# Optimally Streaming
# Greedy Regular Expression Parsing[⋆]

Niels Bjørn Bugge Grathwohl, Fritz Henglein, and Ulrik Terp Rasmussen

Department of Computer Science, University of Copenhagen (DIKU), Denmark

**Abstract.** We study the problem of *streaming* regular expression parsing: Given a regular expression and an input stream of symbols, how to output a serialized syntax tree representation as an output stream *during* input stream processing.

We show that *optimally streaming* regular expression parsing, outputting bits of the output as early as is semantically possible for any regular expression of size $m$ and any input string of length $n$, can be performed in time $O(2^{m \log m} + mn)$ on a unit-cost random-access machine. This is for the wide-spread *greedy* disambiguation strategy for choosing parse trees of grammatically ambiguous regular expressions. In particular, for a fixed regular expression, the algorithm's run-time scales linearly with the input string length. The exponential is due to the need for preprocessing the regular expression to analyze state coverage of its associated NFA, a PSPACE-hard problem, and tabulating all reachable *ordered* sets of NFA-states.

Previous regular expression parsing algorithms operate in multiple phases, *always* requiring processing or storing the whole input string before outputting the first bit of output, not only for those regular expressions and input prefixes where reading to the end of the input is strictly *necessary*.

## 1 Introduction

In programming, regular expressions are often used to extract information from an input, which requires an intensional interpretation of regular expressions as denoting parse trees, and not just their ordinary language-theoretic interpretation as denoting strings.

This is a nontrivial change of perspective. We need to deal with grammatical ambiguity—*which* parse tree to return, not just that it has one—and memory requirements become a critical factor: Deciding whether a string belongs to the language denoted by $(ab)^\star + (a + b)^\star$ can be done in constant space, but outputting the first bit, whether the string matches the first alternative or only the second, may require buffering the whole input string. This is an instructive

case of deliberate grammatical ambiguity to be resolved by the prefer-the-left-alternative policy of greedy disambiguation: Try to match the left alternative; if that fails, return a match according to the right alternative as a fallback. Straight-forward application of automata-theoretic techniques does not help: $(ab)^\star + (a + b)^\star$ denotes the same *language* as $(a + b)^\star$, which is unambiguous and corresponds to a small DFA, but is also useless: it doesn't represent any more when a string consists of a sequence of *ab*-groups.

Previous parsing algorithms [9,3,5,10,13,6] require at least one full pass over the input string before outputting any output bits representing the parse tree. This is the case even for regular expressions requiring only bounded lookahead such as one-unambiguous regular expressions [1].

In this paper we study the problem of *optimally streaming* parsing. Consider $(ab)^\star + (a + b)^\star$, which is ambiguous and in general requires unbounded input buffering, and consider the particular input string `ab`...`abaababababab`.... An *optimally* streaming parsing algorithm needs to buffer the prefix `ab`...`ab` in some form because the complete parse might match either of the two alternatives in the regular expression, but once encountering `aa`, only the right alternative is possible. At this point it outputs this information and the output representation for the buffered string as parsed by the second alternative. After this, it outputs a bit for each input symbol read, with no internal buffering: input symbols are discarded before reading the next symbol. Optimality means that output bits representing the eventual parse tree must be produced *earliest possible*: as soon as they are semantically determined by the input processed so far under the assumption that the parse will succeed.

*Outline.* In Section 2 we recall the *type interpretation* of regular expressions, where a regular expression denotes parse trees, along with the *bit-coding* of parse trees.

In Section 3 we introduce a class of Thompson-style augmented nondeterministic finite automata (aNFAs). Paths in such an aNFA naturally represent *complete* parse trees, and paths to intermediate states represent *partial* parse trees for prefixes of an input string.

We recall the greedy disambiguation strategy in Section 4, which specifies a deterministic mapping of accepted strings to NFA-paths.

Section 5 contains a definition of what it means to be an optimally streaming implementation of a parsing function.

We define what it means for a set of aNFA-states to *cover* another state in Section 6, which constitutes the computationally hardest part needed in our algorithm.

Section 7 contains the main results. We present *path trees* as a way of organizing partial parse trees, and based on these we present our algorithm for an optimally streaming parsing function and analyze its asymptotic run-time complexity.

Finally, in Section 8, the algorithm is demonstrated by illustrative examples alluding to its expressive power and practical utility.

## 2   Preliminaries

In the following section, we recall definitions of regular expressions and their interpretation as types [10].

**Definition 1 (Regular Expression).** *A regular expression (RE) over a finite alphabet $\Sigma$ is an expression $E$ generated by the grammar*

$$E ::= \mathbf{0} \mid \mathbf{1} \mid a \mid E_1 E_2 \mid E_1 + E_2 \mid E_1^\star$$

*where $a \in \Sigma$.*

Concatenation (juxtaposition) and alternation (+) associates to the right; parentheses may be inserted to override associativity. Kleene star ($\star$) binds tightest, followed by concatenation and alternation.

The standard interpretation of regular expressions is as descriptions of regular languages.

**Definition 2 (Language Interpretation).** *Every RE $E$ denotes a language $\mathcal{L}[\![E]\!] \subseteq \Sigma^\star$ given as follows:*

$$\mathcal{L}[\![\mathbf{0}]\!] = \emptyset \qquad \mathcal{L}[\![E_1 E_2]\!] = \mathcal{L}[\![E_1]\!]\mathcal{L}[\![E_2]\!] \qquad \mathcal{L}[\![a]\!] = \{a\}$$
$$\mathcal{L}[\![\mathbf{1}]\!] = \{\epsilon\} \qquad \mathcal{L}[\![E_1 + E_2]\!] = \mathcal{L}[\![E_1]\!] \cup \mathcal{L}[\![E_2]\!] \qquad \mathcal{L}[\![E_1^\star]\!] = \bigcup_{n \geq 0} \mathcal{L}[\![E_1]\!]^n$$

*where we have $A_1 A_2 = \{w_1 w_2 \mid w_1 \in A_1, w_2 \in A_2\}$, and $A^0 = \{\epsilon\}$ and $A^{n+1} = AA^n$.*

Proviso: Henceforth we shall restrict ourselves to REs $E$ such that $\mathcal{L}[\![E]\!] \neq \emptyset$.

For regular expression parsing, we consider an alternative interpretation of regular expressions as types.

