference on Implementation and Application of Automata. Springer, Cham, pp.35-48, 2015.

10. Fahad Ausaf, Roy Dyckhoff, Christian Urban, *POSIX Lexing with Derivatives of Regular Expressions*, International Conference on Interactive Theorem Proving. Springer, Cham, pp. 69-86, 2016.

11. Ulya Trofimovich, *Tagged Deterministic Finite Automata with Lookahead*, arXiv:1907.08837 [cs.FL], 2017.

12. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, *Introduction to algorithms*, 1st edition, MIT Press and McGraw-Hill, ISBN 0-262-03141-8.

13. Mehryar Mohri, *Semiring frameworks and algorithms for shortest-distance problems*, Journal of Automata, Languages and Combinatorics 7 (2002) 3, 321–350, Otto-von-Guericke-Universitat, Magdeburg, 2002.

14. Aaron Karper, *Efficient regular expressions that produce parse trees*, Master's thesis, University of Bern, 2014.

15. Andrew V. Goldberg, Tomasz Radzik, *A heuristic improvement of the Bellman-Ford algorithm*, Elsevier, Applied Mathematics Letters, vol. 6, no. 3, pp. 3-6, 1993.

16. Boris V. Cherkassky, Andrew V. Goldberg, Tomasz Radzik, *Shortest paths algorithms: Theory and experimental evaluation*, Springer, Mathematical programming, vol. 73, no. 2, pp. 129-174, 1996.

17. Boris V. Cherkassky, Loukas Georgiadis, Andrew V. Goldberg, Robert E. Tarjan, and Renato F. Werneck. *Shortest Path Feasibility Algorithms: An Experimental Evaluation*, Journal of Experimental Algorithmics (JEA), 14, 7, 2009.

18. POSIX standard: *POSIX-1.2008* a.k.a. *IEEE Std 1003.1-2008* The IEEE and The Open Group, 2008.

19. Glenn Fowler, *An Interpretation of the POSIX Regex Standard*, https://web.archive.org/web/20050408073627/http://www.research.att.com/~gsf/testregex/re-interpretation.html, 2003.

20. *RE2C*, lexer generator for C/C++. Website: http://re2c.org, source code: http://github.com/skvadrik/re2c.

21. *RE2*, regular expression library. Source code: http://github.com/google/re2.

22. *The GNU C library*, https://www.gnu.org/software/libc/.

23. *Massif, a heap profiler*, http://www.valgrind.org/docs/manual/ms-manual.html.

APPENDIX

Proof of theorem 1

Lemma 5 (Unique position mapping from all PTs to IRE)**.** If $t, s \in PT(e)$ for some IRE e and there is a common position $x \in Pos(t) \cap Pos(s)$, then x corresponds to the same position $x' \in Pos(e)$ in e for both t and s.

Proof. The proof is by induction on the length of x. Induction basis: $x = x' = \Lambda$ (the roots of t and s correspond to the root of e). Induction step: suppose that for any position x of length $|x| < h$ the lemma is true. We will show that if exists a $k \in \mathbb{N}$ such that $x.k \in Pos(t) \cap Pos(s)$, then $x.k$ corresponds to the same position $x'.k'$ in e for both t and s (for some $k' \in \mathbb{N}$). If $e|_{x'}$ is an elementary IRE of the form (i, j, ϵ) or (i, j, a), or if at least one of $t|_x$ and $s|_x$ is $\perp$, then k doesn't exist. Otherwise $e|_{x'}$ is a compound IRE and both $t|_x$ and $s|_x$ are not $\perp$. If $e|_{x'}$ is a union $(i, j, (i_1, j_1, e_1)|(i_2, j_2, e_2))$ or a product $(i, j, (i_1, j_1, e_1)\cdot(i_2, j_2, e_2))$, then both $t|_x$ and $s|_x$ have exactly two subtrees, and positions $x.1$ and $x.2$ in t and s correspond to positions $x'.1$ and $x'.2$ in e. Otherwise, $e|_{x'}$ is a repetition $(i, j, e_1^{n,m})$ and for any $k \geq 1$ position $x.k$ in t and s corresponds to position $x'.1$ in e. □

Theorem 1. P-order $<$ is a strict total order on $PT(e, w)$ for any IRE e and string w.

Proof. We need to show that $<$ is transitive and trichotomous.

(1) Transitivity: we need to show that $\forall t, s, r \in PT(e, w) : (t < s \wedge s < r) \implies t < r$.

Let $t <_x s$ and $s <_y r$ for some positions x, y, and let $z = min(x, y)$.

First, we show that $\|t\|_z^{pos} > \|r\|_z^{pos}$. If $x \leq y$, we have $\|t\|_x^{pos} > \|s\|_x^{pos}$ (implied by $t <_x s$) and $\|s\|_x^{pos} \geq \|r\|_x^{pos}$ (implied by $s <_y r \wedge x \leq y$), therefore $\|t\|_x^{pos} > \|r\|_x^{pos}$. Otherwise $x > y$, we have $\|t\|_y^{pos} > \|r\|_y^{pos}$ (implied by $s <_y r$) and $\|t\|_y^{pos} = \|s\|_y^{pos}$ (implied by $t <_x s \wedge y < x$), therefore $\|t\|_y^{pos} > \|r\|_y^{pos}$.

Second, we show that $\forall z' < z : \|t\|_{z'}^{pos} = \|r\|_{z'}^{pos}$. We have $\|t\|_{z'}^{pos} = \|s\|_{z'}^{pos}$ (implied by $t <_x s \wedge z' < x$) and $\|s\|_{z'}^{pos} = \|r\|_{z'}^{pos}$ (implied by $s <_y r \wedge z' < y$), therefore $\|t\|_{z'}^{pos} = \|r\|_{z'}^{pos}$.

(2) Trichotomy: we need to show that $\forall t, s \in PT(e, w)$ exactly one of $t < s$, $s < t$ or $t = s$ holds. Consider the set of positions where norms of t and s disagree $X = \{x \in Pos(t) \cup Pos(s) : \|t\|_x^{pos} \neq \|s\|_x^{pos}\}$.

(2.1) First case: $X \neq \emptyset$. We show that in this case exactly one of $t < s$ or $s < t$ is true ($t \neq s$ is obvious).

First, we show that at least one of $t < s$ or $s < t$ is true. Let $x = min(X)$; it is well-defined since X is non-empty, finite and lexicographically ordered. For all $y < x$ we have $\|t\|_y^{pos} = \|s\|_y^{pos}$ (by definition of x and because $\|t\|_y^{pos} = \infty = \|s\|_y^{pos}$ if $y \notin Pos(t) \cup Pos(s)$). Since $\|t\|_x^{pos} \neq \|s\|_x^{pos}$, we have either $t <_x s$ or $t <_x s$.

