Sparse Regular Expression Matching[*]

Philip Bille[†] Inge Li Gørtz[‡]

Abstract

A regular expression specifies a set of strings formed by single characters combined with concatenation, union, and Kleene star operators. Given a regular expression R and a string Q, the regular expression matching problem is to decide if Q matches any of the strings specified by R. Regular expressions are a fundamental concept in formal languages and regular expression matching is a basic primitive for searching and processing data. A standard textbook solution [Thompson, CACM 1968] constructs and simulates a nondeterministic finite automaton, leading to an $O(nm)$ time algorithm, where n is the length of Q and m is the length of R. Despite considerable research efforts only polylogarithmic improvements of this bound are known. Recently, conditional lower bounds provided evidence for this lack of progress when Backurs and Indyk [FOCS 2016] proved that, assuming the strong exponential time hypothesis (SETH), regular expression matching cannot be solved in $O((nm)^{1-\epsilon})$, for any constant $\epsilon > 0$. Hence, the complexity of regular expression matching is essentially settled in terms of n and m.

In this paper, we take a new approach and introduce a *density* parameter, Δ, that captures the amount of nondeterminism in the NFA simulation on Q. The density is at most $nm + 1$ but can be significantly smaller. Our main result is a new algorithm that solves regular expression matching in

$$O\left(\Delta \log\log \frac{nm}{\Delta} + n + m\right)$$

time.

This essentially replaces nm with Δ in the complexity of regular expression matching. We complement our upper bound by a matching conditional lower bound that proves that we cannot solve regular expression matching in time $O(\Delta^{1-\epsilon})$ for any constant $\epsilon > 0$ assuming SETH.

The key technical contribution in the result is a new linear space representation of the classic position automaton that supports fast state-set transition computation in near-linear time in the size of the input and output state sets. To achieve this we develop several new insights and techniques of independent interest, including new structural properties of the parse trees of regular expression, a decomposition of state-set transitions based on parse trees, and a fast batched predecessor data structure.

1 Introduction

A regular expression R specifies a set of strings formed by characters from an alphabet Σ combined with concatenation ($\odot$), union ($|$), and Kleene star (*) operators. For instance, $(a|(b \odot a))^*$ describes the set of strings of as and bs such that every b is followed by an a. Given a regular expression R and string Q, the regular expression matching is to decide if Q matches any of the strings specified by R. Regular expressions are a fundamental concept in formal language theory introduced by Kleene in the 1950'ties [48] and regular expression matching is a basic tool in computer science for searching and processing text. Standard tools such as `grep` and `sed` provide direct support for regular expression matching in files, and the scripting language `perl` [72] is a full programming language designed to support regular expression matching easily. Regular expression matching appears in many large-scale data processing applications such as internet traffic analysis [44, 49, 78], data mining [31], data bases [52, 60], computational biology [63], and human-computer interaction [47].

A classic textbook algorithm for regular expression matching, due to Thompson [70] from 1968, constructs and simulates a nondeterministic finite automaton (NFA) A in $O(nm)$ time, where n is the length of Q and m is the number of character symbols in R. The simulation processes Q from left to right and computes a sequence of sets of states $S_0, \ldots, S_n$ such that S_i is the set of states in A to which there is a path from the initial state that matches $Q[1..i]$. In 1985 Galil [29] asked if a faster algorithm could be obtained. A sequence of results [10, 11, 12, 62] improved the $O(nm)$ bound using tabulation or word-level parallelism leading to solutions using either $O(nm\frac{\log\log n}{\log^{1.5} n} + n + m)$ [12] or $O(nm\frac{\log w}{w} + n + m\log m)$ time [10] time, where w is the word length.

[*] The full version of the paper can be accessed at https://arxiv.org/abs/1907.04752

[†] Technical University of Denmark.

[‡] Technical University of Denmark.

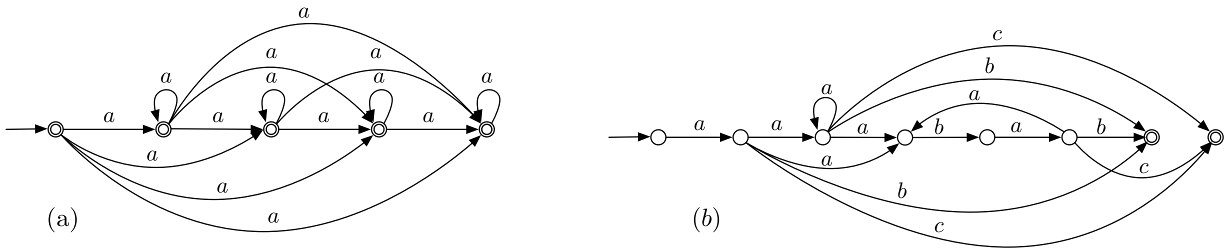


Figure 1: (a) The position automaton for the regular expression $a^*a^*a^*a^*$. (b) The position automaton for $a(a^*)(aba)^*(b|c)$.

Finally, Bille and Thorup [13] gave an algorithm using $O(nk\frac{\log w}{w} + n + m\log k)$ time, where $k \leq m$ is the number of *strings* appearing in the regular expression (see also [8, 26]).

The above solutions are based on the classic NFA simulation algorithm from Thompson's original algorithm [70] and thus achieve the same $O(nm)$ time with some polylogarithmic factors shaved. Recently, conditional lower bounds have provided evidence for the lack of more significant progress. First, Backurs and Indyk [7] showed in 2016 that we cannot solve regular expression matching in time $O((nm)^{1-\epsilon})$, for $\epsilon > 0$, assuming the strong exponential time hypothesis (SETH) [43]. Then, in 2018 Abboud and Bringmann [1] showed that we cannot solve the problem in time $O(nm/\log^{7+\epsilon} n)$, for $\epsilon > 0$, assuming the Formula SAT hypothesis [1]. These results, together with Bringmann, Larsen, and Grønlund [15] and Schepper [67], also studied subclasses of regular expression matching depending on the structure of the operators in the expression, leading to a classification of the complexity of each such subclass. In summary, the complexity of regular expression matching is essentially settled in terms of n and m.

In this paper, we take a new approach and introduce a *density* parameter, Δ, that captures the amount of nondeterminism in the NFA simulation on Q. The density is at most $nm + 1$ but can be significantly smaller. Our main result is a new algorithm that solves regular expression matching in

$$O\left(\Delta \log\log \frac{nm}{\Delta} + n + m\right)$$

time. This essentially replaces nm with Δ in the complexity of regular expression matching. We complement our upper bound by a conditional lower bound that proves that we cannot solve regular expression matching in time $O(\Delta^{1-\epsilon})$ for any constant $\epsilon > 0$ assuming SETH.

1.1 Sparse Regular Expression Matching

Recall that the NFA simulation algorithm constructs a sequence $S_0, \ldots, S_n$ of state sets such that S_i is the set of states in the automaton to which there is a path from the initial state that matches $Q[1..i]$. The goal of this paper is to explore the complexity of regular expression matching if these sets are *sparse*. More precisely, let (R, Q) be an instance of the regular expression matching and let A be a finite automaton that accepts the set of strings defined by R, and let $S_0, \ldots, S_n$ be the sequence of sets of states in the simulation of A on Q. We define the *density* of (R, Q) wrt. A to be

$$\Delta^A_{R,Q} = \sum_{i=0}^{n} |S_i|,$$

i.e., the density is the total size of the state sets in the simulation of A on Q. We will focus on density wrt. to the classic *position automaton* (also known as *Glushkov's automaton*), denoted A_{Pos}, proposed by Glushkov in 1960 [34, 35] and independently by McNaughton and Yamada [57]. For most NFA constructions [4, 17, 33, 70] (including several textbook constructions [2, 3, 24, 51, 56, 69, 77]), the density wrt. to A_{Pos} is a lower bound on the density wrt. to the other NFA construction. The key observation is that the set of states in A_{Pos} naturally corresponds to a *subset* of the set of states in the other constructions. For instance, we can convert Thompson's NFA, A_{T}, into the corresponding position automaton, A_{Pos}, by carefully contracting ϵ-transitions [4, 17, 33]. This implies $\Delta^{A_{\mathsf{Pos}}}_{R,Q} \leq \Delta^{A_{\mathsf{T}}}_{R,Q}$ and thus if an algorithm is efficient in terms of $\Delta^{A_{\mathsf{Pos}}}_{R,Q}$ the same algorithm is also efficient in terms of $\Delta^{A_{\mathsf{T}}}_{R,Q}$. Hence, for the rest of the paper, we define the density, denoted $\Delta_{R,Q}$, to be $\Delta^{A_{\mathsf{Pos}}}_{R,Q}$, and when R and Q are clear from the context we simply write Δ.

Intuitively, the density captures the amount of nondeterminism in the simulation of A_{Pos}. At one extreme $\Delta = n+1$ when all of the $n+1$ state sets are singletons (assuming R matches Q) and at the other extreme $\Delta = nm+1$ when all of the state sets, except the special S_0, consists of all states. The density can be significantly smaller than nm in important practical scenarios. For instance, in internet traffic analysis a stream is matched against a large set of rules specified as a regular expression. Typically, most of these packets will only match a small subset of the rules implying a small density of the problem instance.

A related concept is *deterministic regular expressions* (also known as 1-*unambiguous regular expressions)*. These are defined as regular expressions for which A_{Pos} is deterministic, that is, all state-set transitions on any singleton state set result in a singleton state set. Deterministic regular expressions are widely used in schema languages [14, 19, 30, 61] and have been extensively studied in complexity and automata theory [18, 19, 23, 36, 54, 55, 65]. Groz and Maneth [36] showed how to solve the deterministic regular expression matching problem in $O(n \log\log m + m)$ time. Note that if the regular expression is deterministic we always have that $\Delta \leq n+1$.

1.2 Sparse State-Set Transitions Given a set of S of states and a character α a *state-set transition*, denoted $\delta(S, \alpha)$, is the set of states reachable from S via paths of transitions in the NFA that match α (for ϵ-free NFAs the paths are always single transitions). We can implement the NFA simulation using n state-set transitions by setting S_0 to be the initial state, and computing $S_i = \delta(S_{i-1}, Q[i])$ for $i = 1, \ldots, n$. In our scenario, we are interested in a compact representation of A_{Pos} that supports fast *sparse state-set transitions*, i.e., a state-set computation that is efficient in terms of the sizes of the input set $|S|$ and the output set $|\delta(S, \alpha)|$. Since Δ is the total size of state sets in the simulation this implies an efficient algorithm for sparse regular expression matching.

Surprisingly, few results are known for this problem. If we store A_{Pos} explicitly we can compute $\delta(S, \alpha)$ by computing the union of the endpoints of transitions out of states in S labeled α. This leads to a data structure that uses $O(m^2)$ space and supports state-set transitions in $O(|S||\delta(S, \alpha)|)$ time. Note that since endpoints of the transition may overlap (see Figure 1(a)) we may need to explore $\Omega(|S||\delta(S, \alpha)|)$ transitions in general. A similar worst-case trade-off also holds for the many variants of the position automaton, see e.g. [5, 20, 42, 59]. While ϵ-free NFAs with fewer transitions are known [32, 37, 40, 68] these do not appear to translate to simulations for A_{Pos} nor do they improve the above time bound.

Alternatively, we can store Thompson's automaton, A_{T}, and use the mapping of states mentioned above to convert state-set transitions on A_{T} to state-set transitions on A_{Pos}. Since A_{T} is not an ϵ-free automaton we can compute a state-set transition using a breadth-first search to explore all paths from S that match α. This uses $O(m)$ space and $O(m)$ time. However, it is easy to see that with this approach we may need to traverse large subgraphs of $\Omega(m)$ transitions labeled ϵ even if the sets $|S|$ or $\delta(S, \alpha)$ are sparse. Indeed, the efficient solutions in terms of n and m are based on improving state-set transitions in A_{T} for the *dense* case by polylogarithmic factors.

1.3 Results Our main result is an efficient algorithm for sparse regular expression matching.

THEOREM 1.1. *Given a regular expression R with m positions and a string Q of length n, we can solve the regular expression matching problem in space $O(m)$ and time*

$$O\left(\Delta \log\log \frac{nm}{\Delta} + n + m\right).$$

Since the density Δ is at most $nm+1$, this essentially replaces nm with Δ in the complexity of regular expression matching. As an immediate Corollary of Theorem 1.1 we obtain a solution to deterministic regular expression matching using $O(n \log\log m + m)$ time and $O(m)$ space, thus matching the best known bound of Groz and Maneth [36]. We complement Theorem 1.1 with an essentially matching conditional lower bound.

THEOREM 1.2. *For any $\Delta = n^{1+\gamma}$, for any constant $0 < \gamma \leq 1$, there exists no $O(\Delta^{1-\epsilon})$ time algorithm for regular expression matching for any constant $\epsilon > 0$ assuming SETH.*

Theorem 1.1 is based on a compact representation of the position automaton that supports efficient sparse state-set transitions.

THEOREM 1.3. *Given a regular expression R with m positions, we can represent the position automaton in $O(m)$ space and preprocessing time, such that given any set of states S in sorted order and a character α, we can compute* *the state-set transition $\delta(S,\alpha)$ in time*

$$O\left(|S|\log\log\frac{m}{|S|}+|\delta(S,\alpha)|\right).$$

The output of the state-set transition is also reported in sorted order.

The sorted order of S and $\delta(S,\alpha)$ in Theorem 1.3 refers to the ordering of the corresponding positions in R from left to right (without this condition the $\log\log(m/|S|)$ factor becomes $\log\log|S|$)). Theorem 1.3 significantly improves the previous $O(|S||\delta(S,\alpha)|)$ and $O(m)$ time bounds. Since any solution must use at least $\Omega(|S|+|\delta(S,\alpha)|)$ to read the input and write the output the bound is almost optimal.

1.4 Techniques We develop several new insights and techniques of independent interest, including new structural properties of the parse trees of regular expressions, a novel decomposition of state-set transitions based on parse trees, and a fast batched predecessor data structure.

