A Dichotomy for Regular Expression Membership Testing

Karl Bringmann[*] Allan Grønlund[†] Kasper Green Larsen[‡]

Abstract

We study regular expression membership testing: Given a regular expression of size m and a string of size n, decide whether the string is in the language described by the regular expression. Its classic $O(nm)$ algorithm is one of the big success stories of the 70s, which allowed pattern matching to develop into the standard tool that it is today.

Many special cases of pattern matching have been studied that can be solved faster than in quadratic time. However, a systematic study of tractable cases was made possible only recently, with the first conditional lower bounds reported by Backurs and Indyk [FOCS'16]. Restricted to any "type" of homogeneous regular expressions of depth 2 or 3, they either presented a near-linear time algorithm or a quadratic conditional lower bound, with one exception known as the Word Break problem.

In this paper we complete their work as follows:

- We present two almost-linear time algorithms that generalize all known almost-linear time algorithms for special cases of regular expression membership testing.
- We classify all types, except for the Word Break problem, into almost-linear time or quadratic time assuming the Strong Exponential Time Hypothesis. This extends the classification from depth 2 and 3 to any constant depth.
- For the Word Break problem we give an improved $\tilde{O}(nm^{1/3} + m)$ algorithm. Surprisingly, we also prove a matching conditional lower bound for combinatorial algorithms. This establishes Word Break as the only intermediate problem.

In total, we prove matching upper and lower bounds for any type of bounded-depth homogeneous regular expressions, which yields a full dichotomy for regular expression membership testing.

[*] Max Planck Institute for Informatics, Saarland Informatics Campus. Email: kbringma@mpi-inf.mpg.de

[†] Aarhus University. Email: jallan@cs.au.dk. Supported by Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation, grant DNRF84.

[‡] Aarhus University. Email: larsen@cs.au.dk. Supported by Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation, grant DNRF84, a Villum Young Investigator Grant and an AUFF Starting Grant.

1 Introduction

A regular expression is a term involving an alphabet Σ and the operations concatenation $\circ$, union $|$, Kleene's star $\star$, and Kleene's plus $+$, see Section 2. In regular expression *membership testing*, we are given a regular expression R and a string s and want to decide whether s is in the language described by R. In regular expression *pattern matching*, we instead want to decide whether *any substring* of s is in the language described by R. A big success story of the 70s was to show that both problems have $O(nm)$ time algorithms [17], where n is the length of the string s and m is the size of R. This quite efficient running time, coupled with the great expressiveness of regular expressions, made pattern matching the standard tool that it is today.

Despite the efficient running time of $O(nm)$, it would be desirable to have even faster algorithms. A large body of work in the pattern matching community was devoted to this goal, improving the running time by logarithmic factors [15, 5] and even to near-linear for certain special cases [14, 7, 2].

A systematic study of the complexity of various special cases of pattern matching and membership testing was made possible by the recent advances in the field of conditional lower bounds, where tight running time lower bounds are obtained via fine-grained reductions from certain core problems like satisfiability, all-pairs-shortest-paths, or 3SUM (see, e.g., [9, 22, 1, 16]). Many of these conditional lower bounds are based on the Strong Exponential Time Hypothesis (SETH) [12] which asserts that k-satisfiability has no $O(2^{(1-\varepsilon)n})$ time algorithm for any $\varepsilon > 0$ and all $k \geq 3$.

The first conditional lower bounds for pattern matching problems were presented by Backurs and Indyk [4]. Viewing a regular expression as a tree where the inner nodes are labeled by $\circ, |, \star$, and $+$ and the leafs are labeled by alphabet symbols, they call a regular expression *homogeneous of type* $t \in \{\circ, |, \star, +\}^d$ if in each level i of the tree all inner nodes have type t_i, and the depth of the tree is at most d. Note that leafs may appear in any level, and the degrees are unbounded. This gives rise to natural restrictions *t-pattern matching* and *t-membership*, where we require the regular expression to be homogeneous of type t. The main result of Backurs and Indyk [4] is a characterization of t-pattern matching for all types t of depth $d \leq 3$: For each such problem they either design a near-linear time algorithm or show a quadratic lower bound based on SETH. We observed that the results by Backurs and Indyk actually even yield a classification for all t, not only for depth $d \leq 3$. This is not explicitly stated in [4], so for completeness we prove it in this paper, see Appendix A. This closes the case for t-pattern matching.

For t-membership, Backurs and Indyk also prove a classification into near-linear time and "SETH-hard" for depth $d \leq 3$, with the only exception being $+|\circ$-membership. The latter problem is also known as the Word Break problem, since it can be rephrased as follows: Given a string s and a dictionary D, can s be split into words contained in D? Indeed, a regular expression of type $\circ$ represents a string, so a regular expression of type $|\circ$ represents a dictionary, and type $+|\circ$ then asks whether a given string can be split into dictionary words. A relatively easy algorithm solves the Word Break problem in randomized time $\tilde{O}(nm^{1/2} + m)$, which Backurs and Indyk improved to randomized time $\tilde{O}(nm^{1/2-1/18} + m)$. Thus, the Word Break problem is the only studied special case of membership testing (or pattern matching) for which no near-linear time algorithm or quadratic time hardness is known. In particular, no other special case is "intermediate", i.e., in between near-linear and quadratic running time. Besides the status of Word Break, Backurs and Indyk also leave open a classification for $d > 3$.

1.1 Our Results

In this paper, we complete the dichotomy started by Backurs and Indyk [4] to a full dichotomy for any depth d. In particular, we (conditionally) establish Word Break as the only intermediate problem for (bounded-depth homogeneous) regular expression membership testing. More precisely, our results are as follows.

Word Break Problem. We carefully study the only depth-3 problem left unclassified by Backurs and Indyk. In particular, we improve Backurs and Indyk's $\tilde{O}(nm^{1/2-1/18}+m)$ randomized algorithm to a deterministic $\tilde{O}(nm^{1/3}+m)$ algorithm.

Theorem 1. *The Word Break problem can be solved in time $O(n(m\log m)^{1/3}+m)$.*

We remark that often running times of the form $\tilde{O}(n\sqrt{m})$ stem from a tradeoff of two approaches to a problem. Analogously, our time $\tilde{O}(nm^{1/3}+m)$ stems from trading off three approaches.

Very surprisingly, we also prove a matching conditional lower bound. Our result only holds for combinatorial algorithms, which is a notion without agreed upon definition, intuitively meaning that we forbid unpractical algorithms such as fast matrix multiplication. We use the following hypothesis; a slightly weaker version has also been used in [3] for context free grammar parsing. Recall that the k-Clique problem has a trivial $O(n^k)$ time algorithm, an $O(n^k/\lg^k n)$ combinatorial algorithm [18], and all known faster algorithms use fast matrix multiplication [8].

Conjecture 1. *For all $k \geq 3$, any combinatorial algorithm for k-Clique takes time $n^{k-o(1)}$.*

We provide a (combinatorial) reduction from k-Clique to the Word Break problem showing:

Theorem 2. *Assuming Conjecture 1, the Word Break problem has no combinatorial algorithm in time $(nm^{1/3-\varepsilon}+m)$ for any $\varepsilon > 0$.*

This is a surprising result for multiple reasons. First, $nm^{1/3}$ is a very uncommon time complexity, specifically we are not aware of any other problem where the fastest known algorithm has this running time. Second, it shows that the Word Break problem is an *intermediate* problem for t-membership, as it is neither solvable in almost-linear time nor does it need quadratic time. Our results below show that the Word Break problem is, in fact, the *only* intermediate problem for t-membership, which is quite fascinating.

We leave it as an open problem to prove a matching lower bound without the assumption of "combinatorial". Related to this question, note that the currently fastest algorithm for 4-Clique is based on fast rectangular matrix multiplication and runs in time $O(n^{3.256689})$ [8, 10]. If this bound is close to optimal, then we can still establish Word Break as an intermediate problem (without any restriction to combinatorial algorithms).

Theorem 3. *For any $\delta > 0$, if 4-Clique has no $O(n^{3+\delta})$ algorithm, then Word Break has no $O(n^{1+\delta/3})$ algorithm for $n = m$.*

We remark that this situation of having matching conditional lower bounds only for combinatorial algorithms is not uncommon, see, e.g., Sliding Window Hamming Distance [6].

New Almost-Linear Time Algorithms. We establish two more types for which the membership problem is in almost-linear time.

Theorem 4. *We design a deterministic $\tilde{O}(n)+O(m)$ algorithm for $|+\circ+$-membership and an expected time $n^{1+o(1)}+O(m)$ algorithm for $|+\circ|$-membership. These algorithms also work for t-membership for any subsequence t of $|+\circ+$ or $|+\circ|$, respectively.*

This generalizes *all* previously known almost-linear time algorithms for any t-membership problem, as all such types t are proper subsequences of $|+\circ+$ or $|+\circ|$. Moreover, no further generalization of our algorithms is possible, as shown below.

Dichotomy. We enhance the classification of t-membership started by Backurs and Indyk for $d \leq 3$ to a complete dichotomy for all types t. To this end, we first establish the following simplification rules.

Lemma 1. *For any type t, applying any of the following rules yields a type t' such that t-membership and t'-membership are equivalent under linear-time reductions:*

1. *replace any substring pp, for any $p \in \{\circ, |, \star, +\}$, by p,*
2. *replace any substring $+|+$ by $+|$,*
3. *replace prefix $r\star$ by $r+$ for any $r \in \{+, |\}^*$.*

We say that t-membership simplifies *if one of these rules applies. Applying these rules in any order will eventually lead to an unsimplifiable type.*

We show the following dichotomy. Note that we do not have to consider simplifying types, as they are equivalent to some unsimplifiable type.

Theorem 5. *For any $t \in \{\circ, |, \star, +\}^*$ one of the following holds:*

- *t-membership simplifies,*
- *t is a subsequence of $| + \circ +$ or $| + \circ |$, and thus t-membership is in almost-linear time (by Theorem 4),*
- *$t = +|\circ$, and thus t-membership is the Word Break problem taking time $(nm^{1/3} + m)^{1 \pm o(1)}$ (by Theorems 1 and 2, assuming Conjecture 1), or*
- *t-membership takes time $(nm)^{1-o(1)}$, assuming SETH.*

This yields a complete dichotomy for any constant depth d. We discussed the algorithmic results and the results for Word Break before. Regarding the hardness results, Backurs and Indyk [4] gave SETH-hardness proofs for t-membership on types $\circ\star$, $\circ|\circ$, $\circ + \circ$, $\circ|+$, and $\circ + |$. We provide further SETH-hardness for types $+| \circ +$, $+| \circ |$, and $| + |\circ$. To get from these (hard) *core types* to all remaining hard types, we would like to argue that all hard types contain one of the core types as a *subsequence* and thus are at least as hard. However, arguing about subsequences fails in general, since the definition of "homogeneous with type t" does not allow to leave out layers. This makes it necessary to proceed in a more ad-hoc way.

In summary, we provide matching upper and lower bounds for any type of bounded-depth homogeneous regular expressions, which yields a full dichotomy for the membership problem.

1.2 Organization

The paper is organized as follows. We start with preliminaries in Section 2. For the Word Break problem, we prove the conditional lower bounds in Section 3, followed by the matching upper bound in Section 4. In Section 5 we present our new almost-linear time algorithms for two types, and in Section 6 we prove SETH-hardness for three types. Finally, in Section 7 we prove that our results yield a full dichotomy for homogenous regular expression membership testing.

2 Preliminaries

A regular expression is a tree with leafs labelled by symbols in an alphabet Σ and inner nodes labelled by $\circ$ (at least one child), $|$ (at least one child), $+$ (exactly one child), or $\star$ (exactly one child)[1]. The language described by the regular expression is recursively defined as follows. A leaf v labelled by $c \in \Sigma$ describes the language $L(v) := \{c\}$, consisting of one word of length 1. Consider an inner node v with children $v_1, \ldots, v_\ell$. If v is labelled by $\circ$ then it describes the language $\{s_1 \ldots s_\ell \mid s_1 \in L(v_1), \ldots, s_k \in L(v_\ell)\}$, i.e., all concatenations of strings in the children's languages. If v is labelled by $|$ then it describes the language $L(v) := L(v_1) \cup \ldots L(v_\ell)$. If v is labelled $+$, then its degree ℓ must be 1 and it describes the language $L(v) := \{s_1 \ldots s_k \mid k \geq 1 \text{ and } s_1, \ldots, s_k \in L(v_1)\}$, and if v is labelled $\star$ then the same statement holds with "$k \geq 1$" replaced by "$k \geq 0$". we say that a string s *matches* a regular expression R if s is in the language described by R.

We use the following definition from [4]. We let $\{\circ, |, \star, +\}^*$ be the set of all finite sequences over $\{\circ, |, \star, +\}$; we also call this the set of *types*. For any $t \in \{\circ, |, \star, +\}^*$ we denote its length by $|t|$ and its i-th entry by t_i. We say that a regular expression is *homogeneous of type t* if it has depth at most $|t| + 1$ (i.e., any inner node has level in $\{1, \ldots, |t|\}$), and for any i, any inner node in level i is labelled by t_i. We also say that the type of any inner node at level i is t_i. This does not restrict the appearance of leafs in any level.

