Received xx July 2019; | Revised xx July 2019; | Accepted xx July 2019

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 further develop the POSIX disambiguation algorithm by Okui and Suzuki. We extend its theoretical foundations on a few important practical cases and introduce numerous performance improvements. Our algorithm works in worst-case $O(n\,m^2\,t)$ time and $O(m^2)$ space, 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 groups. Benchmarks show that in practice our algorithm is ~5x slower than leftmost-greedy matching. We present a lazy version of the algorithm that is much faster, but requires memory proportional to the size of input. We study other NFA-based algorithms and show that the Kuklewicz algorithm is slower in practice, and the backward matching algorithm by Cox is incorrect.

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

1 | INTRODUCTION

In this paper we study NFA-based approaches to the problem of POSIX regular expression parsing and submatch extraction. A number of algorithms have been proposed in recent years, but not all of them have been properly studied and formalized. We experimented with different approaches and found that in practice the algorithm by Okui and Suzuki[1] is the most efficient one. In the process, we have discovered a number of improvements that require careful reconstruction of the underlying theory and introduction of new algorithms and proofs. In our experience, the Okui and Suzuki approach is not easy to understand, therefore we include illustrative examples and detailed pseudocode of the extended algorithm. It should be noted that there is a totally different (and very elegant) approach to the problem based on Brzozowski derivatives[5]. We choose to focus on NFA-based approach because in our experience, derivative-based approach is slow in practice (we also discuss theoretical bounds below). Both NFA and derivatives can be used to construct DFA with POSIX longest-match semantics[6 7 9]. The resulting DFA-based algorithms are very fast, because there is no run-time overhead on disambiguation. However, DFA construction is not always viable due to its exponential worst-case complexity, and if viable, it needs to be efficient. Therefore we concentrate on NFA-based algorithms that can be used directly for matching or serve as a basis for DFA construction. We give an overview of existing algorithms, including some that are incorrect but interesting.

Laurikari, 2001 (incorrect). Laurikari described an algorithm based on TNFA, which is an ϵ-NFA with tagged transitions[2]. In his algorithm each submatch group is represented 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 POSIX subexpression hierarchy. Notably, Laurikari used the idea of topological order to avoid worst-case exponential time of ϵ-closure construction. His algorithm doesn't track history of iteration subexpressions and gives incorrect result in cases like `(a|aa)*` and string `aa`. Reported computational complexity is $O(n\,m\,c\,t\,log(t))$, where n is input length, m is TNFA size, c is the time for comparing tag values and t is the number of tags. Memory requirement is $O(m\,t)$.

Kuklewicz, 2007. Kuklewicz fixed Laurikari algorithm by introducing *orbit* tags for iteration subexpressions. He gave only an informal description[3], but the algorithm was later formalized in[9]. It works in the same way as the Laurikari algorithm, except that comparison of orbit tags is based on their previous history, not just the most recent value. The clever idea is to avoid recording full history by compressing histories in a matrix of size $t \times m$, where m is TNFA size and t is the number of tags. t-Th row of the matrix represents ordering of closure states with respect to t-th tag (with possible ties — different states may have the same order). Matrix is updated at each step using continuations of tag histories. The algorithm requires $O(m\,t)$ memory and $O(n\,m\,t\,(m + t\,log(m))$ time, where n is the input length (we assume that worst-case optimal $O(m^2\,t)$ algorithm for ϵ-closure is used, and matrix update takes $O(m\,log(m)\,t^2)$ because for t tags we need to sort m states with $O(t)$ comparison function).

Cox, 2009 (incorrect). Cox came up with the idea of backward POSIX matching[16], which is based on the observation that it is easier to maximize submatch on the last iteration than on the first one because we do not need to remember the history of previous iterations. The algorithm consumes input from right to left and tracks two pairs of offsets for each submatch group: the *active* pair of the most recent offsets (used in disambiguation) and the *final* pair of offsets on the backwards-first (i.e. the last) iteration. The algorithm gives incorrect results under two conditions: (1) ambiguous matches have equal offsets on some iteration, and (2) disambiguation happens too late, when 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, ϵ-closure algorithm sometimes compares ambiguous paths *after* their join point, when both paths have a common suffix with tagged transitions. This is the case with Cox prototype implementation[16]; 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 case of `(aaaa|aaa|a){3,4}` and string `aaaaaaaaaa` we have matches `(aaaa)(aaaa)(a)(a)` and `(aaaa)(aaa)(aaa)` with different number of iterations. Assuming that bounded repetition is modeled by chaining three non-optional sub-automata for `(aaaa|aaa|a)` and the optional fourth one, by the time ambiguous paths meet both have active offsets `(0,4)`. Despite the flaw, Cox algorithm is interesting: if somehow delayed comparison problem was fixed, it would work. The algorithm requires $O(m\,t)$ memory and $O(n\,m^2\,t)$ time (assuming worst-case optimal closure algorithm), where n is the length of input, m it the size of the regular expression and t is the number of submatch groups plus enclosing subexpressions.

Okui and Suzuki, 2013. Okui and Suzuki view the disambiguation problem from the point of comparison of parse trees[1]. 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). 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* (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 heights will diverge: subtree with a smaller norm will already be complete, but the one with a greater norm will not. Height of subtrees is easy to track by attributing it to parentheses and encoding in 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 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 the regular expression and c is the number of submatch groups and repetition operators. However, this estimate leaves out the construction of PAT and precomputation of the precedence relation. Memory requirement is $O(m^2)$. Okui-Suzuki disambiguation is combined with Berry-Sethi construction in[7] in construction of parsing DFA.

Sulzmann and Lu, 2013. Sulzmann and Lu based their algorithm on Brzozowski derivatives[5] (correctness proof is given in[8]). The algorithm unfolds a regular expression into a sequence of derivatives and then folds it back into a parse tree. Each derivative is obtained from the previous one by consuming input symbols in left-to-right order, and each tree is built from the next tree by injecting symbols 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: parse trees are naturally ordered by the longest criterion. Time and space complexity is not entirely clear. In[5] Sulzmann and Lu consider the size of the regular expression as a constant. In[6] 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 the regular expression, 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.

Undoubtedly, other approaches exist, but many of them produce incorrect results or require memory proportional to the length of input (e.g. Glibc implementation[21]). Our contributions are the following:

- We extend Okui-Suzuki algorithm on the case of partially ordered parse trees and introduce the notion of *explicit* and *implicit* submatch groups. This greatly reduces the overhead on disambiguation for regular expressions with only a few submatch groups, which is a common case in practice.
- We extend Okui-Suzuki algorithm on the case of bounded repetition.
- We combine Okui-Suzuki algorithm with Laurikari TNFA. It allows us to omit the preprocessing step at the cost of ϵ-closure construction, which may be preferable in cases when preprocessing time is included in match time.
- We introduce *negative tags* that allow us to handle no-match situation in the same way as match. Negative tags provide a simple way to reset obsolete offsets from earlier iterations, in cases like `(a(b)?)*` and string `aba`.
- We consider ϵ-closure construction as a shortest-path problem and show that path concatenation is right-distributive over path comparison for the subset of paths considered by closure algorithm. This justifies the use of well-known Goldberg-Radzik algorithm based on the idea of topological order, which has worst-case optimal quadratic complexity in the size of closure and guaranteed linear complexity if the closure has no ϵ-loops. This is an improvement over naive exhaustive depth-first search with backtracking, and also an improvement over Laurikari algorithm[9].
- We give a faster algorithm for updating precedence matrices. The straightforward algorithm described by Okui and Suzuki involves pairwise comparison of all states in closure and takes $O(m^2\,t)$ time, assuming m states and $O(t)$ comparison function. We show a pathological example `((a?){0,1000})*` where $t \approx m$. Our algorithm takes $O(m^2)$ time.
- We show how to use our algorithm in order to build either parse trees or POSIX-style offsets.
- We present a lazy version of our algorithm that reduces the overhead on disambiguation at the cost of memory usage that grows with the length of input. The lazy algorithm is well-suited for small inputs.
- We provide a C++ implementation of different NFA-based algorithms[19] and benchmark them against each other and against leftmost greedy implementation that has no overhead on disambiguation and serves as a baseline. We also provide a completely independent Java implementation and a web-page for interactive experiments with our 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 lazy variation of our 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. As usual, we formalize matching problem by giving the interpretation of regular expressions as sets of parse trees. Then we 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 a practical matching algorithm. Therefore we go from parse trees 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 easily converted to an efficient comparison procedure. Finally, we construct TNFA and map parenthesized expressions to its paths, which allows us to compare ambiguous paths using the definition of order on 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 α (where $\alpha \in \Sigma$) are in $\mathcal{R}_\Sigma$.
2. If $e_1, e_2 \in \mathcal{R}_\Sigma$, then union $e_1 | e_2$, product $e_1 e_2$, repetition $e_1^{n,m}$ (where $0 \leq n \leq m \leq \infty$), and submatch group (e_1) are in $\mathcal{R}_\Sigma$.

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

1. Nil tree $\varnothing^i$, empty tree ϵ^i and unit tree α^i (where $\alpha \in \Sigma$ and $i \in \mathbb{Z}$) are in $\mathcal{T}_\Sigma$.
2. If $t_1, \dots, t_n \in \mathcal{T}_\Sigma$ (where $n \geq 1$, 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 $\Diamond$, empty expression ϵ and unit expression α (where $\alpha \in \Sigma$) are in $\mathcal{P}_\Sigma$.
2. If $e_1, e_2 \in \mathcal{P}_\Sigma$, then $e_1 e_2$ and $\langle e_1 \rangle$ are in $\mathcal{P}_\Sigma$.

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

Σ is a finite set of symbols (*alphabet*)
Q is a finite set of *states*
$T \subset \mathbb{N} \times \mathbb{Z} \times \mathbb{N} \times \mathbb{N}$ is a mapping of *tags* to their submatch group, lower nested tag and upper nested tag
$\Delta = \Delta^\Sigma \sqcup \Delta^\epsilon$ is the *transition* relation, consisting of two parts:
    $\Delta^\Sigma \subseteq Q \times \Sigma \times \{\epsilon\} \times Q$ (transitions on symbols)
    $\Delta^\epsilon \subseteq Q \times \mathbb{N} \times \big(\mathbb{Z} \cup \{\epsilon\}\big) \times Q$ (ϵ-transitions, where $\forall (q, n, \_, \_), (q, m, \_, \_) \in \Delta^\epsilon : n \neq m$)
$q_0 \in Q$ is the *initial* state
$q_f \in Q$ is the *final* state

As the reader might notice, our definitions are subtly different from the usual ones in literature. Regular expressions are extended with submatch operator and generalized repetition (note that it 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 also have a special *nil-parenthesis*. TNFA is in essence a nondeterministic finite-state transducer in which some of the ϵ-transitions are marked with *tags* — integer numbers that denote opening and closing parentheses of submatch groups. For i-th group, the opening tag is $2i - 1$ and the closing tag is $2i$ (where $i \in \mathbb{N}$). Tags can be negative, which represents the absence of match and corresponds to nil-parenthesis $\Diamond$ and nil-tree $\varnothing$. Additionally, all ϵ-transitions are marked with *priority* which allows us to impose specific order of TNFA traversal (all ϵ-transitions from the same state have different priority).

```
$match\big(N{=}(\Sigma, Q, T, \Delta, q_0, q_f),\ \alpha_1 \dots \alpha_n\big)$
    $B, D$ : uninitialized matrices in $\mathbb{Z}^{|Q| \times |Q|}$,  $U$ : path context
    $r_0 = initial\_result(T)$
    $u_0 = empty\_path()$
    $X = \big\{(q_0, \varnothing, u_0, r_0)\big\},\ i = 1$
    while $i \leq n \wedge X \neq \emptyset$ do
        $X = closure(N, X, U, B, D)$
        $X = update\_result(T, X, U, i, \alpha_i)$
        $(B, D) = update\_ptables(N, X, U, B, D)$
        $X = \big\{(q, o, u_0, r) \mid (o, \_, \_, r) \in X \wedge (o, \alpha_i, \epsilon, q) \in \Delta^\Sigma\big\}$
        $i = i + 1$
    $X = closure(N, X, U, B, D)$
    if $(q_f, \_, u, r) \in X$ then
        return $final\_result(T, U, u, r, n)$
    else return $\varnothing$
```

ALGORITHM 1: TNFA simulation on a string.

The algorithm takes an automaton N and string $\alpha_1 \dots \alpha_n$ as input, and outputs the match result is some form: it can be a parse tree or a POSIX array of offsets, but for now we leave it unspecified and hide behind functions $initial\_result()$, $update\_result()$ and $final\_result()$. The algorithm works by consuming input symbols, tracking a set of active *configurations* and updating *precedence tables* B and D. Configuration is a tuple (q, o, u, r). The first component q is a TNFA state that is unique for each configuration in the current set. Components o and u keep information about the path by which q was reached: o is the *origin* state used as index in precedence tables, and u is a path fragment constructed by $closure()$. Specific representation of path fragments is hidden behind path context U and function stub $empty\_path()$. Finally, r-component is a partial match result associated with state q. Most of the real work happens inside of $closure()$ and $update\_ptables()$, both of which remain undefined for now. The $closure()$ function builds ϵ-closure of the current configuration set: it explores all states reachable by ϵ-transitions from the q-components and tracks the best path to each reachable state. The $update\_ptables()$ function performs pairwise comparison of all configurations in the new set, recording results in B and D matrices. On the next step q-components become o-components. If paths originating from current configurations join on some future step, $closure()$ will use origin states to lookup comparison results in B and D matrices. If the paths do not join, then comparison performed by $update\_ptables()$ is redundant — unfortunately we do not know in advance which configurations will spawn ambiguous paths.

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 makes submatch information explicit: to each subexpression we assign an *implicit* and *explicit* submatch index. Explicit indices enumerate submatch groups (for all other subexpressions they are zero). Implicit indices enumerate submatch groups and subexpressions that are not submatch groups, but contain nested or sibling groups and need to be considered by disambiguation. This form reflects the POSIX standard, which states that submatch extraction applies only to parenthesized subexpressions, but the longest-match rule applies 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), where $\alpha \in \Sigma$ and $i, j \in \mathbb{Z}$, are in $\mathcal{IR}_\Sigma$.
2. If $r_1, r_2 \in \mathcal{IR}_\Sigma$ and $i, j \in \mathbb{Z}$, then union $(i, j, r_1 \mid r_2)$, product $(i, j, r_1 \cdot r_2)$ and repetition $(i, j, r_1^{n,m})$, where $0 \leq n \leq m \leq \infty$, are in $\mathcal{IR}_\Sigma$.