We show how to decompose any state-set transition $\delta(S,\alpha)$ into a set of *internal transitions* on a set of $O(|S|+|\delta(S,\alpha)|)$ *transition nodes* of the parse tree of R. We have two types of internal transitions: one for $\odot$ and one for $*$. Intuitively, if $R(v)=R(u)\odot R(w)$ then the internal $\odot$-transition of v wrt. α are all the states/positions in $R(w)$ reachable from a state/position in $R(u)$ using a transition labeled α. The internal $*$-transitions are more complicated to describe, but both types of internal transitions are independent of the state set S. We show how to represent R in linear space to efficiently compute internal transitions for any node v and character α.

We identify the set of transition nodes for $\delta(S,\alpha)$ by first computing a compact representation of a *transition tree*, which encodes all paths in R containing transition nodes in $O(|S|)$ space. Then, we find the set of transition nodes using this tree. The key challenge is that even though the representation of the transition tree is small the tree itself can be significantly larger and contain many nodes that are irrelevant for the character α and/or irrelevant for the state set S. Using the structural properties of the parse tree we show how to overcome the challenges and efficiently find the set of transition nodes for any S and α. Computing the internal transitions of all transition nodes could take too long, as the output of these overlap and we could end up using $\Omega(|S||\delta(S,\alpha)|)$ time. However, we prove that these output sets form a laminar family and show how to divide the computations of the internal transitions into computations on a bounded number of non-overlapping intervals.

In combination, the above techniques lead to an $O(m)$ space representation of R that supports state-set transitions in $O(|S|\log\log m+|\delta(S,\alpha)|)$ time. The bottleneck here is computing a $O(|S|)$ predecessor queries in $O(\log\log m)$ time. We present a simple two-level data structure that solves this batched predecessor problem in $O(|S|\log\log\frac{m}{|S|})$ time while maintaining linear space leading to our final structure. Using this solution for sparse-set transitions to implement the NFA simulation implies our main result for sparse regular expression matching of Theorem 1.1.

The lower bound follows from a reduction from the orthogonal vectors problem (OVP). We prove that given $\Delta=n^{1+\gamma}$, for any constant $0<\gamma\leq 1$, we can construct an instance of regular expression matching such that the existence an $O(\Delta^{1-\epsilon})$ algorithm for regular expression matching violates SETH. The reduction is based on the reduction by Backurs and Indyk [7] and is a fairly straightforward generalization of their lower bound.

1.5 Related Work Another NFA construction, by Chang and Paige [21], considered compact representations of A_{Pos} that support efficiently implementing NFA to DFA conversion by subset construction. They presented a linear space representation that supports efficiently computing the set of states S' reachable via *any* character from a state-set S in time $O(|S|+|S'|)$. Since S' can be much larger than $\delta(S,\alpha)$ this does not imply an efficient sparse state-set transition.

Some measures of nondeterminism of NFAs have been studied in automata theory, e.g., width, ambiguity, string tree width, string path width, and cycle height [39,45,46,50,53]. These focus on the complexity of computing measures of the nondeterminism of a given NFA. In contrast, we study the complexity of regular expression matching in terms of the nondeterminism of a simulation on a given NFA and input string.

As mentioned, Bille and Thorup [13] considered the number of strings $k\leq m$ in the regular expression as a parameter for regular expression matching. They gave an algorithm using $O(nk\frac{\log w}{w}+n+m\log k)=O(nk+m\log k)$ time. It is straightforward to construct instances of regular expression matching (for a matching regular expression) such that either $nk=\Theta(nm)$ and $\Delta=\Theta(n)$ or $nk=\Theta(n)$ and $\Delta=\Theta(nm)$ hence this result is incomparable to ours. Cotumaccio, D'Agostino, Policriti, and Prezza [22] studied the indexing version of regular expressions, where the goal is to preprocess a regular expression in order to allow for fast matching given a query string. They considered the co-lexicographic width of an automaton. Applying their construction in the matching setting gives an algorithm that runs in $O(m^2 + np^2 \log(p \cdot \sigma))$, where p is the width of the co-lexicographic order and m^2 comes from the preprocessing of the automaton.

Several papers have studied the related problem of string matching in labeled graphs. For example, Rizzo, Tomescu, and Policriti [66] studied the problem of pattern matching on a labeled graph parameterized in the size of the labeled direct product graph, and Nellore, Nguyen, and Thompson [64] studied string matching in graphs parameterized by the size of the powerset automaton.

Finally, we note that sparsity is a well-studied phenomenon in a wide range of areas in computer science. In particular, sparsity has been extensively studied for other classic pattern matching problems, see, e.g., [6, 27, 28, 41, 73, 74].

1.6 Outline We review regular expressions and automata in Section 2 and the parse tree view of regular expressions in Section 3. We introduce internal transitions, state-set decompositions, and transition trees in Section 4 and present our main algorithm for sparse state-set transitions in Section 5. In Section 6, we present the improved batched predecessor data structure. We use this to obtain the final result for sparse state-set transitions of Theorem 1.3 which we then use to obtain Theorem 1.1. Finally, we show the lower bound of Theorem 1.2 in Section 7.

2 Regular Expressions and Automata

We briefly review the classical concepts used in the paper. For more details see, e.g., Aho et al. [2].

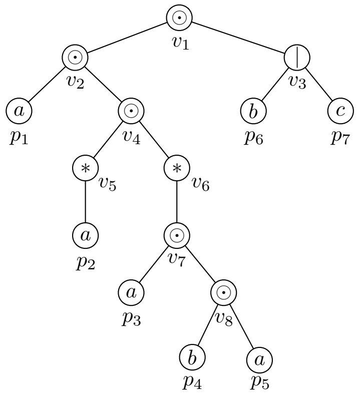
Regular Expressions We consider the set of non-empty regular expressions over an alphabet Σ, defined recursively as follows. If $\alpha \in \Sigma \cup \{\epsilon\}$ then α is a regular expression, and if S and T are regular expressions then so is the *concatenation*, $(S) \odot (T)$, the *union*, $(S)|(T)$, and the *star*, $(S)^*$. We often omit the concatenation $\odot$ when writing regular expressions. The *language* $L(R)$ generated by a regular expression R is defined as follows. If $\alpha \in \Sigma \cup \{\epsilon\}$, then $L(\alpha)$ is the set containing the single string α. If S and T are regular expressions, then $L(S \odot T) = L(S) \odot L(T)$, that is, any string formed by the concatenation of a string in $L(S)$ with a string in $L(T)$, $L(S)|L(T) = L(S) \cup L(T)$, and $L(S^*) = \bigcup_{i \geq 0} L(S)^i$, where $L(S)^0 = \{\epsilon\}$ and $L(S)^i = L(S)^{i-1} \odot L(S)$, for $i > 0$.

Finite Automata A *finite automaton* is a tuple $A = (V, E, \Sigma, \Theta, \Phi)$, where V is a set of nodes called *states*, $E \subseteq (V \times V \times \Sigma \cup \{\epsilon\})$ is a set of directed edges between states called *transitions* each labeled by a character from $\Sigma \cup \{\epsilon\}$, $\Theta \subseteq V$ is a set of *start states*, and $\Phi \subseteq V$ is a set *accepting states*. In short, A is an edge-labeled directed graph with designated subsets of start and accepting nodes. A is a *deterministic finite automaton* (DFA) if A does not contain any ϵ-transitions, all outgoing transitions of any state have different labels, and there is exactly one start state. Otherwise, A is a *nondeterministic finite automaton* (NFA).

Given a string Q and a path p in A we say that p and Q match if the concatenation of the labels on the transitions in p is Q. Given a state s in A and a character α we define the *state-set transition* $\delta_A(s, \alpha)$ to be the set of states reachable from s through paths matching α (note that the paths may include transitions labeled ϵ). For a set of states S we define $\delta_A(S, \alpha) = \bigcup_{s \in S} \delta_A(s, \alpha)$. We say that A *accepts* a string Q if there is a path from a state in Θ to a state in Φ that matches Q. Otherwise, A *rejects* Q. We can use a sequence of state-set transitions to test acceptance of a string Q of length n by computing a sequence of state-sets $S_0, \ldots, S_n$, given by $S_0 = \delta_A(\Theta, \epsilon)$ and $S_i = \delta_A(S_{i-1}, Q[i])$, $i = 1, \ldots, n$. We have that $\Phi \cap S_n \neq \emptyset$ iff A accepts Q.

The Position Automaton Given a regular expression R, we can construct an NFA accepting precisely the strings in $L(R)$ by several classic methods [34, 57, 70]. In particular, Glushkov gave an important construction called the *position automaton* or *Glushkov automaton*. The position automaton is an ϵ-free NFA consisting of only $m + 1$ states and $O(m^2)$ transitions (See Figure 1). Each state except the start state corresponds to a position. Intuitively, each state-set in a state-set simulation is the set of positions in R that correspond to a match of a prefix of Q.

We review the details of the position automaton in the following. Let R be a regular expression with m character symbols from an alphabet Σ. The *position* of a character in R is the index of the character in the



v	first(v)	last(v)
v_1	$\{p_1\}$	$\{p_6, p_7\}$
v_2	$\{p_1\}$	$\{p_1, p_2, p_5\}$
v_3	$\{p_6, p_7\}$	$\{p_6, p_7\}$
v_4	$\{p_2, p_3\}$	$\{p_2, p_5\}$
v_5	$\{p_2\}$	$\{p_2\}$
v_6	$\{p_3\}$	$\{p_5\}$
v_7	$\{p_3\}$	$\{p_5\}$
v_8	$\{p_4\}$	$\{p_5\}$

p	follow(v_1, p)	follow(v_2, p)
p_1	$\{p_2, p_3, p_6, p_7\}$	$\{p_2, p_3\}$
p_2	$\{p_2, p_3, p_6, p_7\}$	$\{p_2, p_3\}$
p_3	$\{p_4\}$	$\{p_4\}$
p_4	$\{p_5\}$	$\{p_5\}$
p_5	$\{p_3, p_6, p_7\}$	$\{p_3\}$
p_6	ϵ	-
p_7	ϵ	-

Figure 2: The parse tree for the expression $a(a^*)(aba)^*(b|c)$, the corresponding first and last sets, and the follow sets for v_1 and v_2.

left-to-right order among the characters in R. The set of positions in R, denoted $\mathsf{Pos}(R)$, is the set $\{1, \ldots, m\}$. The *label* of a position p, denoted $\mathsf{label}(p)$, is the character at position p. The subset of positions labeled α is denoted $\mathsf{Pos}_\alpha(R)$. When R is clear from the context we abbreviate $\mathsf{Pos}(R)$ to phb Pos.

The *marked regular expression* of R, denoted $\overline{R}$, is obtained from R by subscripting each character in R with its position. Similarly, the *marked alphabet*, denoted $\overline{\Sigma}$, is obtained from Σ by adding subscripts. The marked regular expression $\overline{R}$ defines the language $L(\overline{R})$ over the marked alphabet $\overline{\Sigma}$. Note that $\mathsf{Pos}(\overline{R}) = \mathsf{Pos}(R)$. Given a position p we define $\overline{\mathsf{label}}(p)$ to be the label of p in $\overline{R}$. The first and last set of R represent the positions that match the first and last character, respectively, in some string in $L(\overline{R})$. Given a position p, the follow set of R and p is the set of positions that can follow a position p in $L(\overline{R})$. More precisely,

$$\begin{aligned}
\mathsf{first}(R) &= \{p \in \mathsf{Pos}(R) \mid \exists s \in \overline{\Sigma}^*, \overline{\mathsf{label}}(p) \odot s \in L(\overline{R}))\} \\
\mathsf{last}(R) &= \{p \in \mathsf{Pos}(R) \mid \exists s \in \overline{\Sigma}^*, s \odot \overline{\mathsf{label}}(p) \in L(\overline{R}))\} \\
\mathsf{follow}(R, p) &= \{q \in \mathsf{Pos}(R) \mid \exists s, t \in \overline{\Sigma}^*, s \odot \overline{\mathsf{label}}(p) \odot \overline{\mathsf{label}}(q) \odot t \in L(\overline{R}))\}
\end{aligned}$$

We then define the position automaton for R as the NFA $A = (V, E, \{0\}, F)$, where

$$\begin{aligned}
V &= \mathsf{Pos}(R) \cup \{0\}, \\
E &= \{(0, q, \mathsf{label}(q)) \mid q \in \mathsf{first}(R)\} \;\cup\; \bigcup_{p \in \mathsf{Pos}} \{(p, q, \mathsf{label}(q)) \mid q \in \mathsf{follow}(R, p)\} \\
F &= \begin{cases} \{0\} \cup \mathsf{last}(R) & \text{if } \epsilon \in L(R), \\ \mathsf{last}(R) & \text{otherwise} \end{cases}
\end{aligned}$$

3 Regular Expressions as Trees

Throughout the rest of the paper, let R be a regular expression with m positions and let δ denote the state-set transition function of the position automaton for R. For simplicity in the presentation, we will focus on implementing δ on the positions of R and ignore the extra start state of the position automaton. The extra start state is straightforward to represent with additional linear space and is only needed in the initial step of state-set simulations.

We identify regular expressions by their parse trees (see Figure 2). Note that the leaves in left-to-right order are the positions. We call the three types of internal nodes $\odot$-nodes, $*$-nodes, and $|$-nodes. For a $|$-node or $\odot$-node v the left and right child are denoted $\mathsf{left}(v)$ and $\mathsf{right}(v)$, respectively, and for a $*$-node the single child is denoted $\mathsf{left}(v)$. The *depth* of a node v in R is the number of edges on the path from the root of R to v. We denote the subtree (equivalently subexpression) rooted at a node v by $R(v)$. If $u \in R(v)$ then v is an ancestor of u, denoted $v \preceq u$, and if $u \in R(v)$ and $u \neq v$ then v is a proper ancestor of u, denoted $v \prec u$. If v is a (proper) ancestor of u then u is a (proper) descendant of v. A node w is a common ancestor of u and v if it is an ancestor of both u and

v. The *lowest common ancestor* of u and v, $\mathsf{lca}(u, v)$, is the common ancestor of u and v of greatest depth. The *lowest star ancestor* of a node v, denoted $\mathsf{parent}^*(v)$, is the lowest ancestor of v that is a $*$-node.