Second, we show that at most one of $t < s$ or $s < t$ is true, i.e. $<$ is asymmetric: $\forall t, s \in PT(e, w) : t < s \implies s \not< t$. Suppose, on the contrary, that $t <_x s$ and $s <_y t$ for some x, y. Without loss of generality let $x \leq y$. On one hand $t <_x s$ implies $\|t\|_x^{pos} > \|s\|_x^{pos}$. But on the other hand $s <_y t \wedge x \leq y$ implies $\|t\|_x^{pos} \leq \|s\|_x^{pos}$. Contradiction.

(2.2) Second case: $X = \emptyset$. We show that in this case $t = s$.

We have $Pos(t) = Pos(s)$ — otherwise there is a position with norm ∞ in only one of the trees. Therefore t and s have identical node structure. By lemma 5 any position in t and s corresponds to the same position in e. Since any position in e corresponds to a unique explicit submatch index, it must be that submatch indices of all nodes in t and s coincide. Consider some position $x \in Pos(t)$. If x corresponds to an inner node, then both $t|_x$ and $s|_x$ are of the form $T^i(\ldots)$. Otherwise, x corresponds to a leaf node, which can be either $\perp$ or ϵ or α. Since all three have different norms (-1, 0 and 1 respectively), and since $\|t\|_x^{pos} = \|s\|_x^{pos}$, it must be that $t|_x$ and $s|_x$ are identical.

□

Proof of theorem 2

Lemma 6. If $t, s \in PT(e, w)$ for some IRE e and string w, then $t \sim s \Leftrightarrow \forall x : \|t\|_x^{sub} = \|s\|_x^{sub}$.

Proof. Forward implication: let $t \sim s$ and suppose, on the contrary, that $\exists x = min\{y \mid \|t\|_y^{sub} \neq \|s\|_y^{sub}\}$, then either $t \prec_x s$ (if $\|t\|_x^{sub} > \|s\|_x^{sub}$) or $s \prec_x t$ (if $\|t\|_x^{sub} < \|s\|_x^{sub}$), both cases contradict $t \sim s$. Backward implication: $\forall x : \|t\|_x^{sub} = \|s\|_x^{sub}$ implies $\nexists x : t \prec_x s$ and $\nexists y : s \prec_y t$, which implies $t \sim s$. □

Theorem 2. S-order $\prec$ is a strict weak order on $PT(e, w)$ for any IRE e and string w.

Proof. We need to show that $\prec$ is asymmetric and transitive, and incomparability relation $\sim$ is transitive.

(1) Asymmetry: we need to show that $\forall t, s \in PT(e, w) : t \prec s \implies s \not\prec t$.

Suppose, on the contrary, that $t \prec_x s$ and $s \prec_y t$ for some x, y. Without loss of generality let $x \leq y$. On one hand $t \prec_x s$ implies $\|t\|_x^{sub} > \|s\|_x^{sub}$. But on the other hand $s \prec_y t \wedge x \leq y$ implies $\|t\|_x^{sub} \leq \|s\|_x^{sub}$. Contradiction.

(2) Transitivity: we need to show that $\forall t, s, r \in PT(e, w) : (t \prec s \wedge s \prec r) \implies t \prec r$.

Let $t \prec_x s$ and $s \prec_y r$ for some positions x, y, and let $z = min(x, y)$.

First, we show that $\|t\|_z^{sub} > \|r\|_z^{sub}$. If $x \leq y$, we have $\|t\|_x^{sub} > \|s\|_x^{sub}$ (implied by $t \prec_x s$) and $\|s\|_x^{sub} \geq \|r\|_x^{sub}$ (implied by $s \prec_y r \wedge x \leq y$), therefore $\|t\|_x^{sub} > \|r\|_x^{sub}$. Otherwise $x > y$, we have $\|t\|_y^{sub} > \|r\|_y^{sub}$ (implied by $s \prec_y r$) and $\|t\|_y^{sub} = \|s\|_y^{sub}$ (implied by $t \prec_x s \wedge y < x$), therefore $\|t\|_y^{sub} > \|r\|_y^{sub}$.

Second, we show that $\forall z' < z : \|t\|_{z'}^{sub} = \|r\|_{z'}^{sub}$. We have $\|t\|_{z'}^{sub} = \|s\|_{z'}^{sub}$ (implied by $t \prec_x s \wedge z' < x$) and $\|s\|_{z'}^{sub} = \|r\|_{z'}^{sub}$ (implied by $s \prec_y r \wedge z' < y$), therefore $\|t\|_{z'}^{sub} = \|r\|_{z'}^{sub}$.

(3) Transitivity of incomparability: we need to show that $\forall t, s \in PT(e, w) : (t \sim s \wedge s \sim r) \implies t \sim r$.

By forward implication of lemma 6 $t \sim s \Rightarrow \forall x : \|t\|_x^{sub} = \|s\|_x^{sub}$ and $s \sim r \Rightarrow \forall x : \|s\|_x^{sub} = \|r\|_x^{sub}$, therefore $(t \sim s \wedge s \sim r) \Rightarrow \forall x : \|t\|_x^{sub} = \|r\|_x^{sub} \Rightarrow t \sim u$ by backward implication of lemma 6.

□

Proof of Theorem 3

Lemma 7 (Comparability of subtrees)**.** For a given IRE e, string w and position x, if $t, s \in PT(e, w)$, $x \in Sub(t) \cup Sub(s)$ and $\|t\|_y^{sub} = \|s\|_y^{sub}\ \forall y \leq x$, then $\exists e', w' : t|_x, s|_x \in PT(e', w')$.

Proof. By induction on the length of x. Induction basis: $x = \Lambda$, let $e' = e$ and $w' = w$. Induction step: suppose that the lemma is true for any position x of length $|x| < h$, we will show that it is true for any position $x.k$ of length h ($k \in \mathbb{N}$). Assume that $x.k \in Sub(t) \cap Sub(s)$ (otherwise either $x.k \notin Sub(t) \cup Sub(s)$, or exactly one of $\|t\|_{x.k}^{sub}$, $\|s\|_{x.k}^{sub}$ is ∞ — in both cases lemma conditions are not satisfied). Then both $t|_x$ and $s|_x$ have at least one subtree: let $t|_x = T(t_1, \dots, t_n)$ and $s|_x = T(s_1, \dots, s_m)$, where $n, m \geq k$. By induction hypothesis $\exists e', w' : t|_x, s|_x \in PT(e', w')$. We have $w' = str(t_1) \dots str(t_n) = str(s_1) \dots str(s_m)$. We show that $str(t_k) = str(s_k)$. Consider positions $x.j$ for $j \leq k$. By definition the set of submatch positions contains siblings, therefore $x.j \in Sub(t) \cap Sub(s)$. By lemma conditions $\|t\|_{x.j}^{sub} = \|s\|_{x.j}^{sub}$ (because $x.j \leq x.k$), therefore $|str(t_1) \dots str(t_{k-1})| = \sum_{j=1}^{k-1} \|t\|_j^{sub} = \sum_{j=1}^{k-1} \|s\|_j^{sub} = |str(s_1) \dots str(s_{k-1})|$ and $|str(t_k)| = |str(s_k)|$. Consequently, $str(t_k)$ and $str(s_k)$ start and end at the same character in w' and therefore are equal. Finally, have $t|_{x.k}, s|_{x.k} \in PT(r|_{x.k}, str(t_k))$ and induction step is complete. □

Theorem 3. Let t_{min} be the $<$-minimal tree in $PT(e, w)$ for some IRE e and string w, and let T_{min} be the class of $\prec$-minimal trees in $PT(e, w)$. Then $t_{min} \in T_{min}$.