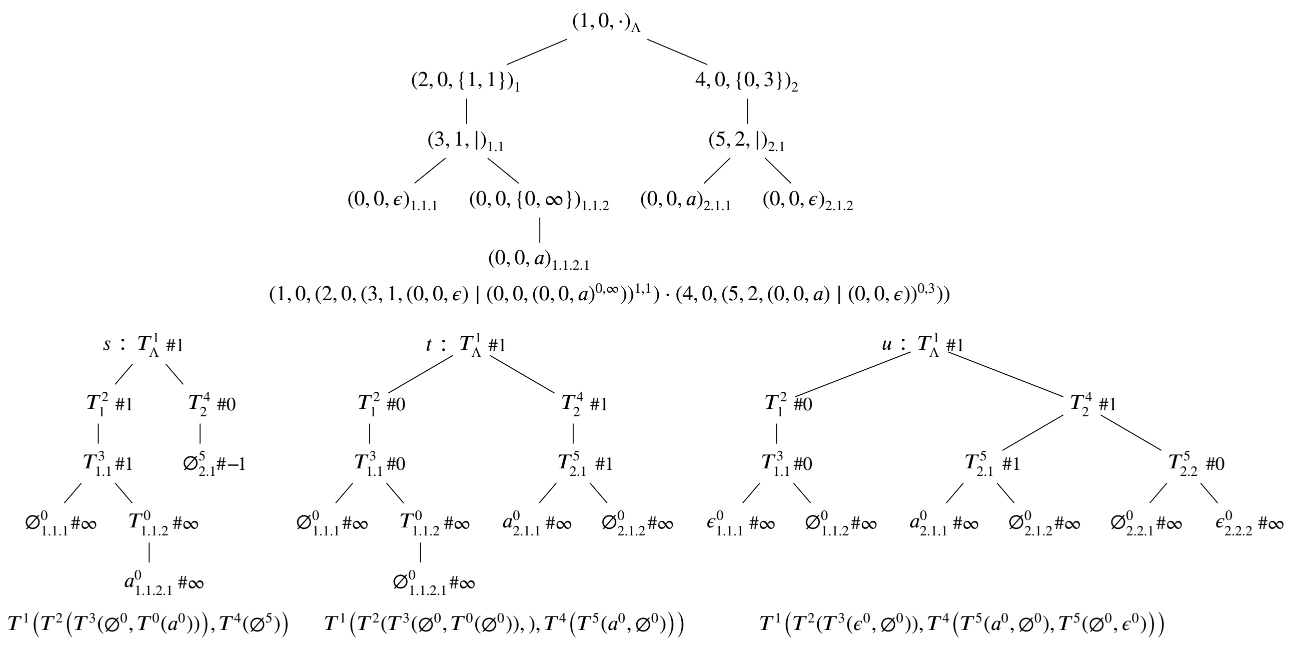
Function IRE transforms RE into IRE. It is defined via a composition of two functions, $mark()$ that transforms RE into IRE with submatch indices in the boolean range $\{0, 1\}$, and $enum()$ that substitutes boolean indices with consecutive numbers. An example of constructing an IRE from a RE is given on figure 1 .

$$
\begin{aligned}
& mark : \mathcal{R}_\Sigma \longrightarrow \mathcal{IR}_\Sigma \\
& mark(x) \mid_{x \in \{\epsilon, \alpha\}} = (0, 0, x) \\
& mark(e_1 \circ e_2) \mid_{\circ \in \{|, \cdot\}} = (i, 0, (i, j_1, r_1) \circ (i, j_2, r_2)) \\
& \qquad \text{where } (i_1, j_1, r_1) = mark(e_1) \\
& \qquad\qquad (i_2, j_2, r_2) = mark(e_2) \\
& \qquad\qquad i = i_1 \vee i_2 \\
& mark(e^{n,m}) \mid_{e = (e_1)} = (1, 0, (1, 1, r)) \\
& \qquad \text{where } (\_, \_, r) = mark(e_1) \\
& mark(e^{n,m}) \mid_{e \neq (e_1)} = (i, 0, (i, j, r)) \\
& \qquad \text{where } (i, j, r) = mark(e) \\
& mark((e)) = mark((e)^{1,1})
\end{aligned}
$$

$$
\begin{aligned}
& enum : \mathbb{Z} \times \mathbb{Z} \times \mathcal{IR}_\Sigma \longrightarrow \mathbb{Z} \times \mathbb{Z} \times \mathcal{IR}_\Sigma \\
& enum(\bar{i}, \bar{j}, (i, j, x)) \mid_{x \in \{\epsilon, \alpha\}} = (\bar{i} + i, \bar{j} + j, (\bar{i} \times i, \bar{j} \times j, x)) \\
& enum(\bar{i}, \bar{j}, (i, j, r_1 \circ r_2)) \mid_{\circ \in \{|, \cdot\}} = (i_2, j_2, (\bar{i} \times i, \bar{j} \times j, \bar{r}_1 \circ \bar{r}_2)) \\
& \qquad \text{where } (i_1, j_1, \bar{r}_1) = enum(\bar{i} + i, \bar{j} + j, r_1) \\
& \qquad\qquad (i_2, j_2, \bar{r}_2) = enum(i_1, j_1, r_2) \\
& enum(\bar{i}, \bar{j}, (i, j, r^{n,m})) = (i_1, j_1, (\bar{i} \times i, \bar{j} \times j, \bar{r}^{n,m})) \\
& \qquad \text{where } (i_1, j_1, \bar{r}) = enum(\bar{i} + i, \bar{j} + j, r) \\
& \\
& IRE : \mathcal{R}_\Sigma \rightarrow \mathcal{IR}_\Sigma \\
& IRE(e) = r \\
& \qquad \text{where } (\_, \_, r) = enum(1, 1, mark(e))
\end{aligned}
$$

The relation between regular expressions and parse trees is given by the operator PT. 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 $PT(r, w)$ denotes the set $\big\{t \in PT(r) \mid str(t) = w\big\}$ of all PTs for IRE r and a string w.

$$
\begin{aligned}
PT &: \mathcal{IR}_\Sigma \rightarrow 2^{\mathcal{T}_\Sigma} \\
PT\big((i, \_, \epsilon)\big) &= \{\epsilon^i\} \\
PT\big((i, \_, \alpha)\big) &= \{\alpha^i\} \\
PT\big((i, \_, (i_1, j_1, r_1) \mid (i_2, j_2, r_2))\big) &= \big\{T^i(t, \varnothing^{i_2}) \mid t \in PT\big((i_1, j_1, r_1)\big)\big\} \cup \big\{T^i(\varnothing^{i_1}, t) \mid t \in PT\big((i_2, j_2, r_2)\big)\big\} \\
PT\big((i, \_, (i_1, j_1, r_1) \cdot (i_2, j_2, r_2))\big) &= \big\{T^i(t_1, t_2) \mid t_1 \in PT\big((i_1, j_1, r_1)\big), t_2 \in PT\big((i_2, j_2, r_2)\big)\big\} \\
PT\big((i, \_, (i_1, j_1, r_1)^{n,m})\big) &= \begin{cases} \big\{T^i(t_1, \dots, t_m) \mid t_k \in PT\big((i_1, j_1, r_1)\big) \; \forall k = \overline{1, m}\big\} \cup \{T^i(\varnothing^{i_1})\} & \text{if } n = 0 \\ \big\{T^i(t_n, \dots, t_m) \mid t_k \in PT\big((i_1, j_1, r_1)\big) \; \forall k = \overline{n, m}\big\} & \text{if } n > 0 \end{cases}
\end{aligned}
$$



$(1,0,\cdot)_\Lambda$
$(2,0,\{1,1\})_1$
$4,0,\{0,3\})_2$
$(3,1,|)_{1.1}$
$(5,2,|)_{2.1}$
$(0,0,\epsilon)_{1.1.1}$
$(0,0,\{0,\infty\})_{1.1.2}$
$(0,0,a)_{2.1.1}$
$(0,0,\epsilon)_{2.1.2}$
$(0,0,a)_{1.1.2.1}$
$(1,0,(2,0,(3,1,(0,0,\epsilon) \mid (0,0,(0,0,a)^{0,\infty}))^{1,1}) \cdot (4,0,(5,2,(0,0,a) \mid (0,0,\epsilon))^{0,3}))$
$s : T^1_\Lambda$ #1
T^2_1 #1
T^4_2 #0
$T^3_{1.1}$ #1
$\varnothing^5_{2.1}$#−1
$\varnothing^0_{1.1.1}$#∞
$T^0_{1.1.2}$ #∞
$a^0_{1.1.2.1}$ #∞
$T^1(T^2(T^3(\varnothing^0,T^0(a^0))),T^4(\varnothing^5))$
$t : T^1_\Lambda$ #1
T^2_1 #0
T^4_2 #1
$T^3_{1.1}$ #0
$T^5_{2.1}$ #1
$\varnothing^0_{1.1.1}$#∞
$T^0_{1.1.2}$ #∞
$a^0_{2.1.1}$ #∞
$\varnothing^0_{2.1.2}$#∞
$\varnothing^0_{1.1.2.1}$#∞
$T^1(T^2(T^3(\varnothing^0,T^0(\varnothing^0)),),T^4(T^5(a^0,\varnothing^0)))$
$u : T^1_\Lambda$ #1
T^2_1 #0
T^4_2 #1
$T^3_{1.1}$ #0
$T^5_{2.1}$ #1
$T^5_{2.2}$ #0
$\epsilon^0_{1.1.1}$ #∞
$\varnothing^0_{1.1.2}$#∞
$a^0_{2.1.1}$ #∞
$\varnothing^0_{2.1.2}$#∞
$\varnothing^0_{2.2.1}$#∞
$\epsilon^0_{2.2.2}$ #∞
$T^1(T^2(T^3(\epsilon^0,\varnothing^0)),T^4(T^5(a^0,\varnothing^0),T^5(\varnothing^0,\epsilon^0)))$

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

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 p is $p.i$ (we shorten $\|t\|_\Lambda$ as $\|t\|$). The *length* of position p, denoted $|p|$, is defined as 0 for Λ and $|p|+1$ for $p.i$. The subtree of a tree t at position p is denoted $t|_p$. Position p is a *prefix* of position q iff $q = p.p'$ for some p', and a *proper prefix* if additionally $p \neq q$. Position p is a *sibling* of position q iff $q = q'.i, p = q'.j$ for some q' 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 $Sub(t) = \{p \mid \exists t|_p = s^i \,:\, i \neq 0\}$ — a subset of $Pos(t)$ that contains positions of subtrees with nonzero implicit submatch index. Intuitively, this is the set of positions important from disambiguation perspective: 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:

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

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

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 $s|_p$ corresponding to an empty repetition, and another tree t has no subtree at position p, then the infinite norm $\|t\|_p$ "outranks" $\|s\|_p$. We define two orders on PTs:

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

Definition 8 (S-order on PTs)**.** Given parse trees $t, s \in PT(r, w)$ for some IRE r and string w, we say that $t \prec_p s$ w.r.t. *decision position* p iff $\|t\|^{sub}_p > \|s\|^{sub}_p$ and $\|t\|^{sub}_q = \|s\|^{sub}_q \;\forall q < p$. We say that $t \prec s$ iff $t \prec_p s$ for some p.

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}. In practice, this means that adding more parentheses in RE does not drastically change submatch results. 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 u on figure 1 : on one hand $t \prec_{2.2} u$, because $\|t\|^{sub}_{2.2} = \infty > 0 = \|u\|^{sub}_{2.2}$ and s-norms at all preceding submatch positions agree; on the other hand $u <_{1.1} t$, because $\|t\|^{pos}_{1.1} = -1 < 0 = \|u\|^{pos}_{1.1}$ 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 the $\prec$-minimal trees in $PT(e, w)$. Then $t_{min} \in T_{min}$.

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 $\Diamond$ 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 Φ transforms PT at the given height into PE:

$$\Phi : \mathbb{Z} \times \mathcal{T}_\Sigma \to \mathcal{P}_\Sigma$$

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

For a given IRE r and string w the set of all PEs $\{\Phi_0(t) \mid t \in PT(r, w)\}$ is denoted $PE(r, w)$, and the set of all prefixes in $PE(r, w)$ is denoted $PR(r, w)$. Each PE α can be represented as $\alpha_0 a_1 \alpha_1 \dots a_n \alpha_n$, where α_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). PE fragments α and β are *comparable* if they have the same number of frames and $\alpha, \beta \in PR(r, w)$ for some r and w. For fragments α and β, $\alpha \sqcap \beta$ denotes their longest common prefix, $\alpha \backslash \beta$ denotes the suffix of α after removing $\alpha \sqcap \beta$, $lasth(\alpha)$ denotes the height of the last parenthesis in α (or ∞ if α is empty or begins with an alphabet symbol), $minh(\alpha)$ denotes the minimal height of parenthesis in α (or ∞ if α is empty or begins with an alphabet symbol), $first(\alpha)$ denotes the first parenthesis in α (or $\bot$ if α is empty or begins with an alphabet symbol). For comparable PE fragments α and β the index of the first distinct pair of frames is called *fork*.

Definition 10. 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 $(\rho_0, \dots, \rho_n)$, where:

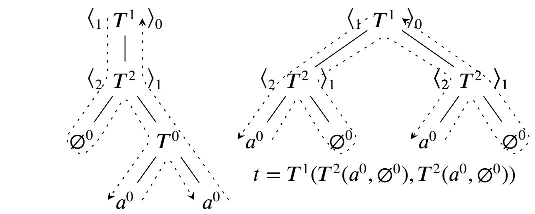
$$\rho_i = \begin{cases} -1 & \text{if } i < k \\ min(lasth(\alpha_i \sqcap \beta_i), minh(\alpha_i \backslash \beta_i)) & \text{if } i = k \\ min(\rho_{i-1}, minh(\alpha_i)) & \text{if } i > k \end{cases}$$

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

Definition 11. (Longest precedence.) Let α, β be comparable PE prefixes and $traces(\alpha, \beta) = \big((\rho_0, \dots, \rho_n), (\rho'_0, \dots, \rho'_n)\big)$. Then $\alpha \sqsubset \beta \Leftrightarrow \exists i \leq n : \big(\rho_i > \rho'_i\big) \wedge \big(\rho_j = \rho'_j \; \forall j > i\big)$. If neither $\alpha \sqsubset \beta$, nor $\beta \sqsubset \alpha$, then α, β are *longest-equivalent*: $\alpha \sim \beta$ (note that in this case $\rho_i = \rho'_i \; \forall i = \overline{1, n}$).