Definition 1. *A linear-time reduction from t-membership to t'-membership is an algorithm that, given a regular expression R of type t and length m and a string s of length n, in total time $O(n+m)$ outputs a regular expression R' of type t' and size $O(m)$, and a string s' of length $O(n)$ such that s matches R if and only if s' matches R'.*

The Strong Exponential Time Hypothesis (SETH) was introduced by Impagliazzo, Paturi, and Zane [13] and is defined as follows.

Conjecture 2. *For no $\varepsilon > 0$, k-SAT can be solved in time $O(2^{(1-\varepsilon)N})$ for all $k \geq 3$.*

Very often it is easier to show SETH-hardness based on the intermediate problem Orthogonal Vectors (OV): Given two sets of d-dimensinal vectors $A, B \subseteq \{0,1\}^d$ with $|A| = |B| = n$, determine if there exist vectors $a \in A, b \in B$ such that $\sum_{i=1}^{d} a[i] \cdot b[i] = 0$. The following OV-conjecture follows from SETH [21].

Conjecture 3. *For any $\varepsilon > 0$ there is no algorithm for OV that runs in time $O(n^{2-\varepsilon}\mathrm{poly}(d))$.*

To start off the proof for the dichotomy, we have the following hardness results from [4].

Theorem 6. *For any type t among $\circ\star$, $\circ|\circ$, $\circ + \circ$, $\circ|+$, and $\circ + |$, any algorithm for t-membership takes time $(nm)^{1-o(1)}$ unless SETH fails.*

3 Conditional Lower Bound for Word Break

In this section we prove our conditional lower bounds for the Word Break problem, Theorems 2 and 3. Both theorems follow from the following reduction.

[1] All our algorithms work in the general case where $\circ$ and $|$ may have degree 1. For the conditional lower bounds, it may be unnatural to allow degree 1 for these operations. If we restrict to degrees at least 2, it is possible to adapt our proofs to prove the same results, but this is tedious and we think that the required changes would be obscuring the overall point. We discuss this issue in more detail when it comes up, see footnote 2 on page 20.

Theorem 7. *For any $k \geq 4$, given a k-Clique instance on n vertices, we can construct an equivalent Word Break instance on a string of length $O(n^{k-1})$ and a dictionary D of total size $\|D\| = \sum_{d \in D} |d| = O(n^3)$. The reduction is combinatorial and runs in linear time in the output size.*

First let us see how this reduction implies Theorems 2 and 3.

Proof of Theorem 2. Suppose for the sake of contradiction that Word Break can be solved combinatorially in time $O(nm^{1/3-\varepsilon} + m)$. Then our reduction yields a combinatorial algorithm for k-Clique in time $O(n^{k-1} \cdot (n^3)^{1/3-\varepsilon}) = O(n^{k-3\varepsilon})$, contradicting Conjecture 1. This proves the theorem. □

Proof of Theorem 2. Assuming that 4-Clique has no $O(n^{3+\delta})$ algorithm for some $\delta > 0$, we want to show that Word Break has no $O(n^{1+\delta/3})$ algorithm for $n = m$.

Setting $k = 4$ in the above reduction yields a string and a dictionary, both of size $O(n^3)$ (which can be padded to the same size). Thus, an $O(n^{1+\delta/3})$ algorithm for Word Break with $n = m$ would yield an $O(n^{3+\delta})$ algorithm for 4-Clique, contradicting the assumption. □

It remains to prove Theorem 7. Let $G = (V, E)$ be an n-node graph on which we want to determine whether there is a k-clique. The main idea of our reduction is to construct a gadget that for any $(k-2)$-clique $S \subset V$ can determine whether there are two nodes $u, v \in V \setminus S$ such that $(u, v) \in E$ and both u and v are connected to all nodes in S, i.e., $S \cup \{u, v\}$ forms a k-clique in G. For intuition, we first present a simplified version of our gadgets and then show how to modify them to obtain the final reduction.

Simplified Neighborhood Gadget. Given a $(k-2)$-clique S, the purpose of our first gadget is to test whether there is a node $u \in V$ that is connected to all nodes in S. Assume the nodes in V are denoted $v_1, \ldots, v_n$. The alphabet Σ over which we construct strings has a symbol i for each v_i. Furthermore, we assume Σ has special symbols # and \$. The simplified neighborhood gadget for $S = \{v_{i_1}, \ldots, v_{i_{k-2}}\}$ has the text T being

$$\$123 \cdots n\#i_1\#123 \cdots n\#i_2\#123 \cdots n\# \cdots n\#i_{k-2}\#123 \cdots n\$$$

and the dictionary D contains for every edge $(v_i, v_j) \in E$, the string:

$$i(i+1) \cdots n\#j\#123 \cdots (i-2)(i-1)$$

and for every node v_i, the two strings

$$\$123 \cdots (i-2)(i-1)$$

and

$$i(i+1) \cdots n\$$$

The idea of the above construction is as follows: Assume we want to break T into words. The crucial observation is that to match T using D, we have to start with $\$123 \cdots (i-2)(i-1)$ for some node v_i. The only way we can possibly match the following part $i(i+1) \cdots n\#i_1\#$ is if D has the string $i(i+1) \cdots n\#i_1\#123 \cdots (i-2)(i-1)$. But this is the case if and only if $(v_i, v_{i_1}) \in E$, i.e. v_i is a neighbor of v_{i_1}. If indeed this is the case, we have now matched the prefix $\#i_1\#123 \cdots (i-2)(i-1)$ of the next block. This means that we can still only use strings starting with $i(i+1) \cdots$ from D. Repeating this argument for all $v_{i_j} \in S$, we conclude that we can break T into words from D if and only if there is some node v_i that is a neighbor of every node $v_{i_j} \in S$.

Simplified k-Clique Gadget. With our neighborhood gadget in mind, we now describe the main ideas of our gadget that for a given $(k-2)$-clique S can test whether there are two nodes v_i, v_j such that $(v_i, v_j) \in E$ and v_i and v_j are both connected to all nodes of S, i.e., $S \cup \{v_i, v_j\}$ forms a k-clique.

Let T_S denote the text used in the neighborhood gadget for S, i.e.

$$T_S = \$123 \cdots n\#i_1\#123 \cdots n\#i_2\#123 \cdots n\# \cdots n\#i_{k-2}\#123 \cdots n\$$$

Our k-clique gadget for S has the following text T:

$$T_S \gamma T_S$$

where γ is a special symbol in Σ. The dictionary D has the strings mentioned in the neighborhood gadget, as well as the string

$$i(i+1) \cdots n\$\gamma\$123 \cdots (j-1)$$

for every edge $(v_i, v_j) \in E$. The idea of this gadget is as follows: Assume we want to break T into words. We have to start using the dictionary string $\$123 \cdots (i-1)$ for some node v_i. For such a candidate node v_i, we can match the prefix

$$\$123 \cdots n\#i_1\#123 \cdots n\#i_2\#123 \cdots n\# \cdots n\#i_{k-2}\#$$

of $T_S \gamma T_S$ if and only if v_i is a neighbor of every node in S. Furthermore, the only way to match this prefix (if we start with $\$123 \cdots (i-1)$) covers precisely the part:

$$\$123 \cdots n\#i_1\#123 \cdots n\#i_2\#123 \cdots n\# \cdots n\#i_{k-2}\#123 \cdots (i-2)(i-1)$$

Thus if we want to also match the γ, we can only use strings

$$i(i+1) \cdots n\$\gamma\$123 \cdots (j-1)$$

for an edge $(v_i, v_j) \in E$. Finally, by the second neighborhood gadget, we can match the whole string $T_S \gamma T_S$ if and only if there are some nodes v_i, v_j such that v_i is a neighbor of every node in S (we can match the first T_S), and $(v_i, v_j) \in E$ (we can match the γ) and v_j is a neighbor of every node in S (we can match the second T_S), i.e., $S \cup \{v_i, v_j\}$ forms a k-clique.

Combining it all. The above gadget allows us to test for a given $(k-2)$-clique S whether there are some two nodes v_i and v_j we can add to S to get a k-clique. Thus, our next step is to find a way to combine such gadgets for all the $(k-2)$-cliques in the input graph. The challenge is to compute an *OR* over all of them, i.e. testing whether at least one can be extended to a k-clique. For this, our idea is to replace every symbol in the above constructions with 3 symbols and then carefully concatenate the gadgets. When we start matching the string T against the dictionary, we are matching against the first symbol of the first $(k-2)$-clique gadget, i.e. we start at an offset of zero. We want to add strings to the dictionary that *always* allow us to match a clique gadget if we have an offset of zero. These strings will then leave us at offset zero in the next gadget. Next, we will add a string that allow us to change from offset zero to offset one. We will then ensure that if we have an offset of one when starting to match a $(k-2)$-clique gadget, we can only match it if that clique can be extended to a k-clique. If so, we ensure that we will start at an offset of two in the next gadget. Next, we will also add strings to the dictionary that allow us to match any gadget if we start at an offset of two, and these strings will ensure we continue to have an offset of two. Finally, we append symbols at the end of the text that can *only* be matched if we have an offset of two after matching the last gadget. To summarize: Any breaking of T into words will start by using an offset of zero and simply skipping over $(k-2)$-cliques that cannot be extended to a k-clique. Then once a proper $(k-2)$-clique is found, a string of the dictionary is used to change the start offset from zero to one. Finally, the clique is matched and leaves us at an offset of two, after which the remaining string is matched while maintaining the offset of two.

We now give the final details of the reduction. Let $G = (V, E)$ be the n-node input graph to k-clique. We do as follows:

1. Start by iterating over every set of $(k-2)$ nodes S in G. For each such set of nodes, test whether they form a $(k-2)$-clique in $O(k^2)$ time. Add each found $(k-2)$-clique S to a list $\mathcal{L}$.

2. Let $\alpha, \beta, \gamma, \mu, \#$ and \$ be special symbols in the alphabet. For a string $T = t_1 t_2 t_3 \cdots t_m$, let

   $$[T]^{(0)}_{\alpha,\beta} = \alpha t_1 \beta \alpha t_2 \beta \cdots \alpha t_m \beta$$

   and

   $$[T]^{(1)}_{\alpha,\beta} = t_1 \beta \alpha t_2 \beta \alpha t_3 \beta \cdots \alpha t_m \beta \alpha$$

   For each node $v_i \in V$, add the following two strings to the dictionary D:

   $$[\$123 \cdots (i-2)(i-1)]^{(1)}_{\alpha,\beta}$$

   and

   $$[i(i+1) \cdots n]^{(1)}_{\alpha,\beta}$$

3. For each edge $(v_i, v_j) \in E$, add the following two strings to the dictionary:

   $$[i(i+1) \cdots n \# j \# 123 \cdots (i-2)(i-1)]^{(1)}_{\alpha,\beta}$$

   and

   $$[i(i+1) \cdots n \$ \gamma \$ 123 \cdots (j-1)]^{(1)}_{\alpha,\beta}$$

4. For each symbol σ amongst $\{1, \ldots, n, \$, \#, \gamma, \mu\}$, add the following two string to D:

   $$\alpha \sigma \beta$$

   and

   $$\beta \alpha \sigma$$

   Intuitively, the first of these strings is used for skipping a gadget if we have an offset of zero, and the second is used for skipping a gadget if we have an offset of two.

5. Also add the three strings

   $$\alpha \mu \beta \alpha \quad \$ \beta \alpha \mu \quad \beta \mu \mu$$

   to the dictionary. The first is intuitively used for changing from an offset of zero to an offset of one (begin matching a clique gadget), the second is used for changing from an offset of one to an offset of two in case a clique gadget could be matched, and the last string is used for matching the end of T if an offset of two has been achieved.

6. We are finally ready to describe the text T. For a $(k-2)$-clique $S = \{v_{i_1}, \ldots, v_{i_{k-2}}\}$, let T_S be the neighborhood gadget from above, i.e.

$$T_S := \$123 \cdots n\#i_1\#123 \cdots n\#i_2\#123 \cdots n\# \cdots n\#i_{k-2}\#123 \cdots n\$$$

For each $S \in \mathcal{L}$ (in some arbitrary order), we append the string:

$$[\mu T_S \gamma T_S \mu]^{(0)}_{\alpha,\beta}$$

to the text T. Finally, once all these strings have been appended, append another two μ's to T. That is, the text T is:

$$T := \left(\circ_{S \in \mathcal{L}} [\mu T_S \gamma T_S \mu]^{(0)}_{\alpha,\beta} \right) \mu\mu$$

We want to show that the text T can be broken into words from the dictionary D iff there is a k-clique in the input graph. Assume first there is a k-clique S in $G = (V, E)$. Let S' be an arbitrary subset of $k-2$ nodes from S. Since these form a $(k-2)$-clique, it follows that T has the substring $[\mu T_{S'} \gamma T_{S'} \mu]^{(0)}_{\alpha,\beta}$. To match T using D, do as follows: For each S'' preceeding S' in $\mathcal{L}$, keep using the strings $\alpha\sigma\beta$ from step 4 above to match. This allows us to match everything preceeding $[\mu T_{S'} \gamma T_{S'} \mu]^{(0)}_{\alpha,\beta}$ in T. Then use the string $\alpha\mu\beta\alpha$ to match the beginning of $[\mu T_{S'} \gamma T_{S'} \mu]^{(0)}_{\alpha,\beta}$. Now let v_i and v_j be the two nodes in $S \backslash S'$. Use the string $[\$123 \cdots (i-2)(i-1)]^{(1)}_{\alpha,\beta}$ to match the next part of $[\mu T_{S'} \gamma T_{S'} \mu]^{(0)}_{\alpha,\beta}$. Then since S is a k-clique, we have the string $[i(i+1) \cdots n\#h\#123 \cdots (i-2)(i-1)]^{(1)}_{\alpha,\beta}$ in the dictionary for every $v_h \in S'$. Use these strings for each $v_h \in S'$. Again, since S is a k-clique, we also have the edge $(v_i, v_j) \in E$. Thus we can use the string

$$[i(i+1) \cdots n\$\gamma\$123 \cdots (j-1)]^{(1)}_{\alpha,\beta}$$

to match across the γ in $[\mu T_{S'} \gamma T_{S'} \mu]^{(0)}_{\alpha,\beta}$. We then repeat the argument for v_j and repeatedly use the strings $[j(j+1) \cdots n\#h\#123 \cdots (j-2)(j-1)]^{(1)}_{\alpha,\beta}$ to match the second $T_{S'}$. We finish by using the string $[j(j+1) \cdots n]^{(1)}_{\alpha,\beta}$ followed by using $\$\beta\alpha\mu$. We are now at an offset where we can repeatedly use $\beta\alpha\sigma$ to match across all remaining $[\mu T_{S''} \gamma T_{S''} \mu]^{(0)}_{\alpha,\beta}$. Finally, we can finish the match by using $\beta\mu\mu$ after the last substring $[\mu T_{S''} \gamma T_{S''} \mu]^{(0)}_{\alpha,\beta}$.

