Text Indexing for Simple Regular Expressions

Hideo Bannai
M&D Data Science Center, Institute of Integrated Research, Institute of Science Tokyo, Japan

Philip Bille
Technical University of Denmark, Lyngby, Denmark

Inge Li Gørtz
Technical University of Denmark, Lyngby, Denmark

Gad M. Landau
Department of Computer Science, University of Haifa, Haifa, Israel

Gonzalo Navarro
Department of Computer Science, University of Chile, Chile
Center for Biotechnology and Bioengineering (CeBiB), Chile

Nicola Prezza
DAIS, Ca' Foscari University of Venice, Venice, Italy

Teresa Anna Steiner
University of Southern Denmark, Odense, Denmark

Simon Rumle Tarnow
Technical University of Denmark, Lyngby, Denmark

Abstract

We study the problem of indexing a text $T[1..n] \in \Sigma^n$ so that, later, given a query regular expression pattern R of size $m = |R|$, we can report all the *occ* substrings $T[i..j]$ of T matching R. The problem is known to be hard for arbitrary patterns R, so in this paper we consider the following two types of patterns. (1) *Character-class Kleene-star* patterns of the form $P_1 D^* P_2$, where P_1 and P_2 are strings and $D = \{c_1, \ldots, c_k\} \subset \Sigma$ is a *character-class* that is shorthand for the regular expression $(c_1|c_2|\cdots|c_k)$. (2) *String Kleene-star* patterns of the form $P_1 P^* P_2$ where P, P_1 and P_2 are strings. In case (1), we describe an index of $O(n \log^{1+\epsilon} n)$ space (for any constant $\epsilon > 0$) solving queries in time $O(m + \log n / \log\log n + occ)$ on constant-sized alphabets. We also describe a more general solution working on any alphabet size. This result is conditioned on the existence of an *anchor*: a character of $P_1 P_2$ that does not belong to D. We justify this assumption by proving that if an anchor is not present, no efficient indexing solution can exist unless the Set Disjointness Conjecture fails. In case (2), we describe an index of size $O(n)$ answering queries in time $O(m + (occ + 1) \log^{\epsilon} n)$ on any alphabet size.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Text indexing, regular expressions, data structures

Digital Object Identifier 10.4230/LIPIcs.CPM.2025.1

Funding *Hideo Bannai*: JSPS KAKENHI Grant Number JP24K02899
Philip Bille: Danish Research Council grant DFF-8021-002498
Inge Li Gørtz: Danish Research Council grant DFF-8021-002498
Gonzalo Navarro: Basal Funds FB0001 and AFB240001, ANID, Chile.
Nicola Prezza: Funded by the European Union (ERC, REGINDEX, 101039208).
Teresa Anna Steiner: Supported by a research grant (VIL51463) from VILLUM FONDEN.

Acknowledgements This work was initiated at Dagstuhl Seminar 24472 "Regular Expressions: Matching and Indexing"

1 Introduction

A regular expression R specifies a set of strings formed by characters from an alphabet Σ combined with concatenation $(\cdot)$, union $(|)$, and Kleene star $(^*)$ operators. For instance, $(a|(b \cdot a))^*$ describes the set of strings of as and bs such that every b is followed by an a. The *text indexing for regular expressions problem* is to preprocess a text T to support efficient *regular expression matching queries* on T, that is, given a regular expression R, report all occurrences of R in T. Here, an occurrence is a substring $T[i..j]$ that matches any of the strings belonging to the regular language of R. We also consider *existential regular expression matching queries*, that is, determining whether or not there is an occurrence of R in T. The goal is to obtain a compact data structure while supporting efficient queries.

Regular expressions are a fundamental concept in formal language theory introduced by Kleene in the 1950s [23], and regular expression pattern matching queries are a basic tool in computer science for searching and processing text. Standard tools such as `grep` and `sed` provide direct support for regular expression matching in files, and the scripting language `perl` [43] is a complete programming language designed to support regular expression matching queries easily. Regular expression matching appears in many large-scale data processing applications, such as internet traffic analysis [21, 27, 44], data mining [17], databases [31, 32], computational biology [35], and human-computer interaction [22]. Most of the solutions are based on the efficient algorithms for the classic *regular expression matching problem*, where we are given both the text T and the regular expression R as input, and the goal is to report the occurrences of R in T. However, in many scenarios, the text T is available before we are given the regular expressions, and we may want to ask multiple regular expression matching queries on T. In this case, we ideally want to take advantage of preprocessing to speed up the queries, and thus, the indexing version of the problem applies.

While the regular expression matching problem is a well-studied classic problem [2, 3, 5, 6, 8, 12, 13, 14, 33, 41, 42], surprisingly few results are known for the text indexing for regular expressions problem. Let n and m be the length of T and R, respectively. Gibney and Thankachan [18] recently showed that text indexing for regular expression is hard to solve efficiently under popular complexity conjectures. More precisely, they showed that conditioned on the online matrix-vector multiplication conjecture, even with arbitrary polynomial preprocessing time, we cannot answer existential queries in $O(n^{1-\varepsilon})$ for any $\varepsilon > 0$. They also show that if conditioned on a slightly stronger assumption, we cannot even answer existential queries in $O(n^{3/2-\varepsilon})$ time, for any $\varepsilon > 0$. Gibney and Thankachan also studied upper bound time-space trade-offs with exponential preprocessing. Specifically, given a parameter t, $1 \leq t \leq n$, fixed at preprocessing, we can solve the problem using $2^{O(tn)}$ space and preprocessing time and $O(nm/t)$ query time.