Definition 12. (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: $\bot < \rangle < \langle < \Diamond$.

Definition 13. (Longest-leftmost precedence.) Let α, β be comparable PE prefixes, then $\alpha < \beta \Leftrightarrow \big(\alpha \sqsubset \beta\big) \vee \big(\alpha \sim \beta \wedge \alpha \subset \beta\big)$.


$\langle_1 T^1 \rangle_0$
$\langle_2 T^2 \rangle_1$
$\varnothing^0$
T^0
a^0
a^0
$\langle_1 T^1 \rangle_0$
$\langle_2 T^2 \rangle_1$
$\langle_2 T^2 \rangle_1$
a^0
$\varnothing^0$
a^0
$\varnothing^0$

$t = T^1(T^2(a^0,\varnothing^0),T^2(a^0,\varnothing^0))$

$s = T^1(T^2(\varnothing^0,T^0(a^0,a^0)))$

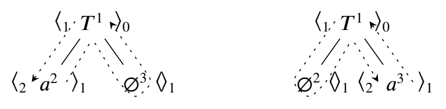
$$\alpha = \Phi_0(s) = \overbracket{\langle_1\langle_2}\, a \quad a\, \overbracket{\rangle_1\rangle_0}$$
$$\beta = \Phi_0(t) = \overbracket{\langle_1\langle_2}\, a\, \overbracket{\rangle_1\langle_2}\, a\, \overbracket{\rangle_1\rangle_0}$$

$traces(\alpha,\beta)$:

$$\begin{bmatrix} \rho_0 = \rho'_0 = -1 \\ \rho_1 = min(lasth(\langle_1\langle_2), minh(\epsilon)) = min(2,\infty) = 2 \\ \rho'_1 = min(lasth(\langle_1\langle_2), minh(\rangle_1\langle_2)) = min(2,1) = 1 \\ \rho_2 = min(\rho_1, minh(\rangle_1\rangle_0)) = min(2,0) = 0 \\ \rho'_2 = min(\rho'_1, minh(\rangle_1\rangle_0)) = min(1,0) = 0 \end{bmatrix}$$

$$\rho_1 > \rho'_1 \;\wedge\; \rho_2 = \rho'_2 \;\Rightarrow\; \alpha \sqsubset \beta \;\Rightarrow\; \alpha < \beta$$
$$\|s\|^{sub}_1 = 2 > 1 = \|t\|^{sub}_1 \wedge \|s\|^{sub}_p = \|t\|^{sub}_p \;\forall p < 1 \;\Rightarrow\; s \prec_1 t$$

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


$\langle_1 T^1 \rangle_0$
$\langle_2 a^2 \rangle_1$
$\varnothing^3 \Diamond_1$
$\langle_1 T^1 \rangle_0$
$\varnothing^2 \Diamond_1$
$\langle_2 a^3 \rangle_1$

$s = T^1(a^2,\varnothing^3)$ $\qquad$ $t = T^1(\varnothing^2,a^3)$

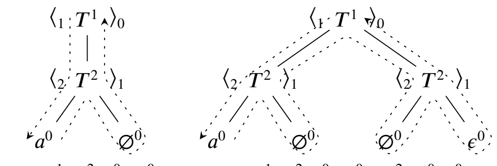
$$\alpha = \Phi_0(s) = \overbracket{\langle_1\langle_2\, a}\, \overbracket{\rangle_1\Diamond_1\rangle_0}$$
$$\beta = \Phi_0(t) = \langle_1 \overbracket{\Diamond_1\langle_2}\, a\, \overbracket{\rangle_1}\rangle_0$$

$traces(\alpha,\beta)$:

$$\begin{bmatrix} \rho_0 = min(lasth(\langle_1), minh(\langle_2)) = min(1,2) = 1 \\ \rho'_0 = min(lasth(\langle_1), minh(\Diamond_1\langle_2)) = min(1,1) = 1 \\ \rho_1 = min(\rho_0, minh(\rangle_1\Diamond_1\rangle_0)) = min(1,0) = 0 \\ \rho'_1 = min(\rho'_0, minh(\rangle_1\rangle_0)) = min(1,0) = 0 \end{bmatrix}$$

$$\rho_i = \rho'_i \;\forall i \;\wedge\; first(\alpha\backslash\beta) = \langle < \Diamond = first(\beta\backslash\alpha) \;\Rightarrow\; \alpha \sim \beta \;\wedge\; \alpha \subset \beta \;\Rightarrow\; \alpha < \beta$$
$$\|s\|^{sub}_1 = 1 > -1 = \|t\|^{sub}_1 \wedge \|s\|^{sub}_p = \|t\|^{sub}_p \;\forall p < 1 \;\Rightarrow\; s \prec_1 t$$

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


$\langle_1 T^1 \rangle_0$
$\langle_2 T^2 \rangle_1$
a^0
$\varnothing^0$
$\langle_1 T^1 \rangle_0$
$\langle_2 T^2 \rangle_1$
$\langle_2 T^2 \rangle_1$
a^0
$\varnothing^0$
$\varnothing^0$
ϵ^0

$s = T^1(T^2(a^0,\varnothing^0))$ $\qquad$ $t = T^1(T^2(a^0,\varnothing^0),T^2(\varnothing^0,\epsilon^0))$

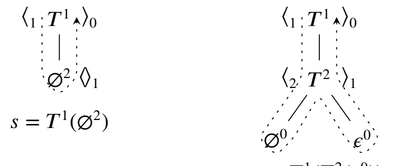
$$\alpha = \Phi_0(s) = \overbracket{\langle_1\langle_2}\, a\, \overbracket{\rangle_1\rangle_0}$$
$$\beta = \Phi_0(t) = \overbracket{\langle_1\langle_2}\, a\, \overbracket{\rangle_1\langle_2\rangle_1\rangle_0}$$

$traces(\alpha,\beta)$:

$$\begin{bmatrix} \rho_0 = \rho'_0 = -1 \\ \rho_1 = min(lasth(\rangle_1), minh(\rangle_0)) = min(1,0) = 0 \\ \rho'_1 = min(lasth(\rangle_1), minh(\langle_2\rangle_1\rangle_0)) = min(1,0) = 0 \end{bmatrix}$$

$$\rho_i = \rho'_i \;\forall i \;\wedge\; first(\alpha\backslash\beta) = \rangle < \langle = first(\beta\backslash\alpha) \;\Rightarrow\; \alpha \sim \beta \;\wedge\; \alpha \subset \beta \;\Rightarrow\; \alpha < \beta$$
$$\|s\|^{sub}_2 = \infty > 0 = \|t\|^{sub}_2 \wedge \|s\|^{sub}_p = \|t\|^{sub}_p \;\forall p < 2 \;\Rightarrow\; s \prec_2 t$$

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


$\langle_1 T^1 \rangle_0$
$\varnothing^2 \Diamond_1$
$\langle_1 T^1 \rangle_0$
$\langle_2 T^2 \rangle_1$
$\varnothing^0$
ϵ^0

$s = T^1(\varnothing^2)$ $\qquad$ $t = T^1(T^2(\epsilon^0))$

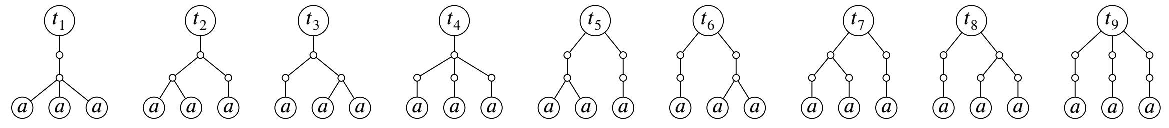
$$\alpha = \Phi_0(s) = \overbracket{\langle_1\Diamond_1\rangle_0}$$
$$\beta = \Phi_0(t) = \overbracket{\langle_1\langle_2\rangle_1\rangle_0}$$

$traces(\alpha,\beta)$:

$$\begin{bmatrix} \rho_0 = min(lasth(\langle_1), minh(\Diamond_1\rangle_0)) = min(1,0) = 0 \\ \rho'_0 = min(lasth(\langle_1), minh(\langle_2\rangle_1\rangle_0)) = min(1,0) = 0 \end{bmatrix}$$

$$\rho_i = \rho'_i \;\forall i \;\wedge\; first(\alpha\backslash\beta) = \Diamond > \langle = first(\beta\backslash\alpha) \;\Rightarrow\; \alpha \sim \beta \;\wedge\; \beta \subset \alpha \;\Rightarrow\; \beta < \alpha$$
$$\|s\|^{sub}_2 = 0 > -1 = \|s\|^{sub}_2 \wedge \|s\|^{sub}_p = \|t\|^{sub}_p \;\forall p < 2 \;\Rightarrow\; t \prec_2 s$$

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



t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	
-1 -1 3 0 -1 -1 2 0	-1 3 3 0 -1 2 2 0	-1 3 3 0 -1 2 2 0	-1 -1 3 0 -1 -1 1 0	-1 3 3 0 -1 1 1 0	-1 3 3 0 -1 2 1 0	-1 3 3 0 -1 1 1 0	-1 3 3 0 -1 1 1 0	t_1
	-1 3 2 0 -1 2 2 0	-1 3 2 0 -1 2 2 0	-1 -1 2 0 -1 -1 1 0	-1 3 2 0 -1 1 1 0	-1 3 2 0 -1 2 1 0	-1 3 2 0 -1 1 1 0	-1 3 2 0 -1 1 1 0	t_2
		-1 -1 3 0 -1 -1 2 0	-1 2 2 0 -1 3 1 0	-1 2 2 0 -1 1 1 0	-1 -1 3 0 -1 -1 1 0	-1 2 2 0 -1 1 1 0	-1 2 2 0 -1 1 1 0	t_3
			-1 2 2 0 -1 3 1 0	-1 2 2 0 -1 1 1 0	-1 -1 2 0 -1 -1 1 0	-1 2 2 0 -1 1 1 0	-1 2 2 0 -1 1 1 0	t_4
				-1 3 1 0 -1 1 1 0	-1 3 1 0 -1 2 1 0	-1 3 1 0 -1 1 1 0	-1 3 1 0 -1 1 1 0	t_5
					-1 1 1 0 -1 2 1 0	-1 -1 3 0 -1 -1 2 0	-1 -1 3 0 -1 -1 1 0	t_6
						-1 2 1 0 -1 1 1 0	-1 2 1 0 -1 1 1 0	t_7
							-1 -1 2 0 -1 -1 1 0	t_8

$$\Phi_0(t_1) = \overbracket{\langle_1\langle_2\langle_3\langle_4}\, a\, \overbracket{\rangle_3\langle_4}\, a\, \overbracket{\rangle_3\langle_4}\, a\, \overbracket{\rangle_3\rangle_2\rangle_1\rangle_0}$$
$$\Phi_0(t_2) = \overbracket{\langle_1\langle_2\langle_3\langle_4}\, a\, \overbracket{\rangle_3\langle_4}\, a\, \overbracket{\rangle_3\rangle_2\langle_3\langle_4}\, a\, \overbracket{\rangle_3\rangle_2\rangle_1\rangle_0}$$
$$\Phi_0(t_3) = \overbracket{\langle_1\langle_2\langle_3\langle_4}\, a\, \overbracket{\rangle_3\rangle_2\langle_3\langle_4}\, a\, \overbracket{\rangle_3\langle_4}\, a\, \overbracket{\rangle_3\rangle_2\rangle_1\rangle_0}$$
$$\Phi_0(t_4) = \overbracket{\langle_1\langle_2\langle_3\langle_4}\, a\, \overbracket{\rangle_3\rangle_2\langle_3\langle_4}\, a\, \overbracket{\rangle_3\rangle_2\langle_3\langle_4}\, a\, \overbracket{\rangle_3\rangle_2\rangle_1\rangle_0}$$
$$\Phi_0(t_5) = \overbracket{\langle_1\langle_2\langle_3\langle_4}\, a\, \overbracket{\rangle_3\langle_4}\, a\, \overbracket{\rangle_3\rangle_2\rangle_1\langle_2\langle_3\langle_4}\, a\, \overbracket{\rangle_3\rangle_2\rangle_1\rangle_0}$$
$$\Phi_0(t_6) = \overbracket{\langle_1\langle_2\langle_3\langle_4}\, a\, \overbracket{\rangle_3\rangle_2\rangle_1\langle_2\langle_3\langle_4}\, a\, \overbracket{\rangle_3\langle_4}\, a\, \overbracket{\rangle_3\rangle_2\rangle_1\rangle_0}$$
$$\Phi_0(t_7) = \overbracket{\langle_1\langle_2\langle_3\langle_4}\, a\, \overbracket{\rangle_3\rangle_2\langle_3\langle_4}\, a\, \overbracket{\rangle_3\rangle_2\rangle_1\langle_2\langle_3\langle_4}\, a\, \overbracket{\rangle_3\rangle_2\rangle_1\rangle_0}$$
$$\Phi_0(t_8) = \overbracket{\langle_1\langle_2\langle_3\langle_4}\, a\, \overbracket{\rangle_3\rangle_2\rangle_1\langle_2\langle_3\langle_4}\, a\, \overbracket{\rangle_3\rangle_2\langle_3\langle_4}\, a\, \overbracket{\rangle_3\rangle_2\rangle_1\rangle_0}$$
$$\Phi_0(t_9) = \overbracket{\langle_1\langle_2\langle_3\langle_4}\, a\, \overbracket{\rangle_3\rangle_2\rangle_1\langle_2\langle_3\langle_4}\, a\, \overbracket{\rangle_3\rangle_2\rangle_1\langle_2\langle_3\langle_4}\, a\, \overbracket{\rangle_3\rangle_2\rangle_1\rangle_0}$$
$$t_1 \prec t_2 \prec t_3 \prec t_4 \prec t_5 \prec t_7 \prec t_6 \prec t_8 \prec t_9$$

(e) – Pairwise comparison of all PEs for RE $(((a)^{1,3})^{1,3})^{1,3}$ and string aaa. Table entry (t_i,t_j) contains $traces(\Phi_0(t_i),\Phi_0(t_j))$.