For the other direction, assume it is possible to break T into words from D. By construction, the last word used has to be $\beta\mu\mu$. Now follow the matching backwards until a string not of the form $\beta\alpha\sigma$ was used. This must happend eventually since T starts with α. We are now at a position in T where the suffix can be matched by repeatedly using $\beta\alpha\sigma$, and then ending with $\beta\mu\mu$. By construction, T has $\alpha\sigma$ just before this suffix for some $\sigma \in \{1, \ldots, n, \$, \#, \gamma, \mu\}$. The only string in D that could match this without being of the form $\beta\alpha\sigma$ is the one string $\$\beta\alpha\mu$. It follows that we must be at the end of some substring $[\mu T_{S'} \gamma T_{S'} \mu]^{(0)}_{\alpha,\beta}$ and used $\$\beta\alpha\mu$ for matching the last μ. To match the preceeding n in the last $T_{S'}$, we must have used a string $[j(j+1) \cdots n]^{(1)}_{\alpha,\beta}$ for some v_j. The only strings that can be used preceeding this are strings of the form $[j(j+1) \cdots n\#h\#123 \cdots (j-2)(j-1)]^{(1)}_{\alpha,\beta}$. Since we have matched T, it follows that (v_j, v_h) is in E for every $v_h \in S'$. Having traced back the match across the last $T_{S'}$ in $[\mu T_{S'} \gamma T_{S'} \mu]^{(0)}_{\alpha,\beta}$, let v_i be the node such that the string $[i(i+1) \cdots n\$\gamma\$123 \cdots (j-1)]^{(1)}_{\alpha,\beta}$ was used to match the γ. It follows that we must have $(v_i, v_j) \in E$. Tracing the matching through the first $T_{S'}$ in $[\mu T_{S'} \gamma T_{S'} \mu]^{(0)}_{\alpha,\beta}$, we conclude that we must also have $(v_i, v_h) \in E$ for every $v_h \in S'$. This establishes that $S' \cup \{v_i, v_j\}$ forms a k-clique in G.

Finishing the proof. From the input graph G, we constructed the Word Break instance in time $O(n^{k-2}k^2)$ plus the time needed to output the text and the dictionary. For every edge $(v_i, v_j) \in E$, we added two strings to D, both of length $O(n)$. Furthermore, D had two $O(n)$ length strings for each node $v_i \in V$ and another $O(n)$ strings of constant length. Thus the total length of the strings in D is $M = O(|E|n + n) = O(n^3)$. The text T has the substring $[\mu T_{S'} \gamma T_{S'} \mu]^{(0)}_{\alpha,\beta}$ for every $(k-2)$-clique S. Thus T has length $N = O(n^{k-1})$ (assuming k is constant). The entire reduction takes $O(n^{k-1} + n^3)$ time for constant k. This finishes the reduction and proves Theorem 7.

4 Algorithm for Word Break

In this section we present an $\tilde{O}(nm^{1/3} + m)$ algorithm for the Word Break problem, proving Theorem 1. Our algorithm uses many ideas of the randomized $\tilde{O}(nm^{1/2-1/18} + m)$ algorithm by Backurs and Indyk [4], in fact, it can be seen as a cleaner execution of their main ideas. Recall that in the Word Break Problem we are given a set of strings $D = \{d_1, \ldots, d_k\}$ (the dictionary) and a string s (the text) and we want to decide whether s can be (D-)*partitioned*, i.e., whether we can write $s = s_1 \ldots s_r$ such that $s_i \in D$ for all i. We denote the length of s by n and the total size of D by $m := \|D\| := \sum_{i=1}^{k} |d_i|$.

We say that we can (D-)*jump* from j to i if the substring $s[j+1..i]$ is in D. Note that if $s[1..j]$ can be partitioned and we can jump from j to i then also $s[1..i]$ can be partitioned. Moreover, $s[1..i]$ can be partitioned if and only if there exists $0 \le j < i$ such that $s[1..j]$ can be partitioned and we can jump from j to i. For any power of two $q \ge 1$, we let $D_q := \{d \in D \mid q \le |d| < 2q\}$.

In the algorithm we want to compute the set T of all indices i such that $s[1..i]$ can be partitioned (where $0 \in T$, since the empty string can be partitioned). The trivial $O(nm)$ algorithm computes $T \cap \{0, \ldots, i\}$ one by one, by checking for each i whether for some string d in the dictionary we have $s[i - |d| + 1..i] = d$ and $i - |d| \in T$, since then we can extend the existing partitioning of $s[1..i - |d|]$ by the string d to a partitioning of s.

In our algorithm, when we have computed the set $T \cap \{0, \ldots, x\}$, we want to compute all possible "jumps" from a point before x to a point after x using dictionary words with length in $[q, 2q)$ (for any power of two q). This gives rise to the following query problem.

Lemma 2. *On dictionary D and string s, consider the following queries:*

- Jump-Query*: Given a power of two $q \ge 1$, an index x in s, and a set $S \subseteq \{x - 2q + 1, \ldots, x\}$, compute the set of all $x < i \le x + 2q$ such that we can D_q-jump from some $j \in S$ to i.*

We can preprocess D, s in time $O(n \log m + m)$ such that queries of the above form can be answered in time $O(\min\{q^2, \sqrt{qm \log q}\})$, where m is the total size of D and $n = |s|$.

Before we prove that jump-queries can be answered in the claimed running time, let us show that this implies an $\tilde{O}(nm^{1/3} + m)$-time algorithm for the Word Break problem.

Proof of Theorem 1. The algorithm works as follows. After initializing $T := \{0\}$, we iterate over $x = 0, \ldots, n-1$. For any x, and any power of two $q \le n$ dividing x, define $S := T \cap \{x - 2q + 1, \ldots, x\}$. Solve a jump-query on (q, x, S) to obtain a set $R \subseteq \{x + 1..x + 2q\}$, and set $T := T \cup R$.

To show correctness of the resulting set T, we have to show that $i \in \{0, \ldots, n\}$ is in T if and only if $s[1..i]$ can be partitioned. Note that whenver we add i to T then $s[1..i]$ can be partitioned, since this only happens when there is a jump to i from some $j \in T$, $j < i$, which inductively yields a partitioning of $s[1..i]$. For the other direction, we have to show that whenever $s[1..i]$ can be partitioned then we eventually add i to T. This is trivially true for the empty string ($i = 0$).

For any $i > 0$ such that $s[1..i]$ can be partitioned, consider any $0 \leq j < i$ such that $s[1..j]$ can be partitioned and we can jump from j to i. Round down $i - j$ to a power of two q, and consider any multiple x of q with $j \leq x < i$. Inductively, we correctly have $j \in T$. Moreover, this holds already in iteration x, since after this time we only add indices larger than x to T. Consider the jump-query for q, x, and $S := T \cap \{x - 2q + 1, \dots, x\}$ in the above algorithm. In this query, we have $j \in S$ and we can jump from j to i, so by correctness of Lemma 2 the returned set R contains i. Hence, we add i to T, and correctness follows.

For the running time, since there are $O(n/q)$ multiples of $1 \leq q \leq n$ in $\{0, \dots, n-1\}$, there are $O(n/q)$ invocations of the query algorithm with power of two $q \leq n$. Thus, the total time of all queries is up to constant factors bounded by

$$\sum_{i=0}^{\log n} \frac{n}{2^\ell} \cdot \min\left\{(2^\ell)^2, \sqrt{2^\ell m \log(2^\ell)}\right\} = n \cdot \sum_{i=0}^{\log n} \min\left\{2^\ell, \sqrt{m\ell/2^\ell}\right\}.$$

We split the sum at a point ℓ^* where $2^{\ell^*} = \Theta((m \log m)^{1/3})$ and use the first term for smaller ℓ and the second for larger. Using $\sum_{i=a}^{b} 2^i = O(2^b)$ and $\sum_{i=a}^{b} \sqrt{i/2^i} = O(\sqrt{a/2^a})$, we obtain the upper bound

$$\leq n \cdot \sum_{i=0}^{\ell^*} 2^\ell + n \cdot \sum_{i=\ell^*+1}^{\log n} \sqrt{m\ell/2^\ell} = O\Big(n 2^{\ell^*} + n\sqrt{m\ell^*/2^{\ell^*}}\Big) = O\big(n(m \log m)^{1/3}\big),$$

since $\ell^* = O(\log m)$ by choice of $2^{\ell^*} = \Theta((m \log m)^{1/3})$. Together with the preprocessing time $O(n \log m + m)$ of Lemma 2, we obtain the desired running time $O(n(m \log m)^{1/3} + m)$. □

It remains to design an algorithm for jump-queries. We present two methods, one with query time $O(q^2)$ and one with query time $O(\sqrt{qm \log q})$. The combined algorithm, where we first run the preprocessing of both methods, and then for each query run the method with the better guarantee on the query time, proves Lemma 2.

4.1 Jump-Queries in Time $O(q^2)$

The dictionary matching algorithm by Aho and Corasick [2] yields the following statement.

Lemma 3. *Given a set of strings D', in time $O(\|D'\|)$ one can build a data structure allowing the following queries. Given a string s' of length n', we compute the set Z of all substrings of s' that are contained in D', in time $O(n' + |Z|) \leq O(n'^2)$.*

With this lemma, we design an algorithm for jump-queries as follows. In the preprocessing, we simply build the data structure of the above lemma for each D_q, in total time $O(m)$.

For a jump-query (q, x, S), we run the query of the above lemma on the substring $s[x - 2q + 1..x + 2q]$ of s. This yields all pairs (j, i), $x - 2q < j < i \leq x + 2q$, such that we can D_q-jump from j to i. Iterating over these pairs and checking whether $j \in S$ gives a simple algorithm for solving the jump-query. The running time is $O(q^2)$, since the query of Lemma 3 runs in time quadratic in the length of the substring $s[x - 2q + 1..x + 2q]$.

4.2 Jump-Queries in Time $O(\sqrt{qm \log q})$

The second algorithm for jump-queries is more involved. Note that if $q > m$ then $D_q = \emptyset$ and the jump-query is trivial. Hence, we may assume $q \leq m$, in addition to $q \leq n$.

Preprocessing. We denote the reverse of a string d by d^{rev}, and let $D_q^{\mathrm{rev}} := \{d^{\mathrm{rev}} \mid d \in D_q\}$. We build a trie $\mathcal{T}_q$ for each D_q^{rev}. Recall that a trie on a set of strings is a rooted tree with each edge labeled by an alphabet symbol, such that if we orient edges away from the root then no node has two outgoing edges with the same labels. We say that a node v in the trie *spells* the word that is formed by concatenating all symbols on the path from the root to v. The set of strings spelled by the nodes in $\mathcal{T}_q$ is exactly the set of all prefixes of strings in D_q^{rev}. Finally, we say that the nodes spelling strings in D_q^{rev} are *marked*. We further annotate the trie $\mathcal{T}_q$ by storing for each node v the lowest marked ancestor m_v.

In the preprocessing we also run the algorithm of the following lemma.

Lemma 4. *The following problem can be solved in total time $O(n \log m + m)$. For each power of two $q \le \min\{n, m\}$ and each index i in string s, compute the minimal $j = j(i)$ such that $s[j..i]$ is a suffix of a string in D_q. Furthermore, compute the node $v(q, i)$ in $\mathcal{T}_q$ spelling the string $s[j(i)..i]$.*

Note that the second part of the problem is well-defined: $\mathcal{T}_q$ stores the reversed strings D_q^{rev}, so for each suffix x of a string in D_q there is a node in $\mathcal{T}_q$ spelling x.

Proof. First note that the problem decomposes over q. Indeed, if we solve the problem for each q in time $O(\|D_q\| + n)$, then over all q the total time is $O(m + n \log m)$, as the D_q partition D and there are $O(\log m)$ powers of two $q \le m$.

Thus, fix a power of two $q \le \min\{n, m\}$. It is natural to reverse all involved strings, i.e., we instead want to compute for each i the maximal j such that $s^{\mathrm{rev}}[i..j]$ is a prefix of a string in D_q^{rev}.