**Definition 3 (Type Interpretation).** *Let the syntax of* values *be given by*

$$v ::= () \mid \mathsf{inl}\ v_1 \mid \mathsf{inr}\ v_1 \mid \langle v_1, v_2 \rangle \mid [v_1, v_2, ..., v_n]$$

*Every RE $E$ can be seen as a* type *describing a set $\mathcal{T}[\![E]\!]$ of well-typed values:*

$$\mathcal{T}[\![\mathbf{0}]\!] = \emptyset \qquad \mathcal{T}[\![E_1 E_2]\!] = \{\langle v_1, v_2 \rangle \mid v_1 \in \mathcal{T}[\![E_1]\!], v_2 \in \mathcal{T}[\![E_2]\!]\}$$
$$\mathcal{T}[\![\mathbf{1}]\!] = \{()\} \qquad \mathcal{T}[\![E_1 + E_2]\!] = \{\mathsf{inl}\ v \mid v \in \mathcal{T}[\![E_1]\!]\} \cup \{\mathsf{inr}\ v \mid v \in \mathcal{T}[\![E_2]\!]\}$$
$$\mathcal{T}[\![a]\!] = \{a\} \qquad \mathcal{T}[\![E_1^\star]\!] = \{[v_1, \ldots, v_n] \mid n \geq 0 \wedge \forall 1 \leq i \leq n.v_i \in \mathcal{T}[\![E_1]\!]\}$$

We write $|v|$ for the *flattening* of a value, defined as the word obtained by doing an in-order traversal of $v$ and writing down all the symbols in the order they are visited. We write $\mathcal{T}_w[\![E]\!]$ for the restricted set $\{v \in \mathcal{T}[\![E]\!] \mid |v| = w\}$. Regular expression *parsing* is a generalization of the acceptance problem of determining whether a word $w$ belongs to the language of some RE $E$, where additionally we produce a parse tree from $\mathcal{T}_w[\![E]\!]$. We say that an RE $E$ is *ambiguous* iff there exists a $w$ such that $|\mathcal{T}_w[\![E]\!]| > 1$.

Any well-typed value can be serialized into a sequence of bits.

**Definition 4 (Bit-Coding).** *Given a value $v \in \mathcal{T}[\![E]\!]$, we denote its bit-code by $\ulcorner v \urcorner \subseteq \{0,1\}^\star$, defined as follows:*

$$\ulcorner () \urcorner = \epsilon \qquad\qquad \ulcorner a \urcorner = \epsilon \qquad\qquad \ulcorner \mathsf{inl}\ v \urcorner = 0 \ulcorner v \urcorner$$
$$\ulcorner \langle v_1, v_2 \rangle \urcorner = \ulcorner v_1 \urcorner \ulcorner v_2 \urcorner \qquad \ulcorner [v_1, ..., v_n] \urcorner = 0 \ulcorner v_1 \urcorner ... 0 \ulcorner v_n \urcorner 1 \qquad \ulcorner \mathsf{inr}\ v \urcorner = 1 \ulcorner v \urcorner$$

We write $\mathcal{B}[\![E]\!]$ for the set $\{\ulcorner v \urcorner \mid v \in \mathcal{T}[\![E]\!]\}$ and $\mathcal{B}_w[\![E]\!]$ for the set restricted to bit-codes for values with a flattening $w$. Note that for any RE $E$, bit-coding is an isomorphism when seen as a function $\ulcorner \cdot \urcorner_E : \mathcal{T}[\![E]\!] \to \mathcal{B}[\![E]\!]$.

# 3 Augmented Automata

In this section we recall from [6] the construction of finite automata from regular expressions. Our construction is similar to that of Thompson [15], but augmented with extra annotations on non-deterministic $\epsilon$-transitions. The resulting automata can be seen as non-deterministic transducers which for each accepted input string in the language of the underlying regular expression outputs the bit-codes for the corresponding parse trees.

**Definition 5 (Augmented Non-deterministic Finite Automaton).** *An augmented non-deterministic finite automaton (aNFA) is a tuple $(\mathsf{State}, \delta, q^{\mathsf{in}}, q^{\mathsf{fin}})$, where $\mathsf{State}$ is a finite set of* states, $q^{\mathsf{in}}, q^{\mathsf{fin}} \in \mathsf{State}$ *are* initial *and* final *states, respectively, and $\delta \subseteq \mathsf{State} \times \Gamma \times \mathsf{State}$ is a labeled transition relation with labels $\Gamma = \Sigma \uplus \{0, 1, \epsilon\}$.*

Transition labels are divided into the disjoint sets $\Sigma$ (symbol labels); $\{0, 1\}$ (bit-labels); and $\{\epsilon\}$ ($\epsilon$-labels). $\Sigma$-transitions can be seen as input actions, and bit-transitions as output actions.

**Definition 6 (aNFA construction).** *Let $E$ be an RE and define an aNFA $M_E = (\mathsf{State}_E, \delta_E, q_E^{\mathsf{in}}, q_E^{\mathsf{fin}})$ by induction on $E$. We give the definition diagrammatically by cases:*
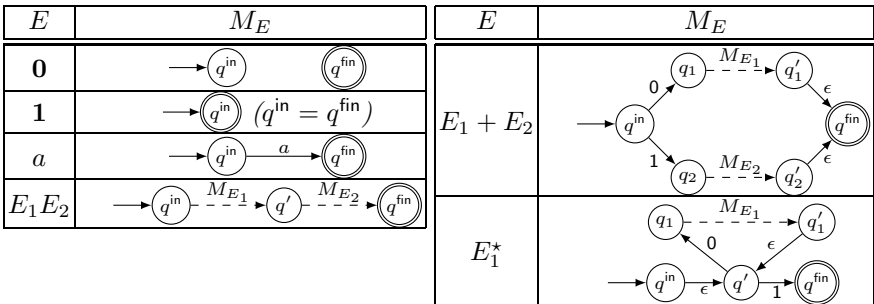


In the above, the notation $(q_1)\text{-}\text{-}^M\text{-}\blacktriangleright(q_2)$ means that $q_1, q_2$ are initial and final states, respectively, in some (sub-)automaton $M$.

See Figure 1 for an example.

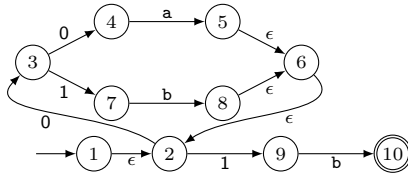**Fig. 1.** Example automaton for the RE $(\mathtt{a} + \mathtt{b})^{\star}\mathtt{b}$

**Definition 7 (Path).** *A* path *in an aNFA is a finite non-empty sequence* $\alpha \in \mathsf{State}^{\star}$ *of the form* $\alpha = p_0\,p_1\ldots p_{n-1}$ *such that for each* $i < n$, *we have* $(p_i, \gamma_i, p_{i+1}) \in \delta_E$ *for some* $\gamma_i$. *As a shorthand for this fact we might write* $p_0 \overset{\alpha}{\rightsquigarrow} p_{n-1}$ *(note that a single state is a path to itself).*

Each path $\alpha$ is associated with a (possibly empty) sequence of labels $\mathsf{lab}(\alpha)$: we let $\mathsf{read}(\alpha)$ and $\mathsf{write}(\alpha)$ refer to the corresponding subsequences of $\mathsf{lab}(\alpha)$ filtered by $\Sigma$ and $\{0, 1\}$, respectively. An automaton *accepts* a word $w$ iff $q^{\mathsf{in}} \overset{\alpha}{\rightsquigarrow} q^{\mathsf{fin}}$ for some $\alpha$ where $\mathsf{read}(\alpha) = w$. There is a one-to-one correspondence between bit-codes and accepting paths:

**Proposition 1.** *For any RE $E$ with aNFA $M_E$, we have for each $w \in \mathcal{L}[\![E]\!]$ that*

$$\{\mathsf{write}(\alpha) \mid q^{\mathsf{in}} \overset{\alpha}{\rightsquigarrow} q^{\mathsf{fin}}, \mathsf{read}(\alpha) = w\} = \mathcal{B}_w[\![E]\!].$$

*Determinization.* Given a state set $Q$, define its *closure* as the set $\mathsf{closure}(Q) = \{q' \mid q \in Q \land \exists \alpha.\mathsf{read}(\alpha) = \epsilon \land q \overset{\alpha}{\rightsquigarrow} q'\}$. For any aNFA $M = (\mathsf{State}, \delta, q^{\mathsf{in}}, q^{\mathsf{fin}})$, let $D(M) = (\mathsf{DState}_M, I_M, F_M, \Delta_M)$ be the deterministic automaton obtained by applying the standard subset construction: Here, $I_M = \mathsf{closure}(\{q^{\mathsf{in}}\})$ is the *initial state*, and $\mathsf{DState}_M \subseteq 2^{\mathsf{State}}$ is the set of states, defined to be the smallest set containing $I_M$ and closed under the transition function $\Delta_M(Q, a) = \mathsf{closure}(\{q' \mid (q, a, q') \in \delta, q \in Q\})$. The set of *final states* $F_M$ is the set $\{Q \in \mathsf{DState}_M \mid q^{\mathsf{fin}} \in Q\}$.