On the other hand, a few text indexing solutions have been studied for highly restricted kinds of regular expressions or regular expression-like patterns. These include text indexing for string patterns (simple strings corresponding to regular expressions that only use concatenations) and string patterns with *wildcards* and *gaps* (strings that include special characters or sequences of special characters that match any other character) [7, 9, 11, 15, 20, 26, 28, 29, 30, 39].

Thus, we should not hope to efficiently solve text indexing for general regular expressions, and efficient solutions are only known for highly restricted regular expressions. Hence, a natural question is if there are simple regular expressions for which efficient solutions are possible and that form a large subset of those used in practice. This paper considers the following two such kinds of regular expressions and provides either efficient solutions or conditional lower bounds to them:

- **Character-class Kleene-star patterns.** These are patterns of the form $P_1D^*P_2$ where P_1 and P_2 are strings and $D = \{c_1, \ldots, c_k\} \subset \Sigma$ is a *character-class* that is shorthand for the regular expression $(c_1|c_2|\cdots|c_k)$.
- **String Kleene-star patterns.** These are patterns of the form $P_1P^*P_2$ where P, P_1 and P_2 are strings.

In other words, we provide solutions (or lower bounds) for all regular patterns containing only *concatenations and at most one occurrence of a Kleene star* (either of a string or a character-class). Using the notation introduced by the seminal paper of Backurs and Indyk [3] on the hardness of (non-indexed) regular expression matching, *character-class Kleene-star* patterns belong to the "$\cdot|$" type: a concatenation of (possibly degenerate, i.e. $|D| = 1$) unions. To see this, observe that the characters of P_1 and P_2 can be interpreted as degenerate unions of one character. *String Kleene-star* patterns, on the other hand, belong to the "$\cdot * \cdot$" type: a concatenation of Kleene stars of concatenations. Again (as discussed in [3]), since any level of the regular expression tree is allowed to contain leaves (i.e. an individual character), patterns of the form $P_1P^*P_2$ belong to this type by interpreting the characters of P_1 and P_2 as leaves in the regular expression tree. Our main results are new text indices that use near-linear space while supporting both kind of queries in time near-linear in the length of the pattern (under certain unavoidable assumptions discussed in detail below: if the assumptions fail, we show that the problem becomes again hard). Below, we introduce our results and discuss them in the context of the results obtained in [3].

1.1 Setup and Results

We first consider text indexing for character-class Kleene-star patterns $R = P_1D^*P_2$, where D is a characters class. We say that the pattern is *anchored* if either P_1 or P_2 has a character that is *not* in D, and we call such a character an *anchor*. If the pattern is anchored, we show the following result.

▶ **Theorem 1.** *Let T be a text of length n over an alphabet Σ. Given a parameter $k_{\max} < |\Sigma|$ and a constant $\epsilon > 0$ fixed at preprocessing time, we can build a data structure that uses $O(k_{\max}\, n \log^{1+\epsilon} n)$ space and supports anchored character-class Kleene-star queries $P_1D^*P_2$, where D is a characters class with $|D| = k \leq k_{\max}$ characters in $O(m+2^k \log n/\log\log n+\text{occ})$ time with high probability. Here, $m = |P_1| + |D| + |P_2|$ and occ is the number of occurrences of the pattern in T.*

In particular, our solution supports queries in almost optimal $O(m + \log n/\log\log n)$ time for constant-sized alphabets. We also extend Theorem 1 result to handle slightly more general *character-class interval patterns* of the form $P_1D^{\geq l}P_2$, $P_1D^{\leq r}P_2$, and $P_1D^{[l,r]}P_2$, meaning that there are at least, at most, and between l and r copies of characters from D.

Intuitively, our strategy is to identify all the right-maximal substrings $T[i..j]$ of T, for every possible starting position i, that contain only symbols in D for every possible set D. Such a substring will form the "D^*" part of the occurrences. For each such $T[i..j]$, we then insert in a range reporting data structure a three-dimensional point with (lexicographically-sorted) coordinates $(T[1..i-1]^{rev}, T[1..j]^{rev}, T[j+1..n])$. The data structure is labeled by set D. We finally observe that the pattern R can be used to query the right range data structure and report all matches of R in T.

Conversely, we show the following conditional lower bound if the pattern is not anchored.

▶ **Theorem 2.** *Let T be a text of length n over an alphabet Σ with $|\Sigma| \geq 4$ and let $\delta \in [0, 1/2]$. Assuming the strong Set Disjointness Conjecture, any data structure that supports existential* *(non-anchored) character-class Kleene-star pattern matching queries $P_1D^*P_2$, where D is a character class with at least 3 characters, in $O(n^\delta)$ time, requires $\tilde{\Omega}(n^{2-2\delta-o(1)})$ space.*

With $\delta = 1/2$, Theorem 2 implies that any near linear space solution must have query time $\tilde{\Omega}(\sqrt{n})$. On the other hand, with $\delta = 0$, Theorem 2 implies that any solution using time independent from n must use $\tilde{\Omega}(n^{2-o(1)})$ space.

To get Theorem 2, we reduce from the Set Disjointness Problem: I.e., preprocessing some sets so we can quickly answer, for any pair of sets, if they are disjoint or not. [10] showed that wlog, we can assume every element appears in the same number of sets. The idea is then to define a string gadget representing any set, and a block for each element in the universe containing the string gadget for every set it is included in. The blocks are separated by a character not in the block. This way, the intersection of two sets is non-empty if and only if their gadgets appear somewhere in the string only separated by characters which appear in a block.

As noted above, *character-class Kleene-star* patterns belong to the "$\cdot|$" type. Backurs and Indyk [3] show that offline pattern matching on this type of pattern can be performed in time $O(n \log m)$. This result is, however, incomparable with ours: their solution is offline and the lower bound of Theorem 2 only applies to the regimes where the query time is $O(\sqrt{n})$ (while Backurs and Indyk's solution could equivalently be interpreted as an index solving queries in $O(n \log m)$ time).