Recall that a suffix tree is a compressed trie containing all suffixes of a given string s'. In particular, "compressed" means that if the trie would contain a path of degree 1 nodes, labeled by the symbols of a substring $s'[i..j]$, then this path is replaced by an edge, which is succinctly labeled by the pair (i, j). We call each node of the uncompressed trie a *position* in the compressed trie, in other words, a position in a compressed trie is either one of its nodes or a pair (e, k), where e is one of the edges, labeled by (i, j), and $i < k < j$. A position p is an *ancestor* of a position p' if the corresponding nodes in the uncompressed tries have this relation, i.e., if we can reach p from p' by going up the compressed trie. It is well-known that suffix trees have linear size and can be computed in linear time [19]. In particular, iterating over all *nodes* of a suffix tree takes linear time, while iterating over all *positions* can take up to quadratic time (as each of the n suffixes may give rise to $\Omega(n)$ positions on average).

We compute a suffix tree $\mathcal{S}$ of s^{rev}. Now we determine for each node v in $\mathcal{T}_q$ the position p_v in $\mathcal{S}$ spelling the same string as v, if it exists. This task is easily solved by simultaneously traversing $\mathcal{T}_q$ and $\mathcal{S}$, for each edge in $\mathcal{T}_q$ making a corresponding move in $\mathcal{S}$, if possible. During this procedure, we store for each node in $\mathcal{S}$ the corresponding node in $\mathcal{T}_q$, if it exists. Moreover, for each edge e in $\mathcal{S}$ we store (if it exists) the pair (v, k), where k is the lowest position (e, k) corresponding to some node in $\mathcal{T}_q$, and v is the corresponding node in $\mathcal{T}_q$. Note that this procedure runs in time $O(\|D_q\|)$, as we can charge all operations to nodes in $\mathcal{T}_q$.

Since $\mathcal{S}$ is a suffix tree of s^{rev}, each leaf u of $\mathcal{S}$ corresponds to some suffix $s^{\mathrm{rev}}[i..n]$ of s^{rev}. With the above annotations of $\mathcal{S}$, iterating over all nodes in $\mathcal{S}$ we can determine for each leaf u the lowest ancestor position p of u corresponding to some node v in $\mathcal{T}_q$. It is easy to see that the string spelled by v is the longest prefix shared by $s^{\mathrm{rev}}[i..n]$ and any string in D_q^{rev}. In other words, denoting by ℓ the length of the string spelled by v (which is the depth of v in $\mathcal{T}_q$), the index $j := i + \ell - 1$ is maximal such that $s^{\mathrm{rev}}[i..j]$ is a prefix of a string in D_q^{rev}. Undoing the reversing, $j' := n + 1 - j$ is minimal such that $s[j'..n + 1 - i]$ is a suffix of a string in D_q. Hence, setting $v(q, n + 1 - i) := v$ solves the problem.

This second part of this algorithm performs one iteration over all nodes in $\mathcal{S}$, taking time $O(n)$, while we charged the first part to the nodes in $\mathcal{T}_q$, taking time linear in the size of D_q. In total over all q, we thus obtain the desired running time $O(n \log m + m)$. □

For each $\mathcal{T}_q$, we also compute a maximal packing of paths with many marked nodes, as is made precise in the following lemma. Recall that in the trie $\mathcal{T}'$ for dictionary D' the marked nodes are the ones spelling the strings in D'.

Lemma 5. *Given any trie $\mathcal{T}$ and a parameter λ, a* λ-packing *is a family $\mathcal{B}$ of pairwise disjoint subsets of $V(\mathcal{T})$ such that (1) each $B \in \mathcal{B}$ is a directed path in $\mathcal{T}$, i.e., it is a path from some node r_B to some descendant v_B of r_B, (2) r_B and v_B are marked for any $B \in \mathcal{B}$, and (3) each $B \in \mathcal{B}$ contains exactly λ marked nodes.*

In time $O(|V(\mathcal{T})|)$ we can compute a maximal (i.e., non-extendable) λ-packing.

Proof. We initialize $\mathcal{B} = \emptyset$. We perform a depth first search on $\mathcal{T}$, remembering the number ℓ_v of marked nodes on the path from the root to the current node v. When v is a leaf and $\ell_v < \lambda$, then v is not contained in any directed path containing λ marked nodes, so we can backtrack. When we reach a node v with $\ell_v = \lambda$, then from the path from the root to v we delete the (possibly empty) prefix of unmarked nodes to obtain a new set B that we add to $\mathcal{B}$. Then we restart the algorithm on all unvisited subtrees of the path from the root to v. Correctness is immediate. □

For any power of two $q \le \min\{n, m\}$, we set $\lambda_q := \left(\frac{m}{q} \log q\right)^{1/2}$ and compute a λ_q-packing $\mathcal{B}_q$ of $\mathcal{T}_q$, in total time $O(m)$. In $\mathcal{T}_q$, we annotate the highest node r_B of each path $B \in \mathcal{B}$ as being the *root* of B. This concludes the preprocessing.

Query Algorithm. Consider a jump-query (q, x, S) as in Lemma 2. For any $B \in \mathcal{B}$ let d_B be the string spelled by the root r_B of B in $\mathcal{T}_q$, and let $\pi_B = (u_1, \dots, u_k)$ be the path from the root of $\mathcal{T}$ to the root r_B of B (note that the labels of π_B form d_B). We set $S_B := \{1 \le i \le k \mid u_i \text{ is marked}\}$, which is the set containing the length of any prefix of d_B that is contained in D_q^{rev}, as the marked nodes in $\mathcal{T}_q$ correspond to the strings in D_q^{rev}.

As the first part of the query algorithm, we compute the sumsets $S{+}S_B := \{i{+}j \mid i \in S,\ j \in S_B\}$ for all $B \in \mathcal{B}$.

Now consider any $x < i \le x + 2q$. By the preprocessing (Lemma 4), we know the minimal j such that $s[j..i]$ is a suffix of some $d \in D_q$, and we know the node $v := v(q, i)$ in $\mathcal{T}_q$ spelling $s[j..i]$. Observe that the path σ from the root to v in $\mathcal{T}_q$ spells the reverse of $s[j..i]$. It follows that the strings $d \in D_q$ such that $s[i - |d| + 1..i] = d$ correspond to the marked nodes on σ. To solve the jump-query (for i) it would thus be sufficient to check for each marked node u on σ whether for the depth j of u we have $i - j \in S$, as then we can D_q-jump from $i - j$ to j and have $i - j \in S$. Note that we can efficiently enumerate the marked nodes on σ, since each node in $\mathcal{T}_q$ is annotated with its lowest marked ancestor. However, there may be up to $\Omega(q)$ marked nodes on σ, so this method would again result in running time $\Theta(q)$ for each i, or $\Theta(q^2)$ in total.

Hence, we change this procedure as follows. Starting in $v = v(q, i)$, we repeatedly go the lowest marked ancestor and check whether it gives rise to a partitioning of $s[1..i]$, until we reach the root r_B of some $B \in \mathcal{B}$. Note that by maximality of $\mathcal{B}$ we can visit less than λ_q marked ancestors before we meet any node of some $B \in \mathcal{B}$, and it takes less than λ_q more steps to lowest marked ancestors to reach the root r_B. Thus, this part of the query algorithm takes time $O(\lambda_q)$. Observe that the remainder of the path σ equals π_B. We thus can make use of the sumset $S + S_B$ as follows. The sumset $S{+}S_B$ contains i if and only if for some $1 \le j \le |\pi_B|$ we have $i - j \in S$ and we can D_q-jump from $i - j$ to i. Hence, we simply need to check whether $i \in S + S_B$ to finish the jump-query for i.

Running Time. As argued above, the second part of the query algorithm takes time $O(\lambda_q)$ for each i, which yields $O(q \cdot \lambda_q)$ in total.

For the first part of computing the sumsets, first note that D_q contains at most m/q strings, since its total size is at most m and each string has length at least q. Thus, the total number of marked nodes in $\mathcal{T}_q$ is at most m/q. As each $B \in \mathcal{B}$ contains exactly λ_q marked nodes, we have

$$|\mathcal{B}| \leq m/(q \cdot \lambda_q). \tag{1}$$

For each $B \in \mathcal{B}$ we compute a sumset $S + S_B$. Note that S and S_B both live in universes of size $O(q)$, since $S \subseteq \{x-2q+1, \dots, x\}$ by definition of jump-queries, and all strings in D_q have length less than $2q$ and thus $|S_B| \subseteq \{1, \dots, 2q\}$. After translation, we can even assume that $S, S_B \subseteq \{1, \dots, O(q)\}$. It is well-known that computing the sumset of $X, Y \subseteq \{1, \dots, U\}$ is equivalent to computing the Boolean convolution of their indicator vectors of length U. The latter in turn can be reduced to multiplication of $O(U \log U)$-bit numbers, by padding every bit of an indicator vector with $O(\log U)$ zero bits and concatenating all padded bits. Since multiplication is in linear time on the Word RAM, this yields an $O(U \log U)$ algorithm for sumset computation. Hence, performing a sumset computation $S + S_B$ can be performed in time $O(q \log q)$. Over all $B \in \mathcal{B}$, we obtain a running time of $O(|\mathcal{B}| \cdot q \log q) = O((m \log q)/\lambda_q)$, by the bound (1).

Summing up both parts of the query algorithm yields running time $O(q \cdot \lambda_q + (m \log q)/\lambda_q)$. Note that our choice of $\lambda_q = \left(\frac{m}{q} \log q\right)^{1/2}$ minimizes this time bound and yields the desired query time $O(\sqrt{qm \log q})$. This finishes the proof of Lemma 2.

5 Almost-linear Time Algorithms

In this section we prove Theorem 4, i.e. we present an $\tilde{O}(n) + O(m)$ time algorithm for $| + \circ +$-membership and an $n^{1+o(1)} + O(m)$ time algorithm for $| + \circ|$-membership. We start with presenting the solution for $| + \circ|$-membership as many of the ideas carry over to $| + \circ +$-membership.

5.1 Almost-linear Time for $| + \circ|$

For a given length-n string T and length-m regular expression R of type $| + \circ|$, over an alphabet Σ, let $R_1, \dots, R_k$ denote the regular expressions of type $\circ|$ such that $R = R_1^+ | R_2^+ | \cdots | R_k^+ | \sigma_1 | \cdots | \sigma_j$. Here the σ_j's are characters from Σ (recall that in the definition of homogenous regular expressions we allow leaves in any depth, so we can have the single characters σ_i in R). Since the σ_i's are trivial to handle, we ignore them in the remainder.

For convenience, we index the characters of T by $T[0], \dots, T[n-1]$. For R to match T, it must be the case that R_i^+ matches T for some index i. Letting ℓ_i be the number of $\circ$'s in R_i, we define $S_{i,j} \subseteq \Sigma$ for $j = 0, \dots, \ell_i$ as the set of characters from Σ such that

$$R_i = (|_{\sigma \in S_{i,0}} \sigma) \circ (|_{\sigma \in S_{i,1}} \sigma) \circ \cdots \circ (|_{\sigma \in S_{i,\ell_i}} \sigma).$$

Note that if a leaf appears in the $|$-level, then the set $S_{i,j}$ is simply a singleton set.

We observe that T matches R_i^+ iff $(\ell_i + 1)$ divides $|T| = n$ and $T[j] \in S_{i,j \bmod (\ell_i+1)}$ for all $j = 0, \dots, n-1$. In other words, if $(\ell_i + 1)$ divides n and we define sets $T_j^{\ell_i} \subseteq \Sigma$ for $j = 0, \dots, \ell_i$, such that

$$T_j^{\ell_i} = \bigcup_{h=0}^{n/(\ell_i+1)-1} \{T[h(\ell_i + 1) + j]\},$$

then we see that T matches R_i^+ iff $T_j \subseteq S_{i,j}$ for $j = 0, \dots, \ell_i$.

Note that the sets $T_j^{\ell_i}$ depend only on T and ℓ_i, i.e. the number of $\circ$'s in R_i. We therefore start by partitioning the expressions R_i into groups having the same number of $\circ$'s $\ell = \ell_i$. This takes time $O(m)$. We can immediately discard all groups where $(\ell+1)$ does not divide n. The crucial property we will use is that an integer n can have no more than $2^{O(\lg n/\lg\lg n)}$ distinct divisors [20], so we have to consider at most $2^{O(\lg n/\lg\lg n)}$ groups.

Now let $R_{i_1}, \ldots, R_{i_k}$ be the regular expressions in a group, i.e., $\ell = \ell_{i_1} = \ell_{i_2} = \cdots = \ell_{i_k}$. By a linear scan through T, we compute in $O(n)$ time the sets T_j^{ℓ} for $j = 0, \ldots, \ell$. We store the sets in a hash table for expected constant time lookups, and we store the sizes $|T_j^{\ell}|$. We then check whether there is an R_{i_h} such that $T_j^{\ell} \subseteq S_{i_h,j}$ for all j. This is done by examining each R_{i_h} in turn. For each such expression, we check whether $T_j^{\ell} \subseteq S_{i_h,j}$ for all j. For one $S_{i_h,j}$, this is done by taking each character of $S_{i_h,j}$ and testing for membership in T_j^{ℓ}. From this, we can compute $|T_j^{\ell} \cap S_{i_h,j}|$. We conclude that $T_j^{\ell} \subseteq S_{i_h,j}$ iff $|T_j^{\ell} \cap S_{i_h,j}| = |T_j^{\ell}|$.