Proof. Consider any $t \in T_{min}$. From t we can construct another tree t' in the following way. Consider all positions $x \in Sub(t)$ which are not proper prefixes of another position in $Sub(t)$. For each such position, $t|_x$ is itself a PT for some sub-IRE e' and substring $w' : t|_x \in PT(e', w')$. Let t'_{min} be the $<$-minimal tree in $PT(e', w')$ and substitute $t|_x$ with t'_{min}. Let t' be the tree resulting from all such substitutions (note that they are independent of the order in which we consider positions x). Since substitutions preserve s-norm at submatch positions, we have $t' \in T_{min}$. We will show that $t' = t_{min}$.

Suppose, on the contrary, that $t' \neq t_{min}$. Since t_{min} is $<$-minimal, it must be that $t_{min} <_x t'$ for some decision position x. By definition it means that $\|t_{min}\|_x^{pos} > \|t'\|_x^{pos}$ and $\|t_{min}\|_y^{pos} = \|t'\|_y^{pos}\ \forall y < x$. It follows that all positions $y \leq x$ that are in $Pos(t_{min})$ are also in $Pos(t')$, otherwise we get a contradiction $\|t'\|_y^{pos} = \infty > \|t_{min}\|_y^{pos}$. By lemma 5 position y corresponds to the same position in the IRE e for both t_{min} and t'. Consequently, any $y \leq x$ is either a submatch position in both t_{min} and t', or in neither of them. Let x' be the longest prefix of x in $Sub(t')$. Since $x' \leq x$, by the above reasoning x' is also in $Sub(t_{min})$. All prefixes of a submatch position are also submatch positions, therefore for $y \leq x'$ s-norms coincide with p-norms: $\|t_{min}\|_y^{sub} = \|t_{min}\|_y^{pos}$ and $\|t'\|_y^{sub} = \|t'\|_y^{pos}$. It follows that $x' \neq x$, otherwise $t_{min} <_x t'$ implies $t_{min} \prec_x t'$, which contradicts to $t' \in T_{min}$. Therefore $x' < x$,

and since $\|t_{min}\|_y^{pos} = \|t'\|_y^{pos}\ \forall y < x$ and s-norms coincide with p-norms, we have $\|t_{min}\|_y^{sub} = \|t'\|_y^{sub}\ \forall y \le x'$. Then by lemma 7 subtrees $t'_{x'}$ and $t_{min}|_{x'}$ are comparable: $\exists e', w' : t'|_{x'}, t_{min}|_{x'} \in PT(e', w')$. By construction of t', subtree $t'_{x'}$ is $<$-minimal in $PT(e', w')$, but at the same time $t_{min} <_{x'.x''} t'$ implies $t_{min}|_{x'} <_{x''} t'|_{x'}$. Contradiction. □

Theorem 4. Let e, e' be two REs, such that e' is constructed from e by adding a pair of parentheses around some subexpression, and let $\bar{e} = IRE(e)$ and $\bar{e}' = IRE(e')$. If T_{min}, T'_{min} are the classes of $\prec$-minimal trees in $PT(\bar{e}, w)$ and $PT(\bar{e}', w)$ respectively for some string w, then $T'_{min} \subseteq T_{min}$.

Proof. First, note that adding parentheses in RE does not affect the set of its positions: we have $Pos(\bar{e}) = Pos(\bar{e}')$. Consequently, p-order on $PT(\bar{e}, w)$ and $PT(\bar{e}', w)$ is identical, and the $<$-minimal tree t_{min} is the same in both cases. By theorem 3 we have $t_{min} \in T_{min}$ and $t_{min} \in T'_{min}$, therefore the classes of $\prec$-minimal trees have at least one common tree: $t_{min} \in T_{min} \cap T'_{min}$

Second, note that adding parentheses in RE may only increase the set of its submatch positions: we have $Sub(\bar{e}) \subseteq Sub(\bar{e}')$. Let S, S' denote $Sub(t_{min})$ for $\bar{e}$ and $\bar{e}'$ respectively, then we have $S \subseteq S'$.

Consider any tree $t \in T'_{min}$. It is incomparable with t_{min} in the s-order on $PT(\bar{e}', w)$, therefore by lemma 6 (forward implication) all s-norms at positions in S' in t and t_{min} are equal: $\|t\|_x^{sub} = \|t_{min}\|_x^{sub}\ \forall x \in S'$. Since $S \subseteq S'$, it follows that $\|t\|_x^{sub} = \|t_{min}\|_x^{sub}\ \forall x \in S$. Therefore by lemma 6 (backward implication) t is incomparable with t_{min} in the s-order on $PT(\bar{e}, w)$, and consequently $t \in T_{min}$. Since t is an arbitrary tree in T'_{min}, it follows that $T'_{min} \subseteq T_{min}$. □

Proof of Theorem 5

Lemma 8. Let $s, t \in PT(e, w)$ for some IRE e and string w. If $s \sim t$, then $\Phi_h(s) = \Phi_h(t)\ \forall h$.

Proof. By induction on the height of e. Induction basis: for height 1 we have $|PT(e, w)| \le 1\ \forall w$, therefore $s = t$ and $\Phi_h(s) = \Phi_h(t)$. Induction step: height is greater than 1, therefore $s = T^o(s_1, \dots, s_n)$ and $t = T^o(t_1, \dots, t_m)$. If $o = 0$, then $\Phi_h(s) = str(s) = w = str(t) = \Phi_h(t)$. Otherwise $o \neq 0$. By lemma 6 we have $s \sim t \Rightarrow \|s\|_x^{sub} = \|t\|_x^{sub}\ \forall x$. This implies $n = m$ (otherwise the norm of subtree at position $min(n, m) + 1$ is ∞ for only one of s, t). Therefore $\Phi_h(s) = \langle_{h+1}\Phi_{h+1}(s_1), \dots, \Phi_{h+1}(s_n)\rangle_h$ and $\Phi_h(t) = \langle_{h+1}\Phi_{h+1}(t_1), \dots, \Phi_{h+1}(t_n)\rangle_h$. It suffices to show that $\forall i \le n : \Phi_{h+1}(s_i) = \Phi_{h+1}(t_i)$. We have $\|s_i\|_x^{sub} = \|t_i\|_x^{sub}\ \forall x$ (implied by $\|s\|_x^{sub} = \|t\|_x^{sub}\ \forall x$), therefore by lemma 6 $s_i \sim t_i$, and by lemma 7 $\exists e', w' : s_i, t_i \in PT(e', w')$, where the height of e' is less than the height of e. By induction hypothesis $\Phi_{h+1}(s_i) = \Phi_{h+1}(t_i)$. □

Lemma 9. Let $s, t \in PT(e, w)$ for some IRE e and string w. If $s \prec_x t$ and $|x| = 1$, then $\Phi_h(s) < \Phi_h(t)\ \forall h$.