We then consider text indexing for String Kleene-star patterns $R = P_1P^*P_2$. We show the following result.

▶ **Theorem 3.** *Let T be a text of length n over an alphabet Σ. Given a constant $\epsilon > 0$ fixed at preprocessing time, we can build a data structure that uses $O(n)$ space and supports String Kleene-star patterns $P_1P^*P_2$ in time $O(m + (\text{occ} + 1)\log^\epsilon n)$, where $m = |P_1| + |P| + |P_2|$ and occ is the number of occurrences of the pattern in T.*

As discussed above, *String Kleene-star* patterns belong to the "$\cdot * \cdot$" type. For this type of patterns, Backurs and Indyk [3] proved a conditional lower bound of $\Omega((mn)^{1-\epsilon})$ (for any constant $\epsilon > 0$) in the offline setting for both pattern matching and membership queries. Our result, instead, implies an offline solution running in $O(m + \log^\epsilon n)$ time (by stopping after locating the first pattern occurrence) after the indexing phase. This does not contradict Backurs and Indyk's lower bound, since our patterns $P_1P^*P_2$ are a very specific case of the (broader) type "$\cdot * \cdot$". Equivalently, this indicates that including more than one Kleene star makes the problem hard again and thus justifies an index for the simpler case $P_1P^*P_2$.

The main idea behind the strategy for Theorem 3 is to preprocess all runs in the string, so we can quickly find patterns ending just before or starting just after a run. However, there are some difficulties to overcome: firstly, P may be periodic - e. g. if $P = ww$, we do not want to report occurrences of $P_1w^3P_2$; secondly, a run may end with a partial occurrence of the period; and lastly, P may share a suffix with P_1 or a prefix with P_2, in which case their occurrences should overlap with the run. We show how to deal with these difficulties in Section 4.

2 Preliminaries

A string T of length $|T| = n$ is a sequence $T[1] \cdots T[n]$ of n characters drawn from an ordered alphabet Σ of size $|\Sigma|$. The string $T[i] \cdots T[j]$, denoted $T[i..j]$, is called a *substring* of T; $T[1..j]$ and $T[i..n]$ are called the j^{th} *prefix* and i^{th} *suffix* of T, respectively. We use ϵ to denote the empty string (i.e., the string of length 0). The *reverse string* of a string T of length n, denoted by T^{rev}, is given by $T^{rev} = T[n] \dots T[1]$. Let P and T be strings over an alphabet Σ. We say that the range $[i..j]$ is an *occurrence* of P in T iff $T[i..j] = P$.

Lexicographic order and Lyndon words. The order of the alphabet defines a *lexicographic order* on the set of strings as follows: For two strings $T_1 \neq T_2$, let i be the length of the longest common prefix of T_1 and T_2. We have $T_1 < T_2$ if and only if either i) $|T_1| = i$ or ii) both T_1 and T_2 have a length at least $i+1$ and $T_1[i+1] < T_2[i+1]$. A string T is a *Lyndon word* if it is lexicographically smaller than any of its proper cyclic shifts, i.e., $T < T[i..n]T[1..i-1]$, for all $1 < i \leq n$.

Concatenation of strings. The concatenation of two strings A and B is defined as $AB = A[1] \cdots A[|A|]B[1] \cdots B[|B|]$. The concatenation of k copies of a string A is denoted by A^k, where $k \in \mathbb{N}$; i.e. $A^0 = \epsilon$ and $A^k = AA^{k-1}$. A string B is called *primitive* if there is no string A and $k > 1$ such that $B = A^k$.

Sets of strings. We denote by $A^{\geq l} = \bigcup_{k \geq l}\{A^k\}$, $A^{\leq r} = \bigcup_{k \leq r}\{A^k\}$, $A^{[l,r]} = \bigcup_{l \leq k \leq r}\{A^k\}$, and $A^* = A^{\geq 0}$. The concatenation of a string A with a set of strings S is defined as $AS = \{AB : B \in S\}$. Similarly, the concatenation of two sets of strings S_1 and S_2 is defined as $S_1S_2 = \{AB : A \in S_1, B \in S_2\}$. We define $S^{\geq l}$, $S^{\leq r}$, $S^{[l,r]}$, and $S^* = S^{\geq 0}$ for sets analogously. We say that the range $[i..j]$ is an *occurrence* of a set of strings S if there is a $P \in S$ such that $[i..j]$ is an occurrence of P in T.

Period of a string. An integer p is a *period* of a string T of length n if and only if $T[i] = T[i+p]$ for all $1 \leq i \leq n-p$. A string T is called *periodic* if it has a period $p \leq n/2$. The smallest period of T will be called *the period* of T.

Tries and suffix trees. A *trie* for a collection of strings $\mathcal{C} = \{T_1, \dots, T_n\}$, is a rooted labeled tree $\mathcal{T}$ such that: (1) The label on each edge is a character in some T_i ($i \in [1,n]$). (2) Each string in $\mathcal{C}$ is represented by a path in $\mathcal{T}$ going from the root down to some node (obtained by concatenating the labels on the edges of the path). (3) Each root-to-leaf path represents a string from $\mathcal{C}$. (4) Common prefixes of two strings share the same path maximally. A *compact trie* is obtained from $\mathcal{T}$ by dissolving all nodes except the root, the branching nodes, and the leaves, and concatenating the labels on the edges incident to dissolved nodes to obtain *string* labels for the remaining edges.

Let T be a string over an alphabet Σ. The *suffix tree* of a string T is the compacted trie of the set of all suffixes of T. Throughout this paper, we assume that nodes in a compact trie or the suffix tree use deterministic dictionaries to store their children.