All the membership testings, summed over the entire execution of the algorithm, take expected $O(m)$ time as we make at most one query per symbol of the input regular expression. Computing the sets T_j^{ℓ} for each divisor $(\ell+1)$ of n takes $n2^{O(\lg n/\lg\lg n)}$ time. Thus, we conclude that $|+\circ|$-membership testing can be solved in expected time $n^{1+o(1)} + O(m)$.

Sub-types. We argue that the above algorithm also solves any type t where t is a subsequence of $|+\circ|$. Type $+\circ|$ simply corresponds to the case of just one R_i and is thus handled by our algorithm above. Moreover, since there is only one R_i and thus only one divisor ℓ_i+1, the running time of our algorithm improves to $O(n+m)$. Type $|\circ|$ can be solved by first discarding all R_i with $\ell_i \neq n-1$ and then running the above algorithm. Again this leaves only one value of ℓ_i and thus the above algorithm runs in time $O(n+m)$. The type $|+|$ corresponds to the case where each $\ell_i = 0$ and is thus also handled by the above algorithm. Again the running time becomes $O(n+m)$ as there is only one value of ℓ_i. Type $|+\circ$ is the case where all sets $S_{i,j}$ are singleton sets and is thus also handled by the above algorithm. However, this type is also a subsequence of $|+\circ+$ and using the algorithm developed in the next section, we get a faster algorithm for $|+\circ$ than using the one above.

Type $||$, $|\circ$, $|+$ are trivial. Type $+|$ corresponds to the case of just one R_i having $\ell_i = 0$ and is thus solved in $O(n+m)$ time using our algorithm. Type $+\circ$ corresponds to just one R_i and only singleton sets $S_{i,j}$ and thus is also solved in $O(n+m)$ time by the above algorithm. The type $\circ|$ is the special case of $|\circ|$ in which there is only one set R_i and is thus also solved in $O(n+m)$ time. Types with just one operator are trivial.

5.2 Near-linear Time for $|+\circ+$

For a given length-n text T and length-m regular expression R of type $|+\circ+$, over an alphabet Σ, let $R_1, \ldots, R_k$ denote the regular expressions of type $\circ+$ such that $R = R_1^+|R_2^+|\cdots|R_k^+|\sigma_1|\cdots|\sigma_j$. As in Section 5.1, the σ_i's are single characters. These can easily be tested against T and thus from now on we ignore them.

Our new algorithm uses some of the ideas from Backurs and Indyk [4] for $+\circ+$-membership. From the text T, define its run-length encoding $r(T)$ as follows: Set $T' = T$ and let $r(T)$ be an initially empty list of tuples. While $|T'| > 0$, let σ be the first symbol of T' and let $\ell > 0$ be the largest integer such that σ^{ℓ} is a prefix of T' (i.e. T' starts with ℓ σ's). We remove the prefix σ^{ℓ} from T' and append the tuple (σ, ℓ) to $r(T)$.

Following Backurs and Indyk [4], we also define the run-length encoding of a regular expression R_i of type $\circ+$ as follows: Let $R_i' = R_i$ and let $r(R_i)$ be an initially empty sequence of tuples. While $|R_i'| > 0$, let $\ell > 0$ be the largest integer such that there exists a length-ℓ prefix of R_i' of the form

$\sigma\sigma^+\sigma^+\sigma\sigma\dots$ (an arbitrary concatenation of σ and σ^+) for a symbol $\sigma \in \Sigma$. Define $\ell' \geq 0$ as the number of σ's in the prefix (and $\ell - \ell'$ is the number of σ^+'s). If $\ell' = \ell$, we append the tuple $(\sigma, = \ell)$ to $r(R_i)$. Otherwise, we append the tuple $(\sigma, \geq \ell)$ to $r(R_i)$. We then delete the prefix and repeat until R_i' has length 0.

Backurs and Indyk observed the following for matching T and R_i^+ for an R_i of type $\circ+$: If the first and last character of R_i are distinct, then T matches R_i^+ iff the following two things hold:

1. $|r(R_i)|$ divides $|r(T)|$.
2. For every $j = 0, \dots, |r(T)| - 1$, if (σ, ℓ) denotes the j'th tuple of $r(T)$, we must have that the $(j \bmod |r(R_i)|)$'th tuple of $r(R_i)$ (counting from 0) is either of the form $(\sigma, = \ell)$ or $(\sigma, \geq \ell')$ for some $\ell' \leq \ell$.

The case where the first and last character of R_i are the same can be efficiently reduced to the case of distinct characters. We argue how at the end of this section and now proceed under the assumption that each R_i has distinct first and last character.

Compared to the $+ \circ +$-case solved by Backurs and Indyk, we need to additionally handle an outer $|$. Our solution for $| + \circ|$ hints at how: We start by partitioning the R_i's into groups such that all R_i's with the same value of $|r(R_i)|$ are placed in the same group. This can easily be done in $O(m)$ time. As argued in Section 5.1, there are at most $2^{O(\lg n / \lg\lg n)}$ groups we need to care about, as property 1 above implies that $|r(R_i)|$ must divide $|r(T)|$ for T to possibly match R_i^+.

For a length s dividing $|r(T)|$, we check T against all the R_i^+'s with $|r(R_i)| = s$ as follows: First let (σ_j, ℓ_j) denote the j'th tuple in $r(T)$ for $j = 0, \dots, |r(T)| - 1$. If there are two tuples (σ_j, ℓ_j) and (σ_h, ℓ_h) with $\sigma_j \neq \sigma_h$ and $j \bmod s = h \bmod s$, we can conclude that T cannot possible match R_i^+ for any R_i in the group (due to property 2 above). If this is not the case, assume for now that we have somehow computed the following values α_j^s and β_j^s for $j = 0, \dots, s - 1$:

$$\alpha_j^s := \min_{h=0}^{|r(T)|/s-1} \ell_{hs+j}.$$

$$\beta_j^s := \max_{h=0}^{|r(T)|/s-1} \ell_{hs+j}.$$

Also, let $\sigma_j \in \Sigma$ denote the character such that $\sigma_h = \sigma_j$ for all h satisfying $h \bmod s = j$.

We then test T against each R_i^+ as follows: For $j = 0, \dots, s-1$, consider the j'th tuple in $r(R_i)$. We have two cases:

a If the j'th tuple is of the form $(\mu, = \ell)$, we check whether $\mu = \sigma_j$. If not, we conclude that T and R_i^+ cannot match. Otherwise, we check whether $\alpha_j^s = \beta_j^s = \ell$. If not, we also conclude that T and R_i^+ cannot match.

b If the j'th tuple is of the form $(\mu, \geq \ell)$, we check whether $\mu = \sigma_j$. If not, we conclude that T and R_i^+ cannot match. Otherwise, we check whether $\ell \leq \alpha_j^s$. If not, we conclude that T and R_i^+ cannot match.

It follows from property 2 above that T and R_i^+ do not match if and only if the above procedure concludes that they do not match.

Assuming the availability of α_j^s and β_j^s, testing T against an R_i^+ thus takes time $O(|r(R_i)|)$. Summing over all R_i, this is $O(m)$ in total. Thus we only need an efficient algorithm for computing the α_j^s's and β_j^s's. We could compute them in $O(n)$ time per divisor s, resulting in an algorithm with running time $n2^{O(\lg n / \lg\lg n)} + O(m)$ as in Section 5.1. However, for this problem we can do better. To compute the α_j^s's and β_j^s's for all divisors s of $|r(T)|$, we start by forming a tree over all the divisors of $|r(T)|$. This tree is formed by first computing all $2^{O(\lg n/\lg\lg n)}$ divisors of $|r(T)|$ in $O(n)$ time. For each divisor s, we compute a prime factorization of s in time $O(s)$. We let the divisor $|r(T)|$ be the root of the tree, and each divisor $s < |r(T)|$ is assigned as a child of the smallest divisor t of $|r(T)|$ such that s divides t. Note that the smallest such divisor t can be found in $O(\lg n/\lg\lg n)$ time from the prime factorization of $|r(T)|$ and s (simply multiply s by the smallest prime that occurs more times in the factorization of $|r(T)|$ than in s and note that $|r(T)|$ has at most $O(\lg n/\lg\lg n)$ distinct prime factors). We compute the values α_j^s and β_j^s for all s by a top-down sweep of the constructed tree. We start at the root divisor $|r(T)|$ where we compute the $\alpha_j^{|r(T)|}$'s and $\beta_j^{|r(T)|}$'s trivially in $O(n)$ time. When processing a divisor s, let t be its parent and let p be the prime such that $p = t/s$. We observe that

$$\alpha_j^s = \min_{h=0}^{|r(T)|/s-1} \ell_{hs+j} = \min_{q=0}^{p-1} \min_{h=0}^{|r(T)|/t-1} \ell_{ht+qs+j} = \min_{q=0}^{p-1} \alpha_{qs+j}^t$$

and

$$\beta_j^t = \max_{q=0}^{p-1} \beta_{qs+j}^t.$$

Thus for a fixed s, we can compute all these values in time $O(sp) = O(t)$ given access to the α_j^t's where t is the parent of s. Now each node of the tree has at most $O(\lg n/\lg\lg n)$ children, as this is the maximum number of distinct prime factors of an integer no greater than n. We thus conclude that the total time for computing all the α_j^s's over all the divisors s is at most $O(\lg n/\lg\lg n) \cdot \sum_{s:s \text{ divides } |r(T)|} s$. The sum over all divisors of an integer n is known to be $O(n \lg\lg n)$ [11], so we conclude that the total time for computing the α_j^s's and β_j^s's is $O(n \lg n) = \tilde{O}(n)$.

Note that we also need to compute for each divisor s of $|r(T)|$ if there are two distinct characters $\sigma_j \neq \sigma_h$ in tuples (σ_j, ℓ_j) and (σ_h, ℓ_h) with $j \bmod s = h \bmod s$. If this was the case, we knew that T couldn't possibly match any of the R_i^+'s with $|r(R_i)| = s$. Observing that if s divides t and we know that two such characters exist for the divisor t, then this is also the case for s. Thus as for the α_j^s's and β_j^s's, we can compute this for all divisors using a top-down sweep of the tree in time $O(n \lg n)$. We conclude that our algorithm runs in time $O(n \lg n + m) = \tilde{O}(n) + O(m)$.

Handling Identical First and Last Characters. Given an input where some of the R_i's start and end with the same character, we extract all these R_i's. For each R_i, we first check if R_i and T both end and start with the same character. If not, we conclude that T cannot possible match R_i^+ and thus we can discard R_i. Next, if all characters of R_i are the same ($|r(R_i)| = 1$), testing it against T is trivial (we can precompute whether T has only one character, and thus we can test for matching in $O(|R_i|)$ time for each such R_i).

The remaining R_i's start and end with the same character as T. Let $r(R_i)$ be the run-length encoding of R_i and let $r(T)$ be the run-length encoding of T. We check whether the last tuple of T can possible match the last tuple of $r(R_i)$: If (σ, ℓ) is the last tuple of T, we check whether the last tuple of $r(R_i)$ is either $(\sigma, = \ell)$ or $(\sigma, \geq \ell')$ for an $\ell' \leq \ell$. If not, we can discard R_i. We make the same test with the first tuple of $r(R_i)$ and $r(T)$.

We now "rotate" the text and the remaining patterns R_i as follows: Let σ be the first and last character of T and all the remaining R_i's. We take all occurrences of σ at the end of T and move them to the front (think of it as a cyclic rotation of T). We do the same thing with the R_i's, i.e. take all occurrences of σ and σ^+ at the end of R_i and move to the front. The observation is that the "rotated" T matches a "rotated" remaining pattern R_i iff they matched before the rotation.

Moreover, T and all the R_i's now start with σ and end with something else. Thus we have reduced to the case of distinct first and last characters. The reduction took time $O(n+m)$.

Sub-types. Type $+\circ+$ corresponds to the case of only one R_i. As this means there is only one size $s = |r(R_i)|$, we can compute the α_j^s and β_j^s values directly in $O(n)$ time, giving total time $O(n+m)$. Type $|\circ+$ is solved by first discarding all R_i where $|r(R_i)| \neq |r(T)|$ (assuming we have already reduced to the case of distinct first and last symbols). This similarly leaves just one divisor of $|r(T)|$ to compute α_j^s and β_j^s values for and thus the running time improves to $O(n+m)$. Type $|++$ is trivial. Type $|+\circ$ corresponds to the case where there are no tuples $(\sigma, \geq \ell)$ in any $r(R_i)$ (i.e. all tuples are of the form $(\sigma, = \ell)$). We thus get $\tilde{O}(n) + O(m)$ for this case by using the above algorithm. Using the more directly taylored algorithm of Backurs and Indyk [4], this can be reduced to $O(n+m)$.

The only length two types not covered by the algorithm in Section 5.1 are $++$ and $\circ+$. Type $++$ is trivial. Type $\circ+$ is the type $|\circ+$ with just one set R_i. It is thus solved in time $O(n+m)$ by the above algorithm.

6 SETH-Based Lower Bounds

In this section we prove SETH-based lower bounds for t-membership for types $+|\circ+$, $+|\circ|$, and $|+|\circ$.