## 4   Disambiguation

A regular expression parsing algorithm has to produce a parse tree for an input word whenever the word is in the language for the underlying RE. In the case of ambiguous REs, the algorithm has to choose one of several candidates. We do not want the choice to be arbitrary, but rather a parse tree which is uniquely identified by a *disambiguation policy*. Since there is a one-to-one correspondence between words in the language of an RE $E$ and accepting paths in $M_E$, a disambiguation policy can be seen as a deterministic choice between aNFA paths recognizing the same string.

We will focus on greedy disambiguation, which corresponds to choosing the first result that would have been found by a backtracking regular expression parsing algorithm such as the one found in the Perl programming language [16].

The greedy strategy has successfully been implemented in previous work [5,6], and is simpler to define and implement than other strategies such as POSIX [8,4] whose known parsing algorithms are technically more complicated [11,13,14].

Greedy disambiguation can be seen as picking the accepting path with the lexicographically least bitcode. A well-known problem with backtracking parsing is non-termination in the case of regular expressions with nullable subexpressions under Kleene star, which means that the lexicographically least path is not always well-defined. This problem can easily be solved by not considering paths with non-productive loops, as in [5].

# 5   Optimal Streaming

In this section we specify what it means to be an *optimally streaming* implementation of a function from sequences to sequences.

We write $w \sqsubseteq w''$ if $w$ is a *prefix* of $w''$, that is $ww' = w''$ for some $w'$. Note that $\sqsubseteq$ is a partial order with greatest lower bounds for nonempty sets: $\bigsqcap L = w$ if $w \sqsubseteq w''$ for all $w'' \in L$ and $\forall w'.(\forall w'' \in S.w' \sqsubseteq w'') \Rightarrow w' \sqsubseteq w$. $\bigsqcap L$ is the longest common prefix of all words in $L$.

**Definition 8 (Completions).** *The set of* completions $C_E(w)$ *of $w$ in $E$ is the set of all words in $\mathcal{L}[\![E]\!]$ that have $w$ as a prefix:*

$$C_E(w) = \{w'' \mid w \sqsubseteq w'' \wedge w'' \in \mathcal{L}[\![E]\!]\}.$$

Note that $C_E(w)$ may be empty.

**Definition 9 (Extension).** *For nonempty $C_E(w)$ the unique* extension $\hat{w}_E$ *of $w$ under $E$ is the longest extension of $w$ with a suffix such that all successful extensions of $w$ to an element of $\mathcal{L}[\![E]\!]$ are also extensions of $\hat{w}$:*

$$\hat{w}_E = \bigsqcap C_E(w).$$

*Word $w$ is* extended *under $E$ if $w = \hat{w}$; otherwise it is* unextended.

Extension is a closure operation: $\hat{\hat{w}} = \hat{w}$; in particular, extensions are extended.

**Definition 10 (Reduction).** *For empty $C_E(w)$ the unique* reduction $\bar{w}_E$ *of $w$ under $E$ is the longest prefix $w'$ of $w$ such that $C_E(w') \neq \emptyset$.*

Given parse function $\mathsf{P}_E(\cdot) : \mathcal{L}[\![E]\!] \to \mathcal{B}[\![E]\!]$ for complete input strings, we can now define what it means for an implementation of it to be optimally streaming:

**Definition 11 (Optimally Streaming).** *The* optimally streaming *function corresponding to $\mathsf{P}_E(\cdot)$ is*

$$O_E(w) = \begin{cases} \bigsqcap\{\mathsf{P}_E(w'') \mid w'' \in C_E(w)\} & \text{if } C_E(w) \neq \emptyset \\ (\bigsqcap O_E(\bar{w}))\sharp & \text{if } C_E(w) = \emptyset. \end{cases}$$

The first condition expresses that after seeing prefix $w$ the function must output *all* bits that are a common prefix of all bit-coded parse trees of words in $\mathcal{L}[\![E]\!]$ that $w$ can be extended to. The second condition expresses that as soon as it is clear that a prefix has no extension to an element of $\mathcal{L}[\![E]\!]$, an indicator $\sharp$ of failure must be emitted, with no further output after that. In this sense $O_E$ is *optimally* streaming: It produces output bits at the semantically earliest possible time during input processing.

It is easy to check that $O_E$ is a streaming function:

$$w \sqsubseteq w' \Rightarrow O_E(w) \sqsubseteq O_E(w')$$

The definition has the, at first glance, surprising consequence that $O_E$ may output bits for parts of the input it has not even read yet:

**Proposition 2.** $O_E(w) = O_E(\hat{w})$

E.g. for $E = (\mathsf{a} + \mathsf{a})(\mathsf{a} + \mathsf{a})$ we have $O_E(\epsilon) = 00$; that is, $O_E$ outputs $00$ off the bat, before reading any input symbols, in anticipation of $\mathsf{aa}$ being the only possible successful extension. Assume the input is $\mathsf{ab}$. After reading $\mathsf{a}$ it does not output anything, and after reading $\mathsf{b}$ it outputs $\sharp$ to indicate a failed parse, the total output being $00\sharp$.

# 6    Coverage

Our algorithm is based on simulating aNFAs in lock-step, maintaining a set of partial paths reading the prefix $w$ of the input that has been consumed so far. In order to be optimally streaming, we have to identify partial paths which are guaranteed not to be a prefixes of a greedy parse for a word in $C_E(w)$.

In this section, we define a *coverage relation* which our parsing algorithm relies on in order to detect the aforementioned situation. In the following, fix an RE $E$ and its aNFA $M_E = (\mathsf{State}_E, \delta_E, q_E^{\mathsf{in}}, q_E^{\mathsf{fin}})$.

**Definition 12 (Coverage).** *Let $p \in \mathsf{State}_E$ be a state and $Q \subseteq \mathsf{State}_E$ a state set. We say that $Q$ covers $p$, written $Q \sqsupseteq p$, iff*

$$\{\mathsf{read}(\alpha) \mid q \overset{\alpha}{\leadsto} q^{\mathsf{fin}}, q \in Q\} \supseteq \{\mathsf{read}(\beta) \mid p \overset{\beta}{\leadsto} q^{\mathsf{fin}}\} \tag{1}$$

Coverage can be seen as a slight generalization of language inclusion. That is, if $Q \sqsupseteq p$, then every word suffix read by a path from $p$ to the final state can also be read by a path from one of the states in $Q$ to the final state.