FIGURE 2 Examples: (a) – (d): four main rules of POSIX comparison, (e) – pairwise comparison of PEs.

Theorem 4. If $s, t \in PT(e, w)$ for some IRE e and string w, then $s \prec t \Leftrightarrow \Phi_h(s) < \Phi_h(t)\ \forall h$.

Next, we go from comparison of PEs to comparison of TNFA paths. 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 symbols and tags which corresponds to a fragment of PE: positive opening tags map to $\langle$, positive closing tags map to $\rangle$, and negative tags map to $\langle\rangle$. We write $q_1 \overset{s|\alpha}{\rightsquigarrow} q_2$ to denote the fact that a path from q_1 to q_2 induces alphabet string s and PE fragment α. We extend the notion of order from PEs to paths: given paths $\pi_1 = q_1 \overset{s|\alpha}{\rightsquigarrow} q_2$ and $\pi_2 = q_1 \overset{s|\beta}{\rightsquigarrow} q_3$ we say that $\pi_1 < \pi_2$ if $\alpha < \beta$. For a given IRE e we say that a path in TNFA for e is *minimal* if it induces $\alpha = PE(t)$ for some minimal tree $t \in PT(e)$. Two paths are *ambiguous* if their start and end states coincide and they induce the same alphabet string. Two paths have a *join point* if they have ambiguous prefixes. 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 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 algorithms for closure construction: GOR1, named after the well-known Goldberg-Radzik algorithm[13], and GTOP, named after "global topological order". Both have the usual structure of shortest-path finding algorithms. The algorithm starts with a set of initial configurations, empty queue and empty set of resulting configurations. Initial configurations are enqueued and the algorithm loops until the queue becomes empty. At each iteration it dequeues configuration (q, o, u, r) and scans ϵ-transitions from state q. For transition $(q, \_, \gamma, p)$ it constructs a new configuration (p, o, v, r) that combines u and γ in an extended path v. If the resulting set contains another configuration for state p, then 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 was changed, the new configuration is enqueued for further scanning. Eventually all states in 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[13]. 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[14]. 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 ϵ-closure (both can be approximated by TNFA size)[15]. 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 setting. In order to do that, we recall the framework for solving shortest-path algorithms based on *closed semirings* described in[10] (section 26.4) and show that our problem fits into this framework. A *semiring* is a structure $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$, where $\mathbb{K}$ is a set, $\oplus : \mathbb{K} \times \mathbb{K} \to \mathbb{K}$ is an associative and commutative operation with identity element $\overline{0}$, $\otimes : \mathbb{K} \times \mathbb{K} \to \mathbb{K}$ is an associative operation with identity element $\overline{1}$, $\otimes$ distributes over $\oplus$ and $\overline{0}$ 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,

```
$closure\_gor1(N = (\Sigma, Q, T, \Delta, q_0, q_f), X, U, B, D)$
context: $C = (N, U, B, D$
    , $topsort, linear$ : stacks of states $q \in Q$
    , $result$ : $Q \to \mathbb{C} \cup \{\varnothing\}$
    , $status$ : $Q \to \{OFF, TOP, LIN\}$
    , $indeg$ : $Q \to \mathbb{Z}$ // in-degree of state
    , $active$ : $Q \to \mathbb{B}$ // true if state needs rescan
    , $etrans$ : $Q \to 2^{\Delta^\epsilon}$ // $\epsilon$-transitions ordered by priority
    , $next$ : $Q \to \mathbb{Z}$) // index of current transition
    $result(q) \equiv \varnothing$
    $status(q) \equiv OFF$
    $active(q) \equiv false$
    $next(q) \equiv 1$
    for $x = (\_, q, \_, \_) \in X$ sorted by inverted $prec()$ do
        $result(q) = x$
        $push(topsort, q)$
    while $topsort$ is not empty do
        while $topsort$ is not empty do
            $q = pop(topsort)$
            if $status(q) \neq LIN$ then
                $status(q) = TOP$
                $push(topsort, q)$
                if $\neg scan(q, C, false)$ then
                    $status(q) = LIN$
                    $pop(topsort)$
                    $push(linear, q)$
        while $linear$ is not empty do
            $q = pop(linear)$
            if $active(q)$ then
                $next(q) = 1$
                $active(q) = false$
                $scan(q, C, true)$
            $status(q) = OFF$
    return $prune(result, N)$

$scan(q, C, all)$
    $any = false$
    while $next(q) < n$ do
        $(q, \epsilon, \tau, p) = etrans(q)_{next(q)}$
        $next(q) = next(q) + 1$
        $x = result(p),\ (o, q, u, r) = result(q)$
        $y = (o, p, extend\_path(H, u, \tau), r)$
        if $x = \varnothing \vee indeg(p) < 2 \vee less(y, x, C)$ then
            $result(p) = y$
            if $status(q) = OFF$ then
                $any = true$
                $next(p) = 1$
                $push(topsort, p)$
                if $\neg all$ then $break$
            else $active(p) = 1$
    return $any$
```

```
$closure\_gtop(N = (\Sigma, Q, T, \Delta, q_0, q_f), X, U, B, D)$
context: $C = (N, U, B, D$
    , $queue$ : priority queue of states $q \in Q$
    , $result$ : $Q \to \mathbb{C} \cup \{\varnothing\}$
    , $status$ : $Q \to \{IN, OUT\}$
    , $indeg$ : $Q \to \mathbb{Z}$ // in-degree of state
    , $topord$ : $Q \to \mathbb{Z}$ // topological index of state
    , $etrans$ : $Q \to 2^{\Delta^\epsilon}$ // $\epsilon$-transitions
    $result(q) \equiv \varnothing$
    $status(q) \equiv OUT$
    for $x = (\_, q, \_, \_) \in X$ do
        $y = result(q)$
        if $y = \bot \vee less(x, y, C)$ then
            $result(q) = x$
            if $status(q) \neq IN$ then
                $insert\_with\_priority(queue, q, topord(q))$
                $status(q) = IN$
    while $queue$ is not empty do
        $q = extract\_min(queue)$
        $status(q) = OUT$
        for $(q, \epsilon, \tau, p) \in etrans(q)$ do
            $x = result(p),\ (o, q, u, r) = result(q)$
            $y = (o, p, extend\_path(H, u, \tau), r)$
            if $x = \varnothing \vee indeg(p) < 2 \vee less(y, x, C)$ then
                $result(p) = y$
                if $status(p) \neq IN$ then
                    $insert\_with\_priority(queue, p, topord(p))$
                    $status(p) = IN$
    return $prune(result, N)$

$prune(X, N)$
    return $\{(\_, q, \_, \_) \in X \mid q \in F \vee \exists (q, \alpha, \_, \_) \in \Delta^\Sigma\}$

$less(x, y, C)$
    $(\_, \_, l) = compare(x, y, U, B, D)$
    return $l < 0$

$prec(x, y, D)$
    $(q, \_, \_, \_) = x,\ (p, \_, \_, \_) = y$
    return $D[q][p] < 0$
```

ALGORITHM 2: Closure algorithms GOR1 (on the left) and GTOP (on the right). Definition of functions of *push*(), *pop*(), *insert_with_priority*(), *extract_min*(), *indeg*() and *topord*() is omitted for brevity. Definitions of *compare*() and *extend_path*() are given in sections 7 and 5. $\mathbb{C}$ is the set of all configurations.

distributivity and idempotence apply to countable $\oplus$-sums. Mohri generalizes this definition and notes that either left or right distributivity is sufficient[11]. 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 $(\mathbb{K}, min, \cdot, \varnothing, \epsilon)$, 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 $\mathbb{K}$), $\varnothing$ corresponds to artificial infinitely long path, and ϵ is the empty path. It is easy to show that min is commutative and associative, $\varnothing$ is identity for min ($min(\pi, \varnothing) = min(\varnothing, \pi) = \pi$), $\cdot$ is associative, ϵ is identity for $\cdot$ ($\pi \cdot \epsilon = \epsilon \cdot \pi = \pi$), $\varnothing$ is annihilator for $\cdot$ ($\pi \cdot \varnothing = \varnothing \cdot \pi = \varnothing$), and 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 $\pi_\alpha = q_0 \overset{u|\alpha}{\rightsquigarrow} q_1$ and $\pi_\beta = q_0 \overset{u|\beta}{\rightsquigarrow} q_1$ be ambiguous paths in TNFA f for IRE e, and let $\pi_\gamma = q_1 \overset{\epsilon|\gamma}{\rightsquigarrow} q_2$ 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 path context U and functions $empty\_path()$ and $extend\_path()$ used in previous sections. An obvious way to represent tagged path is to use a sequence of tags, such as a list or an array: in that case $empty\_path()$ can be implemented as an empty sequence, and $extend\_path()$ is just an append operation. However, a more efficient representation is possible if we consider the structure formed by paths in ϵ-closure. This structure is a *prefix tree* of tags. Some care is necessary with TNFA construction in order to ensure prefixness, but that is easy to accommodate and we give the details in section 9. Storing paths in a prefix tree achieves two purposes: first, we save on the duplicated prefixes, and second, copying paths becomes as simple as copying a pointer to a tree leaf — no need to copy the full sequence. This technique was used by many researches, e.g. Laurikari mentions a *functional data structure*[2] and Karper describes it as the *flyweight pattern*[12].

```
empty_path()
    return 0
extend_path(U, n, τ)
    if τ ≠ ε then
        m = |U| + 1
        append m to succ(U, n)
        append (n, ∅, τ) to U
        return m
    else return n

unroll_path(U, n)
    u = ε
    while n ≠ 0 do
        u = u · tag(U, n)
        n = pred(U, n)
    return reverse(u)
```

ALGORITHM 3: Operations on tag tree.

A convenient representation of tag tree is an indexed sequence of nodes. Each node is a triple (p, s, t) where p is the index of predecessor node, s is a set of indices of successor nodes and t is a tag (positive or negative). Forward links are only necessary if the advanced algorithm for $update\_ptables()$ is used (section 7), otherwise successor component can be omitted. Now we can represent u-components of configurations with indices in the U-tree: root index is 0 (which corresponds to the empty path), and each u-component is a tree index from which we can trace predecessors to the root (function $unroll\_path()$ demonstrates this). In the implementation, it is important to use numeric indices rather than pointers because it allows to use the "two-fingers" algorithm to find fork of two paths (section 7). We assume the existence of functions $pred(U, n)$ that returns p-component of n-th node, $succ(U, n)$ that returns s-component of n-th node and $tag(U, n)$ that returns t-component of n-th node.

6 | REPRESENTATION OF MATCH RESULTS

In this section we show two ways to construct match results: POSIX offsets and a parse tree. In the first case, r-component of configurations is an array of offset pairs $pmatch$. 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. We need the most recent value of each tag, therefore we take care to update tags at most once. Negative tags are updated using helper functions $low()$ and $upp()$ that map each tag to the range of tags covered by it (which includes itself, its pair tag and all nested tags). Helper function $sub()$ maps each tag to the corresponding submatch group. For a given tag t, functions $sub()$, $low()$ and $upp()$ are defined as the 2nd, 3rd and 4th components of $(t, s, l, u) \in T$. Section 9 shows how this mapping is constructed.

In the second case, r-component of configurations 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.

```
$initial\_result(T)$
    return uninitialized array of pairs $(rm\_so, rm\_eo)$

$update\_result(T, X, U, k, \_)$
    return $\{(q, o, u, apply(T, U, u, r, k)) \mid (q, o, u, r) \in X\}$

$final\_result(T, U, u, r, k)$
    $pmatch = apply(T, U, u, r, k)$
    $pmatch[0].rm\_so = 0,\ pmatch[0].rm\_eo = k$
    return $pmatch$

$apply(T, U, n, pmatch, k)$
    $done(\_) \equiv false$
    while $n \neq 0$ do
        $t = tag(U, n)$ $s = sub(T, |t|)$
        if $t < 0 \wedge (s = 0 \vee \neg done(s))$ then
            for $t' = low(T, |t|), upp(T, |t|)$ do
                $s' = sub(T, t')$
                if $s' \neq 0 \wedge \neg done(s')$ then
                    $done(s') = true$
                    $set\_tag(pmatch, t', s', -1)$
        else if $s \neq 0 \wedge \neg done(s)$ then
            $done(s) = true$
            $set\_tag(pmatch, t, s, k)$
        $n = pred(U, n)$
    return $pmatch$

$set\_tag(pmatch, t, s, pos)$
    if $t \bmod 2 \equiv 1$ then $pmatch[s].rm\_so = pos$
    else $pmatch[s].rm\_eo = pos$
```

```
$initial\_result(\_)$
    return $\epsilon$

$update\_result(\_, X, U, \_, \alpha)$
    return $\{(q, o, u, r \cdot unroll\_path(U, u) \cdot \alpha)$
                                  $\mid (q, o, u, r) \in X\}$

$final\_result(\_, U, u, r, \_)$
    return $parse\_tree(r \cdot unroll\_path(U, u), 1)$

$parse\_tree(u, i)$
    if $u = (2i-1) \cdot (2i)$ then
        return $T^i(\epsilon)$
    if $u = (1-2i) \cdot \ldots$ then
        return $T^i(\varnothing)$
    if $u = (2i-1) \cdot \alpha_1 \ldots \alpha_n \cdot (2i) \wedge \alpha_1, \ldots, \alpha_n \in \Sigma$ then
        return $T^i(a_1, \ldots, a_n)$
    if $u = (2i-1) \cdot \beta_1 \ldots \beta_m \cdot (2i) \wedge \beta_1 = 2j-1 \in T$ then
        $n = 0, k = 1$
        while $k \leq m$ do
            $l = k$
            while $|\beta_{k+1}| > 2j$ do $k = k + 1$
            $n = n + 1$
            $t_n = parse\_tree(\beta_l \ldots \beta_k, j)$
        return $T^i(t_1, \ldots, t_n)$
    return $\varnothing$ // ill-formed PE
```