All three proofs are reductions from Orthogonal Vectors and follow the same overall approach. Let A, B be two sets of input vectors for the Orthogonal Vectors problem of size n and m respectively. Each reduction makes the set of vectors A into a string and the set of vectors B into a regular expression of the considered type, such that the string constructed from A matches the regular expression constructed from B iff the Orthogonal Vectors instance has an orthogonal pair of vectors.

The idea is to create a regular expression for each vector $b \in B$ that is matched by strings that encode vectors that are orthogonal to b. The string for the vectors in A are concatenated together in such a way that the regular expression can test for each vector in B, if any vector in A is orthogonal to it. The *OR* of these checks is implemented in the the string and regular expression by the offset construction described in the k-Clique reduction in Section 3. The offset construction is implemented with the regular expression $+|$ that is a part of all three considered regular expression types. As mentioned earlier the $+|$ type can implement a dictionary of regular expression to match against. The $|$ allows to pick a regular expression from the dictionary (subtree) for the input string to match, and the $+$ allows doing that the repeatedly, matching substrings from the input string to dictionary elements from left to right. The dictionary we construct contains for each $b \in B$ a regular expression that is matched only by orthogonal vectors and then regular expressions for the offset construction.

We directly apply the notation from the reduction in Section 3 as needed.

Theorem 8. *$+|\circ|$-membership takes time $(nm)^{1-o(1)}$ unless SETH fails.*

Proof. The string $T(A)$ is constructed as

$$T(A) = \circ_{a \in A} [\mu a \$ \mu]^{(0)}_{\alpha,\beta} \circ \mu\mu$$

using the α, β encoding as defined Section 3. This is the concatenation of all vectors in A with α, β surrounding each bit symbol, $\mu, \$\mu$ symbols around each encoded vector, and finally two μ symbols at the end of the string.

The regular expression must be of the form $(p_1|p_2|\ldots|p_t)^+$ where each p_i is of type $\circ|$ and these we construct the following way. Let $c(1) = 0, c(0) = 0|1$ encode simple regular expressions, and note that $c(1)$ is matched only by 0 while $c(0)$ is matched by both 0 and 1. For each $b \in B$ define $C(b) = \circ_{i=1}^{d} c(b[i])$, the concatenation of the $c-$encoding of the bits in the vector. By construction, a vector $a \in A$ is orthogonal to the vector $b \in B$ iff a read as a string matches the regular expression $C(b)$. Note that $C(b)$ is of the required form $\circ|$.

The full list $\mathcal{L} = p_1, \ldots, p_t$ of patterns is defined as follows. First, the regular expressions for checking orthogonality (notice the different offsets compared to the encoding of A as a string)

$$[C(b)]^{(1)}_{\alpha,\beta}, \quad \forall b \in B$$

Next, add regular expressions that allows skipping a prefix of the A vectors in the string $T(A)$ before matching an orthogonal vectors pair, and the patterns that allows skipping vectors in the string $T(A)$ after an orthogonal vectors pair has occured:

$$\alpha\sigma\beta \text{ and } \beta\alpha\sigma \text{ for } \sigma \in \{0, 1, \$, \mu\}.$$

Then, we need the regular expressions for controlling the offset which are

$$\alpha\mu\beta\alpha, \$\beta\alpha\mu, \beta\mu\mu,$$

the regular expression that must be used before the start of an orthogonal vectors match, the regular expression for changing offset after a succesfull orthogonal vectors match, and finally the regular expression that allows finishing the string match if an orthogonal vector match regular expression has been succesfully used. Note that all the regular expressions for implementing the offset are concatenation of symbols (leafs) and thus off the right form. The final regular expression becomes $R(B) = (|_{\ell \in \mathcal{L}} \ell)^+$.

This finishes the reduction. There is a pair of orthogonal vectors $a \in A, b \in B$ iff the string $T(A)$ matches the regular expression $R(B)$. The proof of that is essentially the K-clique proof for Word-Break from Section 3 and is omitted. The string $T(A)$ has size $O(nd)$ and and the regular expression $R(B)$ has size $O(md)$ and the total construction time is linear $O(d(n+m))$. Thus, any algorithm for $+|\circ|$-membership that runs in time $f(n,m)$ gives an algorithm for Orthogonal Vectors that runs in time $O(d(n+m) + f(dn, dm))$. In particular any algorithm for $+|\circ|$ membership that runs in $O((nm)^{1-\varepsilon})$ time for any $\varepsilon > 0$ gives an algorithm for Orthogonal Vectors that runs in time $O((nd)^{2-2\varepsilon}) = O(n^{2-2\varepsilon}\text{poly}(d))$ contradicting Conjecture 3.

□

Theorem 9. *$+|\circ+$-membership takes time $(nm)^{1-o(1)}$ unless SETH fails.*

Proof. This reduction is very similar to the reduction above, the only difference is how the ortogonality check is encoded in the regular expressions now using $\circ+$ instead of $\circ|$. This is achieved as follows.

For each vector $a \in A$ construct a string where each zero bit is replaced with $t(0) = 001$, and each one bit is replaced with $t(1) = 01$. Let $c(0) = 0^+1, c(1) = 01$ and for each $b \in B$, define $C(b) = \circ_{i=1}^{d} c(b[i])$. An encoded zero in b is matched by both $t(0)$ and $t(1)$, an encoded one is matched only by $t(0)$, and each may only be matched by the encoding of one bit from the string. Thus, a vector a is orthogonal to a vector b iff the string encoding of vector a matches the regular expression $C(b)$.

The remaining part of the reduction is identical to above and Theorem 9 follows. □

Theorem 10. *$|+|\circ$-membership takes time $(nm)^{1-o(1)}$ unless SETH fails.*

Proof. The regular expression we consider is now of the form $(p_1|\dots|p_m)$. In this reduction we construct regular expressions p_i that is matched by the string constructed from the vectors in A iff the i'th vector in B is orthogonal to a vector in A. The outer $|$ functions acts as an OR, the following $+|$ allows the offset implementation as above, and finally the concatenation operator $\circ$ is used to create regular expression that implements orthogonality checking.

The orthogonality checks for vectors in A, B are constructed as follows. For each $a = (a[1], \dots, a[d])$ we encode the position in the vector for each bit into a string as follows:

$$t(a) = \#_1 a[1] \dots \#_d a[d],$$

where $\#_{[1-d]}$ are d unique symbols.

The string $T(A)$ is defined as

$$T(A) = \circ_{a \in A} [\mu t(a) \$ \mu]^{(0)}_{\alpha,\beta} \circ \mu\mu$$

the concatenation of the encoding of each vector in A, again encoded for the offset construction.

The regular expression for each $b \in B$ is constructed by making a list $\mathcal{L}$ of regular expressions of the from $|\circ$ which is then used to make the regular expression $R(b) = (|_{p \in \mathcal{L}} p)^+$. The list $\mathcal{L}$ for b is defined as follows. For each bit $b[i]$, we add the regular expression $\#_i 0$ to $\mathcal{L}$ and if $b[i]$ is equal to zero we also add the regular expression $\#_i 1$. Denote that list of expressions $\mathcal{L}_b$. Then by construction vectors a and b are orthgonal if and only if $t(a)$ matches the regular expression $(|_{p \in \mathcal{L}_{b_i}} p)^+$.

To finalize the construction, we replace the regular expressions in $\mathcal{L}_{b_i}$ with their α, β encodings and add the offset regular expression exactly as above to get $\mathcal{L}$. This construction is used for each input vector $b \in B$, and combined with $|$ to get the regular expression for $R(B) = (|_{b \in B} R(b))$.

The correctness arguments remain the same. The string $T(A)$ only matches the regular expression $R(b)$ if there is a vector $a \in A$ that is orthogonal to b. The top level $|$ for the regular expression then allows to pick any b that is orthogonal to a vector in A. Thus, if there is a pair of orthogonal vectors $a \in A, b \in B$ the string $T(A)$ matches the regular expression $R(b)$, and thus the regular expression for B. If there is no orthogonal pair of vectors in A, B then no $R(b)$ is matched by A, and thus the regular expression for B, the union of them, is not matched by A.

The reduction constructs a string linear in the size of A, and a regular expression linear in the size of B in total time linear in the size of A and B. Thus, for the same reasons as above, Theorem 10 follows □

7 Dichotomy

In this section we prove Theorem 5, i.e., we show that the remaining results in this paper yield a complete dichotomy.

We first provide a proof of Lemma 1.

Proof of Lemma 1. Let $t \in \{\circ, |, \star, +\}^*$ and let R be a homogeneous regular expression of type t. For each claimed simplification rule (from t to some type t') we show that there is an easy transformation of R into a regular expression R' which is homogeneous of type t' and describes the same language as R (except for rule 3, where the language is slightly changed by removing the empty string). This transformation can be performed in linear time. Together with a similar transformation in the opposite direction, this yields an equivalence of t-membership and t'-membership under linear-time reductions.

(1) Suppose that t contains a substring pp, say $t_i = t_{i+1} = p \in \{\circ, |, \star, +\}$, and denote by t' the simplified type, resulting from t by deleting the entry t_{i+1}. In R, for any node on level i put a direct edge to any descendant in level $i+2$ and then delete all internal nodes in level $i+1$. This yields a regular expression R' of type t'. For any $p \in \{\circ, |, \star, +\}$ it is easy to see that both regular expressions descibe the same language. In particular, this follows from the facts $(E^\star)^\star = E^\star$ for any regular expression E (and similarly for $+$) and $(E_{1,1} \circ \ldots \circ E_{1,k(1)}) \circ \ldots \circ (E_{\ell,1} \circ \ldots \circ E_{\ell,k(\ell)}) = E_{1,1} \circ \ldots \circ E_{1,k(1)} \circ \ldots \circ E_{\ell,1} \circ \ldots \circ E_{\ell,k(\ell)}$ for any regular expressions $E_{i,j}$ (and similarly for $|$). This yields a linear-time reduction from t-membership to t'-membership. For the opposite direction, if the i-th layer is labeled p then we may introduce a new layer between $i-1$ and i containing only degree 1 nodes, labelled by p. This means we replace $E^\star$ by $(E^\star)^\star$, and similarly for $+$. For $\circ$ and $|$ the degree 1 vertices are not visible in the written form of regular expressions[2].

(2) For any regular expressions $E_1, \ldots, E_k$, the expression $\big((E_1^+)| \ldots |(E_k^+)\big)^+$ describes the same language as $\big(E_1| \ldots |E_k\big)^+$. Indeed, any string described by the former expression can be written as a concatenation of some number of strings in the union over all languages described by $E_1, \ldots, E_k$, which is exactly the language described by the latter expression. Thus, the inner $+$-operation is redundant and can be removed. Specifically, for a homogeneous regular expression of type t with $t_i, t_{i+1}, t_{i+2} = +|+$ we may contract all edges between layer $i+1$ and $i+2$ to obtain a homogeneous regular expression R' of type t' describing the same language as R. This yields a linear-time reduction from t-membership to t'-membership. For the opposite direction, we may introduce a redundant $+$-layer below any $+|$-layers without changing the language.

(3) Note that for any regular expression E we can check in linear time whether it describes the empty string, by a simple recursive algorithm: No leaf describes the empty string. Any $\star$-node describes the empty string. A $+$-node describes the empty string if its child does so. A $|$-node describes the empty string if at least one of its children does so. A $\circ$-node describes the empty string if all of its children do so. Perform this recursive algorithm to decide whether the root describes the empty string.

Now suppose that t has prefix $r\star$ for some $r \in \{+, |\}^*$, and denote by t' the type where we replaced the prefix $r\star$ by $r+$. (Note that we could restrict our attention to the case where r is a subsequence of $| + |$, since otherwise rules 1 or 2 apply.) Let R be homogeneous of type t, and s a string for which we want to decide whether it is described by R. If s is the empty string, then as described above we can solve the instance in linear time. Otherwise, we adapt R by labeling any

[2] This is the only place where we need to use degree 1 vertices; all other proofs in this paper also work with the additional requirement that each $|$- or $\circ$-node has degree at least 2 (cf. footnote 1 on page 4). Note that for our construction here in Lemma 1 it is essential to use degree 1 nodes: Two levels of $\circ$-nodes with each node having degree at least 2 have at least 4 children, so it seems to be impossible to embed a regular expression of the shorter type (where the degree could be 2 or 3) into a regular expression of the longer type (where the combined degree is at least 4). For $|$-nodes, there is an easy trick by adding dummy leaves that are labeled by fresh symbols not appearing in the input string s, cf. footnote 3 on page 21. For $\circ$-nodes, we are not aware of any similar trick.