We extend the definition of labels to internal nodes. For each internal node v, the label of v, denoted $\mathsf{label}(v)$, is a set of characters such that $\alpha \in \mathsf{label}(v)$ iff $v = \mathsf{lca}(p, q)$ for some positions p and q both labeled α. In Figure 2, $\mathsf{label}(v_2) = \{a\}$ since $v_2 = \mathsf{lca}(p_1, p_2)$ and $\mathsf{label}(p_1) = \mathsf{label}(p_2) = \{a\}$. Since the total number of internal nodes containing label α is $|\mathsf{Pos}_\alpha| - 1$ the total size of all labels is $O(m)$. For a node v we extend our notation to define $\mathsf{Pos}(v)$, $\mathsf{first}(v)$, $\mathsf{last}(v)$, and $\mathsf{follow}(v, p)$ to denote the sets on the subexpression $R(v)$ (see Figure 2).

In our setting, we will often view the first and last sets from the perspective of a single position p and consider the nodes for which p appears in the corresponding first and last sets, respectively. Specifically, we define the *first extent* and *last extent* of a position p, respectively, to be the set of nodes in R given by $\mathsf{firstextent}(p) = \{v \mid p \in \mathsf{first}(v)\}$ and $\mathsf{lastextent}(p) = \{v \mid p \in \mathsf{last}(v)\}$. Furthermore, for a set of positions P, we write $\mathsf{firstextent}(P) = \bigcup_{p \in P} \mathsf{firstextent}(p)$ and $\mathsf{lastextent}(P) = \bigcup_{p \in P} \mathsf{lastextent}(p)$. For instance, in Figure 2 we have $\mathsf{firstextent}(\{p_3, p_6\}) = \{p_3, v_7, v_6, v_4, p_6, v_3\}$. We define the first extent and last extent of an internal node v, to be the sets $\mathsf{firstextent}(v) = \{u \mid u \preceq v \text{ and } u \in \mathsf{firstextent}(\mathsf{Pos}(v))\}$ and $\mathsf{lastextent}(v) = \{u \mid u \preceq v \text{ and } u \in \mathsf{lastextent}(\mathsf{Pos}(v))\}$, respectively.

The first sets and the last sets, respectively, form a laminar family. That is, for any two nodes in the parse tree, their first sets, respectively, last sets, are either disjoint or one is contained in the other. That implies that the set of nodes in $\mathsf{firstextent}(p)$, respectively, $\mathsf{lastextent}(p)$, forms a path from position p to an ancestor of p.

LEMMA 3.1. *Let p be a position in a regular expression R and let v and u be nodes in R such that $u \preceq v \preceq p$. If $u \in \mathsf{firstextent}(p)$, then $v \in \mathsf{firstextent}(p)$ and if $u \in \mathsf{lastextent}(p)$, then $v \in \mathsf{lastextent}(p)$.*

Proof. We have that $R(v)$ is a subexpression of $R(u)$ and p is a position in $R(v)$. Then, if $p \in \mathsf{first}(u)$ then $p \in \mathsf{first}(v)$. Similarly, if $p \in \mathsf{last}(u)$ then $p \in \mathsf{last}(v)$. $\square$

4 Internal Transitions, State-Set Decompositions, and Transition Trees

We now introduce the main structural properties of state-set transitions that we need for our fast sparse state-set transition algorithm in Section 5. We first characterize state-set transitions in the position automaton in terms of $\mathsf{firstextent}$ and $\mathsf{lastextent}$ using the following important property.

LEMMA 4.1. *Let $p \in \mathsf{Pos}$ and $q \in \mathsf{Pos}_\alpha$ and $v = \mathsf{lca}(p, q)$. Then, $q \in \delta(p, \alpha)$ iff either*

(i) v is a $\odot$-node, $\mathsf{left}(v) \in \mathsf{lastextent}(p)$, and $\mathsf{right}(v) \in \mathsf{firstextent}(q)$, or

(ii) $\mathsf{parent}^*(v) \in \mathsf{lastextent}(p) \cap \mathsf{firstextent}(q)$.

Lemma 4.1 has appeared in various forms in earlier work [21, 36, 65]. For our purposes, we state it in terms of lowest common ancestors and first extents and last extents. Lemma 4.1 states that a position q can only appear in $\delta(p, \alpha)$ through a $\odot$-node or a $*$-node. We write $q \in \delta^{\odot}(p, \alpha)$ if (i) is satisfied and $q \in \delta^*(p, \alpha)$ if (ii) is satisfied.

4.1 Internal Transitions

Given an internal node v, an internal transition on v and a character α will correspond to the conditions on q and v in Lemma 4.1 while ignoring the condition on p. In general, we will also specify a range of positions we are interested in. Formally, given an internal node v, a character α, and positions l and r, define the *internal $\odot$-transition* and *internal $*$-transition*, denoted $\delta^{\odot}(v, \alpha)$ and $\delta^*(v, \alpha)$, respectively, as follows.

$$\begin{aligned}
\delta^{\odot}_{[l,r]}(v, \alpha) &= \{q \in \mathsf{Pos}_\alpha \mid \mathsf{right}(v) \in \mathsf{firstextent}(q) \text{ and } q \in [l, r]\} && \text{if } v \text{ is a } \odot\text{-node} \\
\delta^{*}_{[l,r]}(v, \alpha) &= \{q \in \mathsf{Pos}_\alpha \cap \mathsf{Pos}(v) \mid \mathsf{parent}^*(v) \in \mathsf{firstextent}(q) \text{ and } q \in [l, r]\} && \text{if } v \text{ is a } *\text{-node}
\end{aligned}$$

When the range includes all positions we drop the subscript, that is, $\delta^{\odot}(v, \alpha) = \delta^{\odot}_{[1,m]}(v, \alpha)$ and $\delta^*(v, \alpha) = \delta^*_{[1,m]}(v, \alpha)$. For instance, in Figure 2 we have $\delta^{\odot}(v_2, a) = \{p_2, p_3\}$, $\delta^{\odot}_{[3,5]}(v_2, a) = \{p_3\}$, $\delta^{\odot}(v_1, c) = \{p_7\}$ and $\delta^*(v_7, a) = \{p_3\}$.

4.2 Transition Nodes

Given a state-set transition $\delta(P, \alpha)$, the transitions nodes are a set of nodes N such that if we compute the union of internal transitions on N we obtain $\delta(P, \alpha)$. Formally, we define the *$\odot$-transition* *nodes* and $*$-*transition nodes* of $\delta(P,\alpha)$, denoted $N^{\odot}(P,\alpha)$ and $N^{*}(P,\alpha)$, respectively, as

$$\begin{aligned} N^{\odot}(P,\alpha) &= \{v \mid v \text{ is a } \odot\text{-node and } \mathsf{left}(v) \in \mathsf{lastextent}(P) \text{ and } \mathsf{right}(v) \in \mathsf{firstextent}(\mathsf{Pos}_\alpha)\} \\ N^{*}(P,\alpha) &= \{v \mid \text{there exists } q \in \mathsf{Pos}_\alpha \text{ and } p \in P \text{ such that } v = \mathsf{lca}(p,q) \\ &\qquad\qquad \text{and } \mathsf{parent}^*(v) \in \mathsf{lastextent}(p) \cap \mathsf{firstextent}(q)\} \end{aligned}$$

In combination, the set of *transition nodes* is the union of the $\odot$-transition nodes and the $*$-transition nodes.

Lemma 4.2. *For any set of positions P and a character α,*

$$\delta(P,\alpha) = \bigcup_{v\in N^{\odot}(P,\alpha)} \delta^{\odot}(v,\alpha) \qquad \cup \qquad \bigcup_{v\in N^{*}(P,\alpha)} \delta^{*}(v,\alpha)\ . \tag{4.1}$$

Proof. Let RH denote the right handside of (4.1). We first show that $\delta(P,\alpha) \subseteq \text{RH}$. Let $p \in P$ and $q \in \mathsf{Pos}$ be positions with $v = \mathsf{lca}(p,q)$ such that $q \in \delta(p,\alpha)$. Then, $q \in \mathsf{Pos}_\alpha$ and hence p, q, and v satisfies either case (i) or (ii) in Lemma 4.1. If (i) is satisfied, v is a $\odot$-node and $\mathsf{left}(v) \in \mathsf{lastextent}(p) \subseteq \mathsf{lastextent}(P)$ and $\mathsf{right}(v) \in \mathsf{firstextent}(q) \subseteq \mathsf{lastextent}(\mathsf{Pos}_\alpha)$. By definition, $v \in N^{\odot}(P,\alpha)$, and thus $q \in \delta^{\odot}(v,\alpha) \subseteq \cup_{v\in N^{\odot}(P,\alpha)}\delta^{\odot}(v,\alpha)$. Similarly, if (ii) is satisfied, then $\mathsf{parent}^*(v) \in \mathsf{lastextent}(p) \cap \mathsf{firstextent}(q)$, and it follows $v \in N^*(P,\alpha)$. This implies that $q \in \delta^*(v,\alpha) \subseteq \cup_{v\in N^*(P,\alpha)}\delta^*(v,\alpha)$.

To show $\text{RH} \subseteq \delta(P,\alpha)$ first suppose $q \in \cup_{v\in N^{\odot}(P,\alpha)}\delta^{\odot}(v,\alpha)$. Then, $q \in \mathsf{Pos}_\alpha$ and there is a $\odot$-node v such that $\mathsf{right}(v) \in \mathsf{firstextent}(q)$ and $\mathsf{left}(v) \in \mathsf{lastextent}(P)$, which implies that $v = \mathsf{lca}(p,q)$ for some $p \in P$. By Lemma 4.1(i) $q \in \delta(P,\alpha)$. If $q \in \cup_{v\in N^*(P,\alpha)}\delta^*(v,\alpha)$ then there exists a $v \in N^*(P,\alpha)$ such that $q \in \mathsf{Pos}(v)$ and $\mathsf{parent}^*(v) \in \mathsf{firstextent}(q)$. It follows from the definition of $N^*(P,\alpha)$ and By Lemma 4.1(ii) that there exists a $p \in P$ and a $q' \in Pos_\alpha$ such that $v = \mathsf{lca}(p,q')$ and $\mathsf{parent}^* \in \mathsf{lastextent}(p) \cap \mathsf{firstextent}(q')$. Since both p and q are descendants of v we have that $u = \mathsf{lca}(p,q)$ is a (not necessarily proper) descendant of v. It follows from Lemma 3.1 that $u \in \mathsf{lastextent}(p)$ and $\mathsf{parent}^*(u) \in \mathsf{firstextent}(q)$. Thus by Lemma 4.1(ii) we have $q \in \delta(P,\alpha)$. □

We show that the total size of the two sets $N^{\odot}(P,\alpha)$ and $N^*(P,\alpha)$ is $O(|P| + |\delta(P,\alpha)|)$.

Lemma 4.3. *We have $|N^{\odot}(P,\alpha)| \le |P| + |\delta^{\odot}(P,\alpha)| - 1$ and $|N^*(P,\alpha)| \le |P| + |\delta^*(P,\alpha)| - 1$.*

Proof. By definition every node in $N^{\odot}(P,\alpha)$ is the lowest common ancestor of some position $p \in P$ and some position $q \in \delta^{\odot}(P,\alpha)$. The number of distinct pairwise lowest common ancestors of a subset of ℓ leaves in a tree cannot exceed $\ell - 1$. Therefore, the number of lowest common ancestors between positions in P and positions in $\delta^{\odot}(P,\alpha)$ can never be larger than $|P| + |\delta^{\odot}(P,\alpha)| - 1$. The same argument holds for the number of nodes in $|N^*(P,\alpha)|$. □

The internal transitions on the set of transition nodes are not disjoint and hence we cannot afford to compute internal transitions on each of the transition nodes explicitly. Fortunately, by Lemma 3.1, the internal $\odot$-transitions (resp. $*$-transitions) of the nodes from $N^{\odot}(P,\alpha)$ (resp. $N^*(P,\alpha)$) form a laminar family. We use this to divide the computations of the internal transitions into computations on a bounded number of non-overlapping intervals. We implement this idea by compactly encoding all relevant transition nodes in the *transition tree* defined in the following.

4.3 Transition Trees Let P be a set of positions. Given a state-set transition $\delta(P,\alpha)$, we define the *transition tree* T as the subtree of R induced by all nodes in P and their ancestors (see Figure 4(a)). A *segment* in T is a path from a leaf or a branching node to (but not including) the nearest branching node above it (or to the root if no such branching node exists). The root node is its own segment (see Figure 4(b)). The bottom node of a segment s is denoted $\mathsf{bot}(s)$. Note that $\mathsf{bot}(s)$ is always a branching node, a leaf, or the root. Any branching node in T is the lowest common ancestor of two nodes in P and vice versa. Hence, we can compactly store T in $O(|P|)$ space by storing P and the branching nodes with pointers into R.

The following observation follows immediately from the fact that all nodes in $\mathsf{lastextent}(P)$ and their ancestors are contained in T.