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 $update\_ptables()$. The pseudocode follows definition 13 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 use 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. We assume the existence of helper function $height(T, t)$ that maps each tag to its height.

```
$\underline{compare(c_1, c_2, U, B, D)}$
    $(\_, o_1, n_1, \_) = c_1,\ (\_, o_2, n_2, \_) = c_2$
    if $o_1 = o_2 \wedge n_1 = n_2$ then return $(\infty, \infty, 0)$
    $fork = o_1 = o_2$
    if $fork$ then $h_1 = h_2 = \infty$
    else $h_1 = B[o_1][o_2],\ h_2 = B[o_2][o_1]$
    $m_1 = m_2 = \bot$
    while $n_1 \neq n_2$ do
        if $n_1 > n_2$ then
            $h_1 = min(h_1, height(T, tag(U, n_1)))$
            $m_1 = n_1,\ n_1 = pred(U, n_1)$
        else
            $h_2 = min(h_2, height(T, tag(U, n_2)))$
            $m_2 = n_2,\ n_2 = pred(U, n_2)$
    if $n_1 \neq \bot$ then
        $h = height(T, tag(U, n_1))$
        $h_1 = min(h_1, h),\ h_2 = min(h_2, h)$
    if $h_1 > h_2$ then $l = -1$
    else if $h_1 < h_2$ then $l = 1$
    else if $\neg fork$ then $l = D[o_1][o_2]$
    else $l = leftprec(m_1, m_2, U)$
    return $(h_1, h_2, l)$

$\underline{leftprec(n_1, n_2, U)}$
    if $n_1 = n_2$ then return $0$
    if $n_1 = \bot$ then return $-1$
    if $n_2 = \bot$ then return $1$
    $t_1 = tag(U, n_1),\ t_2 = tag(U, n_2)$
    if $t_1 < 0$ then return $1$
    if $t_2 < 0$ then return $-1$
    if $t_1 mod\ 2 \equiv 0$ then return $-1$
    if $t_2 mod\ 2 \equiv 0$ then return $1$
    return $0$

$\underline{update\_ptables(N, X, U, B, D)}$
    for $x_1 = (q_1, \_, \_, \_) \in X$ do
        for $x_2 = (q_2, \_, \_, \_) \in X$ do
            $(h_1, h_2, l) = compare(x_1, x_2, U, B, D)$
            $B'[q_1][q_2] = h_1,\ D'[q_1][q_2] = l$
            $B'[q_2][q_1] = h_2,\ D'[q_2][q_1] = -l$
    return $(B', D')$

$\underline{update\_ptables(N, X, U, B, D)}$
    $i = 0,\ next(n) \equiv 1,$ empty stack $S,$ empty array $L$
    $push(S, 0)$
    while $S$ is not empty do
        $n = pop(S)$
        if $next(n) < k$ then
            $push(S, n)$
            $push(S, succ(U, n)_{next(n)})$
            $next(n) = next(n) + 1$
            $continue$
        $h = height(T, tag(U, n)),\ i_1 = i$
        for $(q, o, n_1, \_) \in X \mid n_1 = n$ do
            $i = i + 1,\ L[i] = (q, o, \bot, h)$
        for $j_1 = \overline{i_1 + 1, i}$ do
            for $j_2 = \overline{j_1, i}$ do
                $(q_1, o_1, \_, \_) = L[j_1]$
                $(q_2, o_2, \_, \_) = L[j_2]$
                if $n = 0 \wedge o_1 \neq o_2$ then
                    $h_1 = B[o_1][o_2],\ h_2 = B[o_2][o_1]$
                    $l = D[o_1][o_2]$
                else $h_1 = h_2 = h,\ l = 0$
                $B'[q_1][q_2] = h_1,\ D'[q_1][q_2] = l$
                $B'[q_2][q_1] = h_2,\ D'[q_2][q_1] = -l$
        for $m \in succ(U, n)$ in reverse do
            $i_2 = i_1$
            while $i_2 > 0 \wedge L[i_2].n = m$ do $i_2 = i_2 - 1$
            for $j_1 = \overline{i_2, i_1}$ do
                $L[j_1].h = min(L[j_1].h, h);$
                for $j_2 = \overline{i_1, i}$ do
                    $(q_1, o_1, n_1, h_1) = L[j_1]$
                    $(q_2, o_2, n_2, h_2) = L[j_2]$
                    if $n = 0 \wedge o_1 \neq o_2$ then
                        $h_1 = min(h_1, B[o_1][o_2])$
                        $h_2 = min(h_2, B[o_2][o_1])$
                    if $h_1 > h_2$ then $l = -1$
                    else if $h_1 < h_2$ then $l = 1$
                    else if $o_1 \neq o_2$ then $l = D[o_1][o_2]$
                    else $l = leftprec(n_1, n_2, U)$
                    $B'[q_1][q_2] = h_1,\ D'[q_1][q_2] = l$
                    $B'[q_2][q_1] = h_2,\ D'[q_2][q_1] = -l$
            $i_1 = i_2$
        for $j = \overline{i_1, i}$ do $L[j].n = n$
    return $(B', D')$
```

ALGORITHM 5: Disambiguation procedures.

We give two alternative algorithms for $update\_ptables()$: a simple one with $O(m^2\,t)$ complexity (on the left) and a complex one with $O(m^2)$ complexity (on the right). Worst case is demonstrated by RE $((a|\epsilon)^{0,k})^{0,\infty}$ where $n \in \mathbb{N}$, for which the simple algorithm takes $O(k^3)$ time and the complex algorithm takes $O(k^2)$ time. The idea of 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 L of *level items* (q, o, u, h), where q and o are state and origin 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, o, u, r) 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 B and D matrices for each pair of q-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 overhead in our algorithm comes from updating B and D matrices at each step. It is all the more unfortunate since many comparisons performed by *update_ptables*() 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 real ambiguity (when they meet in closure) and avoid computation of precedence matrices altogether. We can do it with a few modifications to our original algorithm. First, we no longer need B and D matrices and *update_ptables*() function. Instead, we introduce cache C that maps a pair of tree indices (n_1, n_2) to a triple of precedence values (h_1, h_2, l). Cache stores the "useful" part of B and D 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 tree representation of paths in the following way: forward links are no longer needed (parent links are sufficient), and tree nodes must be augmented with information about current step and origin state (we assume the existence of helper functions *step*() and *origin*()). Third, instead of using *empty_path*() to initialize path fragments in configurations we need to set them to path fragments of their 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. Below is the modified lazy version of *compare*(), the only part of the algorithm that requires non-trivial change.

```
compare(c_1, c_2, U, C)
    $(\_, \_, n_1, \_) = c_1,\ (\_, \_, n_2, \_) = c_2$
    return $compare1(n_1, n_2, U, C)$

compare1(n_1, n_2, U, C)
    if $C(n_1, n_2) = \varnothing$ then
        $C(n_1, n_2) = compare2(n_1, n_2, U, C)$
    return $C(n_1, n_2)$

compare2(n_1, n_2, U, C)
    if $n_1 = n_2$ then return $(\infty, \infty, 0)$
    $h_1 = h_2 = \infty$
    $o_1 = origin(U, n_1),\ o_2 = origin(U, n_2)$
    $s_1 = step(U, n_1),\ s_2 = step(U, n_2),\ s = max(s_1, s_2)$
    $fork = o_1 = o_2 \wedge s_1 = s_2$
    $m_1 = m_2 = \bot$
    while $n_1 \neq n_2 \wedge (s_1 \geq s \vee s_2 \geq s)$ do
        if $s_1 \geq s \wedge (n_1 > n_2 \vee s_2 < s)$ then
            $h_1 = min(h_1, height(T, tag(U, n_1)))$
            $m_1 = n_1,\ n_1 = pred(U, n_1),\ s_1 = step(U, n_1)$
        else
            $h_2 = min(h_2, height(T, tag(U, n_2)))$
            $m_2 = n_2,\ n_2 = pred(U, n_2),\ s_2 = step(U, n_2)$
    if $\neg fork$ then
        $(h'_1, h'_2, l) = compare1(n_1, n_2, U, C)$
        $h_1 = min(h_1, h'_1),\ h_2 = min(h_2, h'_2)$
    else if $n_1 \neq \bot$ then
        $h = height(T, tag(U, n_1))$
        $h_1 = min(h_1, h),\ h_2 = min(h_2, h)$
    if $h_1 > h_2$ then $l = -1$
    else if $h_1 < h_2$ then $l = 1$
    else if $fork$ then $l = leftprec(m_1, m_2, U)$
    return $(h_1, h_2, l)$
```

ALGORITHM 6: Lazy disambiguation procedures (we assume that cache C 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 reduction of precedence matrices), but it is definitely infeasible for very long or streaming inputs. A possible solution may be a hybrid approach that uses lazy disambiguation, but every k steps fully calculates precedence matrices and "forgets" path prefixes. Another possible solution is to keep both algorithms and choose between them depending on the length of input.

9 | TNFA CONSTRUCTION

TNFA construction is given by the function *tnfa*() that accepts IRE r and state y and returns TNFA for r with final state y (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 slower.
- Bounded repetition $r^{n,m}$ is unrolled in a way that duplicates r 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).
- Compact representation of nested tags as ranges in T is possible because nested tags occupy consecutive numbers.
- Passing the final state y 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).

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.

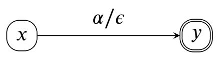
The first step, conversion of RE to IRE, is given by the functions $mark()$ and $enum()$ from section 3. For each sub-RE, $mark()$ constructs a corresponding sub-IRE, and $enum()$ performs a linear visit of the IRE (which doesn't change its structure), therefore IRE size is $O(m)$. Each subexpression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

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 $(i, j, r^{n,m})$ automaton for r is duplicated exactly m times if $m \neq \infty$, or $max(1, n)$ times otherwise (at each step of recursion the counter is decremented and one copy of automaton is constructed). 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 T 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.

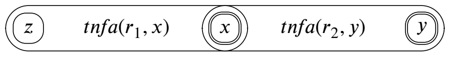
The third step, TNFA simulation, is given by algorithm 1. Initialization at lines 2-5 takes $O(1)$ time. Main loop at lines 6-11 is executed at most n times. The size of closure is bounded by TNFA size, which is $O(m)$ (typically closure is much smaller than

```
$\underline{tnfa(r, y)}$
    if $r = (0, 0, \epsilon)$ then
        return $(\Sigma, \{y\}, \emptyset, \emptyset, y, y)$
    else if $r = (0, 0, \alpha) \mid_{\alpha \in \Sigma}$ then
        return $(\Sigma, \{x, y\}, \emptyset, \{(x, \alpha, \epsilon, y)\}, x, y)$
    else if $r = (0, 0, r_1 \cdot r_2)$ then
        $(\Sigma, Q_2, T_2, \Delta_2, x, y) = tnfa(r_2, y)$
        $(\Sigma, Q_u, T_u, \Delta_2, z, x) = tnfa(r_1, x)$
        return $(\Sigma, Q_1 \cup Q_2, T_1 \cup T_2, \Delta_1 \cup \Delta_2, z, y)$
    else if $r = (0, 0, r_1 \mid r_2)$ then
        $(\Sigma, Q_2, T_2, \Delta_2, x_2, y) = tnfa(r_2, y)$
        $(\Sigma, Q_2', T_2, \Delta_2', x_2', y) = ntags(T_2, y)$
        $(\Sigma, Q_1, T_1, \Delta_1, x_1, x_2') = tnfa(r_2, x_2')$
        $(\Sigma, Q_1', T_1, \Delta_1', x_1', x_2) = ntags(T_1, x_2)$
        $Q = Q_1 \cup Q_1' \cup Q_2 \cup Q_2' \cup \{x\}$
        $\Delta = \Delta_1 \cup \Delta_1' \cup \Delta_2 \cup \Delta_2' \cup \big\{(x, 1, \epsilon, x_1), (x, 2, \epsilon, x_1')\big\}$
        return $(\Sigma, Q, T_1 \cup T_2, \Delta, x, y)$
    else if $r = (0, 0, r_1^{n,m}) \mid_{1<n\leq m\leq\infty}$ then
        $(\Sigma, Q_1, T_1, \Delta_1, x, y) = tnfa((0, 0, r_1^{n-1,m-1}), y)$
        $(\Sigma, Q_2, T_2, \Delta_2, z, x) = tnfa(r_1, x)$
        return $(\Sigma, Q_1 \cup Q_2, T_1 \cup T_2, \Delta_1 \cup \Delta_2, z, y)$
    else if $r = (0, 0, r_1^{0,m})$ then
        $(\Sigma, Q_1, T_1, \Delta_1, x_1, y) = tnfa((0, 0, r_1^{1,m}), y)$
        $(\Sigma, Q_1', T_1, \Delta_1', x_1', y) = ntags(T_1, y)$
        $Q = Q_1 \cup Q_1' \cup \{x\}$
        $\Delta = \Delta_1 \cup \Delta_1' \cup \big\{(x, 1, \epsilon, x_1), (x, 2, \epsilon, x_1')\big\}$
        return $(\Sigma, Q, T_1, \Delta, x, y)$
    else if $r = (0, 0, r_1^{1,\infty})$ then
        $(\Sigma, Q_1, T_1, \Delta_1, z, x) = tnfa(r_1, \_)$
        $Q = Q_1 \cup \{y\}$
        $\Delta = \Delta_1 \cup \big\{(x, 1, \epsilon, z), (x, 2, \epsilon, y)\big\}$
        return $(\Sigma, Q, T_1, \Delta, z, y)$
    else if $r = (0, 0, r_1^{1,1})$ then
        return $tnfa(r_1, y)$
    else if $r = (0, 0, r_1^{1,m}) \mid_{1<m<\infty}$ then
        $(\Sigma, Q_1, T_1, \Delta_1, x, y) = tnfa((0, 0, r_1^{1,m-1}), y)$
        $(\Sigma, Q_2, T_2, \Delta_2, w, z) = tnfa(r_1, z)$
        $\Delta = \Delta_1 \cup \Delta_2 \cup \big\{(z, 1, \epsilon, y), (z, 2, \epsilon, x)\big\}$
        return $(\Sigma, Q_1 \cup Q_2, T_1 \cup T_2, \Delta, w, y)$
    else if $r = (i, j, r_1) \mid_{i\neq 0}$ then
        $(\Sigma, Q_1, T_1, \Delta_1, z, x) = tnfa((0, 0, r_1), x)$
        $Q = Q_1 \cup \{w, y\}$
        $T = T_1 \cup \big\{(2i-1, j, 0, -1), (2i, j, 0, -1)\big\}$
        $\Delta = \Delta_1 \cup \big\{(w, 1, 2i-1, z), (x, 1, 2i, y)\big\}$
        return $(\Sigma, Q, T, \Delta, w, y)$
$\underline{ntags(T, y)}$
    $(t_{open}, t_{last}) = min\_max\big\{t \mid (t, \_, \_, \_) \in T\big\}$
    $(t_{clos}, s, \_, \_) = (t, \_, \_, \_) \in T \mid t = t_{open} + 1$
    $T' = \big\{(t, \_, \_, \_) \in T \mid t \neq t_{clos}\big\} \cup \big\{(t_{clos}, s, t_{open}, t_{last})\big\}$
    $\Delta = \big\{(x, 1, -t_{clos}, y)\big\}$
    return $(\Sigma, \{x, y\}, T', \Delta, x, y)$
```