Proof. By lemma conditions $|x| = 1$, therefore $x \in \mathbb{N}$. At least one of $s|_x$ and $t|_x$ must exist (otherwise $\|s\|_x^{sub} = \infty = \|t\|_x^{sub}$ which contradicts $s \prec_x t$), therefore e is a compound IRE and s, t can be represented as $s = T^o(s_1, \dots, s_n)$ and $t = T^o(t_1, \dots, t_m)$ where $o \neq 0$ because Λ is a prefix of decision position x. Let k be the number of frames and let j be the fork, then:

$$\begin{array}{rcl|l} \Phi_h(s) & = & \langle_{h+1}\Phi_{h+1}(s_1)\dots\Phi_{h+1}(s_n)\rangle_h & = \beta_0 a_1 \dots a_j \beta_j \mid \gamma_j a_{j+1} \dots a_k \gamma_k \\ \Phi_h(t) & = & \langle_{h+1}\Phi_{h+1}(t_1)\dots\Phi_{h+1}(t_m)\rangle_h & = \beta_0 a_1 \dots a_j \beta_j \mid \delta_j a_{j+1} \dots a_k \delta_k \end{array}$$

Consider any $i < x$ ($i \in \mathbb{N}$). By lemma conditions $s \prec_x t$, therefore $\|s\|_y^{sub} = \|t\|_y^{sub}\ \forall y < x$. In particular $\|s_i\|_y^{sub} = \|t_i\|_y^{sub}\ \forall y$, therefore by lemma 6 $s_i \sim t_i$, therefore by lemma 8 $\Phi_{h+1}(s_i) = \Phi_{h+1}(t_i)$. Let $traces(\Phi_h(s), \Phi_h(t)) = \big((h_0, \dots, h_k), (h'_0, \dots, h'_k)\big)$.

(1) Case $\infty = \|s\|_x^{sub} > \|t\|_x^{sub}$. In this case s_x does not exist and fork happens immediately after $\Phi_{h+1}(s_{x-1})$, $\Phi_{h+1}(t_{x-1})$:

$$\begin{array}{rcl|l} \Phi_h(s) & = & \langle_{h+1}\Phi_{h+1}(s_1)\dots\Phi_{h+1}(s_{x-1}) & \rangle_h \\ \Phi_h(t) & = & \langle_{h+1}\Phi_{h+1}(t_1)\dots\Phi_{h+1}(t_{x-1}) & \Phi_{h+1}(t_x)\dots\Phi_{h+1}(t_m)\rangle_h \end{array}$$

Fork frame is the last one, therefore both γ_j and δ_j contain the closing parenthesis $\rangle_h$ and we have $h_j = h'_j = h$. For all $i < j$ we have $h_i = h'_i = -1$. Therefore $h_i = h'_i\ \forall i$ and $\Phi_h(s) \sim \Phi_h(t)$. Since $first(\gamma_j)$ is $\rangle$ and $first(\delta_j)$ is one of $\langle$ and $\langle\rangle$, we have $\Phi_h(s) \subset \Phi_h(t)$. Therefore $\Phi_h(s) < \Phi_h(t)$.

(2) Case $\infty > \|s\|_x^{sub} > \|t\|_x^{sub} = -1$. In this case both s_x and t_x exist, s_x is not $\perp$ and t_x is $\perp$, and fork happens immediately after $\Phi_{h+1}(s_{x-1})$, $\Phi_{h+1}(t_{x-1})$:

$$\begin{array}{rcl|l} \Phi_h(s) & = & \langle_{h+1}\Phi_{h+1}(s_1)\ldots\Phi_{h+1}(s_{x-1}) & \langle_{h+2}\ \beta'\ \rangle_{h+1}\ \Phi_{h+1}(s_{x+1})\ldots\Phi_{h+1}(s_n)\rangle_h \\ \Phi_h(t) & = & \langle_{h+1}\Phi_{h+1}(t_1)\ \ldots\Phi_{h+1}(t_{x-1}) & \Diamond_{h+1}\qquad\quad \Phi_{h+1}(t_{x+1})\ \ldots\Phi_{h+1}(t_m)\rangle_h \end{array}$$

(2.1) If the fork frame is the last one, then both γ_j and δ_j contain the closing parenthesis $\rangle_h$ and we have $h_j = h'_j = h$.

(2.2) Otherwise the fork frame is not the last one. We have $minh(\gamma_j)$, $minh(\delta_j) \geq h+1$ and $lasth(\beta_j) = h+1$ (the last parenthesis in β_j is either $\langle_{h+1}$ if $x = 1$ and s_{x-1} does not exist, or else one of $\rangle_{h+1}$ and $\Diamond_{h+1}$), therefore $h_j = h'_j = h+1$. For subsequent frames i such that $j < i < k$ we have $h_i = h'_i = h+1$ (on one hand $h_i, h'_i \leq h+1$ because $h_j = h'_j = h+1$, but on the other hand $minh(\gamma_i)$, $minh(\delta_i) \geq h+1$). For the last pair of frames we have $h_k = h'_k = h$ (they both contain the closing parenthesis $\rangle_h$).

In both cases $h_i = h'_i\ \forall i \geq j$. Since $h_i = h'_i = -1\ \forall i < j$, we have $h_i = h'_i\ \forall i$ and therefore $\Phi_h(s) \sim \Phi_h(t)$. Since $first(\gamma_j) = \langle < \Diamond = first(\delta_j)$ we have $\Phi_h(s) \subset \Phi_h(t)$. Therefore $\Phi_h(s) < \Phi_h(t)$.