Let $\overline{M_e}$ refer to the automaton obtained by reversing the direction of all transitions and swapping the initial and final states. It can easily be verified that if (1) holds for some $Q, p$, then the following property also holds in the *reverse* automaton $\overline{M_E}$:

$$\{\mathsf{read}(\alpha) \mid q^{\mathsf{in}} \overset{\alpha}{\leadsto} q, q \in Q\} \supseteq \{\mathsf{read}(\beta) \mid q^{\mathsf{in}} \overset{\alpha}{\leadsto} p\} \tag{2}$$

If we consider $D(\overline{M_E})$, the deterministic automaton generated from $\overline{M_E}$, then we see that (2) is satisfied iff

$$\forall S \in \mathsf{DState}_{\overline{M_E}}.\ p \in S \Rightarrow Q \cap S \neq \emptyset \qquad (3)$$

This is true since a DFA state $S$ is reachable by reading a word $w$ in $D(\overline{M_E})$ iff every $q \in S$ is reachable by reading $w$ in $\overline{M_E}$. Since a DFA accepts the same language as the underlying aNFA, this implies that condition (2) must hold iff $Q$ has a non-empty intersection with *all* DFA states containing $p$.

The equivalence of (1) and (3) gives us a method to decide $\sqsupseteq$ in an aNFA $M$, provided that we have computed $D(\overline{M})$ beforehand. Checking (3) for a particular $Q$ and $p$ can be done by intersecting all states of $\mathsf{DState}_{\overline{M_E}}$ with $Q$, using time $O(|Q||\mathsf{DState}_{\overline{M_E}}|) = O(|Q|2^{O(m)})$, where $m$ is the size of the RE $E$.

The exponential cost appears to be unavoidable – the problem of deciding coverage is inherently hard to compute:

**Proposition 3.** *The problem of deciding coverage, that is the set $\{(E, Q, p) \mid Q \subseteq \mathsf{State}_E \wedge Q \sqsupseteq p\}$, is PSPACE-hard.*

*Proof.* We can reduce regular expression equivalence to coverage: Given regular expressions $E$ and $F$, produce an aNFA $M_{E+F}$ for $E + F$ and observe that $M_E$ and $M_F$ are subautomata. Now observe that there is a path $q^{\mathsf{in}}_{E+F} \overset{\alpha}{\leadsto} q^{\mathsf{fin}}_E$ (respectively $q^{\mathsf{in}}_{E+F} \overset{\beta}{\leadsto} q^{\mathsf{fin}}_F$) in $M_{E+F}$ iff there is a path $q^{\mathsf{in}}_E \overset{\alpha'}{\leadsto} q^{\mathsf{fin}}_E$ with $\mathsf{read}(\alpha) = \mathsf{read}(\alpha')$ in $M_E$ (respectively $q^{\mathsf{in}}_F \overset{\beta'}{\leadsto} q^{\mathsf{fin}}_F$ with $\mathsf{read}(\beta) = \mathsf{read}(\beta')$ in $M_F$). Hence, we have $\{q^{\mathsf{in}}_F\} \sqsupseteq q^{\mathsf{in}}_E$ in $M_{E+F}$ iff $\mathcal{L}[\![E]\!] \subseteq \mathcal{L}[\![F]\!]$. Since regular expression containment is PSPACE-complete [12] this shows that coverage is PSPACE-hard.          □

Even after having computed a determinized automaton, the decision version of the coverage problem is still NP-complete, which we show by reduction to and from MIN-COVER, a well-known NP-complete problem. Let STATE-COVER refer to the problem of deciding membership for the language $\{(M, D(M), p, k) \mid \exists Q.\ |Q| = k \wedge p \notin Q \wedge Q \sqsupseteq p \text{ in } M\}$. Recall that MIN-COVER is the problem of deciding membership for the language $\{(X, \mathcal{F}, k) \mid \exists \mathcal{C} \subseteq \mathcal{F}.|\mathcal{C}| = k \wedge X = \bigcup \mathcal{C}\}$.

**Proposition 4.** STATE-COVER *is NP-complete.*

*Proof.* STATE-COVER $\Rightarrow$ MIN-COVER: Let $(M, D(M), p, k)$ be given. Define $X = \{S \in \mathsf{DState}_M \mid p \in S\}$ and $\mathcal{F} = \{R_q \mid q \in \bigcup X\}$ where $R_q = \{S \in X \mid q \in S\}$. Then any $k$-sized set cover $\mathcal{C} = \{R_{q_1}, ..., R_{q_k}\}$ gives a state cover $Q = \{q_1, ..., q_k\}$ and vice-versa.

MIN-COVER $\Rightarrow$ STATE-COVER: Let $(X, \mathcal{F}, k)$ be given, where $|X| = m$ and $|\mathcal{F}| = n$. Construct an aNFA $M_{X,\mathcal{F}}$ over the alphabet $\Sigma = X \uplus \{\$\}$. Define its states to be the set $\{q^{\mathsf{in}}, q^{\mathsf{fin}}, p\} \cup \{F_1, ..., F_n\}$, and for each $F_i$, add transitions $F_i \overset{\$}{\to} q^{\mathsf{fin}}$ and $q^{\mathsf{in}} \overset{x_{ij}}{\to} F_i$ for each $x_{ij} \in F_i$. Finally add transitions $p \overset{\$}{\to} q^{\mathsf{fin}}$ and $q^{\mathsf{in}} \overset{x}{\to} p$ for each $x \in X$.

Observe that $D(M_{X,\mathcal{F}})$ will have states $\{\{q^{\mathsf{in}}\}, \{q^{\mathsf{fin}}\}\} \cup \{S_x \mid x \in X\}$ where $S_x = \{F \in \mathcal{F} \mid x \in F\} \cup \{p\}$, and $\Delta(\{q^{\mathsf{in}}\}, x) = S_x$. Also, the time to

compute $D(M_{X,\mathcal{F}})$ is bounded by $O(|X||\mathcal{F}|)$. Then any $k$-sized state cover $Q = \{F_1, ..., F_k\}$ is also a set cover.                                                      $\square$

# 7   Algorithm

Our parsing algorithm produces a bit-coded parse tree from an input string $w$ for a given RE $E$. We will simulate $M_E$ in lock-step, reading a symbol from $w$ in each step. The simulation maintains a set of all partial paths that read the prefix of $w$ that has been consumed so far; there are always only finitely many paths to consider, since we restrict ourselves to paths without non-productive loops. When a path reaches a non-deterministic choice, it will "fork" into two paths with the same prefix. Thus, the path set can be represented as a tree of states, where the root is the initial state, the edges are transitions between states, and the leaves are the reachable states.

**Definition 13 (Path Trees).** *A* path tree *is a rooted, ordered, binary tree with internal nodes of outdegrees 1 or 2. Nodes are labeled by aNFA-states and edges by $\Gamma = \Sigma \cup \{0, 1\} \cup \{\epsilon\}$. Binary nodes have a pair of $0$- and $1$-labeled edges (in this order only), respectively.*

We use the following notation:

  - $\mathsf{root}(T)$ is the root node of path tree $T$.
  - $\mathsf{path}(n, c)$ is the path from $n$ to $c$, where $c$ is a descendant of $n$.
  - $\mathsf{init}(T)$ is the path from the root to the first binary node reachable or to the unique leaf of $T$ if it has no binary node.
  - $\mathsf{leaves}(T)$ is the *ordered list* of leaf nodes.
  - $\mathsf{Tr_{empty}}$ is the empty tree.