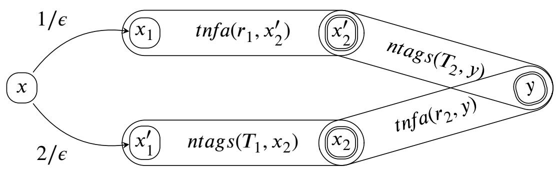
(a) $tnfa\big((0, 0, \epsilon), y\big)$



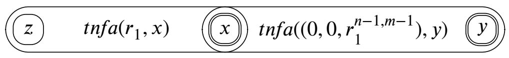
(b) $tnfa\big((0, 0, a), y\big)$



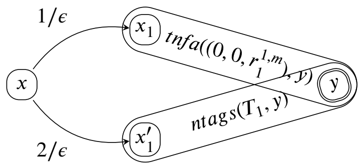
(c) $tnfa\big((0, 0, r_1 \cdot r_2), y\big)$



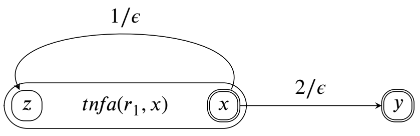
(d) $tnfa\big((0, 0, r_1 \mid r_2), y\big)$



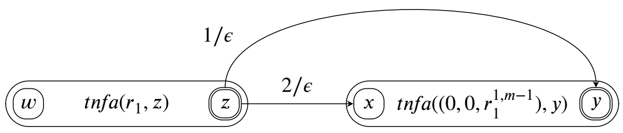
(e) $tnfa\big((0, 0, r_1^{n,m}), y\big) \mid_{1<n\leq m\leq\infty}$



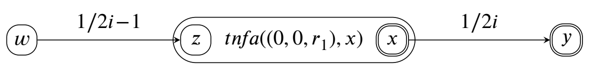
(f) $tnfa\big((0, 0, r_1^{0,m}), y\big)$



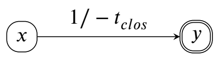
(g) $tnfa\big((0, 0, r_1^{1,\infty}), y\big)$



(h) $tnfa\big((0, 0, r_1^{1,m}), y\big) \mid_{1<m<\infty}$



(i) $tnfa\big((i, j, r_1), y\big) \mid_{i\neq 0}$



(j) $ntags\big(T, y\big)$

ALGORITHM 7: TNFA construction.

that). Each iteration of the loop includes the following steps. At line 7 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 8 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 9 the call to $update\_ptables()$ takes $O(m^2)$ time if the advanced algorithm from section 7 is used. At line 10 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 6-11 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 B and D 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 path context U 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.

11 | BENCHMARKS

In order to present benchmark results in a meaningful way, we show the time of each algorithm relative to the "baseline" leftmost greedy algorithm, which has no overhead on disambiguation and thus represents best-case matching time. We measure the speed of each algorithm in characters per second and divide it by the speed of leftmost greedy algorithm. This allows us to show the net overhead on POSIX disambiguation. To relate our implementations to the real world, we include Google RE2 library (it uses leftmost greedy disambiguation and serves as a reference, not as a competing implementation). All implementations can be found in RE2C source code[19]. Our set of benchmarks consists of three subsets:

1. Real-world benchmarks. These include very large REs containing thousands of characters and hundreds of capturing groups (parser for HTTP message headers conforming to RFC-7230, URI parser conforming to RFC-3986, IPv6 address parser); medium-sized REs containing hundreds of characters and dozens of capturing groups (simplified parsers for HTTP headers and URI, IPv4 address parser, simple date parser); and small REs with less than a hundred characters and about five capturing groups (simplified parsers for IPv4 and date).
2. Artificial benchmarks with high level of ambiguity. All these REs are restricted to a single alphabet symbol used with various combinations of RE operators (union, product, iteration and bounded repetition).
3. A series of pathological examples that demonstrates worst-case behavior of the naive $update\_ptables()$ algorithm.

We benchmark four variations of our algorithm: "Okui-Suzuki" is the main variation (it uses advanced $update\_ptables()$ algorithm and GOR1), "GTOP Okui-Suzuki" uses GTOP, "naive Okui-Suzuki" uses naive $update\_ptables()$ algorithm, and "lazy Okui-Suzuki" differs from the main variation as described in section 8. Besides our algorithm, we also benchmark Kuklewicz and Cox algorithms. Kuklewicz algorithm is described in detail in[9]. As for the Cox algorithm, the only description we are aware of is the prototype implementation[16]. We found a number of flaws in it, as described in the introduction. Our implementation, therefore, differs from the original: we add support for bounded repetition, we use GOR1 for closure construction, and we use a fast forward pre-processing phase to find the matching string prefix before running the backward phase (forward phase ignores submatch and merely performs recognition). Benchmark results show the following:

- Okui-Suzuki algorithm degrades with increased closure size. This is understandable, as the algorithm performs pairwise comparison of closure states to compute precedence matrices. Naive $update\_ptables()$ algorithm degrades extremely fast, and the advanced algorithm behaves much better (though it may incur slight overhead in simple cases).
- Kuklewicz algorithms degrades with increased closure size and increased number of tags. This is not surprising, as the algorithm has per-state and per-tag loop used to compute precedence matrix. On real-world tests with many capturing groups Kuklewicz algorithm is much slower than Okui-Suzuki algorithm.

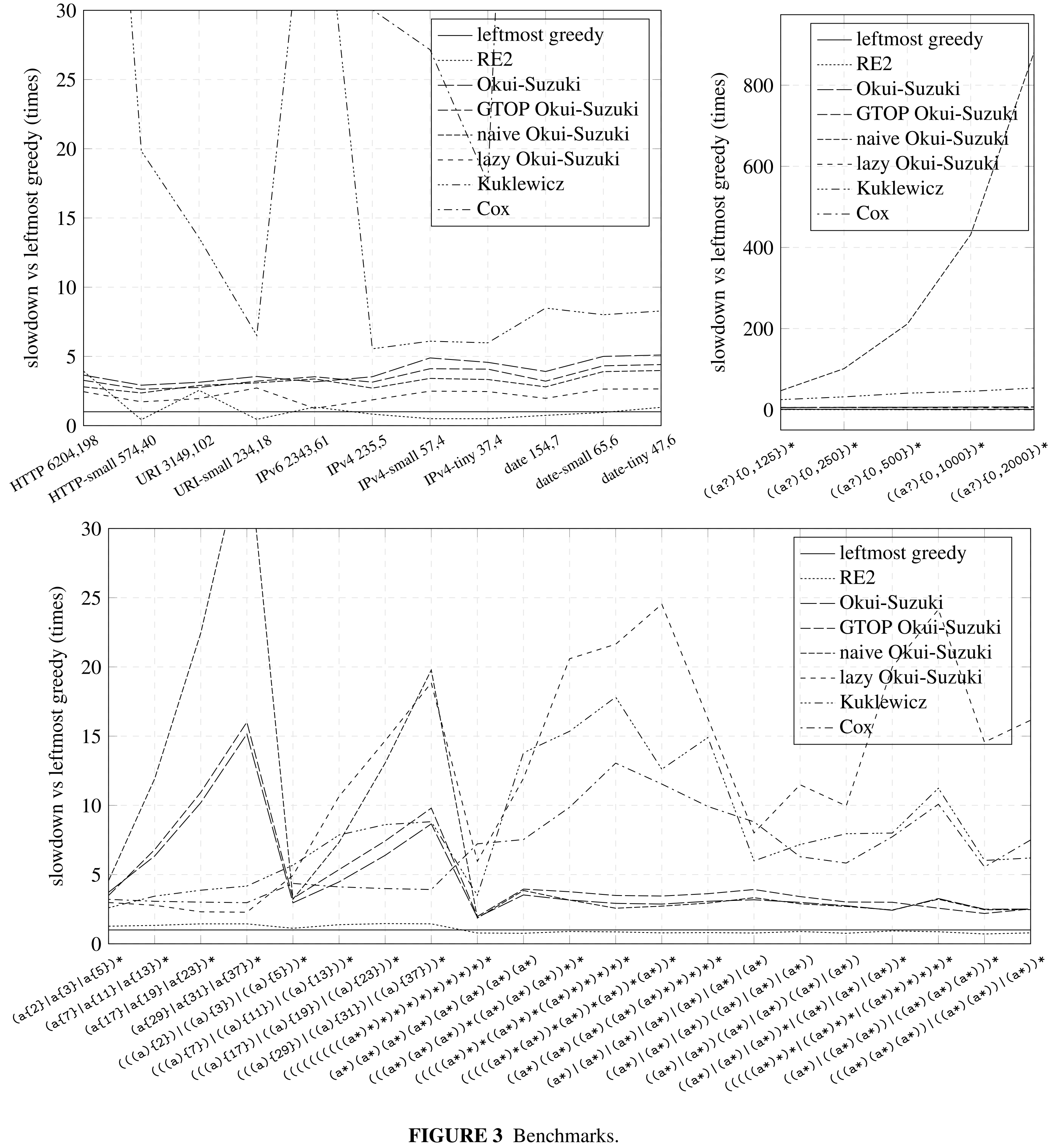


FIGURE 3 Benchmarks.
Real-world tests have labels of the form “title *m*,*k*”, where *m* is RE size and *k* is the number of capturing groups.

- Cox algorithm degrades with increased number of tags. The bottleneck of the algorithm is copying of offset arrays (each array contains a pair of offsets per tag). Using GOR1 instead of naive depth-first search increases the amount of copying (though asymptotically faster), because depth-first scan order allows to use a single buffer array that is updated and restored in-place. However, copying is required elsewhere in the algorithm, and in general it is not suited for RE with many submatch groups. On real-world tests the Cox algorithm is so slow that it did not fit into the plot space.

- Lazy variation of Okui-Suzuki degrades with increased cache size and the size of path context. This may happen because of long input strings and because of high level of ambiguity in RE (in such cases lazy algorithm does all the work of non-lazy algorithm, but with the additional overhead on cache lookups/insertions and accumulation of data from the previous steps). On real-world tests lazy variation of Okui-Suzuki is fast.
- GOR1 and GTOP performance is similar.
- RE2 performance is close to our leftmost greedy implementation.

One particularly interesting group of tests that show the above points are RE of the form $(a^{k_1} | \dots | a^{k_n})^{0,\infty}$ (artificial tests 1-4) and their variations with more capturing groups (artificial tests 5-8). For example, consider `(a{2}|a{3}|a{5})*` and `(((a){2})|((a){3})|((a){5}))*`. Given input string `a...a`, submatch on the last iteration varies with the length of input: it equals `aaaaa` for $5n$-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 infinitely with a period of five characters. We can increase variation period and the range of possible submatch results by choosing larger counter values. This causes increased closure size — hence the slowdown of Okui-Suzuki algorithm on tests 1 to 4 and 5 to 8 (especially pronounced for the "naive Okui-Suzuki" variation), and the more gentle slowdown of Kuklewicz algorithm on the same ranges. Adding more capturing groups increases the number of tags — hence the slowdown of Kuklewicz and Cox algorithms on 5-8 group compared to 1-4 group.

In closing, we would like to point out that correctness of all benchmarked implementations has been tested on a subset of Glenn Fowler test suite[18] (we removed tests for backreferences and start/end anchors), extended by Kuklewicz and further extended by ourselves to some 500 tests. All algorithms except Cox algorithm have passed the tests (Cox algorithm fails in about 10 cases for the reasons discussed in the introduction).

12 | CONCLUSIONS AND FUTURE WORK

The main result of our work is a practical POSIX matching algorithm that can be used on real-world regular expressions, does not require complex preprocessing and incurs relatively modest disambiguation overhead compared to other algorithms. We tried to present the algorithm in full, with a few useful variations, in order to make implementation easy for the reader.

We see a certain tradeoff between speed and memory usage: bounded-memory version of the algorithm performs a lot of redundant work, and the lazy version avoids redundant work at the expense of potentially unbounded memory usage. Both approaches seem not ideal; perhaps in practice a hybrid approach can be used.

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 match results in such order that longest-leftmost result is always first. We experimented with recursive descent parsers that embrace the same ordering idea and construsted a prototype implementation.