(3) Case $\infty > \|s\|_x^{sub} > \|t\|_x^{sub} \geq 0$. In this case both s_x and t_x exist and none of them is $\perp$, and fork happens somewhere after the opening parenthesis $\langle_{h+2}$ and before the closing parenthesis $\rangle_{h+1}$ in $\Phi_h(s_x)$, $\Phi_h(t_x)$:

$$\begin{array}{rcl|l} \Phi_h(s) & = & \langle_{h+1}\Phi_{h+1}(s_1)\ldots\Phi_{h+1}(s_{x-1})\ \langle_{h+2}\ \beta' & \gamma'\ \rangle_{h+1}\ \Phi_{h+1}(s_{x+1})\ldots\Phi_{h+1}(s_n)\rangle_h \\ \Phi_h(t) & = & \langle_{h+1}\Phi_{h+1}(t_1)\ \ldots\Phi_{h+1}(t_{x-1})\ \langle_{h+2}\ \beta' & \delta'\ \rangle_{h+1}\ \Phi_{h+1}(t_{x+1})\ \ldots\Phi_{h+1}(t_m)\rangle_h \end{array}$$

From $\|s\|_x^{sub} > \|t\|_x^{sub} \geq 0$ it follows that s_x contains more alphabet symbols than t_x. Consequently $\Phi_{h+1}(s_x)$ contains more alphabet symbols, and thus spans more frames than $\Phi_{h+1}(t_x)$. Let l be the index of the frame δ_l that contains the closing parenthesis $\rangle_{h+1}$ of $\Phi_{h+1}(t_x)$. By the above reasoning $\Phi_{h+1}(s_x)$ does not end in frame γ_l, therefore γ_l does not contain the closing parenthesis $\rangle_{h+1}$ and we have $minh(\gamma_l) \geq h+2$ and $minh(\delta_l) = h+1$. Furthermore, note that $minh(\beta')$, $minh(\gamma')$, $minh(\delta') \geq h+2$, therefore $lasth(\beta_j) \geq h+2$ (including the case when β' is empty), and for all frames i such that $j \leq i < l$ (if any) we have $h_i, h'_i \geq h+2$. Consequently, for l-th frame we have $h_l \geq h+2$ and $h'_l = h+1$, thus $h_l > h'_l$. For subsequent frames i such that $l < i < k$ we have $minh(\gamma_i)$, $minh(\delta_i) \geq h+1$, therefore $h_i \geq h+1$ and $h'_i = h+1$, thus $h_i \geq h'_i$. For the last pair of frames we have $h_k = h'_k = h$, as they both contain the closing parenthesis $\rangle_h$. Therefore $\Phi_h(s) \sqsubset \Phi_h(t)$, which implies $\Phi_h(s) < \Phi_h(t)$.

□

Lemma 10. Let $s, t \in PT(e, w)$ for some IRE e and string w. If $s \prec_x t$, then $\Phi_h(s) < \Phi_h(t)\ \forall h$.

Proof. The proof is by induction on the length of x. Induction basis for $|x| = 1$ is given by lemma 9. Induction step: suppose that the lemma is correct for all x of length $|x| < h$ and let $|x| = h$ ($h \geq 2$). Let $x = x'.x''$ where $x' \in \mathbb{N}$. At least one of $s|_x$ and $t|_x$ must exist (otherwise $\|s\|_x^{sub} = \infty = \|t\|_x^{sub}$ which contradicts $s \prec_x t$), therefore both e and $e|_{x'}$ are compound IREs and s, t can be represented as $s = T^o(s_1, \ldots, s_n)$ and $t = T^o(t_1, \ldots, t_m)$ where $s' = s_{x'} = T^{o'}(s'_1, \ldots, s'_{n'})$ and $t' = t_{x'} = T^{o'}(t'_1, \ldots, t'_{m'})$ and both $o, o' \neq 0$ (because Λ and x' are prefixes of decision position x). Therefore $\Phi_h(s)$, $\Phi_h(t)$ can be represented as follows:

$$\begin{array}{rcl} \Phi_h(s) & = & \langle_{h+1}\Phi_{h+1}(s_1)\ldots\Phi_{h+1}(s_{x'-1})\ \overbrace{\langle_{h+2}\Phi_{h+2}(s'_1)\ldots\Phi_{h+2}(s'_{n'})\rangle_{h+1}}^{\Phi_{h+1}(s')}\ \Phi_{h+1}(s_{x'+1})\Phi_{h+1}(s_n)\rangle_h \\ \Phi_h(t) & = & \langle_{h+1}\Phi_{h+1}(t_1)\ldots\Phi_{h+1}(t_{x'-1})\ \underbrace{\langle_{h+2}\Phi_{h+2}(t'_1)\ldots\Phi_{h+2}(t'_{m'})\rangle_{h+1}}_{\Phi_{h+1}(t')}\ \Phi_{h+1}(t_{x'+1})\Phi_{h+1}(t_m)\rangle_h \end{array}$$

Consider any $i < x'$. By lemma conditions $s \prec_x t$, therefore $\|s\|_y^{sub} = \|t\|_y^{sub}\ \forall y < x$, and in particular $\|s_i\|_y^{sub} = \|t_i\|_y^{sub}\ \forall y$, therefore by lemma 6 $s_i \sim t_i$, therefore by lemma 8 $\Phi_{h+1}(s_i) = \Phi_{h+1}(t_i)$. Since $x' < x$ we have $\|s\|_y^{sub} = \|t\|_y^{sub}\ \forall y \leq x'$ and by lemma 7 $\exists e', w' : s', t' \in PT(e', w')$. Since $\|s'\|_y^{sub} = \|s\|_{x'.y}^{sub}\ \forall y$ and $\|t'\|_y^{sub} = \|t\|_{x'.y}^{sub}\ \forall y$, we have $s' \prec_{x''} t'$. Since $|x''| < |x|$

by induction hypothesis we have $\Phi_{h+1}(s') < \Phi_{h+1}(t')$. If j is the fork and $f \le j \le k$, then $\Phi_h(s)$, $\Phi_h(t)$ can be represented as:

$$\begin{aligned}\Phi_h(s) &= \beta_0 a_1 \dots a_f \beta_f^1 \overbrace{\beta_f^2 a_{f+1} \dots a_j \beta_j \mid \gamma_j a_{j+1} \dots a_k \gamma_k^1}^{\Phi_{h+1}(s')} \gamma_k^2 a_{k+1} \dots a_l \gamma_l \\ \Phi_h(t) &= \beta_0 a_1 \dots a_f \beta_f^1 \underbrace{\beta_f^2 a_{f+1} \dots a_j \beta_j \mid \delta_j a_{j+1} \dots a_k \delta_k^1}_{\Phi_{h+1}(t')} \delta_k^2 a_{k+1} \dots a_l \delta_l\end{aligned}$$

Let $traces(\Phi_h(s), \Phi_h(t)) = \big((h_0, \dots, h_l), (h'_0, \dots, h'_l)\big)$ and $traces(\Phi_{h+1}(s'), \Phi_{h+1}(t')) = \big((\sigma_h, \dots, \sigma_k), (\sigma'_h, \dots, \sigma'_k)\big)$. We show that for frames i such that $j \le i < k$ we have $h_i = \sigma_i \wedge h'_i = \sigma'_i$ and for subsequent frames $k \le i \le l$ we have $h_i = h'_i$.