Observation 4.4. *Let T be the transition tree for P in R. If v is a transition node for $\delta(P,\alpha)$ then v is a node on a segment in T.*

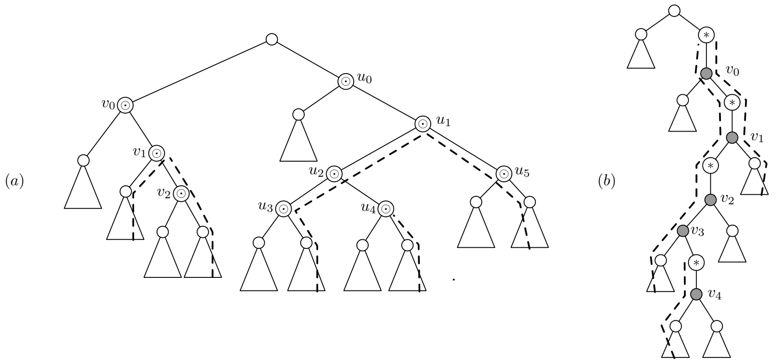


Figure 3: The dotted lines edges indicate the $\mathsf{firstextent}$ paths for the positions in Pos_α. (a) We have $\delta^{\odot}(v_2, \alpha) = \delta^{\odot}(v_1, \alpha)$ and they are contained in but not equal to $\delta^{\odot}(v_0, \alpha)$. Both $\delta^{\odot}(u_3, \alpha)$ and $\delta^{\odot}(u_5, \alpha) = \delta^{\odot}(u_1, \alpha)$ are contained in $\delta^{\odot}(u_0, \alpha)$. Whereas, $\delta^{\odot}(u_2, \alpha) \not\subseteq \delta^{\odot}(u_0, \alpha)$. Note, that we cannot dismiss e.g. $\delta^{\odot}(u_3, \alpha)$ at preprocessing time, since $\mathsf{left}(u_0)$ might not be in $\mathsf{lastextent}(P)$. (b) The grey nodes are lcas of a position in P and a position in Pos_α. The set $\delta^*(v_4, \alpha)$ is not included in any of the others. Nodes v_2 and v_1 has the same $*$-parent and thus $\delta^*(v_3, \alpha) \subseteq \delta^*(v_2, \alpha)$. Since $\mathsf{Pos}_\alpha \cap \mathsf{Pos}(v_2) = \mathsf{Pos}_\alpha \cap \mathsf{Pos}(v_3)$ in the example then $\delta^*(v_3, \alpha) = \delta^*(v_2, \alpha)$. We also have $\delta^*(v_2, \alpha) \subset \delta^*(v_1, \alpha) = \delta^*(v_0, \alpha)$. We cannot dismiss any of the nodes at preprocessing time as might be the case that only a subset (or none) of them is in $\mathsf{lastextent}(P)$.

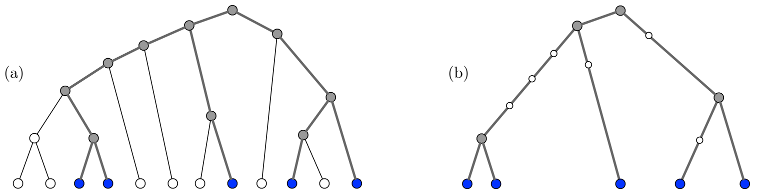


Figure 4: (a) The parse tree R. The blue leaves are the positions in P. (b) The transition tree T of P. Branching nodes are grey and internal nodes on segments are white.

Thus it is enough to consider the nodes in the transition tree when computing the set of transition nodes.

Observation 4.5. *Any $\odot$-node u that is an internal node on a segment and where $\mathsf{left}(u) \in \mathsf{lastextent}(P)$ must have its left child on the same segment.*

To see why, observe that since $\mathsf{left}(u) \in \mathsf{lastextent}(P)$, then $\mathsf{left}(u)$ is an ancestor of a node in P, and thus by definition of T, $\mathsf{left}(u)$ belongs to T. Since u is an internal node on a segment s, only one of its children belongs to T. Therefore, $\mathsf{left}(u)$ must also belong to segment s.

We will show in Section 5.2 that the transition tree T and the key information we need in our algorithm can be computed in $O(|P|)$ time.

5 Sparse State-Set Transitions

We now present our $O(m)$ space data structure that supports computing a state-set transition $\delta(P, \alpha)$ in $O(|P| \log\log m + |\delta(|P|, \alpha)|)$ time. We describe the data structure and analyze the space and preprocessing time in Section 5.1. The high-level idea of our sparse state-set algorithm is to identify the transition nodes for $\delta^{\odot}(P, \alpha)$ and $\delta^*(P, \alpha)$ using the transition tree. The state-set transitions for each set of nodes induce a partition of Pos_α into nested intervals. We partition these intervals into non-overlapping intervals and then compute the internal transitions for each new interval. We then compute the union of these, which by the decomposition in Lemma 4.2, is precisely the set $\delta(P, \alpha)$. We describe how to construct the transition tree in Section 5.2. In Section 5.3 we describe how to find the set transition nodes and how to construct the intervals. In Section 5.4 we we show how to compute the internal transitions efficiently. Finally, in Section 5.5 we put everything together to get an algorithm for computing a state-set transition $\delta(P, \alpha)$ in $O(|P| \log\log m + |\delta(P, \alpha)|)$ time.

5.1 Data Structure We store the regular expression with labels on leaves and internal nodes together with the following components.

- For each node v in R, we store the range of positions that are descendants of v, the depth of v, the highest node in $\mathsf{lastextent}(v)$, and the highest node in $\mathsf{firstextent}(v)$. We also store a pointer $\mathsf{parent}^{\odot}(v)$ to the lowest ancestor u of v such that u is an $\odot$-node and $v \in T(\mathsf{left}(u))$, and a pointer $\mathsf{parent}^{*}(v)$ to its lowest star ancestor.

- At each branching node v we store the position of the rightmost leaf in $\mathsf{left}(v)$ and the position of the leftmost leaf in $\mathsf{right}(v)$.

- Data structures for R that supports *lowest common ancestor queries* and *first label queries*. Given a node v and a character α, a first label query, denoted $\mathsf{firstlabel}(v, \alpha)$, returns the lowest ancestor of v whose label contains α.

Furthermore, we store the following information for each $\alpha \in \Sigma$.

- Arrays A_α and D_α, where $A_\alpha[i]$ is the ith position labeled α in the left-to-right ordering of Pos_α, and $D_\alpha[i]$ is the depth of the highest node in $\mathsf{firstextent}(A_\alpha[i])$.

- A data structure supporting predecessor and successor queries on the positions in Pos_α. That is, given any position p in R the predecessor (successor) query returns the position in A_α of the nearest position labeled α to the left (right) of p including p itself. For a branching node $v \in R$, we define the successor of v in Pos_α as the successor in Pos_α of the leftmost leaf in $\mathsf{right}(v)$. Note that this corresponds to the first position labeled α after v in the order obtained by an inorder traversal of the nodes in R. Similarly, we define the predecessor of v in Pos_α as the predecessor in Pos_α of the rightmost leaf in $\mathsf{left}(v)$.

- A data structure on D_α that supports *range minimum queries*. Given any pair of indices l and r, the range minimum query on D_α returns a minimum value in the subarray $D_\alpha[l, r]$.

- For each node v containing label α:
  - A pointer $\mathsf{next}^{\odot}(v, \alpha)$ to the lowest proper ancestor u of v such that $v \in T(\mathsf{left}(u))$, $\alpha \in \mathsf{label}(u)$, and $\delta^{\odot}(u, \alpha)$ is non-empty. If no such u exists we store a null pointer to indicate this.
  - A pointer $\mathsf{next}^{*}(v, \alpha)$ to the lowest proper ancestor u of v labeled α such that there exists a $q \in \delta^{*}(u, \alpha)$ where $q \in T(\mathsf{right}(u))$ if $v \in T(\mathsf{left}(u))$ and $q \in T(\mathsf{left}(u))$ if $v \in T(\mathsf{right}(u))$. If no such u exists we store a null pointer to indicate this.
  - The range of positions in A_α that are descendants of v, of $\mathsf{left}(v)$, and of $\mathsf{right}(v)$, respectively.

The idea of the $\mathsf{next}^{\odot}$-pointers is that they form a chain of prospective nodes for $N^{\odot}$ with label α. Any node u from $N^{\odot}(P, \alpha)$ with label α on a segment s has its left child on the path and $\delta^{\odot}(u, \alpha) \neq \emptyset$, so it is included in this chain. We show that at most one node can be from $N^{\odot}(P, \alpha)$ on each segment that does not have label α. Furthermore, given a segment s, the nodes from the chain on s that belong to $N^{\odot}(P, \alpha)$ form a subchain starting from the lowest node of the chain that is on s to (not including) the first node in the chain that is either not in $\mathsf{lastextent}(P)$ or not on s. Similarly, the next^{*}-pointers form a chain of prospective nodes for N^{*} with label α.

Space The regular expression and the labels use $O(m)$ space. The arrays A_α and D_α, $\alpha \in \Sigma$, use $O(m)$ space in total. We use linear space and linear preprocessing time data structures to support lowest common ancestors in constant time [9, 38], first label queries in $O(\log\log m)$ time [25], predecessor queries in $O(\log\log m)$ time [58, 75], and range minimum queries in constant time [9, 38]. For each alphabet character, the total size of these data structures is linear in the number of leaves labeled with that character. Thus in total the space for these data structures is linear in m. The cited data structures for predecessor queries both use randomization, but since we only need a static structure it is straightforward and well-known how to obtain the same bound deterministically by combining deterministic dictionaries [37] with a simple two-level approach (see, e.g., Thorup [71]). We store at most two next pointers for each label in R and a single pointer for each position using $O(m)$ space. The remaining information uses $O(m)$ space.

Preprocessing We compute the range of positions that are descendants of v, the depth of v, the $\mathsf{parent}^{\odot}(v)$ and $\mathsf{parent}^{*}(v)$ pointers, and the positions of $\mathsf{left}(v)$ and $\mathsf{right}(v)$ using tree traversal in linear time. To compute the highest node for each node in R, we first compute for each node $u \in R$ if $\epsilon \in R(u)$ using a linear time bottom-up tree traversal. In top-down traversal we then compute the highest node $H_L(v)$ in $\mathsf{lastextent}(v)$ for each node $v \in R$ using the following rules: If v is the left child of an $\odot$-node and $\epsilon \notin R(\mathsf{right}(v))$ then $H_L(v) = v$. Otherwise, $H_L(v) = H_L(\mathsf{parent}(v))$. We compute the highest node in $\mathsf{firstextent}(v)$ similarly. We construct the arrays A_α and D_α, for all $\alpha \in \Sigma$ in a single tree traversal.

To compute the remaining information we do the following for each $\alpha \in \Sigma$. Construct a tree R_α containing all nodes with label α. To do this, we use lca queries on each consecutive pair of leaves in A_α from left to right. By keeping track of the depths of the nodes and checking if the newest node is an ancestor of the previous node it is straightforward to implement this in linear time. To compute the next-pointers we do a top-down traversal of R_α. In each node u we check if $\delta^{\odot}(u, \alpha)$ and $\delta^{*}(u, \alpha)$ are empty using range minimum queries on D_α. To check if $\delta^{\odot}(u, \alpha)$ we do the range minimum query on the range of positions in Pos_α in $\mathsf{right}(v)$. If the depth returned is less than or equal to $\mathsf{depth}(\mathsf{right}(v))$ then $\delta^{\odot}(u, \alpha)$ is non-empty. For $\delta^{*}(u, \alpha)$ we do the query separately on the intervals corresponding to the left and the right child and we compare with the depth of $\mathsf{parent}^{*}(v)$. With this information, we can compute the next-pointers during the traversal of R_α in constant time per node. The total size of all the R_α trees $O(m)$, since each tree has size $2|\mathsf{Pos}_\alpha| - 1$. Thus the total time used for each α is $O(|\mathsf{Pos}_\alpha|)$. Hence, it follows that the total preprocessing time is $O(m)$.

5.2 Constructing the Transition Tree We say that a node v is a $\odot$-*live node* if $\mathsf{left}(v) \in \mathsf{lastextent}(P)$ and v is a $\odot$-node. Note that any node in $N^{\odot}(P, \alpha)$ is a $\odot$-live node and a node in the transition tree T. It follows from Observation 4.5 that any $\odot$-live node that is an internal node on a segment has its left child on the segment. A segment s in T is called a $*$-*segment* if $\mathsf{bot}(s)$ is not the root and $\mathsf{parent}^{*}(\mathsf{bot}(s)) \in \mathsf{lastextent}(P)$.

We compute the compact representation of the transition tree as follows. Let P be the set of leaves and repeatedly take the lca of adjacent nodes to form the internal nodes and the segments of T. Using a tree traversal on the compact transition tree we also compute for all branching nodes v in T the depth of the highest node in $\mathsf{lastextent}(P \cap \mathsf{Pos}(v))$, all $\odot$-live branching nodes in T, and all $*$-segments of T. Hence, we have the following result.

LEMMA 5.1. *In $O(|P|)$ time we can compute the transition tree T of P, for all branching nodes v in T the depth of the highest node in $\mathsf{lastextent}(P \cap \mathsf{Pos}(v))$, all $\odot$-live branching nodes in T, and all $*$-segments of T.*

OBSERVATION 5.2. *We can check in constant time if a node $v \in T$ is in $\mathsf{lastextent}(P)$ given the segment it is on.*

If v is a leaf or a branching node, we already computed the information. Otherwise, v is an internal node on a segment s. Then we compare the depth of v with the depth d of the highest node in $\mathsf{lastextent}(P \cap \mathsf{Pos}(\mathsf{bot}(s)))$. Now v is in $\mathsf{lastextent}(P)$ if and only if the depth of v is at least d.

5.3 Computing Transitions Nodes and Intervals We construct two sets of nodes $M^{\odot}$ and M^{*} that consists of all nodes in $N^{\odot}(P, \alpha)$ and $N^{*}(P, \alpha)$, respectively, together with a constant number of other nodes per segment. We compute these sets for each segment using a depth-first traversal of the transition tree. We also construct sets $L^{\odot}$ and L^{*}, that partition Pos_α into intervals in order to avoid recomputing overlapping internal transitions. We associate each interval with the lowest node from $M^{\odot}$ (respectively M^{*}) that can contain the positions in its internal transition (see Figure 5).

5.3.1 Computing Transitions Nodes We first explain how to compute the transition nodes using the next-pointers and the transition tree.

Computing $M^{\odot}$ We find for each segment s all $\odot$-transition nodes for $\delta(P, \alpha)$ as follows.

1. *Find first relevant $\odot$-node on s:* Set $x = \mathsf{parent}^{\odot}(\mathsf{bot}(s))$. If x is not on s or x is not an $\odot$-live node stop.

2. *Find first $\odot$-transition node on s:* Compute the successor q of x in Pos_α, i.e., q is the successor in Pos_α of the leftmost leaf in $R(\mathsf{right}(x))$. If no such q exists stop, otherwise set $v = \mathsf{lca}(x, q)$. If v is not on s stop. If v is a $\odot$-live node not labeled α we add v to $M^{\odot}$. Now compute $x = \mathsf{firstlabel}(v, \alpha)$. If x is not a $\odot$-live node set $x = \mathsf{next}^{\odot}(x, \alpha)$.