3 Character-class Kleene-star Patterns

In this section we give our data structure for answering anchored character-class Kleene-star pattern queries. Without loss of generality, we can assume that the anchor belongs to P_2 (the other case is captured by building our structures on the reversed text and querying the reversed pattern).

Recall that we assume $k = |D| \leq k_{\max}$ for some parameter $k_{\max} < |\Sigma|$ fixed at construction time. We first describe a solution for the case $k_{\max} \in O(\log n)$, and then in Section 3.3 show how to handle the case where $k_{\max} \geq \log n$.

Our general strategy is to identify all the right-maximal substrings $T[i..j]$ of T, for every possible starting position i, that contain only symbols in D (we later generalize the solution to consider all the possible subsets of D). Such a substring forms the "D^*" part of the

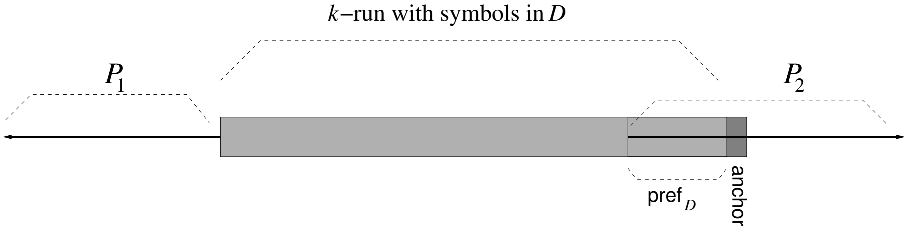

Figure 1 Illustration of the general strategy to capture patterns of the form $P_1 D^* P_2$. A k-run is a right-maximal substring $T[i..j]$ containing exactly k distinct symbols.

occurrences. For this sake, D^* must be preceded by P_1 and followed by P_2. However, if P_2 starts with some symbols in D, those symbols will belong to the right-maximal substring $T[i..j]$. We therefore separate $P_2 = \text{pref}_D \cdot \text{suff}_D$, where pref_D is the longest prefix of P_2 that contains only symbols from D, and suff_D starts with the anchor. The new condition is then that the substring $T[i..j]$ ends with pref_D and is followed by suff_D. See Figure 1.

We need the following definitions.

▶ **Definition 4.** *The D-prefix of P_2, denoted $\text{pref}_D(P_2)$ is the longest prefix of P_2 that is formed only by symbols in D. We define $\text{suff}_D(P_2)$ so that $P_2 = \text{pref}_D(P_2) \cdot \text{suff}_D(P_2)$*

▶ **Definition 5.** *The k-run of T that starts at position i is the maximal range $[i..j]$ such that $T[i..j]$ contains exactly k distinct symbols. If the suffix $T[i..n]$ has less than k different symbols, then there is no k-run starting at i. We call $D_{i,k}$ the set of k symbols that occur in the k-run that starts at position i.*

Note that T contains at most n k-runs, each starting at a distinct position $i \in [1..n]$.

We first show how to find occurrences matching all k symbols of D in the D^* part of the pattern $P_1 D^* P_2$. Then, we complete this solution by allowing matches with any subset of D.

3.1 Matching all k Characters of D

We show how to build a data structure for the case where $k = |D|$ is known at construction time, and we only find the occurrences that match *exactly* all k distinct letters in the D^* part of the occurrence. Recall that we also assume that P_2 contains an anchor.

Data structure. Let $\mathcal{D}_k$ be the set of subsets $D \subseteq \Sigma$ of size k that occur as a k-run in T. Our data structure consists of the following:

- The suffix tree $\mathcal{T}$ of T and the suffix tree $\mathcal{T}^{rev}$ of the reversed text, T^{rev}.
- A data structure S_D for each set $D \in \mathcal{D}_k$ indexing all the text positions $P_D = \{i \mid D_{i,k} = D\}$. The structure consists of an orthogonal range reporting data structure for a four-dimensional grid in $[1..n]^4$ with $|P_D|$ points, one per k-run $[i..j]$ with $i \in P_D$. For each such k-run $[i..j]$ we store a point with coordinates $(x_i, y_i, z_i, j - i + 1)$, where
  - x_i is the lexicographic rank of $T[1..i-1]^{rev}$ among all the reversed prefixes of T.
  - y_i is the lexicographic rank of $T[1..j]^{rev}$ among all the reversed prefixes of T.
  - z_i is the lexicographic rank of $T[j+1..n]$ among all the suffixes of T.

  Each point stores the limits $[i..j]$ of its k-run (so as to report occurrence positions).
- A trie τ_k storing all the strings s_D of length k formed by sorting in increasing order the k characters of D, for every $D \in \mathcal{D}_k$.

Note that the fourth coordinate $j-i+1$ of point $(x_i, y_i, z_i, j-i+1)$ could be avoided (i.e. using a 3D range reporting data structure) by defining y_i to be the lexicographic rank of $T[1..j]^{rev}\$$ (where \$ is a special terminator character) in the set formed by all the reversed prefixes of T and strings of the form $T[1..j]^{rev}\$$, for all k-runs $T[i..j]$. While this solution would work in the same asymptotic space and query time (because we will only need one-sided queries on the fourth coordinate), we will need the fourth dimension in Subsection 3.4.

Basic search. At query time, we first compute $\text{pref}_D(P_2)$. For any occurrence of the query pattern, $\text{pref}_D(P_2)$ will necessarily be the suffix of a k-run. This is why we need P_2 to contain an anchor; P_1 is not restricted because we index every possible initial position i.

We then sort the symbols of D and use the trie τ_k to find the data structure S_D.