As a notational convenience, the tree with a root node labeled $q$ and no children is written $q\langle\cdot\rangle$, where $q$ is an aNFA-state. Similarly, a tree with a root labeled $q$ with children $l$ and $r$ is written $q\langle 0 : l, 1 : r\rangle$, where $q$ is an aNFA-state and $l$ and $r$ are path trees and the edges from $q$ to $l$ and $r$ are labeled $0$ and $1$, respectively. Unary nodes are written $q\langle \ell : c\rangle$, denoting a tree rooted at $q$ with only one $\ell$-labelled child $c$.

In the following we shall use $T_w$ to refer to a path tree created after processing input word $w$ and $T$ to refer to path trees in general, where the input string giving rise to the tree is irrelevant.

**Definition 14 (Path Tree Invariant).** *Let $T_w$ be a path tree and $w$ a word. Define $I(T_w)$ as the proposition that* all *of the following hold:*

  (i) *The $\mathsf{leaves}(T_w)$ have pairwise distinct node labels; all labels are symbol sources, that is states with a single symbol transition, or the accept state.*
  (ii) *All paths from the root to a leaf read $w$:*
     $\forall n \in \mathsf{leaves}(T_w).\ \mathsf{read}(\mathsf{path}(\mathsf{root}(T_w), n)) = w.$
  (iii) *For each leaf $n \in \mathsf{leaves}(T_w)$ there exists $w'' \in C_E(w)$ such that the bit-coded parse of $w''$ starts with $\mathsf{write}(\mathsf{path}(\mathsf{root}(T_w), n))$.*

---

**Algorithm 1.** Optimally streaming greedy regular expression parsing algorithm

---

**Require:** An aNFA $M$, a coverage relation $\sqsupseteq$, and an input stream $S$.
**Ensure:** The greedy leftmost parse tree, emitted in an optimally-streaming fashion.
 1: **function** STREAM-PARSE($M, \sqsupseteq, S$)
 2:     $w \leftarrow \epsilon$
 3:     $(T_\epsilon, \_\_) \leftarrow$ CLOSURE($M, \emptyset, q^{\text{in}}$)   ▷ Initialize path tree as the output of CLOSURE
 4:     **while** $S$ has another input symbol $a$ **do**
 5:         **if** $C_E(wa) = \emptyset$ **then**
 6:             **return** write(init($T_w$)) followed by $\sharp$ and exit.
 7:         $T_{wa} \leftarrow$ ESTABLISH-INVARIANT($T_w, a, \sqsupseteq$)
 8:         Output new bits on the path to the first binary node in $T_{wa}$, if any.
 9:         $w \leftarrow wa$
10:     **if** $q^{\text{fin}} \in$ leaves($T_w$) **then**
11:         **return** write(path(root($T_w$), $q^{\text{fin}}$))
12:     **else**
13:         **return** write(init($T_w$)) followed by $\sharp$

---

**Algorithm 2.** Establishing invariant $I(T_{wa})$

---

**Require:** A path tree $T_w$ satisfying invariant $I(T_w)$, a character $a$, and a coverage
    relation $\sqsupseteq$.
**Ensure:** A path tree $T_{wa}$ satisfying invariant $I(T_{wa})$.
 1: **function** ESTABLISH-INVARIANT($T_w, a, \sqsupseteq$)
 2:     Remove leaves from $T_w$ that do not have a transition on $a$.
 3:     Extend $T_w$ to $T_{wa}$ by following all $a$-transitions.
 4:     **for** each leaf $n$ in $T_{wa}$ **do**
 5:         $(T', \_\_) \leftarrow$ CLOSURE($M, \emptyset, n$).
 6:         Replace the leaf $n$ with the tree $T'$ in $T_{wa}$.
 7:     **return** PRUNE($T_{wa}, \sqsupseteq$)

---

(iv) *For each $w'' \in C_E(w)$ there exists $n \in$ leaves($T_w$) such that the bit-coded parse of $w''$ starts with* write(path(root($T_w$), $n$)).

The path tree invariant is maintained by Algorithm 2: line 2 establishes part i; line 3 establishes part ii; and lines 4–7 establish part iii and iv.

**Theorem 1 (Optimal Streaming Property).** *Assume extended $w$, $C_E(w) \neq \emptyset$. Consider the path tree $T_w$ after reading $w$ upon entry into the while-loop of the algorithm in Algorithm 1. Then* write(init($T_w$)) $= O_E(w)$.

In other words, the initial path from the root of $T_w$ to the first binary node in $T_w$ is the longest common prefix of all paths accepting an extension of $w$. Operationally, whenever that path gets longer by pruning branches, we output the bits on the extension.

*Proof.* Assume $w$ extended, that is $w = \hat{w}$; assume $C_E(w) \neq \emptyset$, that is there exists $w''$ such that $w \sqsubseteq w''$ and $w'' \in \mathcal{L}[\![E]\!]$.

---

**Algorithm 3.** Pruning algorithm

---

**Require:** A path tree $T$ and a covering relation $\sqsupseteq$.
**Ensure:** A pruned path tree $T'$ where all leaves are alive.
 1: **function** PRUNE($T, \sqsupseteq$)
 2:     **for** each $l$ in reverse(leaves($T$)) **do**
 3:         $S \leftarrow \{n \mid n$ comes before $l$ in leaves($T$)$\}$
 4:         **if** $S \sqsupseteq l$ **then**
 5:             $p \leftarrow$ parent($l$)
 6:             Delete $l$ from $T$
 7:             $T \leftarrow$ CUT($T, p$)
 8:     **return** $T$
 9: **function** CUT($T, n$)                               ▷ Cuts a chain of 1-ary nodes.
10:     **if** $|$children($n$)$| = 0$ **then**
11:         $p \leftarrow$ parent($n$)
12:         $T' \leftarrow T$ with $n$ removed
13:         **return** CUT($T', p$)
14:     **else**
15:         **return** $T$

---

Claim: $|$leaves($T_w$)$| \geq 2$ or the unique node in leaves($T_w$) is labeled by the accept state. Proof of claim: Assume otherwise, that is $|$leaves($T_w$)$| = 1$, but its node is not the accept state. By i of $I(T_w)$, this means the node must have a symbol transition on some symbol $a$. In this case, all accepting paths $C_E(wa) = C_E(w)$ and thus $\hat{w} = \hat{wa}$; in particular $\hat{w} \neq w$, which, however, is a contradiction to the assumption that $w$ is extended.

This means we have two cases. The case $|$leaves($T_w$)$| = 1$ with the sole node being labeled by the accept state is easy: It spells a single path from initial to accept state. By ii and iii of $I(T_w)$ we have that that path is correct for $w$. By iv and since the accept state has no outgoing transitions, we have $C_E(w) = \{w\}$, and the theorem follows for this case.