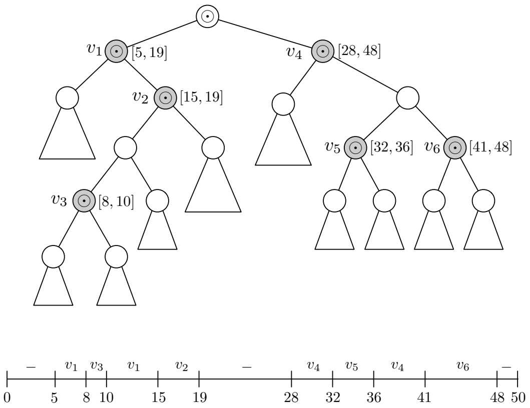


Figure 5: The nodes in $M^\odot$ are colored grey. The interval $[l_v, r_v]$ covered by $\mathsf{right}(v)$ is written next to the nodes. The list $L^\odot = \{(5, 7, v_1), (8, 10, v_3), (11, 14, v_1), (15, 19, v_2), (28, 31, v_4), (32, 36, v_5), (37, 49, v_4), (41, 48, v_6)\}$. In the final algorithm, we compute and return the union of $\delta^\odot_{[5,7]}(v_1, \alpha)$, $\delta^\odot_{[8,10]}(v_3, \alpha)$, $\delta^\odot_{[11,14]}(v_1, \alpha), \ldots, \delta^\odot_{[41,48]}(v_6, \alpha)$.

3. *Repeatedly find next transition node by following* $\mathsf{next}^\odot$*-pointers:* We find all $\odot$-transition nodes by repeatedly following $\mathsf{next}^\odot$-pointers from x and adding the visited nodes to $M^\odot$ as follows. As long as $\mathsf{left}(x) \in \mathsf{lastextent}(P)$ and x is on s we add x to $M^\odot$ and set $x = \mathsf{next}^\odot(x, \alpha)$.

Computing M^* We find for each segment $s = (v, w)$ all $*$-transition nodes for $\delta(P, \alpha)$.

1. If s is not a $*$-segment we stop.

2. *Find first $*$-transition node on s:* Compute the predecessor q^- and successor q^+ of $\mathsf{bot}(s)$ in Pos_α and let v be the lowest of $\mathsf{lca}(\mathsf{bot}(s), q^-)$ and $\mathsf{lca}(\mathsf{bot}(s), q^+)$. If v is not on s stop. If $\alpha \notin \mathsf{label}(v)$ and $\mathsf{parent}^*(v) \in \mathsf{lastextent}(P)$ we add v to M^*.

3. *Repeatedly find next transition node by following* next^**-pointers:* We first compute $x = \mathsf{firstlabel}(v, \alpha)$. We find all $*$-transition nodes by repeatedly following next^*-pointers from x and adding the visited nodes to M^* as follows. As long as $\mathsf{parent}^*(x) \in \mathsf{lastextent}(P)$ and x is on s we add x to M^* and set $x = \mathsf{next}^*(v, \alpha)$.

Complexity We first analyze the time used to find $M^\odot$. We use $O(|M^\odot|)$ time to follow pointers. Additionally, we use $O(|T| \log\log m) = O(|P| \log\log m)$ time to compute the first label queries, as we do one first label query on each of the $|T|$ segments. The time to check if a node is $\odot$-live is constant. For all the bottom nodes of the segments the information is stored in the tree and for all the other nodes v we can check in constant time if $\mathsf{left}(v) \in \mathsf{lastextent}(P)$ as described in the end of Section 5.2 as $\mathsf{left}(v)$ will always be on the current segment. Thus the total time used is $O(|M^\odot| + |P| \log\log m)$. Similarly, we use time $O(|M^*| + |P| \log\log m)$ to compute M^*. Next we analyze the size of $M^\odot$ and M^*.

Lemma 5.3. $|M^\odot| = O(|\delta^\odot(P, \alpha)| + |P|)$.

Proof. We will prove that at most one node from $M^\odot$ from each segment is not in $N^\odot(P, \alpha)$. Recall that $N^\odot(P, \alpha)$ consists of all the $\odot$-nodes v that have $\mathsf{left}(v) \in \mathsf{lastextent}(P)$ and $\mathsf{right}(v) \in \mathsf{firstextent}(\mathsf{Pos}_\alpha)$.

Any node $u \in s$ added to $M^\odot$ in step 3 except the first one has $\delta(u, \alpha) \neq \emptyset$, since they were found using $\mathsf{next}^\odot$-pointers. Thus, $\mathsf{right}(u) \in \mathsf{firstextent}(\mathsf{Pos}_\alpha)$. A node is only added if it is $\odot$-live, i.e., $\mathsf{left}(u) \in \mathsf{lastextent}(P)$. Thus $u \in N^\odot(P, \alpha)$. Therefore, only the first node found on each segment—the node from step 2—might not be in $N^\odot(P, \alpha)$. Since there are $O(|P|)$ segments in T we have $|M^\odot| = O(|N^\odot(P, \alpha)| + |P|)$. By Lemma 4.3 we have $|N^\odot(P, \alpha)| \leq |\delta^\odot(P, \alpha)| + |P|$ and thus $|M^\odot| = O(|\delta^\odot(P, \alpha)| + |P|)$. □

The argument for the size of M^* is similar, but here we show that the number of nodes in M^* that are not in $N^*(P, \alpha)$ is at most $2|P|$.

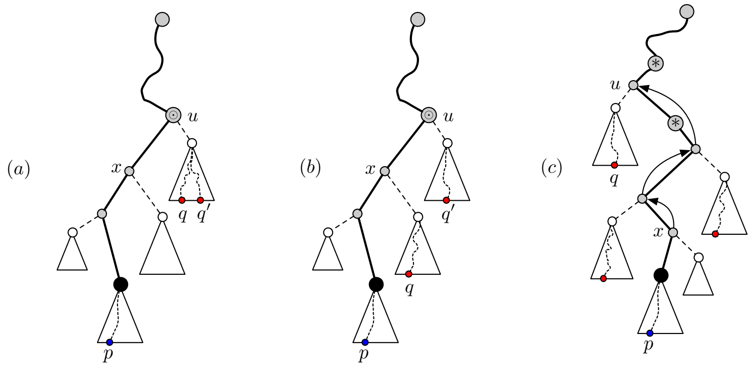


Figure 6: The thick edges are the edges on the segment s and the black node is $\mathsf{bot}(s)$. In (a) both q and q' are positions in the right subtree of u, and $u = \mathsf{lca}(x, q)$. In (b) the positions q and q' are in different subtrees and node u has label α. In (c) the arrows indicates the next^*-pointers.

Lemma 5.4. $|M^*| = O(|\delta^*(P, \alpha)| + |P|)$.

Proof. We will prove that at most two nodes from M^* from each segment is not in $N^*(P, \alpha)$. Recall that $N^*(P, \alpha)$ consists of all the nodes v such that there exists a $q \in \mathsf{Pos}_\alpha$ and a $p \in P$ such that $v = \mathsf{lca}(p, q)$ and $\mathsf{parent}^*(v) \in \mathsf{lastextent}(p) \cap \mathsf{firstextent}(q)$.

Any node u added to M^* by following next^*-pointers has $\delta^*(u, \alpha) \neq \emptyset$ and $\mathsf{parent}^*(u) \in \mathsf{lastextent}(P)$. Furthermore, if $\mathsf{left}(u)$ is on the segment s, then there exists a position $q \in \mathsf{right}(u) \cap \delta^*(u, \alpha)$ and a position $p \in P \cap \mathsf{left}(u)$. This implies that u is in $N^*(P, \alpha)$. The argument for the case where $\mathsf{right}(u)$ is on the segment s is symmetric.

At most two other nodes are added to M^* for each segment (the first two nodes are added to M^* on each segment). It follows that the total number of nodes in M^* is at most $|\delta^*(P, \alpha)| + 2|P|$. By Lemma 4.3 we have $|N^*(P, \alpha)| \leq |\delta^*(P, \alpha)| + |P|$ and thus $|M^*| = O(|\delta^*(P, \alpha)| + |P|)$. □

Combining Lemmas 5.3 and 5.4 and the above discussion, the total time to compute $M^\odot$ and M^* is $O(|\delta^\odot(P, \alpha)| + |P| \log\log m)$ and $O(|\delta^*(P, \alpha)| + |P| \log\log m)$, respectively.

Correctness We argue that the sets $M^\odot$ and M^* include all transition nodes for $\delta(P, \alpha)$. We need the following lemma, which follows from the path structure of the last extent sets.

Lemma 5.5. *Let u be an $\odot$-live node and let s be the segment in T containing u. All $\odot$-nodes below u on s with* $\mathsf{left}(u)$ *on s are also $\odot$-live.*

Proof. If $\mathsf{left}(u)$ is not on s then u is a branching node in T by construction of T. Thus $u = \mathsf{bot}(s)$ and it is trivially true since u has no descendants on s. We will prove the case $\mathsf{left}(u)$ on s by contradiction. Since u is $\odot$-live there exists a node $p \in \mathsf{left}(u) \cap P$ such that $u \in \mathsf{lastextent}(p)$. Furthermore, $\mathsf{bot}(s)$ is an ancestor of some node p' in P. Let $w = \mathsf{lca}(\mathsf{bot}(s), p)$. If $p' = p$ then $w = \mathsf{bot}(s)$. If $p' \neq p$ then w is a branching node in T. Since there are no branching nodes internally on a segment it follows that $w = \mathsf{bot}(s)$. By Lemma 3.1 all nodes on s are in $\mathsf{lastextent}(p)$. □

We are now ready to prove that $N^\odot(P, \alpha)$ is contained in $M^\odot$. Let u be a node in $N^\odot(P, \alpha)$. There are two main cases in the proof depending on whether u labeled α or not. If not, then we show that $u = \mathsf{lca}(x, q)$ in step 2. If u is labeled α, then we show, that either u is the first $\odot$-node on the segment, in which case it is added as the first node in step 3, or it is a node on the path induced by the $\mathsf{next}^\odot$-pointers. As we proved in Lemma 5.5 above, all the nodes on this path below u are also $\odot$-live and we show by induction that all the $\odot$-live nodes from s on this path are added to $M^\odot$ in step 3.

Lemma 5.6. $N^\odot(P, \alpha) \subseteq M^\odot$.

Proof. Let $u \in N^{\odot}(P, \alpha)$. By definition u is $\odot$-live, $\mathsf{right}(v) \in \mathsf{firstextent}(\mathsf{Pos}_\alpha)$, and $\delta(u, \alpha) \neq \emptyset$. Let s be the segment in T containing u. We want to show that u is added to $M^{\odot}$. Define $x = \mathsf{parent}^{\odot}(\mathsf{bot}(s))$ and q as the successor of x in Pos_α as in step 1 and 2 of the algorithm. There are two cases depending on whether u has label α or not.

Case 1: $\alpha \notin \mathsf{label}(u)$. We first show that $u = \mathsf{lca}(x, q)$. Since u is an $\odot$-node we have that u is an ancestor of x. If u is a proper ancestor of x then $x \in R(\mathsf{left}(u))$. Since $\mathsf{right}(u) \in \mathsf{firstextent}(\mathsf{Pos}_\alpha)$ there is a position $q' \in \mathsf{Pos}_\alpha$ such that $q' \in \mathsf{Pos}(\mathsf{right}(u))$. This implies that $q \leq q'$. If $q \in \mathsf{Pos}(\mathsf{right}(u))$ then $u = \mathsf{lca}(x, q)$ and we are done (see Figure 6(a)). Assume $q \notin \mathsf{Pos}(\mathsf{right}(u))$ and let $w = \mathsf{lca}(x, q) \neq u$. Since $q < q'$ we have $u \prec w$. But then $q \in \mathsf{Pos}(w) \subset \mathsf{Pos}(\mathsf{left}(u))$. Thus $u = \mathsf{lca}(q, q')$ and u has label α which is a contradiction (see Figure 6(b)). It follows that $u = \mathsf{lca}(x, q)$. Thus, since u is $\odot$-live and not labeled α, u is added to $M^{\odot}$ in step 2.

Case 2: $\alpha \in \mathsf{label}(u)$. There are two subcases. In the first case $u = \mathsf{parent}^{\odot}(\mathsf{bot}(s))$. Then $u = x$ in step 1. Since $\mathsf{right}(u) \in \mathsf{firstextent}(\mathsf{Pos}_\alpha)$ we have $u = v = \mathsf{lca}(x, q)$ in step 2. Since u is labeled α we have $u = \mathsf{firstlabel}(v, \alpha) = \mathsf{firstlabel}(u, \alpha)$. Since $\mathsf{left}(u) \in \mathsf{lastextent}(P)$ we add u to $M^{\odot}$ in the first iteration in step 3.

In the other case $u \neq \mathsf{parent}^{\odot}(\mathsf{bot}(s))$. Let

$$S_\alpha = \{w \in s \mid w \text{ is a } \odot\text{-live node}, \mathsf{left}(w) \in s, \alpha \in \mathsf{label}(w) \text{ and } \delta^{\odot}(w, \alpha) \neq \emptyset\}\ .$$

Clearly, $u \in S_\alpha$, and thus $u \in M^{\odot}$ follows from the following claim.

Claim 5.7. *Let w be a node in S_α. Then w is added to $M^{\odot}$ in step 3.*

Proof: We prove the claim using induction on the height of w. For the base case let w be the lowest node in S_α. Since w is $\odot$-live it follows from Lemma 5.5 that $\mathsf{parent}^{\odot}(\mathsf{bot}(s))$ is also $\odot$-live and thus we continue from step 1 to step 2. Since $\mathsf{right}(w) \in \mathsf{firstextent}(\mathsf{Pos}_\alpha)$ we have $w \preceq v$, where $v = \mathsf{lca}(x, q)$ as computed in step 2. Now either $w = \mathsf{firstlabel}(v, \alpha)$ and then w is added to $M^{\odot}$ in the first iteration in step 3. Otherwise, $w \prec \mathsf{firstlabel}(v, \alpha)$. Since w is the lowest node in S_α then $x = \mathsf{firstlabel}(v, \alpha)$ has its $\mathsf{next}^{\odot}$-pointer pointing to w. Now either x is an $\odot$-live node, in which case x is set to w in the end of the first iteration of step 3. Otherwise, x is set to w in the end of step 2. In either case, w is added to $M^{\odot}$ in step 3.