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

ACKNOWLEDGEMENTS

I want to thank my parents Vladimir and Elina, my teachers Tatyana Leonidovna and Demian Vladimirovich and the Belarusian State University for the love of mathematics. This work would not be possible without the help of my husband and dearest friend Sergei Trofimovich. Finally, many thanks to the RE2C users! *Ulya Trofimovich*

References

1. 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.
2. Ville Laurikari, *Efficient submatch addressing for regular expressions*, Helsinki University of Technology, 2001.
3. Chris Kuklewicz, *Regular expressions/bounded space proposal*, http://wiki.haskell.org/index.php?title=Regular_expressions/Bounded_space_proposal&oldid=11475 2007.
4. Russ Cox, backward POSIX matching algorithm (source code), https://swtch.com/~rsc/regexp/nfa-posix.y.txt 2009.
5. 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.
6. 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.
7. 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.
8. 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.
9. Ulya Trofimovich, *Tagged Deterministic Finite Automata with Lookahead*, http://re2c.org/2017_trofimovich_tagged_deterministic_finite_automata_with_lookahead.pdf, 2017.
10. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, *Introduction to algorithms*, 1st edition, MIT Press and McGraw-Hill, ISBN 0-262-03141-8.
11. 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.
12. Aaron Karper, *Efficient regular expressions that produce parse trees*, Master's thesis, University of Bern, 2014.
13. 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.
14. 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.
15. 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.
16. Aaron Karper, *Efficient regular expressions that produce parse trees*, epubli GmbH, 2014
17. POSIX standard: *POSIX-1.2008* a.k.a. *IEEE Std 1003.1-2008* The IEEE and The Open Group, 2008.
18. 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.
19. *RE2C*, lexer generator for C/C++. Website: http://re2c.org, source code: http://github.com/skvadrik/re2c.
20. *RE2*, regular expression library. Source code: http://github.com/google/re2.
21. *The GNU C library*, https://www.gnu.org/software/libc/.

APPENDIX

Proof of theorem 1

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

Proof. The proof is by induction on the length of p. Induction basis: $p = p' = \Lambda$ (the roots of t and s correspond to the root of e). Induction step: suppose that for any position p of length $|p| < h$ the lemma is true. We will show that if exists a $k \in \mathbb{N}$ such that $p.k \in Pos(t) \cap Pos(s)$, then $p.k$ corresponds to the same position $p'.k'$ in r for both t and s (for some $k' \in \mathbb{N}$). If $r|_{p'}$ is an elementary IRE of the form (i, j, ϵ) or (i, j, α), or if at least one of $t|_p$ and $s|_p$ is $\varnothing$, then k doesn't exist. Otherwise $r|_{p'}$ is a compound IRE and both $t|_p$ and $s|_p$ are not $\varnothing$. If $r|_{p'}$ is a union $(i, j, (i_1, j_1, r_1)|(i_2, r_2, j_2))$ or a product $(i, j, (i_1, j_1, r_1) \cdot (i_2, r_2, j_2))$, then both $t|_p$ and $s|_p$ have exactly two subtrees, and positions $p.1$ and $p.2$ in t and s correspond to positions $p'.1$ and $p'.2$ in r. Otherwise, $r|_{p'}$ is a repetition $(i, j, r_1^{n,m})$ and for any $k \geq 1$ position $p.k$ in t and s corresponds to position $p'.1$ in r. □

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 $\forall t, s, u \in PT(e, w) : (t < s \wedge s < u) \implies t < u$.

Let $t <_p s$ and $s <_q u$ for some positions p, q, and let $r = min(p, q)$.

First, we show that $\|t\|^{pos}_r > \|u\|^{pos}_r$. If $p \leq q$, we have $\|t\|^{pos}_p > \|s\|^{pos}_p$ (implied by $t <_p s$) and $\|s\|^{pos}_p \geq \|u\|^{pos}_p$ (implied by $s <_q u \wedge p \leq q$), therefore $\|t\|^{pos}_p > \|u\|^{pos}_p$. Otherwise $p > q$, we have $\|t\|^{pos}_q > \|u\|^{pos}_q$ (implied by $s <_q u$) and $\|t\|^{pos}_q = \|s\|^{pos}_q$ (implied by $t <_p s \wedge q < p$), therefore $\|t\|^{pos}_q > \|u\|^{pos}_q$.

Second, we show that $\forall r' < r : \|t\|^{pos}_{r'} = \|u\|^{pos}_{r'}$. We have $\|t\|^{pos}_{r'} = \|s\|^{pos}_{r'}$ (implied by $t <_p s \wedge r' < p$) and $\|s\|^{pos}_{r'} = \|u\|^{pos}_{r'}$ (implied by $s <_q u \wedge r' < q$), therefore $\|t\|^{pos}_{r'} = \|u\|^{pos}_{r'}$.

(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 $P = \{p \in Pos(t) \cup Pos(s) : \|t\|^{pos}_p \neq \|s\|^{pos}_p\}$.

(2.1) First case: $P \neq \emptyset$. 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 $p = min(P)$; it is well-defined since P is non-empty, finite and lexicographically ordered. For all $q < p$ we have $\|t\|^{pos}_q = \|s\|^{pos}_q$ (by definition of p and because $\|t\|^{pos}_q = \infty = \|s\|^{pos}_q$ if $q \notin Pos(t) \cup Pos(s)$). Since $\|t\|^{pos}_p \neq \|s\|^{pos}_p$, we have either $t <_p s$ or $t <_p s$.

Second, we show that at most one of $t < s$ or $s < t$ is true, i.e. $<$ is asymmetric: $\forall t, s \in PT(e, w) : t < s \implies s \nless t$. Suppose, on the contrary, that $t <_p s$ and $s <_q t$ for some p, q. Without loss of generality let $p \leq q$. On one hand $t <_p s$ implies $\|t\|^{pos}_p > \|s\|^{pos}_p$. But on the other hand $s <_q t \wedge p \leq q$ implies $\|t\|^{pos}_p \leq \|s\|^{pos}_p$. Contradiction.

(2.2) Second case: $P = \emptyset$. 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 $p \in Pos(t)$. If p corresponds to an inner node, then both $t|_p$ and $s|_p$ are of the form $T^i(\ldots)$. Otherwise, p corresponds to a leaf node, which can be either $\varnothing$ or ϵ or α. Since all three have different norms (-1, 0 and 1 respectively), and since $\|t\|^{pos}_p = \|s\|^{pos}_p$, it must be that $t|_p$ and $s|_p$ are identical.

□

Proof of theorem 2

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

Proof. Forward implication: let $t \sim s$ and suppose, on the contrary, that $\exists p = min\{q \mid \|t\|^{sub}_q \neq \|s\|^{sub}_q\}$, then either $t \prec_p s$ (if $\|t\|^{sub}_p > \|s\|^{sub}_p$) or $s \prec_p t$ (if $\|t\|^{sub}_p < \|s\|^{sub}_p$), both cases contradict $t \sim s$. Backward implication: $\forall p : \|t\|^{sub}_p = \|s\|^{sub}_p$ implies $\nexists p : t \prec_p s$ and $\nexists q : s \prec_q t$, 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 $t \prec_p s$ and $s \prec_q t$ for some p, q. Without loss of generality let $p \le q$. On one hand $t \prec_p s$ implies $\|t\|^{sub}_p > \|s\|^{sub}_p$. But on the other hand $s \prec_q t \wedge p \le q$ implies $\|t\|^{sub}_p \le \|s\|^{sub}_p$. Contradiction.

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

Let $t \prec_p s$ and $s \prec_q u$ for some positions p, q, and let $r = min(p, q)$.

First, we show that $\|t\|^{sub}_r > \|u\|^{sub}_r$. If $p \le q$, we have $\|t\|^{sub}_p > \|s\|^{sub}_p$ (implied by $t \prec_p s$) and $\|s\|^{sub}_p \ge \|u\|^{sub}_p$ (implied by $s \prec_q u \wedge p \le q$), therefore $\|t\|^{sub}_p > \|u\|^{sub}_p$. Otherwise $p > q$, we have $\|t\|^{sub}_q > \|u\|^{sub}_q$ (implied by $s \prec_q u$) and $\|t\|^{sub}_q = \|s\|^{sub}_q$ (implied by $t \prec_p s \wedge q < p$), therefore $\|t\|^{sub}_q > \|u\|^{sub}_q$.

Second, we show that $\forall r' < r : \|t\|^{sub}_{r'} = \|u\|^{sub}_{r'}$. We have $\|t\|^{sub}_{r'} = \|s\|^{sub}_{r'}$ (implied by $t \prec_p s \wedge r' < p$) and $\|s\|^{sub}_{r'} = \|u\|^{sub}_{r'}$ (implied by $s \prec_q u \wedge r' < q$), therefore $\|t\|^{sub}_{r'} = \|u\|^{sub}_{r'}$.

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

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

□

Proof of Theorem 3

Lemma 7 (Comparability of subtrees)**.** For a given IRE e, string w and position p, if $t, s \in PT(e, w)$, $p \in Sub(t) \cup Sub(s)$ and $\|t\|^{sub}_q = \|s\|^{sub}_q \; \forall q \le p$, then $\exists e', w' : t|_p, s|_p \in PT(e', w')$.

Proof. By induction on the length of p. Induction basis: $p = \Lambda$, let $e' = e$ and $w' = w$. Induction step: suppose that the lemma is true for any position p of length $|p| < h$, we will show that it is true for any position $p.k$ of length h ($k \in \mathbb{N}$). Assume that $p.k \in Sub(t) \cap Sub(s)$ (otherwise either $p.k \notin Sub(t) \cup Sub(s)$, or exactly one of $\|t\|^{sub}_{p.k}$, $\|s\|^{sub}_{p.k}$ is ∞ — in both cases lemma conditions are not satisfied). Then both $t|_p$ and $s|_p$ have at least one subtree: let $t|_p = T(t_1, \dots, t_n)$ and $s|_p = T(s_1, \dots, s_m)$, where $n, m \ge k$. By induction hypothesis $\exists e', w' : t|_p, s|_p \in PT(e', w')$. 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 $p.j$ for $j \le k$. By definition the set of submatch positions contains siblings, therefore $p.j \in Sub(t) \cap Sub(s)$. By lemma conditions $\|t\|^{sub}_{p.j} = \|s\|^{sub}_{p.j}$ (because $p.j \le p.k$), therefore $|str(t_1) \dots str(t_{k-1})| = \sum_{j=1}^{k-1} \|t\|^{sub}_j = \sum_{j=1}^{k-1} \|s\|^{sub}_j = |str(s_1) \dots str(s_{k-1})|$ 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 $t|_{p.k}, s|_{p.k} \in PT(r|_{p.k}, str(t_k))$ 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 the $\prec$-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 $p \in Sub(t)$ which are not proper prefixes of another position in $Sub(t)$. For each such position, $t|_p$ is itself a PT for some sub-IRE r' and substring w': $t|_p \in PT(r', w')$. Let t'_{min} be the $<$-minimal tree in $PT(r', w')$ and substitute $t|_p$ 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 p). 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 $t' \neq t_{min}$. Then $t_{min} <_p t'$ for some decision position p. It must be that $p \notin Sub(t') \cup Sub(t_{min})$, because otherwise $\|t_{min}\|^{sub}_p = \|t_{min}\|^{pos}_p > \|t'\|^{pos}_p = \|t'\|^{sub}_p$ and $\|t_{min}\|^{pos}_p = \|t'\|^{pos}_p \; \forall q < p$ implies $\|t_{min}\|^{sub}_p = \|t'\|^{sub}_p \; \forall q < p$, which means that $t_{min} \prec_p t'$, which contradicts to $t' \in T_{min}$. Thus p is a non-submatch position. Let $p = p'.p''$, where p' is the longest proper prefix of p in $Sub(u) \cup Sub(t_{min})$. For all $q \leq p'$ it must be that $\|u\|_q^{sub} = \|t_{min}\|_q^{sub}$, otherwise $\|u\|_q^{sub} \neq \|t_{min}\|_q^{sub}$ implies $\|u\|_q^{pos} \neq \|t_{min}\|_q^{pos}$, which contradicts to $t_{min} <_p t'$ because $q \leq p' < p$. By lemma 7, subtrees $t'_{p'}$ and $t_{min}|_{p'}$ are comparable: $\exists r', w' : t'|_{p'}, t_{min}|_{p'} \in PT(r', w')$. By construction of t', subtree $t'_{p'}$ is $<$-minimal in $PT(r', w')$, but at the same time $t_{min} <_{p'.p''} u$ implies $t_{min}|_{p'} <_{p''} u|_{p'}$. Contradiction. □

Proof of Theorem 4

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 $|PT(e, w)| \leq 1\ \forall w$, therefore $s = t$ and $\Phi_h(s) = \Phi_h(t)$. Induction step: height is greater than 1, therefore $s = T^d(s_1, \ldots, s_n)$ and $t = T^d(t_1, \ldots, t_m)$. If $d = 0$, then $\Phi_h(s) = str(s) = w = str(t) = \Phi_h(t)$. Otherwise $d \neq 0$. By lemma 6 we have $s \sim t \Rightarrow \|s\|_p^{sub} = \|t\|_p^{sub}\ \forall p$. This implies $n = m$ (otherwise the norm of subtree at position $min(n, m) + 1$ is ∞ for only one of s, t). Therefore $\Phi_h(s) = \langle_{h+1} \Phi_{h+1}(s_1), \ldots, \Phi_{h+1}(s_n)\rangle_h$ and $\Phi_h(t) = \langle_{h+1} \Phi_{h+1}(t_1), \ldots, \Phi_{h+1}(t_n)\rangle_h$. It suffices to show that $\forall i \leq n : \Phi_{h+1}(s_i) = \Phi_{h+1}(t_i)$. We have $\|s_i\|_p^{sub} = \|t_i\|_p^{sub}\ \forall p$ (implied by $\|s\|_p^{sub} = \|t\|_p^{sub}\ \forall p$), 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 $s \prec_p t$ and $|p| = 1$, then $\Phi_h(s) < \Phi_h(t)\ \forall h$.

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

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

Consider any $i < p$ ($i \in \mathbb{N}$). By lemma conditions $s \prec_p t$, therefore $\|s\|_q^{sub} = \|t\|_q^{sub}\ \forall q < p$, and in particular $\|s_i\|_q^{sub} = \|t_i\|_q^{sub}\ \forall q$, therefore by lemma 6 $s_i \sim t_i$, therefore by lemma 8 $\Phi_{h+1}(s_i) = \Phi_{h+1}(t_i)$. Let $traces(\Phi_h(s), \Phi_h(t)) = \big((\rho_0, \ldots, \rho_k), (\rho'_0, \ldots, \rho'_k)\big)$.