Let us consider the case $|$leaves($T_w$)$| \geq 2$ then. Recall that $C_E(w) \neq \emptyset$ by assumption. By iv of $I(T_w)$ the accepting path of every $w'' \in C_E(w)$ starts with path(root($T_w$), $n$) for some $n \in$ leaves($T_w$), and by iii each path from the root to a leaf is the start of some accept path. Since $|$leaves($T_w$)$| \geq 2$ we know that there exists a binary node in $T_w$. Consider the first on the path from the root to a leaf. It has both $0$- and $1$-labeled out-edges. Thus the longest common prefix of $\{$write($p$) $\mid n \in$ leaves($T_w$), $p \in$ path(root($T_w$), $n$)$\}$ is write(init($T_w$)), the bits on the initial path from the root of $T_w$ to its first binary node.                    □

The algorithm, as given, is only optimally streaming for extended prefixes. It can be made to work for all prefixes by enclosing it in an outer loop that for each prefix $w$ computes $\hat{w}$ and calls the given algorithm with $\hat{w}$. The outer loop then checks that subsequent symbols match until $\hat{w}$ is reached. By Proposition 2 the resulting algorithm gives the right result for all input prefixes, not only extended ones.

**Algorithm 4.** $\epsilon$-closure with path tree construction.

---

**Require:** An aNFA $M$, a set of visited states $V$, and a state $q$
**Ensure:** A path tree $T$ and a set of visited states $V'$
 1: **function** CLOSURE($M, V, q$)
 2:     **if** $q \xrightarrow{0} q_l$ and $q \xrightarrow{1} q_r$ **then**
 3:         $(T^l, V_l) \leftarrow$ CLOSURE($M, V \cup \{q\}, q_l$)              ▷ Try left option first.
 4:         $(T^r, V_{lr}) \leftarrow$ CLOSURE($M, V_l, q_r$)      ▷ Use $V_l$ to skip already-visited nodes.
 5:         **return** $(q\langle T^l : T^r \rangle, V_{lr})$
 6:     **if** $q \xrightarrow{\epsilon} p$ **then**
 7:         **if** $p \in V$ **then**                                       ▷ Stop loops.
 8:             **return** $(\mathsf{Tr}_{\mathsf{empty}}, V)$
 9:         **else**
10:             $(T', V') \leftarrow$ CLOSURE($M, V \cup \{q\}, p$)
11:             **return** $(q\langle \epsilon : T' \rangle, V')$
12:     **else**                                  ▷ $q$ is a symbol source or the final state.
13:         **return** $(q\langle \cdot \rangle, V)$

---

**Theorem 2.** *The optimally streaming algorithm can be implemented to run in time $O(2^{m \log m} + mn)$, where $m = |E|$ and $n = |w|$.*

*Proof (Sketch).* As shown in Section 6, we can decide coverage in time $O(m2^{O(m)})$. The set of ordered lists leaves($T$) for any $T$ reachable from the initial state can be precomputed and covered states marked in it. (This requires unit-cost random access since there are $O(2^{m \log m})$ such lists.) The $\epsilon$-closure can be computed in time $O(m)$ for each input symbol, and pruning can be amortized over $\epsilon$-closure computation by charging each edge removed to its addition to a tree path.     □

For fixed regular expression $E$ this is linear time in $n$ and thus asymptotically optimal. An exponential in $m$ as an additive preprocessing cost appears practically unavoidable since we require the coverage relation, which is inherently hard to compute (Proposition 3).

## 8 Example

Consider the RE $(\mathtt{aaa} + \mathtt{aa})^\star$. A simplified version of its aNFA is shown in Figure 2. The following two observations are requirements for an earliest parse of this expression:

- After one $\mathtt{a}$ has been read, the algorithm *must* output a $0$ to indicate that one iteration of the Kleene star been made, but:
- *five* consecutive $\mathtt{a}$s determine that the leftmost possibility in the Kleene star choice was taken, meaning that the first *three* $\mathtt{a}$s are consumed in that branch.

The first point can be seen by noting that any parse of a non-zero number of $\mathtt{a}$s must follow a path through the Kleene star. This guarantees that *if* a

successful parse is eventually performed, it must be the case that at least one iteration was made.

The second point can be seen by considering the situation where only four input as have been read: It is not known whether these are the only four or more input symbols in the stream. In the former case, the correct (and only) parse is two iterations with the right alternative, but in the latter case, the first three symbols are consumed in the left branch instead.

These observations correspond intuitively to what "earliest" parsing is; as soon as it is impossible that an iteration was *not* made, a bit indicating this fact is emitted, and as soon as the first three symbols must have been parsed in the left alternative, this fact is output. Furthermore, a 0-bit is emitted to indicate that (at least) another iteration is performed.

Figure 2 shows the evolution of the path tree during execution with the RE $(\mathsf{aaa} + \mathsf{aa})^\star$ on the input $\mathsf{aaaaa}$.

By similar reasoning as above, after five as it is safe to commit to the left alternative after every third a. Hence, for the inputs $\mathsf{aaaaa}(\mathsf{aaa})^n$, $\mathsf{aaaaa}(\mathsf{aaa})^n\mathsf{a}$, and $\mathsf{aaaaa}(\mathsf{aaa})^n\mathsf{aa}$ the "commit points" are placed as follows ($\cdot$ indicate end-of-input):

$$\underset{0}{\mathsf{a}} \mid \underset{00}{\mathsf{aaaa}} \mid \underbrace{\left(\underset{00}{\mathsf{aaa}} \mid \cdots \mid \underset{00}{\mathsf{aaa}}\right)}_{n \text{ times}} \mid \underset{11}{\cdot} \qquad \underset{0}{\mathsf{a}} \mid \underset{00}{\mathsf{aaaa}} \mid \underbrace{\left(\underset{00}{\mathsf{aaa}} \mid \cdots \mid \underset{00}{\mathsf{aaa}}\right)}_{n \text{ times}} \mid \underset{01}{\mathsf{a}\cdot}$$

$$\underset{0}{\mathsf{a}} \mid \underset{00}{\mathsf{aaaa}} \mid \underbrace{\left(\underset{00}{\mathsf{aaa}} \mid \cdots \mid \underset{00}{\mathsf{aaa}}\right)}_{n \text{ times}} \mid \underset{1011}{\mathsf{aa}\cdot}$$

*Complex coverage.* The previous example does not exhibit any non-trivial coverage, i.e., situations where a state $n$ is covered by $k > 1$ other states. One can construct an expression that contains non-trivial coverage relations by observing that if each symbol source $s$ in the aNFA is associated with the RE representing the language recognized from $s$, coverage can be expressed as a set of (in)equations in Kleene algebra. Thus, the coverage $\{n_0, n_1\} \sqsupseteq n$ becomes $RE(n_0) + RE(n_1) \geq RE(n)$ in KA, where $RE(\cdot)$ is the function that yields the RE from a symbol source in an aNFA.

Any expression of the form $x_1 z y_1 + x_2 z y_2 + x_3 z(y_1 + y_2)$ satisfies the property that two subterms cover a third. If the coverage is to play a role in the algorithm, however, the languages denoted by $x_1$ and $x_2$ must not subsume that of $x_3$, otherwise the part starting with $x_3$ would never play a role due to greedy leftmost disambiguation.