Induction step: Let $w \in S_\alpha$ be a node that is not the lowest node in S_α. Let $w' \in S_\alpha$ be the highest node in S_α that is a proper descendant of w. Then w' points to w. Let $v = \mathsf{next}^{\odot}(w', \alpha)$. By definition of the $\mathsf{next}^{\odot}$-pointers $v \in S_\alpha$ and $v \prec w'$. Now, since $w' \in R(\mathsf{left}(w))$, $\alpha \in \mathsf{label}(w)$ and $\delta^{\odot}(w, \alpha) \neq \emptyset$, we have $v = \mathsf{next}^{\odot}(w', \alpha) \succeq w$. If $v \neq w$ then $w' \succ v \succ w$ contradicting that w' is the highest descendant of w in S_α. Thus $v = w$. By the induction hypothesis, w' was added to $M^{\odot}$ in step 3, whereafter we follow the $\mathsf{next}^{\odot}$-pointer to w and add w in the next iteration. $\blacksquare$ $\square$

We will now prove that $N^*(P, \alpha) \subseteq M^*$. Here is an outline of the proof. Recall that for any node u in $N^*(P, \alpha)$, there exists a position q in Pos_α and a position p in P, such that u is the lca of q and p and $\mathsf{parent}^*(u)$ is in both $\mathsf{firstextent}(q)$ and $\mathsf{lastextent}(p)$. We first prove that the segment s containing u is a $*$-segment. Then it follows easily that if $u = \mathsf{bot}(s)$ then u is added to M^* in step 2 or 3. If u is not the bottom node on s, then due to the properties of the transition tree the child of u not on s contains q in its subtree. We then show that if u is not labeled α then it is added to M^* in step 2. Otherwise, it is either added as the first node in step 3, or it is a node on the path induced by the next^*-pointers on s. By similar arguments as those in the previous proof all nodes below u on this path has their parent^*-node in $\mathsf{lastextent}(p)$, the first node on the path is x found by a firstlabel query in step 3, and thus u is added to M^* in step 3. See Figure 6.

Lemma 5.8. *$N^*(P, \alpha) \subseteq M^*$.*

Proof. Let u be a node in $N^*(P, \alpha)$. Then there exists a position $p \in P$ and a position $q \in \mathsf{Pos}_\alpha$ such that $u = \mathsf{lca}(p, q)$ and $\mathsf{parent}^*(u) \in \mathsf{lastextent}(p) \cap \mathsf{firstextent}(q)$. Let s be the segment u is on. We first prove that s is a $*$-segment: If $u = \mathsf{bot}(s)$ then $\mathsf{parent}^*(\mathsf{bot}(s)) = \mathsf{parent}^*(u) \in \mathsf{lastextent}(P)$ and thus s is a $*$-segment. Otherwise, $u \prec \mathsf{bot}(s)$. By construction of T we have $p \in \mathsf{Pos}(\mathsf{bot}(s))$ and thus by Lemma 3.1 $\mathsf{bot}(s) \in \mathsf{lastextent}(p)$, since $\mathsf{parent}^*(u)$ is an ancestor of $\mathsf{bot}(s)$.

We now prove that $u \in M^*$. Let v be the lowest of $\mathsf{lca}(\mathsf{bot}(s), q-)$ and $\mathsf{lca}(\mathsf{bot}(s), q+)$ as computed in step 2. If $u = \mathsf{bot}(s)$ then $q \in \mathsf{Pos}(\mathsf{bot}(s))$ and thus $v = \mathsf{bot}(s) = u$. If $\alpha \notin \mathsf{label}(u)$ then u is added to M^* in step 2. Otherwise, it is added in step 3.

If $u \neq \mathsf{bot}(s)$, let u_s be the child of u that is on s and let u_o be the child not on s. By construction of T we have $p \in \mathsf{Pos}(u_s)$ and $p \in \mathsf{Pos}(\mathsf{bot}(s))$, since all bottom nodes of segments in T have a position from P in their subtree and any node that is the lca of two nodes in P is a branching node in T. Thus $q \in \mathsf{Pos}(u_o)$. There are two cases depending on whether u is labeled α or not.

Case 1: $\alpha \notin \mathsf{label}(u)$. Since $q \in \mathsf{Pos}(u_o)$ we have $\mathsf{Pos}(u_s) \cap \mathsf{Pos}_\alpha = \emptyset$. Thus, either $q+$ or $q-$ is in $\mathsf{Pos}(u_o)$ and then $v = u$. Therefore, u is added to M^* in step 2.

Case 2: $\alpha \in \mathsf{label}(u)$. Then u is the lca of two positions in Pos_α, which implies that there exists a position labeled α in $\mathsf{Pos}(u_s)$. It follows that either $q-$ or $q+$ is in $\mathsf{Pos}(u_s)$ and thus $u \preceq v$. If $u = v$ then u is added to M^* in the first iteration in step 3. Otherwise, $u \prec v$. Let $x = \mathsf{firstlabel}(v, \alpha)$. If $u = x$ then u is added to M^* in the first iteration in step 3.

If $u \neq x$ then $u \prec x$. Since $q \in \mathsf{Pos}(u_o)$ and $\mathsf{parent}^*(u) \in \mathsf{firstextent}(q)$ we have $q \in \delta^*(u, \alpha)$. Therefore, the highest proper descendant of u labeled α has its next^*-pointer pointing to u. It follows that there is a chain of next^*-pointers from x to u. It remains to show that for all the nodes in this chain $x = x_0, \ldots, x_k = u$ we have $\mathsf{parent}^*(x_i) \in \mathsf{lastextent}(P)$. If this is true we will reach u in step 3. Since $p \in \mathsf{Pos}(\mathsf{bot}(s))$ then all nodes on s are ancestors of p. Since $u \preceq x_i$ we have $\mathsf{parent}^*(u) \preceq \mathsf{parent}^*(x_i)$. By Lemma 3.1 we have $\mathsf{parent}^*(x_i) \in \mathsf{lastextent}(p)$.
□

5.3.2 Computing the Intervals

We now compute the lists $L^\odot$ and L^* of intervals for the nodes in $M^\odot$ and M^*, respectively. We do this by processing the nodes $M^\odot$ and M^* in inorder using a depth-first left-to-right inorder traversal of T.

First, we compute for each node v in $M^\odot$ the range $[l_v, r_v]$ of positions labeled α that are descendants of $\mathsf{right}(v)$. If v is labeled α the range $[l_v, r_v]$ is stored at v and otherwise we use the predecessor data structure to compute it using the range stored at $\mathsf{right}(v)$. Similarly, we compute for each node in M^* the range $[l_v, r_v]$ of positions labeled α that are descendants of v.

Computing $L^\odot$ We compute the list of intervals $L^\odot$ by a depth-first left-to-right inorder traversal of T. We maintain a stack S keeping track of the deepest node not finished and a counter ℓ equal to the left starting point of the currently open interval. If there is no open interval $\ell = -1$. Initially, $S = \emptyset$ and $\ell = -1$.

For each node $v \in M^\odot$ in inorder do the following:

- When we meet v in the traversal after traversing the left subtree of v: If $\ell \neq -1$ append $(\ell, l_v - 1, \mathsf{top}(S))$ to $L^\odot$. Set $\ell = l_v$ and push v onto the stack S.
- When we finish the traversal of the subtree containing v: Note that in this case $\mathsf{top}(S) = v$. If $\ell \leq r_v$ append (ℓ, r_v, v) to $L^\odot$. Pop v from S. If the stack is now empty set $\ell = -1$, otherwise set $\ell = r_v + 1$.

Note that $\mathsf{right}(v)$ might not be in T, in which case the two steps for v follow immediately after each other.

Computing L^* We maintain a stack S and counter ℓ as before.

For each node $v \in M^*$ in inorder do the following:

- First time we meet v in the traversal: If $\ell \neq -1$ append $(\ell, l_v - 1, \mathsf{top}(S))$ to L^*. Set $\ell = l_v$ and push v onto the stack S.
- Last time we meet v in the traversal: Note that in this case $\mathsf{top}(S) = v$. If $\ell \leq r_v$ append (ℓ, r_v, v) to L^*. Pop v from S. If the stack is now empty set $\ell = -1$, otherwise set $\ell = r_v + 1$.

Complexity To compute the ranges use time $O(|M^\odot| + |P| \log\log m)$ as at most one node in $M^\odot$ on each segment is not labeled α. We will store the nodes in $M^\odot$ in increasing order of depth for each segment. This is easy to maintain as we find them in order of decreasing depth. This way we can do the depth-first left-to-right traversal on the nodes of $M^\odot$ in T in linear time in the size of $M^\odot$. Thus computing $L^\odot$ takes time $O(|M^\odot|)$.

Similarly, we use time $O(|M^*|)$ to compute L^*. In summary, we have the following lemma.

LEMMA 5.9. *The sets $M^\odot$ and M^*, and the lists $L^\odot$ and L^*, can be computed in time $O(|\delta(P, \alpha)| + |P| \log\log m)$.*

5.4 Internal Transitions We will compute and return the state-set transition by computing the internal transitions on the nodes in $L^{\odot}$ and L^*. Next, we show how to compute internal transitions efficiently using 3-sided range queries. We assume that the range $[l, r]$ is given as indexes in A_α.

Computing $\delta^{\odot}_{[l,r]}(v, \alpha)$. Given an $\odot$-node v, a character α, and a range $[l, r]$ we compute $\delta^{\odot}_{[l,r]}(v, \alpha)$ as follows. We perform a 3-sided range reporting query $(l, r, \mathsf{depth}(\mathsf{right}(v))$ on D_α. That is, we return all positions in $D_\alpha[l, r]$ with a value less than or equal to $\mathsf{depth}(\mathsf{right}(v))$. This can be done by a standard technique of recursively applying range minimum queries as follows. Let j be the position returned by range minimum query on $D_\alpha[l, r]$. If $D_\alpha[j] \leq \mathsf{depth}(\mathsf{right}(v))$ return $A_\alpha[j]$ and recurse on the ranges $[l, j-1]$ and $[j+1, r]$. We stop if this is not the case or if the range is empty.

For instance, suppose we compute $\delta^{\odot}_{[2,4]}(v_2, a)$ in our example in Figure 2. The range $[2, 4]$ in A_α corresponds to the positions p_2, p_3, and p_5. We find the highest first extent in to be $f = v_4$ corresponding to both p_2 and p_3. Suppose $j = 3$ corresponding to p_3. Then we compare f with $\mathsf{right}(v_2) = v_4$ and since f is an ancestor of v_4 we report p_3 and repeat on the subarrays $[2, 2]$ and $[4, 4]$. On $[2, 2]$ we return p_2, while on $[4, 4]$ we do not get a position since v_8 is a proper descendant of v_4.

Note, that we can get the output in sorted order if we first recurse on the range $[l, j-1]$, then report $A_\alpha[j]$, and then recurse on the range $[j+1, r]$.

The algorithm returns all positions q with label α in $[l, r]$ such that $\mathsf{right}(v) \in \mathsf{firstextent}(q)$ and is thus correct. Each recursive call uses constant time and we repeat at most $2|\delta^{\odot}_{[l,r]}(v, \alpha)| + 1$ times. Hence, in total we use $O(1 + |\delta^{\odot}_{[l,r]}(v, \alpha)|)$ time.

Computing $\delta^{*}_{[l,r]}(v, \alpha)$. To compute an internal transition for the $*$-case we do a 3-sided range reporting query $(l, r, \mathsf{depth}(\mathsf{parent}^*(v))$. Correctness follows as above and the time and space bounds are the same.

In summary, we have the following result.

Lemma 5.10. *Let R be a regular expression of size m. Given R we can build a data structure in $O(m)$ space and preprocessing time, such that given a node v in R, a character $\alpha \in \Sigma$, and a range $[l, r]$ we can compute $\delta^{\odot}_{[l,r]}(v, \alpha)$ in sorted order in time $O(1 + |\delta^{\odot}_{[l,r]}(v, \alpha)|)$ and $\delta^{*}_{[l,r]}(v, \alpha)$ in sorted order in time $O(1 + |\delta^{*}_{[l,r]}(v, \alpha)|)$.*

5.5 Computing State-Set Transitions Given a set of positions P and a character α, we compute the state-set transition $\delta(P, \alpha)$ as follows. For simplicity, we assume that the positions in P are sorted according to their left-to-right order since otherwise we can sort them in additional $O(|P| \log\log m)$ time using integer sorting. The final algorithm for computing fast state-set transitions is as follows.

1. First, we construct the transition tree and all the information from Lemma 5.1 as in Subsection 5.2.

2. We then compute $M^{\odot}$, M^*, $L^{\odot}$, and L^* as in Subsection 5.3.

3. Finally, we compute and return the state-set transition by computing

$$D^{\odot} = \bigcup_{(l,r,u) \in L^{\odot}} \delta^{\odot}_{[l,r]}(u, \alpha) \qquad \text{and} \qquad D^{*} = \bigcup_{(l,r,u) \in L^{*}} \delta^{*}_{[l,r]}(u, \alpha).$$

   by processing $L^{\odot}$ and L^* from left-to-right using the procedure from Subsection 5.4 that computes $\delta^{\odot}_{[l,r]}(u, \alpha)$ and $\delta^{*}_{[l,r]}(u, \alpha)$ in sorted order. Since both lists $L^{\odot}$ and L^* are sorted the lists $D^{\odot}$ and D^* are also sorted. We merge these two lists to get the final output.

Analysis of the algorithm Step 1 uses $O(|P|)$ time by Lemma 5.1, and Step 2 uses $O(|\delta(P, \alpha)| + |P| \log\log m)$ time by Lemma 5.9. By Lemma 5.10 the time to compute all internal transitions in step 3 is

$$O\left(\sum_{(l,r,u) \in L^{\odot}} (1 + |\delta^{\odot}_{[l,r]}(u, \alpha)|) + \sum_{(l,r,u) \in L^{*}} (1 + |\delta^{*}_{[l,r]}(u, \alpha)|) \right) \tag{5.2}$$

The length of list $L^{\odot}$ is $O(|M^{\odot}|)$ as each interval endpoint is due to meeting a node in $M^{\odot}$ for the first or last time in the traversal. Similarly, $|L^*| = O(|M^*|)$. Hence, (5.2) is $O(|M^{\odot}| + |M^*| + |\delta(P, \alpha)|)$. The time to merge the two lists is linear in the total size of the lists since the lists are sorted. Thus the time for step 3 is $O(|M^{\odot}| + |M^*| + |\delta(P, \alpha)|)$. Plugging in the bounds from Lemma 5.3 and 5.4 we get a total running time of $O(|\delta(P, \alpha)| + |P| \log \log m)$.