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

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

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

(2) Case $\infty > \|s\|_p^{sub} > \|t\|_p^{sub} = -1$. In this case both s_p and t_p exist, s_p is not $\varnothing$ and t_p is $\varnothing$, and fork happens immediately after $\Phi_{h+1}(s_{p-1})$, $\Phi_{h+1}(t_{p-1})$:

$$\begin{array}{rcl|ll} \Phi_h(s) & = & \langle_{h+1} \Phi_{h+1}(s_1) \ldots \Phi_{h+1}(s_{p-1}) & \langle_{h+2}\ x\ \rangle_{h+1} & \Phi_{h+1}(s_{p+1}) \ldots \Phi_{h+1}(s_n)\rangle_h \\ \Phi_h(t) & = & \langle_{h+1} \Phi_{h+1}(t_1) \ldots \Phi_{h+1}(t_{p-1}) & \Diamond_{h+1} & \Phi_{h+1}(t_{p+1}) \ldots \Phi_{h+1}(t_m)\rangle_h \end{array}$$

(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 $\rho_j = \rho'_j = h$.

(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 $p = 1$ and s_{p-1} does not exist, or else one of $\rangle_{h+1}$ and $\Diamond_{h+1}$), therefore $\rho_j = \rho'_j = h{+}1$. For subsequent frames i such that $j < i < k$ we have $\rho_i = \rho'_i = h + 1$ (on one hand $\rho_i, \rho'_i \leq h + 1$ because $\rho_j = \rho'_j = h + 1$, but on the other hand $minh(\gamma_i)$, $minh(\delta_i) \geq h + 1$). For the last pair of frames we have $\rho_k = \rho'_k = h$ (they both contain the closing parenthesis $\rangle_h$).

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

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

$$\begin{array}{rcl|l} \Phi_h(s) & = & \langle_{h+1}\Phi_{h+1}(s_1)\ldots\Phi_{h+1}(s_{p-1})\ \langle_{h+2}\ x & y\ \rangle_{h+1}\ \Phi_{h+1}(s_{p+1})\ldots\Phi_{h+1}(s_n)\rangle_h \\ \Phi_h(t) & = & \langle_{h+1}\Phi_{h+1}(t_1)\ldots\Phi_{h+1}(t_{p-1})\ \langle_{h+2}\ x & z\ \rangle_{h+1}\ \Phi_{h+1}(t_{p+1})\ldots\Phi_{h+1}(t_m)\rangle_h \end{array}$$

From $\|s\|^{sub}_p > \|t\|^{sub}_p \geq 0$ it follows that s_p contains more alphabet symbols than t_p. Consequently $\Phi_{h+1}(s_p)$ contains more alphabet symbols, and thus spans more frames than $\Phi_{h+1}(t_p)$. Let l be the index of the frame δ_l that contains the closing parenthesis $\rangle_{h+1}$ of $\Phi_{h+1}(t_p)$. By the above reasoning $\Phi_{h+1}(s_p)$ 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 $minh(x)$, $minh(y)$, $minh(z) \geq h+2$, therefore $lasth(\beta_j) \geq h+2$ (including the case when x is empty), and for all frames i such that $j \leq i < l$ (if any) we have $\rho_i, \rho'_i \geq h+2$. Consequently, for l-th frame we have $\rho_l \geq h+2$ and $\rho'_l = h+1$, thus $\rho_l > \rho'_l$. For subsequent frames i such that $l < i < k$ we have $minh(\gamma_i)$, $minh(\delta_i) \geq h+1$, therefore $\rho_i \geq h+1$ and $\rho'_i = h+1$, thus $\rho_i \geq \rho'_i$. For the last pair of frames we have $\rho_k = \rho'_k = h$, 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 $s, t \in PT(r, w)$ for some IRE r and string w. If $s \prec_p t$, then $\Phi_h(s) < \Phi_h(t)\ \forall h$.

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

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

Consider any $i < p'$. By lemma conditions $s \prec_p t$, therefore $\|s\|^{sub}_q = \|t\|^{sub}_q\ \forall q < p$, and in particular $\|s_i\|^{sub}_q = \|t_i\|^{sub}_q\ \forall q$, therefore by lemma 6 $s_i \sim t_i$, therefore by lemma 8 $\Phi_{h+1}(s_i) = \Phi_{h+1}(t_i)$. Since $p' < p$ we have $\|s\|^{sub}_q = \|t\|^{sub}_q\ \forall q \leq p'$ and by lemma 7 $\exists e', w' : s', t' \in PT(e', w')$. Since $\|s'\|^{sub}_q = \|s\|^{sub}_{p'.q}\ \forall q$ and $\|t'\|^{sub}_q = \|t\|^{sub}_{p'.q}\ \forall q$, we have $s' \prec_{p''} t'$. Since $|p''| < |p|$ by induction hypothesis we have $\Phi_{h+1}(s') < \Phi_{h+1}(t')$. If j is the fork and $f \leq j \leq k$, then $\Phi_h(s)$, $\Phi_h(t)$ can be represented as:

$$\begin{array}{rl|l} \Phi_h(s) & = \beta_0 a_1 \ldots a_f \beta^1_f\ \overbrace{\beta^2_f a_{f+1} \ldots a_j \beta_j} & \overbrace{\gamma_j a_{j+1} \ldots a_k \gamma^1_k}^{\Phi_{h+1}(s')}\ \gamma^2_k a_{k+1} \ldots a_l \gamma_l \\ \Phi_h(t) & = \beta_0 a_1 \ldots a_f \beta^1_f\ \underbrace{\beta^2_f a_{f+1} \ldots a_j \beta_j} & \underbrace{\delta_j a_{j+1} \ldots a_k \delta^1_k}_{\Phi_{h+1}(t')}\ \delta^2_k a_{k+1} \ldots a_l \delta_l \end{array}$$

Let $traces(\Phi_h(s), \Phi_h(t)) = \big((\rho_0, \ldots, \rho_l), (\rho'_0, \ldots, \rho'_l)\big)$ and $traces(\Phi_{h+1}(s'), \Phi_{h+1}(t')) = \big((\sigma_h, \ldots, \sigma_k), (\sigma'_h, \ldots, \sigma'_k)\big)$. We show that for frames i such that $j \leq i < k$ we have $\rho_i = \sigma_i \wedge \rho'_i = \sigma'_i$ and for subsequent frames $k \leq i \leq l$ we have $\rho_i = \rho'_i$.

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

(2) Case $j < i < k \leq l$. The computation of ρ_i, ρ'_i depends only on ρ_j, ρ'_j, or which we have shown $\rho_j = \sigma_j \wedge \rho'_j = \sigma'_j$ in case (1), and on $\Phi_{h+1}(s')$, $\Phi_{h+1}(t')$, which are not changed by appending γ^2_k and δ^2_k since $i < k$. Therefore $\rho_i = \sigma_i \wedge \rho'_i = \sigma'_i$.

(3) Case $j \leq i = k < l$. We have $minh(\gamma^1_k) = minh(\delta^1_k) = h+1$ and $minh(\gamma^2_k) = minh(\delta^2_k) \geq h+1$. None of the preceding frames after the fork contain parentheses with height less than $h+1$, therefore $\rho_k = \rho'_k = h+1$.

(4) Case $j \leq k < i < l$. We have $\rho_i = \rho'_i = h+1$, because $\rho_k = \rho'_k = h+1$ and $minh(\gamma_i)$, $minh(\delta_i) \geq h+1$.

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

We have shown that $\rho_i = \sigma_i \wedge \rho'_i = \sigma'_i \; \forall i : j \leq i < k$ and $\rho_i = \rho'_i \; \forall i : k \leq i \leq l$. 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 4. 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 $s \nprec t$. 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 $s \nprec t$ 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 $\big((\rho_1, \dots, \rho_n), (\rho'_1, \dots, \rho'_n)\big) = traces(\alpha, \beta)$ where $n \geq 1$. 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 $\big((\rho_1, \dots, \rho_n, \rho_{n+1}), (\rho'_1, \dots, \rho'_n, \rho'_{n+1})\big)$.

(1) Case $\alpha \sim \beta \wedge \alpha \subset \beta$. In this case $\rho_i = \rho'_i \; \forall i \leq n$, therefore $\rho_{n+1} = min(\rho_n, minh(\gamma)) = min(\rho'_n, minh(\gamma)) = \rho'_{n+1}$ 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 $\exists j \leq n$ such that $\rho_j > \rho'_j$ and $\rho_i = \rho'_i \; \forall j < i \leq n$. We show that $\exists l \leq n+1$ such that $\rho_l > \rho'_l$ and $\rho_i = \rho'_i \; \forall l < i \leq n+1$, which by definition means that $\alpha c \gamma \sqsubset \beta c \gamma$.

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

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

(2c) Case $j = n$ and $minh(\gamma) \leq \rho'_n$. In this case $\rho_n > \rho'_n$ and we have $\rho_{n+1} = min(\rho_n, minh(\gamma)) = minh(\gamma)$ and $\rho'_{n+1} = min(\rho'_n, minh(\gamma)) = minh(\gamma)$, therefore $\rho_{n+1} = \rho'_{n+1}$ 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 $\pi_1 = q_0 \overset{u|\alpha}{\rightsquigarrow} q$, $\pi_2 = q \overset{\epsilon|\beta}{\rightsquigarrow} q$ is the last tagged ϵ-loop on π and $\pi_3 = q \overset{v|\gamma}{\rightsquigarrow} q_f$. 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 $\big((\rho_1, \dots, \rho_n), (\rho'_1, \dots, \rho'_n)\big) = traces(\alpha\beta\gamma, \alpha\gamma)$ 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:

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

Since x_i is an ϵ-loop, it is contained in the fork frame of $\alpha\beta\gamma$. We have $minh(\beta) = h$ and $minh(\gamma) \leq h - 1$, therefore $\rho_k = \rho'_k \leq h-1$. Subsequent frames $l > k$ (if any) are identical and thus $\rho_l = \rho'_l$. 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 $\alpha\gamma$:

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

Here $y_1 y_2 = x_i$ and $y_1 y_3 = x_{i+1}$ (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 $\rho_k = h$. 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 $\rho'_k > h$. All subsequent frames $l > k$ are identical: if they contain parentheses of height less than h, then $\rho_l = \rho'_l < h$; otherwise $\rho_l \leq h$ and $\rho'_l > h$. 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 $\pi_1 = q_0 \overset{s|\alpha}{\rightsquigarrow} q_1$ and $\pi_2 = q_1 \overset{\epsilon|\gamma}{\rightsquigarrow} q_1$ 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 $\pi'_1 = q_0 \overset{s|\alpha'}{\rightsquigarrow} q_1$ 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 $\big((\rho_1, \dots, \rho_k), (\rho'_1, \dots, \rho'_k)\big) = traces(\alpha', \alpha\gamma)$.

(1) Case $\alpha' = \alpha$. Because α is a proper prefix of $\alpha\gamma$, fork happens at the last frame and we have $\rho_k = lasth(\alpha)$ and $\rho'_k = min(lasth(\alpha), minh(\gamma))$. If $lasth(\alpha) > minh(\gamma)$, then $\rho_k > \rho'_k$ and $\alpha \sqsubset \alpha\gamma$. Otherwise $\rho_k = \rho'_k$ and $\alpha \sim \alpha\gamma$, and we have $first(\alpha \backslash \alpha\gamma) = \bot$ and $first(\alpha\gamma \backslash \alpha) \neq \bot$, 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 $\rho_k = \sigma_k$ and $\rho'_k = min(\sigma'_k, minh(\gamma)) \leq \sigma_k$. If $minh(\gamma) < \sigma'_k$ then $\rho_k > \rho'_k$ and $\alpha' \sqsubset \alpha\gamma$. Otherwise $\rho'_k = \sigma'_k$. 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 $\pi_\alpha = q_0 \overset{u|\alpha}{\rightsquigarrow} q_1$ and $\pi_\beta = q_0 \overset{u|\beta}{\rightsquigarrow} q_1$ be ambiguous paths in TNFA f for IRE e, and let $\pi_\gamma = q_1 \overset{\epsilon|\gamma}{\rightsquigarrow} q_2$ 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 $\big((\rho_1, \dots, \rho_k), (\rho'_1, \dots, \rho'_k)\big) = traces(\alpha, \beta)$ 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 $\rho_i = \sigma_i$ and $\rho'_i = \sigma'_i$. 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 $\rho_i = \rho'_i \;\forall i$ (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 $\exists j \leq k$ such that $\rho_j > \rho'_j$ and $\rho_i = \rho'_i \;\forall i > j$. We show that $\alpha\gamma \sqsubset \beta\gamma$.
   - (2a) Case $j < k$. In this case $\rho_k = \rho'_k$ and appending γ does not change relation on the last frame: $\sigma_k = min(\rho_k, minh(\gamma)) = min(\rho'_k, minh(\gamma)) = \sigma'_k$. Since $\sigma_i = \rho_i$ and $\sigma'_i = \rho'_i$ for all preceding frames $i < k$, we have $\alpha\gamma \sqsubset \beta\gamma$.
   - (2b) Case $j = k$ and $minh(\gamma) > \rho'_k$. In this case $\rho_k > \rho'_k$ and again appending γ does not change relation on the last frame: $\sigma_k = min(\rho_k, minh(\gamma)) > \rho'_k$ and $\sigma'_k = min(\rho'_k, minh(\gamma)) = \rho'_k$, therefore $\sigma_k > \sigma'_k$. Therefore $\alpha\gamma \sqsubset \beta\gamma$.
   - (2c) Case $j = k$ and $minh(\gamma) \leq \rho'_k$ and $\exists l < k$ such that $\rho_l > \rho'_l$ and $\rho_i = \rho'_i$ 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: $\sigma_l = \rho_l > \rho'_l = \sigma'_l$. Therefore $\alpha\gamma \sqsubset \beta\gamma$.

(2d) Case $j = k$ and $minh(\gamma) \leq \rho'_k$ and $\nexists l < k$ such that $\rho_l > \rho'_l$ and $\rho_i = \rho'_i$ 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 ρ_k. By TNFA construction this means that π'_α is contained in a sub-TNFA f' for $e|_p$ at some position p with length $|p| = \rho_k$. 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 ρ_k: by conditions of (2d) either $k = 1$ and there are no such frames, or $\rho'_{k-1} \geq \rho_{k-1}$ which implies $\rho'_{k-1} \geq \rho_k$ (because by definition $\rho_k = min(\rho_{k-1}, minh(\alpha_k)) \leq \rho_{k-1}$). On the last frame of π'_β minimal height is $\rho'_k < \rho_k$. 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 ρ_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.) $\square$

$\square$