An option to still make our proofs work is to consider homogeneous regular expressions of type t where each $\circ$- or $|$-node has degree at least d, giving rise to the (t,d)-membership problem. Then (t,d)-membership has the same complexity as the problem t-membership studied in this paper, irrespective of the (constant) value of d. To show this, for each reduction rule from t-membership to t'-membership of Lemmas 1 and 6 one can easily establish a similar reduction from (t,d)-membership to (t',d')-membership *for some* $d' \geq 2$ depending only on t, t', d. Moreover, since (t,d)-membership is a special case of (t,d')-membership for any $d \geq d'$, it suffices that we prove all algorithmic results for $(t,2)$-membership, and all negative results for (t,d)-membership for any constant $d \geq 2$ to show that the complexity of (t,d)-membership does not depend on d. As our algorithms even work for degree 1 nodes, the former is satisfied. For the latter, it suffices to observe that in the direct hardness proofs of Backurs and Indyk [4] and our Theorems 8–10, all degrees grow with the input size, and thus any constant lower bound does not break the reduction. This is a viable option for proving our results in the more restricted setting with degrees at least two, but we think that it would obscure the overall point of the paper.

internal node in layer $|r| + 1$ by $+$, obtaining a homogeneous regular expression R' of type t'. Then R describes s if and only R' describes s. Indeed, since $\star$ allows more strings than $+$, R describes a superset of R'. Moreover, if R describes s, then we trace the definitions of $|$ and $+$ as follows. We initialize node v as the root and string x as s. If the current vertex v is labelled $|$, then the language of v is the union over the children's languages, so the current string x is contained in the language of at least one child v', and we recursively trace (v', x). If v is labelled $+$, then we can write x as a concatenation $x_1 \dots x_k$ for some $k \geq 1$ and all x_i in the language of the child v' of v. Note that we may remove all x_ℓ which equal the empty string. For each remaining x_ℓ we trace (v', x_ℓ). Running this traceback procedure until layer $|r| + 1$ yields trace calls (v_i, x_i) such that v_i describes x_i for all i, and the x_i partition s. Since by construction each x_i is non-empty, x_i is still in the language of v_i if we relabel v_i by $+$. This shows that s is also described by R', where we relabeled each node in layer $|r| + 1$ by $+$.

Finally, applying these rules eventually leads to an unsimplifiable type, since rules 1 and 2 reduce the length of the type and rule 3 reduces the number of $\star$-operations. Thus, each rule reduces the sum of these two non-negative integers. □

Lemma 6. *For types t, t', there is a linear-time reduction from t'-membership to t-membership if one of the following sufficient conditions holds:*

1. *t' is a prefix of t,*
2. *we may obtain t from the sequence t' by inserting a $|$ at any position,*
3. *we may obtain t from the sequence t' by replacing a $\star$ by $+\star$, or*
4. *t' starts with $\circ$ and we may obtain t from the sequence t' by prepending a $+$.*

Proof. (1) The definition of "homogeneous with type t" does not restrict the appearance of leafs in any level. Thus, any homogeneous regular expression of type t' (i.e., the prefix) can be seen as a homogeneous regular expression of type t (i.e., the longer string) where all leaves appear in levels at most $|t'| + 1$. This shows that trivially t'-membership is a special case of t-membership.

(2) Let t be obtained from t' by inserting a $|$ at position i. Consider any homogeneous regular expression R' of type t'. Viewed as a tree, in R we subdivide any edge from a node in layer $i - 1$ to a node in layer i, and mark the newly created nodes by $|$. This yields a regular expression R of type t. Since a $|$ with degree 1 is trivial, the language described by R is the same as for R'.[3]

(3) Let R' be a homogeneous regular expression of type t' with $t'_i = \star$. Subdivide any edge in R' from a node in layer $i - 1$ to an internal node in layer i, and label the newly created nodes by $+$. This yields a regular expression R of type t. (Note that by this construction any leaf of R' in layer i stays a leaf of R in layer i.) Consider any newly created node v with child u. Since u is an internal node, it is labeled by $\star$ and (its subtree) represents a regular expression $E^\star$. The newly created node v thus represents the regular expression $(E^\star)^+$. Using the fact $(E^\star)^+ = E^\star$ for any regular expression E, it follows that R and R' describe the same language. Hence, we obtain a linear-time reduction from t'-membership to t-membership.

(4) Let $R' = E_1 \circ \ldots \circ E_k$ be a homogeneous regular expression of type t'. Let s' be the input string for which we want to know whether it matches R', and let x be a fresh symbol not occuring in s'. We construct the expression $R := (x \circ E_1 \circ \ldots \circ E_k \circ x)^+$ and the string $s := xs'x$. Note

[3] An alternative to using $|$-nodes of degree 1 is as follows: Let x be a fresh symbol, not occuring in the input string s' for which we want to know whether it matches R'. For each newly created node v by the subdivision process, add a new leaf node ℓ_v marked by x and connect it to v. This again yields a regular expression of type t. Since x is a fresh symbol, the newly added leaves cannot match any symbol in s', and thus s' matches R' if and only if s' matches R.

that R is homogeneous of type t. Moreover, there are exactly two occurences of x in s, and thus s matches R if and only if s' matches $E_1 \circ \ldots \circ E_k = R'$. Thus, we obtain a linear-time reduction from t'-membership to t-membership. □

We are now ready to prove the dichotomy theorem for the membership problem.

Proof of Theorem 5. The natural way of enumerating all types yields a tree with vertex set $\{\circ, |, \star, +\}^*$, where types t and t' are connected by an edge if we obtain t from t' by appending one of the operations $\circ, |, \star, +$. To keep this tree simple, we directly apply the simplification rule Lemma 1 (1), i.e., we do not consider types with consecutive equal operations. We split this tree into the three Figures 1-3 according to the first operation $\circ/\star, +, |$.

In the following we describe these figures. If t-membership is solvable in almost-linear time, then in the figures we mark the node corresponding to t by the fastest known running time, and we refer to [4] or our Theorem 4. We remark that most $O(n + m)$ algorithms are immediate, but for completeness we refer to [4].

If t-membership simplifies, then t-membership is equivalent to t'-membership for some different, non-simplifying type t' in the tree. In this case, we mark t by the corresponding simplification rule of Lemma 1. Note that the simplification rules have the property that if t simplifies then any descendant of t in the tree also simplifies. Thus, we may ignore the subtree of any simplifying type t.

If we can show that any algorithm for t-membership takes time $(nm)^{1-o(1)}$ unless SETH fails, then in the figures we mark t as "hard". Note that by Lemma 6 (1), if t is hard then any descendant of t in the tree is also hard. Thus, we may ignore the subtree of t, and we only mark minimal hardness results. From the results by Backurs and Indyk [4] (Theorem 6) we know that t-membership takes time $(nm)^{1-o(1)}$ under SETH for the types $\circ\star$, $\circ|\circ$, $\circ+\circ$, $\circ|+$, and $\circ+|$. Our Theorems 8-10 add the types $+|\circ+$, $+|\circ|$, and $|+|\circ$. If there is such a direct hardness proof of t-membership, then in the figures we refer to the corresponding theorem. In all other minimal hard cases, there is a combination of the reduction rules, Lemma 6 (2–4), resulting in a type t' such that hardness of t'-membership follows from Theorem 6 and t'-membership has a linear time reduction to t-membership. In this case, in the figures we additionally mark the node corresponding to t by t'. E.g., $|\circ|\star$-membership contains $\circ\star$-membership as a special case (since by Lemma 6 (2) we may remove any $|$ operations) and $\circ\star$-membership is hard by Theorem 6, so we mark the node corresponding to $|\circ|\star$ by "hard $\circ\star$".

It is easy to check that our Figures 1-3 indeed enumerate all cases and thus contain all maximal algorithmic results and minimal hardness results. The claimed dichotomy of Theorem 5 now follows by inspecting these figures. □

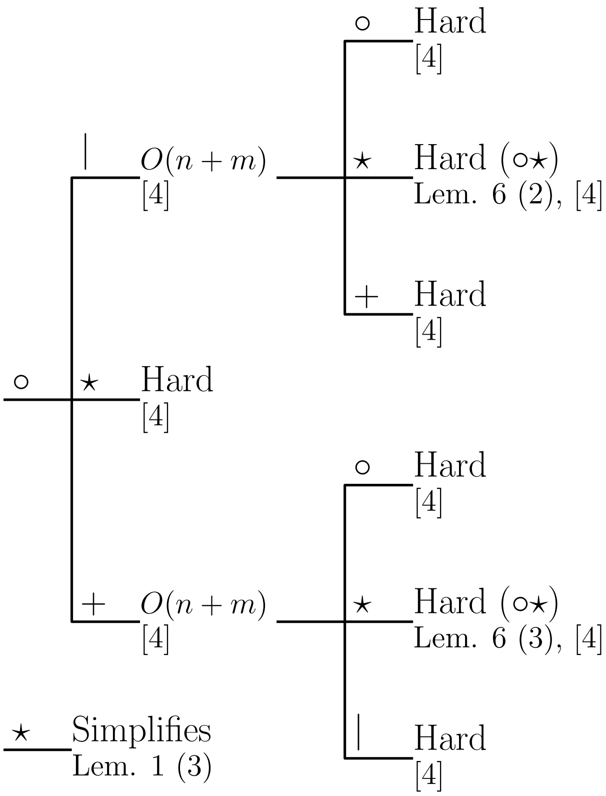

Figure 1: Tree diagram for t-membership with first operation $\circ$ or $\star$. For details see the proof of Theorem 5.

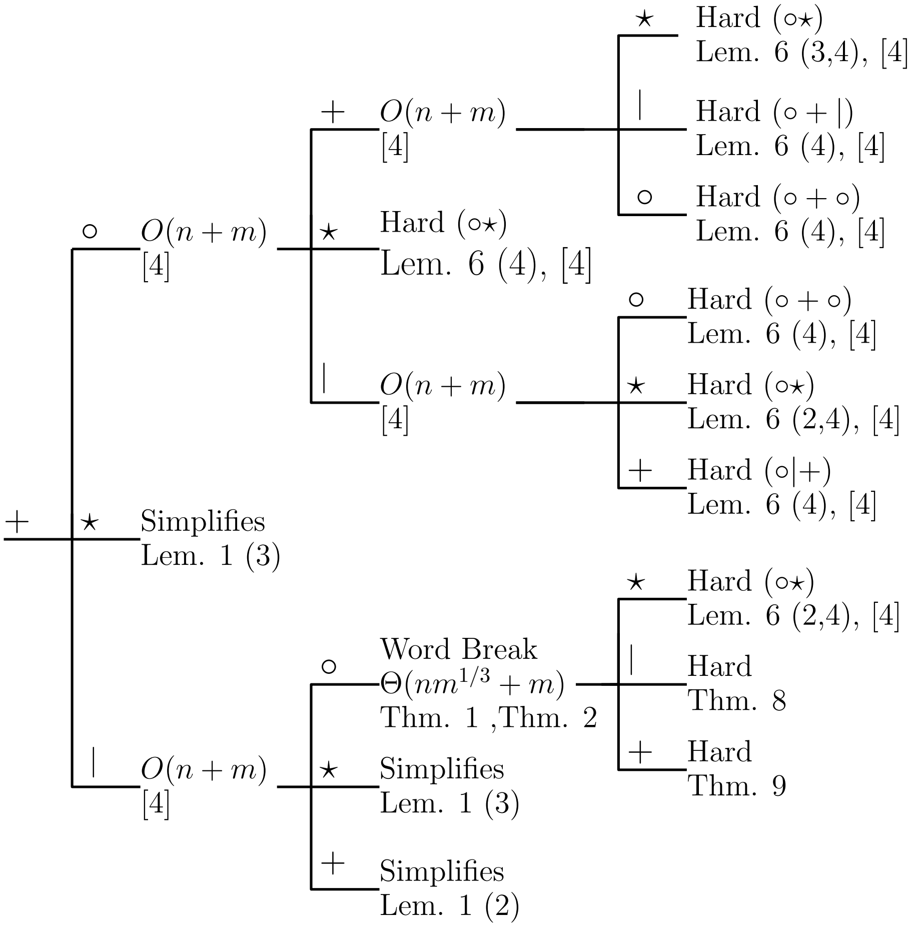

Figure 2: Tree diagram for t-membership with first operation +. For details see the proof of Theorem 5.

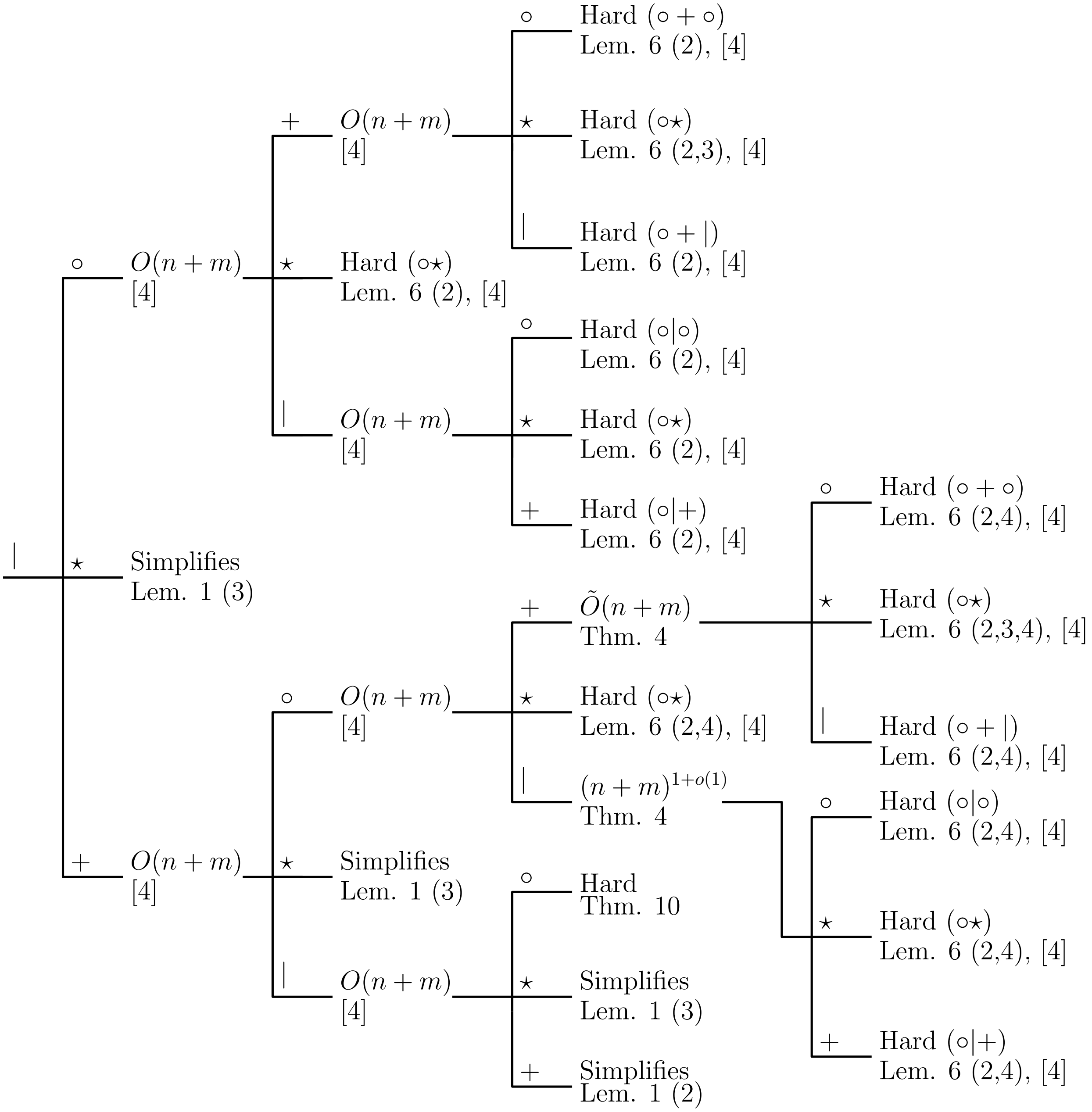

Figure 3: Tree diagram for t-membership with first operation $|$. For details see the proof of Theorem 5.