(1) Case $i = j < k \le l$ (the fork frame). Since we have shown that $\Phi_{h+1}(s_i) = \Phi_{h+1}(t_i)\ \forall i < x'$, and since $\Phi_{h+1}(s')$ and $\Phi_{h+1}(t')$ have nonempty common prefix $\langle_{h+2}$, it follows that $lasth(\Phi_h(s) \sqcap \Phi_h(t)) = lasth(\Phi_{h+1}(s') \sqcap \Phi_{h+1}(t'))$. From $j < k$ it follows that γ_j and δ_j end before a_k and are not changed by appending γ_k^2 and δ_k^2. Therefore $h_j = \sigma_j \wedge h'_j = \sigma'_j$.

(2) Case $j < i < k \le l$. The computation of h_i, h'_i depends only on h_j, h'_j, or which we have shown $h_j = \sigma_j \wedge h'_j = \sigma'_j$ in case (1), and on $\Phi_{h+1}(s')$, $\Phi_{h+1}(t')$, which are not changed by appending γ_k^2 and δ_k^2 since $i < k$. Therefore $h_i = \sigma_i \wedge h'_i = \sigma'_i$.

(3) Case $j \le i = k < l$. We have $minh(\gamma_k^1) = minh(\delta_k^1) = h + 1$ and $minh(\gamma_k^2) = minh(\delta_k^2) \ge h + 1$. None of the preceding frames after the fork contain parentheses with height less than $h + 1$, therefore $h_k = h'_k = h + 1$.

(4) Case $j \le k < i < l$. We have $h_i = h'_i = h + 1$, because $h_k = h'_k = h + 1$ and $minh(\gamma_i)$, $minh(\delta_i) \ge h + 1$.

(5) Case $j \le k \le i = l$. We have $h_l = h'_l = h$, because both γ_l and δ_l contain the closing parenthesis $\rangle_h$.

We have shown that $h_i = \sigma_i \wedge h'_i = \sigma'_i\ \forall i : j \le i < k$ and $h_i = h'_i\ \forall i : k \le i \le l$. It trivially follows that $\Phi_{h+1}(s') \sqsubset \Phi_{h+1}(t') \Rightarrow \Phi_h(s) \sqsubset \Phi_h(t)$ and $\Phi_{h+1}(s') \sim \Phi_{h+1}(t') \Rightarrow \Phi_h(s) \sim \Phi_h(t)$. Because none of $\Phi_{h+1}(s')$, $\Phi_{h+1}(t')$ is a proper prefix of another, $\Phi_{h+1}(s') \subset \Phi_{h+1}(t') \Rightarrow \Phi_h(s) \subset \Phi_h(t)$. Therefore $\Phi_{h+1}(s') < \Phi_{h+1}(t') \Rightarrow \Phi_h(s) < \Phi_h(t)$ (the premise has been shown). □

Theorem 5. If $s, t \in PT(e, w)$ for some IRE e and string w, then $s \prec t \Leftrightarrow \Phi_h(s) < \Phi_h(t)\ \forall h$.

Proof. Forward implication is given by lemma 10. Backward implication: suppose, on the contrary, that $\Phi_h(s) < \Phi_h(t)\ \forall h$, but $s \not\prec t$. Since $\prec$ is a strict weak order (by theorem 2), it must be that either $s \sim t$ (then $\Phi_h(s) = \Phi_h(t)\ \forall h$ by lemma 8), or $t \prec s$ (then $\Phi_h(t) < \Phi_h(s)\ \forall h$ by lemma 10). Both cases contradict $\Phi_h(s) < \Phi_h(t)\ \forall h$, therefore assumption $s \not\prec t$ is incorrect. □

Correctness of incremental path comparison

Lemma 1 (Frame-by-frame comparison of PEs)**.** If α, β are comparable PE prefixes, c is an alphabet symbol and γ is a single-frame PE fragment, then $\alpha < \beta$ implies $\alpha c\gamma < \beta c\gamma$.

Proof. Let $\big((h_1, \dots, h_n), (h'_1, \dots, h'_n)\big) = traces(\alpha, \beta)$ where $n \ge 1$. Since $\alpha c\gamma$, $\beta c\gamma$ have one more frame than α, β and the first n frames are identical to frames of α, β, we can represent $traces(\alpha c\gamma, \beta c\gamma)$ as $\big((h_1, \dots, h_n, h_{n+1}), (h'_1, \dots, h'_n, h'_{n+1})\big)$.

(1) Case $\alpha \sim \beta \wedge \alpha \subset \beta$. In this case $h_i = h'_i\ \forall i \le n$, therefore $h_{n+1} = min(h_n, minh(\gamma)) = min(h'_n, minh(\gamma)) = h'_{n+1}$ and $\alpha c\gamma \sim \beta c\gamma$. Furthermore, $first(\alpha c\gamma \backslash \beta c\gamma) = first(\alpha \backslash \beta)$ and $first(\beta c\gamma \backslash \alpha c\gamma) = first(\beta \backslash \alpha)$, therefore $\alpha \subset \beta \implies \alpha c\gamma \subset \beta c\gamma$.

(2) Case $\alpha \sqsubset \beta$. In this case $\exists j \le n$ such that $h_j > h'_j$ and $h_i = h'_i\ \forall j < i \le n$. We show that $\exists l \le n + 1$ such that $h_l > h'_l$ and $h_i = h'_i\ \forall l < i \le n + 1$, which by definition means that $\alpha c\gamma \sqsubset \beta c\gamma$.

(2a) Case $j < n$. In this case $h_n = h'_n$ and $h_{n+1} = min(h_n, minh(\gamma)) = min(h'_n, minh(\gamma)) = h'_{n+1}$, therefore $l = j$.

(2b) Case $j = n$ and $minh(\gamma) > h'_n$. In this case $h_n > h'_n$ and we have $h_{n+1} = min(h_n, minh(\gamma)) > h'_n$ and $h'_{n+1} = min(h'_n, minh(\gamma)) = h'_n$, therefore $h_{n+1} > h'_{n+1}$ and $l = n + 1$.

(2c) Case $j = n$ and $minh(\gamma) \le h'_n$. In this case $h_n > h'_n$ and we have $h_{n+1} = min(h_n, minh(\gamma)) = minh(\gamma)$ and $h'_{n+1} = min(h'_n, minh(\gamma)) = minh(\gamma)$, therefore $h_{n+1} = h'_{n+1}$ and $l = n$.

In both cases $\alpha c\gamma < \beta c\gamma$. □

Lemma 2. Minimal paths do not contain tagged ϵ-loops.