We now find the lexicographic range $[x_1, x_2] \times [y_1, y_2] \times [z_1, z_2] \times [|\text{pref}_D(P_2)|, +\infty]$ using the suffix tree $\mathcal{T}$ of T and the suffix tree $\mathcal{T}^{rev}$ of the reversed text, T^{rev}. The range $[x_1, x_2]$ then corresponds to the leaf range of the locus of P_1^{rev} in $\mathcal{T}^{rev}$, the range $[y_1, y_2]$ to the leaf range of the locus of $\text{pref}_D(P_2)^{rev}$ in $\mathcal{T}^{rev}$, and the range $[z_1, z_2]$ to the leaf range of the locus of $\text{suff}_D(P_2)$ in $\mathcal{T}$.

Once the four-dimensional range is identified, we extract all the points from S_D in the range using the range reporting data structure.

Time and space The suffix trees use space $O(n)$. The total number of points in the range reporting data structures is $O(n)$ as there are at most n k-runs. Because we will perform one-sided searches on the fourth coordinate, the grid of S_D can be represented in $O(|P_D| \log^{1+\epsilon} n)$ space, for any constant $\epsilon > 0$, so that range searches on it take time $O(\text{occ} + \log n / \log\log n)$ to report the occ points in the range [36, Thm. 7]. Thus, the total space for the range reporting data structures is $O(n \log^{1+\epsilon} n)$. The space of the trie τ_k is $k|\mathcal{D}_k| \in O(kn)$.

The string $\text{pref}_D(P_2)$ can easily be computed in $O(k + |P_2|)$ time with high probability using a dictionary data structure [16]. Sorting D can be done in $O(k \log\log k)$ time [1]. By implementing the pointers of node children in τ_k and in the suffix trees $\mathcal{T}$ and $\mathcal{T}^{rev}$ using fast dictionaries [16], the search in τ_k takes $O(k)$ time with high probability and the three searches in $\mathcal{T}$ and $\mathcal{T}^{rev}$ take total time $O(|P_1| + |P_2|)$ with high probability. The range reporting query takes time $O(\log n / \log\log n + \text{occ})$. In total, a query takes $O(m + k \log\log k + \log n / \log\log n + \text{occ})$ time with high probability.

3.2 Matching any Subset of D

We now show how to find all occurrences of $P_1 D^* P_2$, that is, also the ones containing only a subset of the characters of D in the D^* part of the occurrence.

Our previous search will not capture the $(k-i)$-runs, for $1 \le i < k$, containing only characters appearing in *subsets* of D, as we only find P_1 and $\text{suff}_D(P_2)$ surrounding the k-runs containing all characters from D. To solve this we will build an orthogonal range reporting data structures for all $D \in \bigcup_{1 \le k \le k_{\max}} \mathcal{D}_k$. To capture all the *occ* occurrences of $P_1 D^* P_2$, we search the corresponding grids of all the $2^k - 1$ nonempty subsets of D, which leads to the cost $O(2^k \log n / \log\log n + occ)$. We wish to avoid, however, the cost of searching for P_1, $\text{pref}_{D'}(P_2)$, and $\text{suff}_{D'}(P_2)$ in the suffix trees for every subset D' of D. In the following we show how to do this.

Data Structure Let $\mathcal{D} = \bigcup_{1 \le k \le k_{\max}} \mathcal{D}_k$. Our data structure consists of the following.

- The suffix tree $\mathcal{T}$ of T and the suffix tree $\mathcal{T}^{rev}$ of the reversed text, T^{rev}.
- The data structure S_D from Section 3.1 for each set $D \in \mathcal{D}$.

- A trie τ storing all the strings of length 1 to $k_{\max}$, in increasing order of symbols, that correspond to some $D \in \mathcal{D}$.

The suffix trees uses linear space. The space for each of the k range reporting data structures is $O(n \log^{1+\epsilon} n)$. Added over all $k \in [1..k_{\max}]$, the total space becomes $O(k_{\max}\, n \log^{1+\epsilon} n)$. The space for the trie τ is $O(nk_{\max}^2)$ since there are at most $k_{\max}n$ strings each of length at most $k_{\max}$. Since we assume $k_{\max} \in O(\log n)$, the total space is $O(k_{\max} n \log^{1+\epsilon} n)$.

Search To perform the search we search in τ for all subsets D' of D: In sorted order, we traverse τ to find all the subsets of D: for each next symbol $c \in D$, we try both skipping it or descending by it in τ. In this way we visit all the $2^k - 1$ nodes of τ corresponding to subsets of D. Each time we are in a node in the trie τ corresponding to some set $D' \subseteq D$ which has an associated range reporting data structure $S_{D'}$, we perform a range reporting query $(x_1, x_2, y_1, y_2, z_1, z_2, |\text{pref}_{D'}(P_2)|, \infty)$.

Note that the range $[x_1, x_2]$ is the same for all queries, so we only compute this once. This is done by a search for P_1^{rev} in $\mathcal{T}^{rev}$. The intervals $[y_1, y_2]$ and $[z_1, z_2]$, on the other hand, change during the search, as the split of P_2 into $\text{pref}_{D'}(P_2)$ and $\text{suff}_{D'}(P_2)$ depends on the subset D'. To compute these intervals we first preprocess P_2 as follows. Compute the ranges $[y_1, y_2]$ for all reversed prefixes of P_2 using the suffix tree $\mathcal{T}^{rev}$: Start by looking up the locus for P_2^{rev} and then find the remaining ones by following suffix links. Similarly, we compute the ranges $[z_1, z_2]$ for the suffixes of P_2 following suffix links in $\mathcal{T}$. If we know the length ℓ of $\text{pref}_{D'}(P_2)$ we can then easily look up the intervals the corresponding intervals.

Maintaining ℓ. We now explain how to maintain the length ℓ of $\text{pref}_{D'}(P_2)$ for $D' \subset D$ in constant time for every trie node we meet during the traversal of τ. The difficulty with maintaining $|\text{pref}_{D'}(P_2)|$ while D' changes is that we when traversing the trie we add the characters to D' in lexicographical order and not in the order they occur in P_2 (see Figure 2).