Choose $x_1 = x_2 = (\mathsf{aa})^\star$, $x_3 = \mathsf{a}^\star$, $y_1 = \mathsf{a}$, and $y_2 = \mathsf{b}$. Figure 3 shows the expression

$$(\mathsf{aa})^\star(\mathsf{za} + \mathsf{zb}) + \mathsf{a}^\star\mathsf{z}(\mathsf{a} + \mathsf{b}).$$

The earliest point where any bits can be output is when the z is reached. Then it becomes known whether there was an even or odd number of as. Due to the coverage $\{8, 13\} \sqsupseteq 20$ state 20 is pruned away on the input $\mathsf{aazb}$, thereby causing the path tree to have a large trunk that can be output.
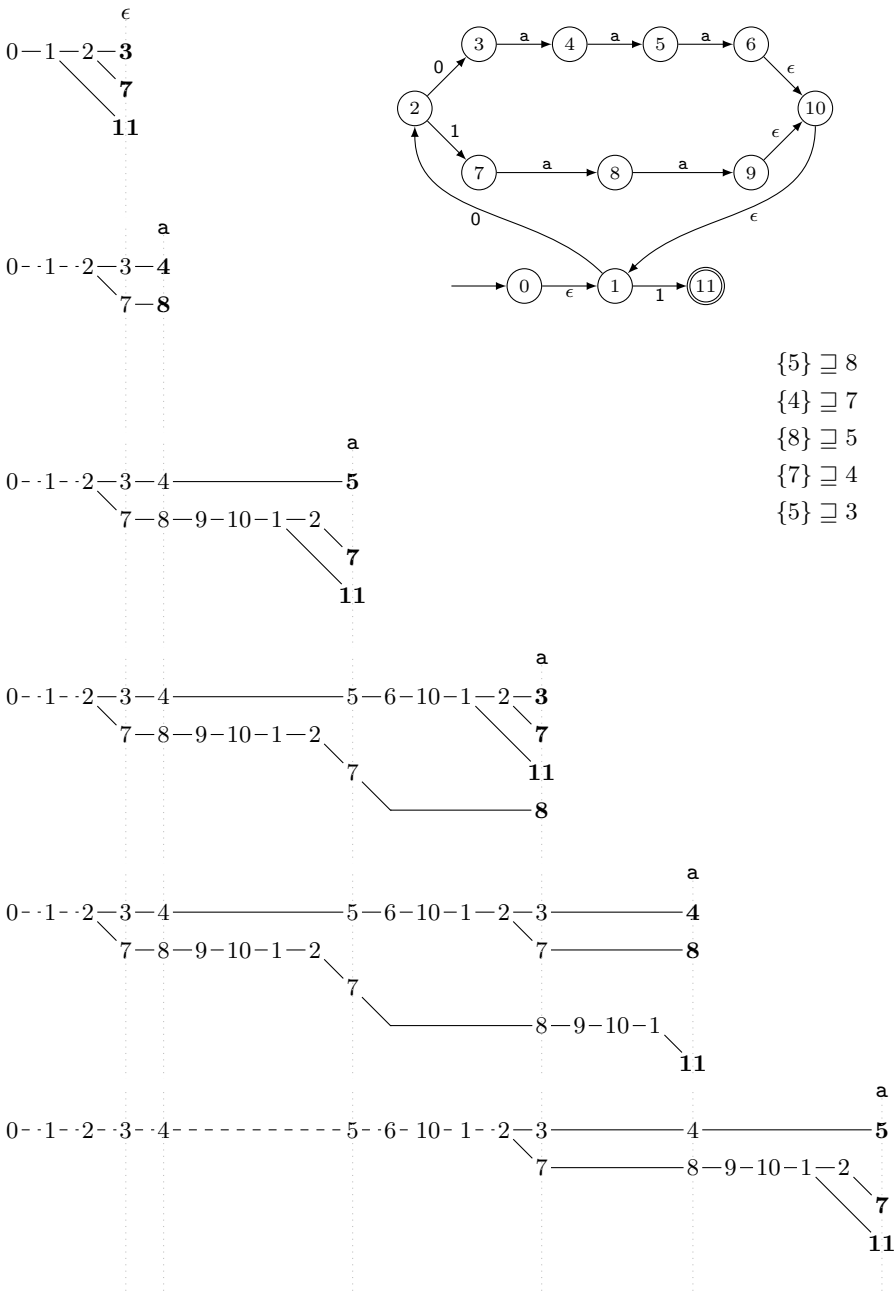
**Fig. 2.** Example run of the algorithm on the regular expression $E = (\texttt{aaa} + \texttt{aa})^\star$ and the input string $\texttt{aaaaa}$. The dashed edges represent the partial parse trees that can be emitted: thus, after one $\texttt{a}$ we can emit a $\texttt{0}$, and after five $\texttt{a}$s we can emit $\texttt{00}$ because the bottom "leg" of the tree has been removed in the pruning step. The automaton for $E$ and its associated minimal covering relation are shown in the inset.
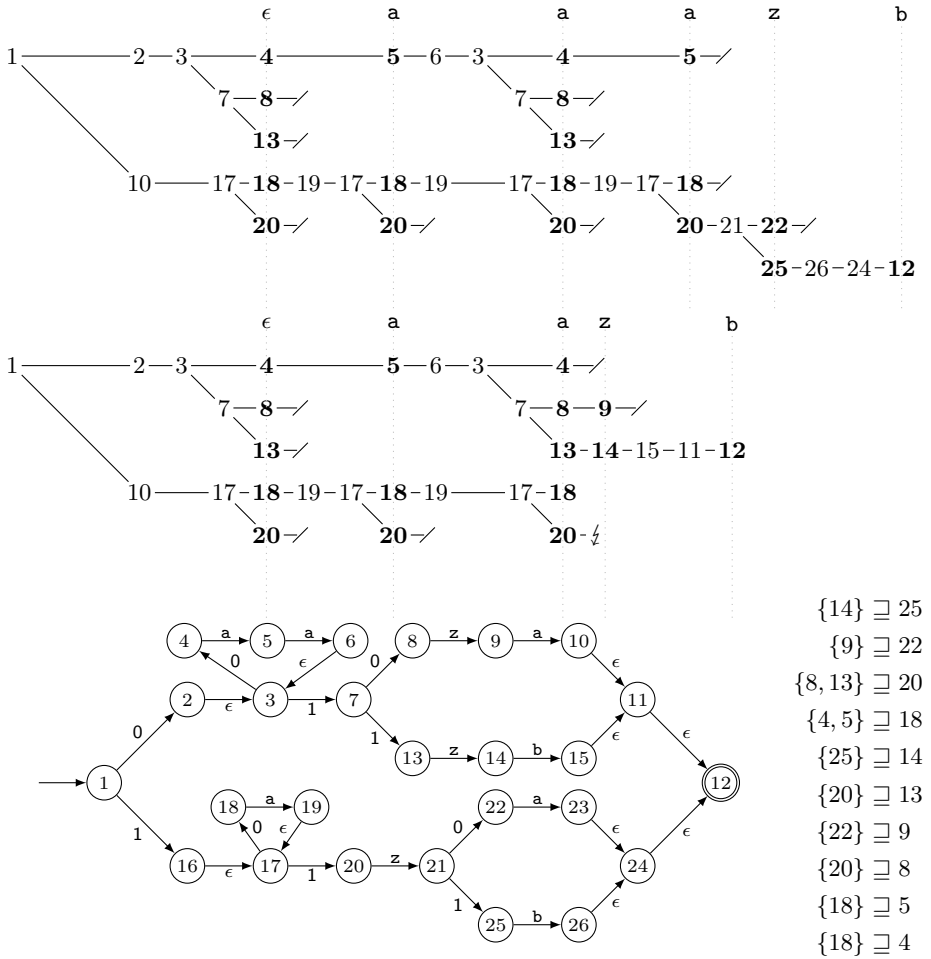
**Fig. 3.** Example run of the algorithm on $E = (\mathtt{aa})^{\star}(\mathtt{za} + \mathtt{zb}) + \mathtt{a}^{\star}\mathtt{z}(\mathtt{a} + \mathtt{b})$. Note that state 20 is covered by *the combination of* states 8 and 13. The earliest time the algorithm can do a commit is when a $\mathtt{z}$ is encountered, which decides whether there is an even or odd number of $\mathtt{a}$s. The topmost figure shows the evolution of the path tree on the input $\mathtt{aaazb}$. There is a long "trunk" from state 1 to state 21 after reading $\mathtt{z}$, as the rest of the branches have been pruned (not shown). The desired output, corresponding to taking the rightmost option in the sum, can be read off the labels on the edges. Likewise in the second figure, we see that if the $\mathtt{z}$ comes after an even number of $\mathtt{a}$s, a binary-node-free path from 1 to 7 emerges. Due to the cover $\{8, 13\} \sqsupseteq 20$, the branch starting from 20 is not expanded further, even though there could be a $\mathtt{z}$-transition on it. This is indicated with ∤. Overall, the resulting parse tree corresponds to the leftmost option in the sum.