Correctness We have already proved that $N^{\odot}(P, \alpha) \subseteq M^{\odot}$ and $N^*(P, \alpha) \subseteq M^*$. Thus, by Lemma 4.2 the set of positions computed in step 4 is $\delta(P, \alpha)$. There are at most $|P|$ nodes in $M^{\odot} \setminus N^{\odot}(P, \alpha)$. These nodes are all in $\mathsf{lastextent}(P)$ and thus the set of internal transitions on these will all be in $\delta(P, \alpha)$. Similarly, for the nodes in $M^* \setminus N^*(P, \alpha)$.

It remains to show that the partition into lists is correct. We show that a position $q \in \bigcup_{v \in M^{\odot}} \delta^{\odot}(v, \alpha)$ is covered by the lowest node in $M^{\odot}$ such that $q \in \delta^{\odot}(u, \alpha)$, i.e., where covered means that $q \in \delta^{\odot}_{[l,r]}(u, \alpha)$ for some l, r such that $(l, r, u) \in L^{\odot}$. This follows easily from the properties of the inorder traversal. The arguments for $q \in \cup_{v \in M^*} \delta^*(v, \alpha)$ are similar. For completeness, the full proof is shown below.

Lemma 5.11. *We have*

$$\bigcup_{v \in M^{\odot}} \delta^{\odot}(v, \alpha) = \bigcup_{(l,r,u) \in L^{\odot}} \delta^{\odot}_{[l,r]}(u, \alpha) \qquad \textit{and} \qquad \bigcup_{v \in M^*} \delta^*(v, \alpha) = \bigcup_{(l,r,u) \in L^*} \delta^*_{[l,r]}(u, \alpha) \,.$$

Proof. We split the proof into two cases.

Case 1: $\bigcup_{v \in M^{\odot}} \delta^{\odot}(v, \alpha) = \bigcup_{(l,r,u) \in L^{\odot}} \delta^{\odot}_{[l,r]}(u, \alpha)$. We are only adding subranges of the range of $\mathsf{right}(v)$ for any node in $v \in M^{\odot}$ to $L^{\odot}$. This immediately implies that $\cup_{(l,r,u) \in L^{\odot}} \delta^{\odot}_{[l,r]}(u, \alpha) \subseteq \cup_{v \in M^{\odot}} \delta^{\odot}(v, \alpha)$.

For the other direction, let q be a position in $\cup_{v \in M^{\odot}} \delta^{\odot}(v, \alpha)$ and let u be the deepest node in $M^{\odot}$ such that $q \in \delta^{\odot}(u, \alpha)$. We will show that $q \in \delta^{\odot}_{[l,r]}(u, \alpha)$ for some l, r such that $(l, r, u) \in L^{\odot}$.

Let $\mathsf{pred}(u) = \arg\max_{x \in M^{\odot}} \{r_x < q\}$ and $\mathsf{succ}(u) = \arg\min_{x \in M^{\odot}} \{l_x > q\}$. If neither $\mathsf{pred}(u)$ nor $\mathsf{succ}(u)$ are in $T(\mathsf{right}(u))$ then nothing happens with ℓ and the stack S after the step where we add u to the top of the stack until we leave u the last time. At this point $\mathsf{top}(S) = v$ and $\ell = l_u$ and $(\ell, r_u, u) = (l_u, r_u, u)$ is appended to $L^{\odot}$. If $\mathsf{pred}(u) \in T(\mathsf{right}(u))$ then when we leave $\mathsf{pred}(u)$, node u will be on top of the stack and $\ell = r_{\mathsf{pred}(u)} + 1$. If also $\mathsf{succ}(u) \in T(\mathsf{right}(u))$ then $\mathsf{succ}(u)$ is the next node from $M^{\odot}$ we process in our inorder traversal. When we meet $\mathsf{succ}(u)$ before traversing its right subtree we add $(\ell, l_{\mathsf{succ}(u)} - 1, \mathsf{top}(S)) = (r_{\mathsf{pred}(u)} + 1, l_{\mathsf{succ}(u)} - 1, u)$ to $L^{\odot}$. Since $r_{\mathsf{pred}(u)} + 1 \leq q \leq l_{\mathsf{succ}(u)} - 1$ we have $q \in \delta^{\odot}_{[r_{\mathsf{pred}(u)}+1, l_{\mathsf{succ}(u)}+1]}(u, \alpha)$. If $\mathsf{succ}(u) \notin T(\mathsf{right}(u))$ then the next change we perform is when leaving u. Here we add $(\ell, r_u - 1, u)$ to $L^{\odot}$. If only $\mathsf{succ}(u) \in T(\mathsf{right}(u))$ then we have $\mathsf{top}(S) = u$ and $\ell = l_u$ when we process $\mathsf{succ}(u)$ the first time (after visiting its left subtree). Then we add $(l_u, l_{\mathsf{succ}(u)} - 1, u)$ to $L^{\odot}$.

Case 2: $\bigcup_{v \in M^*} \delta^*(v, \alpha) = \bigcup_{(l,r,u) \in L^*} \delta^*_{[l,r]}(u, \alpha)$. The arguments are similar to case 1. We are only adding subranges of the range of v for any node in $v \in M^*$ to L^*. This immediately implies that $\cup_{(l,r,u) \in L^*} \delta^*_{[l,r]}(u, \alpha) \subseteq \cup_{v \in M^*} \delta^*(v, \alpha)$.

For the other direction let q be a position in $\bigcup_{v \in M^*} \delta^*(v, \alpha)$ and let u be the deepest node in M^* such that $q \in \delta^*(u, \alpha)$. We will show that $q \in \delta^*_{[l,r]}(u, \alpha)$ for some l, r such that $(l, r, u) \in L^*$.

If there are no other nodes from M^* than u in $T(u)$ then nothing happens between the first and last time we meet u in the traversal and (l_u, r_u, u) is appended to $L^{\odot}$ when we meet u the last time. Similarly to case 1, let $u_{\mathsf{pred}(u)} = \arg\max_{x \in M^*} \{r_x < q\}$ and $\mathsf{succ}(u) = \arg\min_{x \in M^*} \{l_x > q\}$. If $\mathsf{pred}(u) \in T(u)$ then when we leave $\mathsf{pred}(u)$, node u will be on top of the stack and $\ell = r_{\mathsf{pred}(u)} + 1$. If also $\mathsf{succ}(u) \in T(u)$ then $\mathsf{succ}(u)$ is the next node from M^* we process in our traversal. When we meet $\mathsf{succ}(u)$ the first time we add $(\ell, l_{\mathsf{succ}(u)} - 1, \mathsf{top}(S)) = (r_{\mathsf{pred}(u)} + 1, l_{\mathsf{succ}(u)} - 1, u)$ to L^*. Since $r_{\mathsf{pred}(u)} + 1 \leq q \leq l_{\mathsf{succ}(u)} - 1$ we have $q \in \delta^*_{[r_{\mathsf{pred}(u)}+1, l_{\mathsf{succ}(u)}+1]}(u, \alpha)$. If $\mathsf{succ}(u) \notin T(u)$ then the next change we perform is when leaving u. Here we add $(\ell, r_u - 1, u)$ to L^*. If only $\mathsf{succ}(u) \in T(u)$ then we have $\mathsf{top}(S) = u$ and $\ell = l_u$ when we process $\mathsf{succ}(u)$ the first time (after visiting its left subtree). Then we add $(l_u, l_{\mathsf{succ}(u)} - 1, u)$ to L^*. $\square$

In summary, we have the following result.

Lemma 5.12. *Given a regular expression R of size m, we can build a data structure in $O(m)$ space and preprocessing time such that given any set of positions P in R and character $\alpha \in \Sigma$, we can compute $\delta(P, \alpha)$ in $O(|P| \log\log m + |\delta(P, \alpha)|)$ time.*

6 Speeding Up State-Set Transitions

We now show how to improve the run time of computing a state-set transition $\delta(P, \alpha)$ from $O(|P| \log\log m + |\delta(P, \alpha)|)$ to $O(|P| \log\log \frac{m}{|P|} + |\delta(P, \alpha)|)$ while still using linear space. Finally, we show how to use this to obtain the main results of Theorems 1.1. We now require that the input positions in P are sorted and the output positions in $\delta(P, \alpha)$ are reported in sorted order.

First, observe that the $\log\log m$ factor is from computing $|P|$ predecessor queries and $|P|$ first label queries in steps 2 and 3 in the main algorithm in Section 5.5. The first label queries in turn are reduced to computing $O(|P|)$ predecessor queries on the Euler tour of R [25]. In both cases, we need to answer a batch of $b = \Theta(|P|)$ predecessor queries on a set of size $t = \Theta(|\mathsf{Pos}_\alpha|)$ from a universe of size $u = \Theta(m)$. The batch is provided in sorted order and the output should also be sorted.

We use a simple two-level data structure as follows. We first partition the universe into t intervals of size u/t (except possibly the last which may be smaller). For each interval, we store a predecessor data structure over the subset of the elements in the interval using a reduced universe of size u/t. Furthermore, we also store a pointer to the nearest non-empty smaller interval. Using the same predecessor data structure as in Section 5.5 for each interval the total space is $O(t)$. We answer a batch of b predecessor queries according to the following two cases:

1. If $b \leq t$ we process each predecessor query in the batch by identifying the at most two intervals containing the answer and then querying these predecessor data structures. In total, this uses $O(b \log\log(u/t)) = O(b \log\log(u/b))$ time.

2. If $b > t$ we simply merge the sorted batch of queries with the input set using $O(b + t) = O(b)$ time.

Since the batch is sorted we can also return the output in sorted order in $O(b)$ time. It follows that the running time is bounded by $O(b \log\log(u/b)$. Plugging into to the algorithm of Section 5.5, we obtain a data structure that uses $O(m)$ space and supports computing a state-set transition $\delta(P, \alpha)$ in time $O(|P| \log\log \frac{m}{|P|} + |\delta(P, \alpha)|)$. This completes the proof of Theorem 1.3.

Next consider Theorem 1.1. Let Q be a string of length n and let $S_0, \ldots, S_n$ be the state-sets in the simulation of R on Q. We implement the state-set transitions using Theorem 1.3. Note that each state-set transition produces the output in sorted order as required. Since logarithms are concave we have that the total time for the state-set transitions is

$$O\left(\sum_{i=0}^{n} |S_i| \log\log \frac{m}{|S_i|}\right) = O\left((n+1)\frac{\Delta}{n+1} \log\log \frac{m}{\Delta/(n+1)}\right) = O\left(\Delta \log\log \frac{nm}{\Delta}\right) .$$

The algorithm uses $O(m)$ space to store the representation of R and at most two state sets during the simulation. This completes the proof of Theorem 1.1.

7 Conditional Lower Bound

We now prove the conditional lower bound of Theorem 1.2. Our lower bound follows the reduction of Backurs and Indyk [7] from the orthogonal vectors problem (OVP) to regular expression matching.

The orthogonal vectors problems is defined as follows. Given two sets $A, B \subseteq \{0,1\}^d$ such that $|A| = M$, $|B| = N$, determine if there exists $a \in A$ and $b \in B$ such that $a \cdot b = 0$. For any $M = \Theta(N^\alpha)$ for some $\alpha \in (0, 1]$ and any constant $\epsilon > 0$, any algorithm for OVP with running time $O((MN)^{1-\epsilon})$ violates SETH for $d = \omega(\log N)$ [16,76].

Backurs and Indyk [7] showed hardness of regular expression matching using a reduction from OVP. Given an instance of OVP they show how to construct a regular expression R' and a string Q' in $O(Nd)$ time such that Q' matches R' if and only if there exists $a \in A$ and $b \in B$ such that $a \cdot b = 0$. The reduction works in $O(Nd)$ time, the lengths of both R' and Q' is $\Theta(Nd)$, and the alphabet is $\{x, y\}$. The regular expression R' has the form

$$R' = \left(\bigodot_{j=1}^{|Q'|} (x^* y^*)\right) \cdot P \cdot \left(\bigodot_{j=1}^{|Q'|} (x^* y^*)\right) .$$

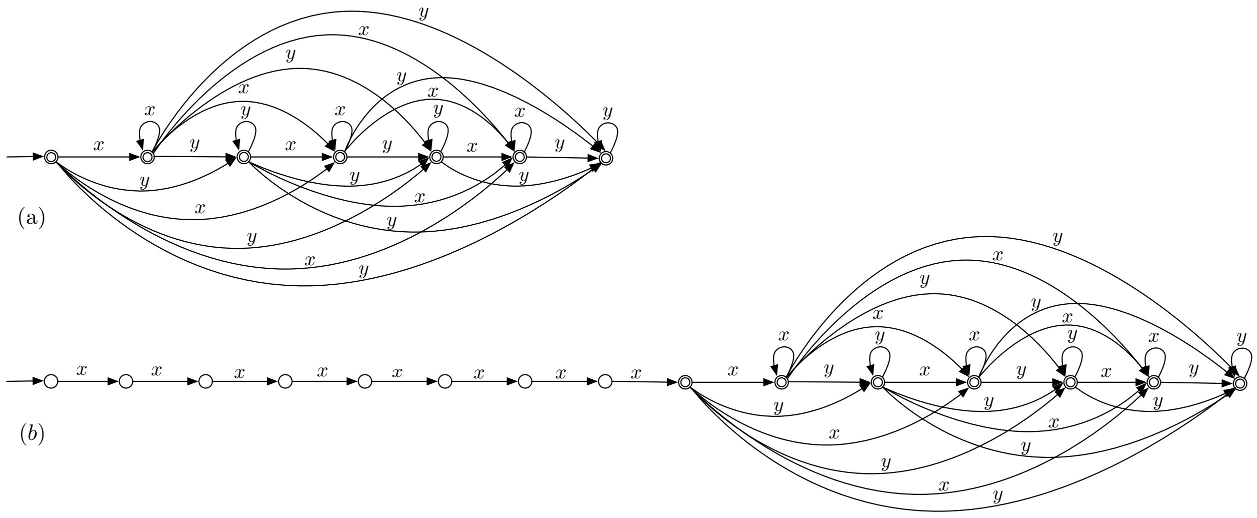


Figure 7: (a) The position automaton for $x^*y^*x^*y^*x^*y^*$. (b) The position automaton for $x^8x^*y^*x^*y^*x^*y^*$.