Proof. Suppose, on the contrary, that π is a minimal path in some TNFA and that π contains at least one tagged ϵ-loop. We show that it is possible to construct another path π' such that $\pi' < \pi$. Path π can be represented as $\pi = \pi_1\pi_2\pi_3$, where $\pi_1 = q_0 \xrightarrow{u|\alpha} q$, $\pi_2 = q \xrightarrow{\epsilon|\beta} q$ is the last tagged ϵ-loop on π and $\pi_3 = q \xrightarrow{v|\gamma} q_f$. Let $\pi' = \pi_1\pi_3$ be the path that is obtained from π by removing the loop π_2. Paths π and π' consume the same input string uv and induce comparable PEs $\alpha\beta\gamma$ and $\alpha\gamma$. Let $\big((h_1, \dots, h_n), (h'_1, \dots, h'_n)\big) = traces(\alpha\beta\gamma, \alpha\gamma)$ and let k be the index of the fork frame. By construction of TNFA the loop π_2 must be contained in a sub-TNFA f for sub-IRE of the form $e = (\_, \_, e_1^{1,\infty})$, as this is the only looping TNFA construct — see algorithm 7. Let f_1 be the sub-TNFA for e_1. Path π enters f and iterates through f_1 at least twice before leaving f (single iteration is not enough to create a loop by TNFA construction). Let j be the total number of iterations through f_1, and let i be the index of the last ϵ-loop iteration (note that not all iterations are necessarily ϵ-loops). Consider two possible cases:

(1) Case $i = j$. In this case fork of $\alpha\beta\gamma$ and $\alpha\gamma$ happens immediately after $(i-1)$-th iteration:

$$\begin{array}{rl|l} \alpha\beta\gamma = x_0\langle_{h-1} & \langle_h x_1\rangle_h \dots \langle_h x_{i-1}\rangle_h & \langle_h x_i\rangle_h \ \rangle_{h-1} x_{j+1} \\ \alpha\gamma = x_0\langle_{h-1} & \langle_h x_1\rangle_h \dots \langle_h x_{i-1}\rangle_h & \rangle_{h-1} x_{j+1} \end{array}$$

Since x_i is an ϵ-loop, it is contained in the fork frame of $\alpha\beta\gamma$. We have $minh(\beta) = h$ and $minh(\gamma) \le h - 1$, therefore $h_k = h'_k \le h-1$. Subsequent frames $l > k$ (if any) are identical and thus $h_l = h'_l$. Furthermore, $first(\gamma) = \rangle < \langle = first(\beta)$. Therefore $\alpha\beta\gamma \sim \alpha\gamma$ and $\alpha\gamma \subset \alpha\beta\gamma$.

(2) Case $i < j$. In this case $(i+1)$-th iteration cannot be an ϵ-loop (because we assumed that i-th iteration is the last ϵ-loop), therefore the fork of $\alpha\beta\gamma$ and $\alpha\gamma$ happens inside of i-th iteration of $\alpha\beta\gamma$ and $(i+1)$-th iteration of $\alpha\gamma$:

$$\begin{array}{rl|ll} \alpha\beta\gamma = x_0\langle_{h-1} & \langle_h x_1\rangle_h \dots \langle_h x_{i-1}\rangle_h\langle_h y_1 & y_2\rangle_h\langle_h x_{i+1}\rangle_h\langle_h x_{i+2}\rangle_h \dots \langle_h x_j\rangle_h \ \rangle_{h-1} x_{j+1} \\ \alpha\gamma = x_0\langle_{h-1} & \langle_h x_1\rangle_h \dots \langle_h x_{i-1}\rangle_h\langle_h y_1 & y_3 \qquad\quad \rangle_h\langle_h x_{i+2}\rangle_h \dots \langle_h x_j\rangle_h \ \rangle_{h-1} x_{j+1} \end{array}$$

Here $y_1y_2 = x_i$ and $y_1y_3 = x_{i+1}$ (i-th iteration is missing from $\alpha\gamma$ by construction of π'). Fragment y_2 is part of the ϵ-loop, therefore fork frame of $\alpha\beta\gamma$ contains a parenthesis $\rangle_h$ and we have $h_k = h$. On the other hand, y_3 contains alphabet symbols, because x_{i+1} is not an ϵ-loop and y_1 is a part of the ϵ-loop. Therefore fork frame of $\alpha\gamma$ ends in y_3 and we have $h'_k > h$. All subsequent frames $l > k$ are identical: if they contain parentheses of height less than h, then $h_l = h'_l < h$; otherwise $h_l \le h$ and $h'_l > h$. Therefore $\alpha\gamma \sqsubset \alpha\beta\gamma$.

In both cases $\alpha\gamma < \alpha\beta\gamma$, which contradicts the fact that π is a minimal path. □

Lemma 3. GOR1 and GTOP discard paths with tagged ϵ-loops.

Proof. Suppose that GOR1/GTOP finds path $\pi_1\pi_2$ where $\pi_1 = q_0 \xrightarrow{s|\alpha} q_1$ and $\pi_2 = q_1 \xrightarrow{\epsilon|\gamma} q_1$ is a tagged ϵ-loop. Both algorithms construct new paths by exploring transitions from the end state of existing paths, so they can only find $\pi_1\pi_2$ after they find π_1. Therefore when GOR1/GTOP finds $\pi_1\pi_2$, it already has some shortest-path candidate $\pi'_1 = q_0 \xrightarrow{s|\alpha'} q_1$ and must compare ambiguous paths $\pi_1\pi_2$ and π'_1. There are two possibilities: either $\alpha' = \alpha$ or $\alpha' < \alpha$ (the latter means that the algorithm has found a shorter path to q_1 in between finding π_1 and $\pi_1\pi_2$). Let $\big((h_1, \dots, h_k), (h'_1, \dots, h'_k)\big) = traces(\alpha', \alpha\gamma)$.

(1) Case $\alpha' = \alpha$. Because α is a proper prefix of $\alpha\gamma$, fork happens at the last frame and we have $h_k = lasth(\alpha)$ and $h'_k = min(lasth(\alpha), minh(\gamma))$. If $lasth(\alpha) > minh(\gamma)$, then $h_k > h'_k$ and $\alpha \sqsubset \alpha\gamma$. Otherwise $h_k = h'_k$ and $\alpha \sim \alpha\gamma$, and we have $first(\alpha\backslash\alpha\gamma) = \bot$ and $first(\alpha\gamma\backslash\alpha) \ne \bot$, therefore $\alpha \subset \alpha\gamma$. In both cases $\alpha < \alpha\gamma$.

(2) Case $\alpha' < \alpha$. Let $\big((\sigma_1, \dots, \sigma_k), (\sigma'_1, \dots, \sigma'_k)\big) = traces(\alpha', \alpha)$. We have $h_k = \sigma_k$ and $h'_k = min(\sigma'_k, minh(\gamma)) \le \sigma_k$. If $minh(\gamma) < \sigma'_k$ then $h_k > h'_k$ and $\alpha' \sqsubset \alpha\gamma$. Otherwise $h'_k = \sigma'_k$. If $\alpha' \sqsubset \alpha$ then $\alpha' \sqsubset \alpha\gamma$. Otherwise $\alpha' \sim \alpha$ and $\alpha' \subset \alpha$. None of α and α' is a proper prefix of the other because otherwise the longer path has an ϵ-loop through q_1, which contradicts our assumption about π_1 and π'_1. Therefore $first(\alpha'\backslash\alpha) = first(\alpha'\backslash\alpha\gamma)$ and $first(\alpha\backslash\alpha') = first(\alpha\gamma\backslash\alpha')$. Consequently $\alpha' \subset \alpha \implies \alpha' \subset \alpha\gamma$. Thus $\alpha' < \alpha\gamma$.