First we compute for each character $c \in D$ the position p_c of the first occurrence of c in $\text{pref}_D(P_2)$. If c does not occur in $\text{pref}_D(P_2)$, we set $p_c = \infty$. For each $c \in D$, we furthermore compute the *position rank* r_c of c, i.e., the rank of p_c in the sorted set $\{p_c : c \in D\}$. We build:

- a dictionary R saving the position rank r_c of each element $c \in D$.
- an array B containing the characters in D in position rank order such that $B[r_c] = c$ for all $c \in D$ (define $B[0] = -1$).
- an array P containing the position of the first occurrence of the characters in D in rank order, i.e. $P[i]$ is the first position of character $B[i]$.

The main idea is to maintain the intervals of characters in position rank order that we have in the sets D'. Before we start the traversal of τ we also construct an array $A[0..|D|+1]$ and initialize all positions in A to 0. We will maintain the invariant that the first, respectively last, position of an interval of nonzero entries in A contains the position of the end, respectively start, of the interval. Initialize $\ell = 0$ and initialize an empty stack S. We now maintain $\ell = |\text{pref}_{D'}(P_2)|$ as follows:

When we go down during the traversal adding a character c to the set we first lookup p_c and r_c. If $p_c = \infty$ there are no changes. Otherwise, we set Set $A[r_c] = r_c$ and compute the leftmost position lp of the nonzero interval containing c: If $A[r_c - 1] = 0$ then set $lp = r_c$. Else $lp = A[r_c - 1]$. To compute the rightmost position rp of the nonzero interval containing c: If $A[r_c + 1] = 0$ then set $rp = r_c$. Else $rp = A[r_c + 1]$. We then push $(lp, A[lp], rp, A[rp], \ell)$ onto the stack to be able to quickly undo the operations later. Then we update A by setting

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
$P_2 =$	b	b	a	a	b	e	a	b	e	e	c	e	a	d	a	h	...	

$D = \{a, b, c, d, e\}$		D in position rank order: $[b, a, e, c, d]$
$D_1 = \{a\}$	$\ell = 0$	$A = [0, 0, 2, 0, 0, 0, 0]$
$D_2 = \{a, b\}$	$\ell = 5$	$A = [0, 2, 1, 0, 0, 0, 0]$
$D_3 = \{a, b, c\}$	$\ell = 5$	$A = [0, 2, 1, 0, 4, 0, 0]$
$D_4 = \{a, b, c, d\}$	$\ell = 5$	$A = [0, 2, 1, 0, 5, 4, 0]$
$D_5 = \{a, b, c, d, e\}$	$\ell = 15$	$A = [0, 5, 1, 3, 5, 1, 0]$
$D_6 = \{b\}$	$\ell = 2$	$A = [0, 1, 0, 0, 0, 0, 0]$
$\vdots$	$\vdots$	$\vdots$

Figure 2 Computing $\ell = |\text{pref}_{D'}(P_2)|$ as D' changes during the traversal of the trie. The array A maintains the intervals of characters in position rank order (the order in which the characters appear in P_2) that are in D'.

$A[lp] = rp$ and $A[rp] = lp$. Finally, we update ℓ: If $A[1] \geq r_c$ set $\ell = P[A[1] + 1] - 1$. Otherwise, ℓ does not change.

When going up in the traversal removing character c we first lookup p_c. If $p_c = \infty$ there are no changes. Otherwise, we pop (lp, lv, rp, rv, ℓ') from the stack and set $A[lp] = lv$, $A[rp] = rv$, $A[r_c] = 0$, and $\ell = \ell'$.

Time It takes $O(|P_1|)$ time to search for P_1^{rev} in $\mathcal{T}^{rev}$. Computing $[y_1, y_2]$ and $[z_1, z_2]$ for all splits of P_2 takes time $O(|P_2|)$. Sorting D can be done in time $O(k \log \log k)$ [1]. Computing p_c for all characters in D, sorting them, computing the ranks r_c, and constructing the arrays B and P and the dictionary R takes linear time in the pattern length with high probability. The size of the subtrie we visit in the search is $O(2^k)$ and in each step we use constant time to compute the length of ℓ. The total time for the range queries is $O(2^k \log n / \log \log n + \text{occ})$. Thus, in total we use $O(m + 2^k \log n / \log \log n + \text{occ})$ time with high probability.

3.3 Solution for $k_{\max} \geq \log n$

In the discussion above, we assumed that $k_{\max} \in O(\log n)$. If $k_{\max} \geq \log n$, we build the data structure described above by replacing $k_{\max}$ with $k'_{\max} = \log n$. The space of the data structure is still $O(k'_{\max}\, n \log^{1+\epsilon} n) \subseteq O(k_{\max}\, n \log^{1+\epsilon} n)$. At query time, if $|D| = k \leq \log n$ we use the data structure to answer queries in $O(m + 2^k \log n / \log \log n + \text{occ})$ time.

If, on the other hand, $|D| = k > \log n$ then $n \in O(2^k \log n / \log \log n)$. We first find all occurrences of P_1 and P_2 using the suffix tree $\mathcal{T}$. Let L_1 be the end positions of the occurrences of P_1 and let P_2 be the start positions of the occurrences of P_2. We sort the lists L_1 and L_2. This can all be done in $O(m + n)$ time and linear space using radix sort. We also mark with a 1 in a bitvector B of length n all text positions i such that $T[i] \in D$. This can be done in $O(n)$ time with high probability, with a simple scan of T and a dictionary over D [16]. We build a data structure over the bitvector supporting rank queries in constant time [40]. We can now find all occurrences of the pattern by considering the occurrences in sorted order in a merge like fashion. Recall, that P_2 has an anchor. We consider the first occurrence p_1 in the list L_1 and find the first occurrence p_2 in L_2 that comes after L_1, i.e. $p_2 > p_1$. If all characters between p_1 and p_2 are from D (constant time with two rank operations over bitvector B) we output the occurrence. We delete p_1 from the list and continue the same way. In total, we find all occurrences in $O(n + \text{occ}) \in O(2^k \log n / \log \log n + \text{occ})$ time with high probability. In summary, this proves Theorem 1.