*CSV files.* The expression $((\mathtt{a}+\mathtt{b})^\star(;(\mathtt{a}+\mathtt{b})^\star)^\star\mathtt{n})^\star$ defines the format of a simple semicolon-delimited data format, with data consisting of words over $\{\mathtt{a},\mathtt{b}\}$ and rows separated by the newline character, $\mathtt{n}$. Our algorithm emits the partial parse trees after each letter has been parsed, as illustrated on the example input below:

```
a;ba;a      a | ; | b | a | ; | a | n | b | ; | ; | a | n | ·
b;;b       000  10  01  00  10  00  11 001  10  10  00  11  1
```

Due to the star-height of three, many widespread implementations would not be able to meaningfully handle this expression using only the RE engine. Capturing groups under Kleene stars return either the first or last match, but not a *list* of matches—and certainly not a list of lists of matches! Hence, if using an implementation like Perl's [16], one is forced to rewrite the expression by removing the iteration in the outer Kleene star and reintroduce it as a looping construct in Perl.

## 9   Related and Future Work

Parsing regular expressions is not new [6,5,3,10,14], and streaming parsing of XML documents has been investigated for more than a decade in the context of XQUERY and XPATH—see, e.g., [2,7,17]. However, *streaming regular expression* parsing appears to be new.

In earlier work [6] we described a compact "lean log" format for storing intermediate information required for two-phase regular expression parsing. The algorithm presented here may degenerate to two passes, but requires often just one pass in the sense being effectively streaming, using only $O(m)$ work space, independent of $n$. The preprocessing of the regular expression and the intermediate data structure durig input string processing are more complex, however. It may be possible to merge the two approaches using a tree of lean log frames with associated counters, observing that edges in the path tree that are *not* labeled $\mathtt{0}$ or $\mathtt{1}$ are redundant. This is future work.

## References

1. Brüggemann-Klein, A., Wood, D.: One-unambiguous regular languages. Information and computation 140(2), 229–253 (1998),
   http://dx.doi.org/10.1006/inco.1997.2688
2. Debarbieux, D., Gauwin, O., Niehren, J., Sebastian, T., Zergaoui, M.: Early nested word automata for XPath query answering on XML streams. In: Konstantinidis, S. (ed.) CIAA 2013. LNCS, vol. 7982, pp. 292–305. Springer, Heidelberg (2013)

3. Dubé, D., Feeley, M.: Efficiently building a parse tree from a regular expression. Acta Informatica 37(2), 121–144 (2000),
   `http://dx.doi.org/10.1007/s002360000037`
4. Fowler, G.: An interpretation of the POSIX regex standard (January 2003),
   `http://www2.research.att.com/~astopen/testregex/re-interpretation.html`
5. Frisch, A., Cardelli, L.: Greedy regular expression matching. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 618–629. Springer, Heidelberg (2004),
   `http://dx.doi.org/10.1007/978-3-540-27836-8_53`
6. Grathwohl, N.B.B., Henglein, F., Nielsen, L., Rasmussen, U.T.: Two-pass greedy regular expression parsing. In: Konstantinidis, S. (ed.) CIAA 2013. LNCS, vol. 7982, pp. 60–71. Springer, Heidelberg (2013),
   `http://dx.doi.org/10.1007/978-3-642-39274-0_7`
7. Gupta, A.K., Suciu, D.: Stream processing of XPath queries with predicates. In: Proc. 2003 ACM SIGMOD International Conference on Management of Data, SIGMOD 2003, pp. 419–430. ACM, New York (2003),
   `http://dx.doi.org/10.1145/872757.872809`
8. IEEE Computer Society: Standard for Information Technology - Portable Operating System Interface (POSIX), Base Specifications, Issue 7. IEEE (2008),
   `http://dx.doi.org/10.1109/IEEESTD.2008.4694976`, IEEE Std 1003.1
9. Kearns, S.: Extending regular expressions with context operators and parse extraction. Software - Practice and Experience 21(8), 787–804 (1991),
   `http://dx.doi.org/10.1002/spe.4380210803`
10. Nielsen, L., Henglein, F.: Bit-coded regular expression parsing. In: Dediu, A.-H., Inenaga, S., Martín-Vide, C. (eds.) LATA 2011. LNCS, vol. 6638, pp. 402–413. Springer, Heidelberg (2011),
    `http://dx.doi.org/10.1007/978-3-642-21254-3_32`
11. Okui, S., Suzuki, T.: Disambiguation in regular expression matching via position automata with augmented transitions. In: Domaratzki, M., Salomaa, K. (eds.) CIAA 2010. LNCS, vol. 6482, pp. 231–240. Springer, Heidelberg (2011),
    `http://dx.doi.org/10.1007/978-3-642-18098-9_25`
12. Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time (preliminary report). In: Proc. Fifth Annual ACM Symposium on Theory of Computing, pp. 1–9. ACM (1973)
13. Sulzmann, M., Lu, K.Z.M.: Regular expression sub-matching using partial derivatives. In: Proc. 14th Symposium on Principles and Practice of Declarative Programming, PPDP 2012, pp. 79–90. ACM, New York (2012),
    `http://dx.doi.org/10.1145/2370776.2370788`
14. Sulzmann, M., Lu, K.Z.M.: POSIX regular expression parsing with derivatives. In: Codish, M., Sumii, E. (eds.) FLOPS 2014. LNCS, vol. 8475, pp. 203–220. Springer, Heidelberg (2014), `http://dx.doi.org/10.1007/978-3-319-07151-0_13`
15. Thompson, K.: Programming techniques: Regular expression search algorithm. Commun. ACM 11(6), 419–422 (1968),
    `http://dx.doi.org/10.1145/363347.363387`
16. Wall, L., Christiansen, T., Orwant, J.: Programming Perl. O'Reilly Media, Incorporated (2000)
17. Wu, X., Theodoratos, D.: A survey on XML streaming evaluation techniques. The VLDB Journal 22(2), 177–202 (2013),
    `http://dx.doi.org/10.1007/s00778-012-0281-y`