References

[1] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *Proceedings of the 2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 434–443, 2014.

[2] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June 1975.

[3] Virginia Vassilevska Williams Amir Abboud, Arturs Backurs. If the current clique algorithms are optimal, so is valiant's parser. In *56th Annual IEEE Symposium on Foundations of Computer Science*, 2015.

[4] Arturs Backurs and Piotr Indyk. Which regular expression patterns are hard to match? In *57th Annual IEEE Symposium on Foundations of Computer Science*, 2016.

[5] Philip Bille and Mikkel Thorup. Faster regular expression matching. In *36th International Colloquium on Automata, Languages and Programming*, pages 171–182, 2009.

[6] Raphael Clifford. Matrix multiplication and pattern matching under hamming norm. http://www.cs.bris.ac.uk/Research/Algorithms/events/BAD09/BAD09/Talks/BAD09-Hammingnotes.pdf.

[7] Richard Cole and Ramesh Hariharan. Verifying candidate matches in sparse and wildcard matching. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, pages 592–601, 2002.

[8] Friedrich Eisenbrand and Fabrizio Grandoni. On the complexity of fixed parameter clique and dominating set. *Theoretical Computer Science*, 326(1):57–67, 2004.

[9] Anka Gajentaan and Mark H. Overmars. On a class of o(n2) problems in computational geometry. *Comput. Geom. Theory Appl.*, 5(3):165–185, October 1995.

[10] François Le Gall. Faster algorithms for rectangular matrix multiplication. In *53rd Annual IEEE Symposium on Foundations of Computer Science*, pages 514–523, 2012.

[11] Thomas Hakon Grönwall. Some asymptotic expressions in the theory of numbers. *Trans. Amer. Math. Soc.*, 14:113–122, 1913.

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

[13] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *J. Computer and System Sciences*, 63(4):512–530, 2001.

[14] D. E. Knuth, J. H. Morris, and V. B. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.

[15] Gene Myers. A four russians algorithm for regular expression pattern matching. *J. ACM*, 39(2):432–448, April 1992.

[16] Mihai Pătraşcu and Ryan Williams. On the possibility of faster sat algorithms. In *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1065–1075, 2010.

[17] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[18] Virginia Vassilevska. Efficient algorithms for clique problems. *Inf. Process. Lett.*, 109(4):254–257, January 2009.

[19] Peter Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[20] S. Wigert. Sur l'ordre de grandeur du nombre des diviseurs d'un entier. *Ark. Math. Astron. Fys.*, 3:113–140, 1906-07.

[21] Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theor. Comput. Sci.*, 348(2):357–365, December 2005.

[22] Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *Proceedings of the 2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pages 645–654, 2010.

A Dichotomy for Pattern Matching

For the regular expression pattern matching problem Backurs and Indyk [4] characterized all types of length at most 3. In this section, we show that their results in fact imply a complete dichotomy for all types of any (constant) depth.

Recall that in the pattern matching problem we are given a regular expression R and a string s and want to decide whether *any substring of* s is in the language described by R (whereas in the membership problem we want to know whether the whole string s is in the language of R). For t-pattern matching, i.e., the natural restriction of pattern matching to regular expressions R that are homogeneous of type t, Backurs and Indyk established that any algorithm takes time $(nm)^{1-o(1)}$ (unless SETH fails) for any type t among $\circ\star$, $\circ|\circ$, $\circ+\circ$, $\circ|+$, $\circ+|$, $|\circ|$, and $|\circ+$. Positive results for t-pattern matching date back to the 70ies, since $\circ$-pattern matching corresponds to standard string matching, which has a classic linear-time algorithm due to Knuth, Morris, and Pratt [14]. Further special cases with near-linear time algorithms are $|\circ$ (dictionary matching [2]), $\circ|$ (superset matching [7]), and $\circ+$ ([4]). An easy observation moreover shows that t-pattern matching is in linear time whenever t starts with $\star$ or $|\star$.

Lemma 7. *For any type t with prefix $\star$ or with prefix $|\star$, t-pattern matching is in linear time.*

Proof. In both cases, any homogeneous regular expression R of type t matches the empty string, and thus the empty substring of any string s is in the language described by R, so that any string s is a YES-instance – unless all leaves of R are too high, so that there is no internal $\star$-node. In the latter case, either the root of R is a leaf, so that the language described by R consists of a single string of length one, or the root is a $|$-node all of whose children are leaves, so that the language described by R is a finite set of strings of length one. In both cases, it is easy to check in linear time whether (any substring of) a given string s matches R. □

For pattern matching, the following two lemmas show similar simplification rules and reduction rules as for the membership problem (cf. Lemmas 1 and 6).

Lemma 8. *For any type t, applying any of the following rules yields a type t' such that t-pattern matching and t'-pattern matching are equivalent under linear-time reductions:*

1. *replace any substring pp, for some $p \in \{\circ, |, \star, +\}$, by p,*

2. *remove prefix $+$,*

3. *replace prefix $|+$ by $|$.*

We say that t-pattern matching simplifies *if one of these rules applies. Applying these rules in any order will eventually lead to an unsimplifiable type.*

Proof. Rule 1 also holds for the membership problem. The same proof as in Lemma 1 works verbatim for pattern matching, since we argued that we can turn any homogeneous regular expression R of type t into one of type t' without changing its described language, and without looking at the input string s (and similarly the other way round).

For rule 2, note that string s contains a substring matching regular expression E^+ if and only if it contains a substring matching E. Indeed, the former means that s has a substring $s' = s'_1 \dots s'_k$ with each s'_i matching E, but then any s'_i is a substring of s matching E. The opposite direction holds since E^+ describes a superset of the language of E. Thus, we may remove any prefix $+$, and the inverse operation of introducing a redundant prefix $+$ also does not change the problem.

For rule 3, we similarly have that string s contains a substring matching regular expression $E_1^+ | \dots | E_k^+$ if and only if it contains a substring matching $E_1 | \dots | E_k$. □

Lemma 9. *For types t, t', there is a linear-time reduction from t'-membership to t-membership if one of the following sufficient conditions holds:*

1. *t' is a prefix of t,*

2. *we may obtain t from the sequence t' by replacing a $\star$ by $+\star$, or*

3. *we may obtain t from the sequence t' by inserting a $|$ at any position.*

Proof. The proof of Lemma 6 works verbatim. □

We are now ready to prove the following dichotomy.

Theorem 11. *For any $t \in \{\circ, |, \star, +\}^*$ one of the following holds:*

- *t-pattern matching simplifies,*

- *t has prefix $\star$ or $|\star$ and thus t-pattern matching is in linear time (Lemma 7),*

- *t is a subsequence of $|\circ$ (dictionary matching [2]), $\circ|$ (superset matching [7]), or $\circ+$ ([4]), and thus t-membership is in near-linear time, or*

- *t-membership takes time $(nm)^{1-o(1)}$, assuming SETH.*

Proof. All algorithmic results and minimal hardness results were known before, we just show that the known results are sufficient to completely characterize all types of any (constant) depth. The proof is similar to the one of Theorem 5. Again we consider a tree containing all types not containing two consecutive equal operations (i.e., not simplifiable by Lemma 8.1), see Figure 4.

When one of the simplification rules of Lemma 8 applies to t, then t-pattern matching is equivalent to t'-pattern matching for some different, unsimplifiable t' in the tree. Since the same simplification rule also applies to all descendants of t in the tree, we can ignore the whole subtree of t.

For the subtrees starting with $\star$ or $|\star$ we know that t-pattern matching is in linear time for each type t by Lemma 7.

For any other type t with a near-linear time algorithm, in the figure we annotate the corresponding node by the fastest known asymptotic running time.

Finally, when there is a SETH-based lower bound for t-pattern matching then the same lower bound also holds for all descendants of t in the tree (by Lemma 9.1), so we have characterized the whole subtree of t. When hardness directly follows from one of the reductions in [4] then we simply mark t as "hard". When we first have to use one of the reductions in Lemma 9 to reduce from a hard type t' of [4], then in the figure we annotate t by "hard t'" as well as the corresponding reduction rule.

It is easy to check that our Figure 4 indeed enumerates all cases and thus contains all maximal algorithmic results and minimal hardness results. The claimed dichotomy of Theorem 11 now follows by inspecting the figure. $\square$

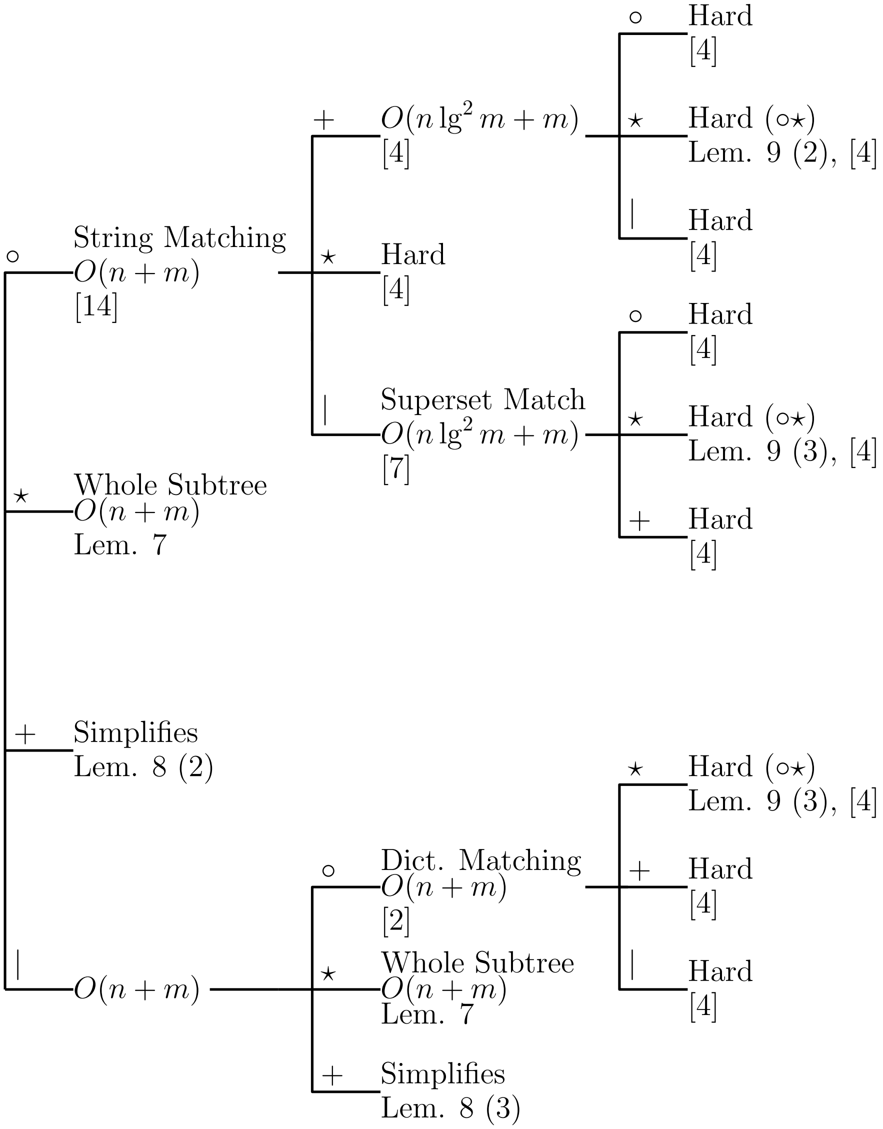

Figure 4: Tree diagram for t-pattern matching. For details see the proof of Theorem 11.