In both cases $\alpha' < \alpha\gamma$, therefore path $\pi_1\pi_2$ is discarded. □

Lemma 4 (Right distributivity of comparison over concatenation for paths without tagged ϵ-loops)**.** Let $\pi_\alpha = q_0 \xrightarrow{u|\alpha} q_1$ and $\pi_\beta = q_0 \xrightarrow{u|\beta} q_1$ be ambiguous paths in TNFA f for IRE e, and let $\pi_\gamma = q_1 \xrightarrow{\epsilon|\gamma} q_2$ be their common ϵ-suffix, such that $\pi_\alpha\pi_\gamma$ and $\pi_\beta\pi_\gamma$ do not contain tagged ϵ-loops. If $\alpha < \beta$ then $\alpha\gamma < \beta\gamma$.

Proof. Let $\big((h_1, \dots, h_k), (h'_1, \dots, h'_k)\big) = traces(\alpha, \beta)$ and $\big((\sigma_1, \dots, \sigma_k), (\sigma'_1, \dots, \sigma'_k)\big) = traces(\alpha\gamma, \beta\gamma)$. Appending γ to α and β changes only the last frame, therefore for frames $i < k$ we have $h_i = \sigma_i$ and $h'_i = \sigma'_i$. Consider two possible cases.

(1) Case $\alpha \sim \beta \wedge \alpha \subset \beta$. We show that $\alpha\gamma \sim \beta\gamma \wedge \alpha\gamma \subset \beta\gamma$. We have $h_i = h'_i \ \forall i$ (implied by $\alpha \sim \beta$), therefore $\sigma_i = \sigma'_i \ \forall i$ and consequently $\alpha\gamma \sim \beta\gamma$. Let $x = first(\alpha \backslash \beta)$, $y = first(\beta \backslash \alpha)$, $x' = first(\alpha\gamma \backslash \beta\gamma)$ and $y' = first(\beta\gamma \backslash \alpha\gamma)$. If one of π_α and π_β is a proper prefix of another, then the longer path contains tagged ϵ-loop through q_1, which contradicts lemma conditions (the suffix of the longer path must be an ϵ-path, because α and β have the same number of frames and the suffix is contained in the last frame). Therefore none of π_α and π_β is a proper prefix of another. Consequently $x = x'$ and $y = y'$, and we have $\alpha \subset \beta \implies x < y \implies x' < y' \implies \alpha\gamma \subset \beta\gamma$.

(2) Case $\alpha \sqsubset \beta$: by definition this means that $\exists j \le k$ such that $h_j > h'_j$ and $h_i = h'_i \ \forall i > j$. We show that $\alpha\gamma \sqsubset \beta\gamma$.

(2a) Case $j < k$. In this case $h_k = h'_k$ and appending γ does not change relation on the last frame: $\sigma_k = min(h_k, minh(\gamma)) = min(h'_k, minh(\gamma)) = \sigma'_k$. Since $\sigma_i = h_i$ and $\sigma'_i = h'_i$ for all preceding frames $i < k$, we have $\alpha\gamma \sqsubset \beta\gamma$.

(2b) Case $j = k$ and $minh(\gamma) > h'_k$. In this case $h_k > h'_k$ and again appending γ does not change relation on the last frame: $\sigma_k = min(h_k, minh(\gamma)) > h'_k$ and $\sigma'_k = min(h'_k, minh(\gamma)) = h'_k$, therefore $\sigma_k > \sigma'_k$. Therefore $\alpha\gamma \sqsubset \beta\gamma$.

(2c) Case $j = k$ and $minh(\gamma) \le h'_k$ and $\exists l < k$ such that $h_l > h'_l$ and $h_i = h'_i$ for $l < i < k$. In this case γ contains parentheses of low height and appending it makes height on the last frame equal: $\sigma_k = \sigma'_k = minh(\gamma)$. However, the relation on the last preceding differing frame is the same: $\sigma_l = h_l > h'_l = \sigma'_l$. Therefore $\alpha\gamma \sqsubset \beta\gamma$.

(2d) Case $j = k$ and $minh(\gamma) \le h'_k$ and $\nexists l < k$ such that $h_l > h'_l$ and $h_i = h'_i$ for $l < i < k$. In this case γ contains parentheses of low height, appending it makes height on the last frame equal: $\sigma_k = \sigma'_k = minh(\gamma)$, and this may change comparison result as the relation on the last preceding differing frame may be different. We show that in this case the extended path $\pi_\beta\pi_\gamma$ contains a tagged ϵ-loop. Consider the fragments of paths π_α and π_β from fork to join, including (if it exists) the common ϵ-transition to the fork state: π'_α and π'_β. Minimal parenthesis height on π'_α is h_k. By TNFA construction this means that π'_α is contained in a sub-TNFA f' for $e|_x$ at some position x with length $|x| = h_k$. As for π'_β, its start state coincides with π'_α and thus is in f'. The minimal height of all but the last frames of π'_β is at least h_k: by conditions of (2d) either $k = 1$ and there are no such frames, or $h'_{k-1} \ge h_{k-1}$ which implies $h'_{k-1} \ge h_k$ (because by definition $h_k = min(h_{k-1}, minh(\alpha_k)) \le h_{k-1}$). On the last frame of π'_β minimal height is $h'_k < h_k$. Therefore all but the last frames of π'_β are contained in f', but the the last frame is not. Now consider π_γ: by conditions of (2d) its minimal height is less than h_k, therefore it is not contained in f', but its start state is the join point of π'_α and π'_β and thus in f'. Taken together, above facts imply that the last frame of $\pi_\beta\pi_\gamma$ starts in f', then leaves f', then returns to f' and joins with $\pi_\alpha\pi_\gamma$, and then leaves f' second time. Since the end state of f' is unique (by TNFA construction), $\pi_\beta\pi_\gamma$ must contain a tagged ϵ-loop through it, which contradicts lemma conditions.

(Note that in the presence of tagged ϵ-loops right distributivity may not hold: we may have paths π_1, π_2 and π_3 such that π_2 and π_3 are two different ϵ-loops through the same subautomaton and $\pi_1\pi_2 < \pi_1\pi_3$, in which case $\pi_1\pi_2\pi_3 < \pi_1\pi_3$, but $\pi_1 < \pi_1\pi_2$ because the first is a proper prefix of the second.) □

□