3.4 Character-Class Interval Patterns

We extend our solution to handle patterns of the form $P_1D^{\geq l}P_2$, $P_1D^{\leq r}P_2$, and $P_1D^{[l,r]}P_2$, meaning that there are at least, at most, and between l and r copies of characters from D. We collectively call these *character-class interval patterns.*

By using one-sided restrictions on the fourth dimension, we can easily handle queries of the form $P_1D^{\geq l}P_2$ in our solution from the previous section. Handling queries of the form $P_1D^{\leq r}P_2$ or $P_1D^{[l,r]}P_2$ requires a two-sided restriction on the fourth dimension. This raises the space of the grid to $O(|P_D|\log^{2+\epsilon} n)$, while retaining its query time [36, Thm. 7] [37]. With these observations we obtain the following results.

▶ **Theorem 6.** *Let T be a text of length n over an alphabet Σ. Given a parameter $k_{\max} < |\Sigma|$ and a constant $\epsilon > 0$ fixed at preprocessing time, we can build a data structure that uses $O(k_{\max}\, n \log^{1+\epsilon} n)$ space and supports anchored character-class interval queries of the form $P_1D^{\geq l}P_2$, where D is a character class with $k \leq k_{\max}$ characters in time $O(m + 2^k \log n/\log\log n + \mathrm{occ})$ and $m = |P_1| + |D| + |P_2|$ and occ is the number of occurrences of the pattern in T.*

▶ **Theorem 7.** *Let T be a text of length n over an alphabet Σ. Given a parameter $k_{\max} < |\Sigma|$ and a constant $\epsilon > 0$ fixed at preprocessing time, we can build a data structure that uses $O(k_{\max}\, n \log^{2+\epsilon} n)$ space and supports anchored character-class interval queries of the form $P_1D^{\leq r}P_2$ or $P_1D^{[l,r]}P_2$, where D is a characters class with $k \leq k_{\max}$ characters in time $O(m+2^k \log n/\log\log n+\mathrm{occ})$ and $m = |P_1|+|D|+|P_2|$ and occ is the number of occurrences of the pattern in T.*

An alternative solution, when longer matches are more interesting than shorter ones, is to store the points (x_i, y_i, z_i) in a three-dimensional grid, and use $j - i + 1$ as the point weights. Three-dimensional grids on weighted points can use $O(|P_D|\log^{2+\epsilon} n)$ space and report points from larger to smaller weight (i.e., $j - i + 1$) in time $O(p + \log n)$ [34, Lem. A.5]. We can use this to report the occurrences from longer to shorter k-runs, thereby stopping when the length drops below $|\mathrm{pref}_D(P_2)|$. We insert the first answer of each of the $2^k - 1$ grids into a priority queue, where the priority will be the length $j - i + 1$ of the matched k'-run $[i..j]$ minus $|\mathrm{pref}_{D'}(P_2)|$, then extract the longest answer and replace it by the next point from the same grid, repeating until returning all the desired answers. The time per returned element now includes a factor $O(\log\log n)$ if we implement the priority queue with a dynamic predecessor search data structure, plus $O(2^k \log\log n)$ for the initial insertions. We can also return t longest answers in this case, within a total time of $O(m + 2^k \log n + t \log\log n)$.

4 String Kleene-star Patterns

In this section we give our data structure for supporting string Kleene-star pattern queries.

As an intermediate step, we first create a structure that, given strings S_1 and S_2, a primitive string w, and numbers $a, b, c, d \in \mathbb{N}$ with $b < a$ and $d < |w|$, where S_1 and w do not share a suffix and S_2 and $w[d + 1..]$ do not share a prefix, finds all occurrences in T of patterns of the form $S_1w^{aq+b}w[1..d]S_2$, where $q \geq c$ and $q \in \mathbb{N}$. Later we will show that this is sufficient to find occurrences of $P_1P^*P_2$. For now, we assume that S_1 and S_2 are not the empty string; we will handle these cases later. We will also assume that w is not the empty string - in our transformation from $P_1P^*P_2$ to $S_1w^{aq+b}w[1..d]S_2$, w will be empty if and only if P is empty. In this case, the problem reduces to matching $P_1P_2 = S_1S_2$ in the suffix tree.

To define our data structures, we need the notion of a run (or maximal repetition) in T.

▶ **Definition 8.** *A* run *of T is a periodic substring $T[i..j]$, such that the period cannot be extended to the left or the right. That is, if the smallest period of $T[i..j]$ is p, then $T[i-1] \neq T[i+p-1]$ and $T[j+1] \neq T[j-p+1]$. We can write $T[i..j] = w^t w[1..r]$, where $t \in \mathbb{N}$, $|w| = p$ and $r < |w|$. We also call $T[i..j]$ a* run of w. *The* Lyndon root *of a run of w is the cyclic shift of w that is a Lyndon word.*

Our general strategy is to preprocess all runs into a data structure, such that we can quickly determine the runs preceded by S_1 and followed by S_2, which additionally end on $w[1..d]$ and have a length that matches the query.