Here P is a regular expression of length $O(Md)$ with the property that a substring of Q' can be derived from P if and only if there exists $a \in A$ and $b \in B$ such that $a \cdot b = 0$. The precise definition of R', Q', and P can be found in [7].

We claim that $\Delta_{R',Q'} = \Theta(|Q'|^2)$. To see this first note that $\Delta_{R',Q'}$ is at most $|R'||Q'| + 1 = O(|Q'|^2)$. For the lower bound, consider the sequence $S_0, S_1, \ldots, S_{|Q'|}$ of state sets in the NFA simulation, and focus on the first $2|Q'|$ positions in R', i.e., the positions corresponding to the subexpression immediately before P. Since Q is a string of xs and ys, we have $|S_1| = |Q'|$ and $|S_i| \geq |S_{i-1}| - 1$. Thus, $\Delta_{R',Q'} = \Omega(|Q'|^2)$ and hence the claim follows. See Figure 7(a).

We can now prove the following theorem.

Theorem 7.1. *Given $A = \{a^1, \ldots, a^N\} \subseteq \{0,1\}^d$ and $B = \{b^1, \ldots, b^M\} \subseteq \{0,1\}^d$ and a constant γ, where $0 < \gamma \leq 1$, we can construct in $O((Nd)^{2/(1+\gamma)})$ time a regular expression R and a string Q, such that there exists $a \in A$ and $b \in B$ where $a \cdot b = 0$ if and only if $Q \in L(R)$. The size of R and Q is $\Theta((Nd)^{2/(1+\gamma)})$ and $\Delta_{R,Q} = \Theta(|Q|^{1+\gamma}) = \Theta(N^2d^2)$.*

Proof. We construct our instance R, Q from R' and Q' as follows.

$$Q = x^\ell \cdot Q' \qquad \text{and} \qquad R = x^\ell \cdot R'$$

where $\ell = (Nd)^{2/(1+\gamma)}$.

Clearly, Q matches R if and only if Q' matches R'. Furthermore, the NFA simulation on the first ℓ characters must produce singleton state sets (see Figure 7(b)). Hence, we have that $\Delta_{R,Q} = \ell + \Delta_{R',Q'} = \Theta((Nd)^2)$. Since $|Q| = \ell + |Q'| = \Theta((Nd)^{2/(1+\gamma)})$ we have that $\Delta = \Theta(|Q|^{1+\gamma})$. □

Theorem 1.2 follows directly from Theorem 7.1, since an $O(\Delta^{1-\epsilon}) = O((Nd)^{2-2\epsilon})$ time algorithm for regular expression matching would imply a $O((Nd)^{2-2\epsilon} + (Nd)^{2/(1+\gamma)})$ algorithm for OVP.

8 Acknowledgments

We thank the anonymous reviewers whose comments and suggestions significantly improved the presentation of the paper.

References

[1] Amir Abboud and Karl Bringmann. Tighter connections between formula-sat and shaving logs. In *Proc. 45th ICALP*, 2018.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., 1986.

[3] Alfred V Aho and Jeffrey D Ullman. *The theory of parsing, translation, and compiling*, volume 1. Prentice-Hall Englewood Cliffs, 1973.

[4] Cyril Allauzen and Mehryar Mohri. A unified construction of the Glushkov, Follow, and Antimirov automata. In *Proc. 36th MFCS*, pages 110–121, 2006.

[5] Valentin M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.

[6] Alberto Apostolico and Concettina Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2(1-4):315–336, 1987.

[7] Arturs Backurs and Piotr Indyk. Which regular expression patterns are hard to match? In *Proc. 57th FOCS*, pages 457–466, 2016.

[8] Djamal Belazzougui and Mathieu Raffinot. Approximate regular expression matching with multi-strings. In *Proc. 18th SPIRE*, pages 55–66, 2011.

[9] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *Proc. 4th LATIN*, pages 88–94, 2000.

[10] Philip Bille. New algorithms for regular expression matching. In *Proc. of the 33rd ICALP*, pages 643–654, 2006.

[11] Philip Bille and Martin Farach-Colton. Fast and compact regular expression matching. *Theor. Comput. Sci.*, 409(3):486–496, 2008.

[12] Philip Bille and Mikkel Thorup. Faster regular expression matching. In *Proc. 36th ICALP*, pages 171–182, 2009.

[13] Philip Bille and Mikkel Thorup. Regular expression matching with multi-strings and intervals. In *Proc. 21st SODA*, pages 1297–1308, 2010.

[14] T Bray, J Paoli, CM Sperberg-McQueen, Y Mailer, and F Yergeau. Extensible markup language (XML) 1.0 5th edition. Technical report, W3C, 2008.

[15] Karl Bringmann, Allan Grønlund, and Kasper Green Larsen. A dichotomy for regular expression membership testing. In *Proc. 58th FOCS*, pages 307–318, 2017.

[16] Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pages 79–97. IEEE, 2015.

[17] Sabine Broda, Markus Holzer, Eva Maia, Nelma Moreira, and Rogério Reis. A mesh of automata. *Inform. and Comput.*, 265:94–111, 2019.

[18] Anne Brüggemann-Klein. Regular expressions into finite automata. *Theoret. Comput. Sci.*, 120(2):197–213, 1993.

[19] Anne Brüggemann-Klein and Derick Wood. One-unambiguous regular languages. *Inform. and Comput.*, 140(2):229–253, 1998.

[20] Janusz A Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[21] Chia-Hsiang Chang and Robert Paige. From regular expressions to DFA's using compressed NFA's. *Theoret. Comput. Sci.*, 178(1-2):1–36, 1997.

[22] Nicola Cotumaccio, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Co-lexicographically ordering automata and regular languages - part I. *J. ACM*, 70(4), 2023.

[23] Wojciech Czerwiński, Claire David, Katja Losemann, and Wim Martens. Deciding definability by deterministic regular expressions. *J. Comput. System Sci.*, 88:75–89, 2017.

[24] Peter J Denning, Jack B Dennis, and Joseph E Qualitz. *Machines, languages, and computation.* Prentice Hall, 1978.

[25] P. F. Dietz. Fully persistent arrays. In *Proc. 1st WADS*, pages 67–74, 1989.

[26] Bartłomiej Dudek, Paweł Gawrychowski, Garance Gourdel, and Tatiana Starikovskaya. Streaming regular expression membership and pattern matching. In *Proc. 33rd SODA*, pages 670–694, 2022.

[27] David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F. Italiano. Sparse dynamic programming i: Linear cost functions. *J. ACM*, 39(3):519–545, 1992.

[28] David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F Italiano. Sparse dynamic programming ii: convex and concave cost functions. *J. ACM*, 39(3):546–567, 1992.

[29] Z. Galil. Open problems in stringology. In A. Apostolico and Z. Galil, editors, *Combinatorial problems on words, NATO ASI Series, Vol. F12*, pages 1–8. 1985.

[30] Shudi Sandy Gao, C Michael Sperberg-McQueen, and Henry Thompson. W3C XML schema definition language (XSD) 1.1 part 1: Structures. Technical report, W3C, 2012.

[31] Minos N Garofalakis, Rajeev Rastogi, and Kyuseok Shim. SPIRIT: Sequential pattern mining with regular expression constraints. In *Proc. 25th VLDB*, pages 223–234, 1999.

[32] Viliam Geffert. Translation of binary regular expressions into nondeterministic ε-free automata with $O(n \log n)$ transitions. *J. Comput. Syst. Sci.*, 66(3):451–472, 2003.

[33] Dora Giammarresi, Jean-Luc Ponty, and Derick Wood. Gluskov and Thompson constructions: a synthesis. Technical report, 1998. `http://www.cs.ust.hk/tcsc/RR/1998-11.ps.gz`.

[34] Victor M. Glushkov. The abstract theory of automata. *Russian Math. Surveys*, 16(5):1–53, 1961.

[35] Viktor M Glushkov. On a synthesis algorithm for abstract automata. *Ukr. Matem. Zhurnal*, 12(2):147–156, 1960.

[36] Benoît Groz and Sebastian Maneth. Efficient testing and matching of deterministic regular expressions. *J. Comput. Syst. Sci.*, 89:372–399, 2017.

[37] Torben Hagerup, Peter Bro Miltersen, and Rasmus Pagh. Deterministic dictionaries. *J. Algorithms*, 41(1):69–85, 2001.

[38] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[39] Juraj Hromkovič, Sebastian Seibert, Juhani Karhumäki, Hartmut Klauck, and Georg Schnitger. Communication complexity method for measuring nondeterminism in finite automata. *Information and Computation*, 172(2):202–217, 2002.

[40] Juraj Hromkovič, Sebastian Seibert, and Thomas Wilke. Translating regular expressions into small ε-free nondeterministic finite automata. *J. Comput. Syst. Sci.*, 62(4):565–588, 2001.

[41] James W Hunt and Thomas G Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, 1977.

[42] Lucian Ilie and Sheng Yu. Follow automata. *Inform. and Comput.*, 186(1):140 – 162, 2003.

[43] Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-SAT. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001.

[44] Theodore Johnson, S. Muthukrishnan, and Irina Rozenbaum. Monitoring regular expressions on out-of-order streams. In *Proc. 23nd ICDE*, pages 1315–1319, 2007.

[45] Chris Keeler and Kai Salomaa. Branching measures and nearly acyclic nfas. *International Journal of Foundations of Computer Science*, 30(06n07):1135–1155, 2019.

[46] Chris Keeler and Kai Salomaa. Structural properties of nfas and growth rates of nondeterminism measures. *Information and Computation*, 284:104690, 2022.

[47] Kenrick Kin, Björn Hartmann, Tony DeRose, and Maneesh Agrawala. Proton: multitouch gestures as regular expressions. In *Proc. SIGCHI*, pages 2885–2894, 2012.

[48] S. C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies, Ann. Math. Stud. No. 34*, pages 3–41. Princeton U. Press, 1956.

[49] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proc. SIGCOMM*, pages 339–350, 2006.

[50] Denis Kuperberg and Anirban Majumdar. Computing the width of non-deterministic automata. *Logical Methods in Computer Science*, 15, 2019.

[51] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1981.

[52] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In *Proc. 27th VLDB*, pages 361–370, 2001.

[53] Christof Löding and Stefan Repke. Decidability results on the existence of lookahead delegators for nfa. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2013)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

[54] Ping Lu, Joachim Bremer, and Haiming Chen. Deciding determinism of regular languages. *Theory Comput. Syst.*, 57:97–139, 2015.

[55] Wim Martens, Frank Neven, and Thomas Schwentick. Complexity of decision problems for xml schemas and chain regular expressions. *SIAM J. Comput.*, 39(4):1486–1530, 2010.

[56] John C Martin. *Introduction to Languages and the Theory of Computation*. McGraw-Hill, 1991.

[57] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Trans. on Electronic Computers*, 9(1):39–47, 1960.

[58] K. Mehlhorn and S. Nähler. Bounded ordered dictionaries in $O(\log\log N)$ time and $O(n)$ space. *Inform. Process. Lett.*, 35(4):183–189, 1990.

[59] Boris G Mirkin. An algorithm for constructing a base in a language of regular expressions. *Engineering Cybernetics*, 5:51–57, 1966.

[60] Makoto Murata. Extended path expressions of XML. In *Proc. 20th PODS*, pages 126–137, 2001.

[61] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of xml schema languages using formal language theory. *ACM Trans. Internet Tech.*, 5(4):660–704, 2005.

[62] E. W. Myers. A four-russian algorithm for regular expression pattern matching. *J. ACM*, 39(2):430–448, 1992.

[63] Gonzalo Navarro and Mathieu Raffinot. Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. *J. Comp. Biology*, 10(6):903–923, 2003.

[64] Abhinav Nellore, Austin Nguyen, and Reid F. Thompson. An invertible transform for efficient string matching in labeled digraphs. In *Proc. 32nd CPM*, volume 191 of *LIPIcs*, pages 20:1–20:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[65] J-L Ponty, Djelloul Ziadi, and J-M Champarnaud. A new quadratic algorithm to convert a regular expression into an automaton. In *Proc. 1st WIA*, pages 109–119. Springer, 1996.

[66] Nicola Rizzo, Alexandru I. Tomescu, and Alberto Policriti. Solving string problems on graphs using the labeled direct

product. *Algorithmica*, 84(10):3008–3033, 2022.

[67] Philipp Schepper. Fine-grained complexity of regular expression pattern matching and membership. In *Proc. 28th ESA*, 2020.

[68] Georg Schnitger. Regular expressions and NFAs without ε-transitions. In *Proc. 23rd STACS*, pages 432–443, 2006.

[69] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory: Volume I Languages and Parsing*. Springer, 1988.

[70] K. Thompson. Regular expression search algorithm. *Commun. ACM*, 11:419–422, 1968.

[71] Mikkel Thorup. Space efficient dynamic stabbing with fast queries. In *Proc. 33rd STOC*, pages 649–658, 2003.

[72] Larry Wall. *The Perl Programming Language*. Prentice Hall Software Series, 1994.

[73] W John Wilbur and David J Lipman. Rapid similarity searches of nucleic acid and protein data banks. *Proc. Nat. Acad. Sci.*, 80(3):726–730, 1983.

[74] W John Wilbur and David J Lipman. The context dependent comparison of biological sequences.

[75] Dan E Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Inform. Process. Lett.*, 17(2):81–84, 1983.

[76] Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science*, 348(2-3):357–365, 2005.

[77] Derrick Wood. *Theory of Computation: A Primer*. Addison-Wesley, 1987.

[78] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proc. ANCS*, pages 93–102, 2006.