Data structure Let $T[i..j+r] = w^t w[1..r]$ be a run in T. For each $1 \leq a \leq t$ we insert a point in a three-dimensional grid $G_{w,a,b}$ where $b = t \bmod a$. Each point stores the positions i, j of the occurrence of the run and has coordinates x, y, z defined as follows:

- x is the lexicographic rank of $T[1..i-1]^{rev}$ among all the reversed prefixes of T.
- y is the lexicographical rank of $T[j+1..n]$ among all the suffixes of T.
- $z = \lfloor t/a \rfloor$

Furthermore, we construct a compact trie of the strings w of all runs and a lookup table for each such that given a and b we can find $G_{w,a,b}$. Finally, we store the suffix tree $\mathcal{T}$ of T and the suffix tree $\mathcal{T}^{rev}$ of the reversed text T^{rev}.

By the runs theorem, the sum of exponents of all runs in T is $O(n)$ [4,25], hence the total number of grids and points is $O(n)$. Let $|G_{w,a,b}|$ be the number of points in the grid $G_{w,a,b}$. We store $G_{w,a,b}$ in the orthogonal range reporting data structure [37] using $O(|G_{w,a,b}|)$ space, so that 5-sided searches on it take time $O((p+1)\log^{\epsilon}|G_{w,a,b}|)$, for any constant $\epsilon > 0$, to report the p points in the range. Hence, our structure uses $O(n)$ space in total.

Query To answer a query as above, we find the query ranges $[x_1, x_2] \times [y_1, y_2]$ using the suffix trees $\mathcal{T}$ and $\mathcal{T}^{rev}$. The ranges $[x_1, x_2]$ and $[y_1, y_2]$ correspond to the leaf ranges of the loci of S_1^{rev} in $\mathcal{T}^{rev}$ and $w[1..d]S_2$ in $\mathcal{T}$, respectively. Finally, we find all occurrences of $S_1 w^{aq+b} w[1..d] S_2$ with $q \geq c$ as the points in $G_{w,a,b}$ inside the 5-sided query $[x_1, x_2] \times [y_1, y_2] \times [c, +\infty]$.

The ranges in $\mathcal{T}$ and $\mathcal{T}^{rev}$ can be found in time $O(|d| + |S_1| + |S_2|) = O(|w| + |S_1| + |S_2|)$ if the suffix tree nodes use deterministic dictionaries to store their children. We then do a single query to the range data structure $G_{w,a,b}$, which reports occ points in $O((\text{occ}+1)\log^{\epsilon} n)$ time. We have proven the following:

▶ **Lemma 9.** *Given a text $T[1..n]$ over alphabet Σ, we can build a data structure that uses $O(n)$ space and can answer the following queries: Given two non-empty strings S_1 and S_2, a primitive string w, and numbers $a, b, c, d \in \mathbb{N}$ with $b < a$ and $d < |w|$, where S_1 and w do not share a suffix and S_2 and $w[d+1..]$ do not share a prefix, find all occurrences in T of patterns of the form $S_1 w^{aq+b} w[1..d] S_2$, where $q \geq c$ and $q \in \mathbb{N}$. The query time is $O(|S_1 S_2 w| + (occ+1)\log^{\epsilon} n)$, where occ is the number of occurrences of $S_1 w^{aq+b} w[1..d] S_2$.*

Transforming $P_1 P^* P_2$ into $S_1 w^{aq+b} w[1..d] S_2$ Given $P_1 P^* P_2$ we compute the strings S_1, w and S_2 and the numbers a, b, c, and d as follows: The string S_1 is $P_1[1..|P_1| - i]$ where i is the length of the longest common suffix of P_1 and $P^{\lceil |P_1|/|P| \rceil}$. Let $P' = P[(-i \bmod |P|) + 1..|P|] \cdot P[1..(-i \bmod |P|)]$ and $P_2' = P_1[|P_1| - i + 1..|P_1|]P_2$. We compute w and a such that $P' = w^a$ and $a \in \mathbb{N}$ is maximal (this can be done in time $O(|P'|)$ e.g. using KMP [24]). By definition of P' and i, we have that $P'[|P'|] = P[-i \bmod |P|] \neq P_1[|P_1| - i]$. Therefore, S_1 and w do not share a suffix.

$$\begin{array}{ll} DBC(ABCABCABC)^*ABCABCABCABCABCB & \\ D(BCABCABCA)^*BCABCABCABCABCABCB & // \quad S_1 = D \text{ and } P \text{ rotated} \\ D(BCA)^{3q}BCABCABCABCABCABCB & // \quad P' \text{ reduced to } w^3 = (BCA)^3 \\ D(BCA)^{3q}BCABCABCB \quad q \geq c = 1 & // \quad w^3 \text{ occurs at least once} \\ D(BCA)^{3q+2}BCB, \quad q \geq c = 1 & // \quad S_1 w^{3q+2} w[1..2] S_2 \end{array}$$

Figure 3 An example of the transformation applied when $P_1 = DBC$, $P = ABCABCABC$, and $P_2 = ABCABCABCABCABCB$. Here $S_1 = D$, $w = BCA$, $S_2 = B$, $a = 3$, $b = 2$, $c = 1$ and $d = 2$.

Let j be the length of the longest common prefix of P_2' and $w^{\lceil |P_2'|/|w| \rceil}$. We define S_2 as $P_2'[j+1..|P_2'|]$ and $d = j \bmod |w|$. Note that by definition of S_2, S_2 and $w[d+1..]$ do not share a prefix. Finally, we let $b = (j-d)/|w| \bmod a$ and $c = \lceil \frac{j-d}{a|w|} \rceil - b$. See Figure 3.

The transformation can be done in $O(|P_1| + |P_2| + |P|)$ time: The longest common suffix of P_1 and $P^{\lceil |P_1|/|P| \rceil}$ can be computed in $O(|P_1|)$ time and the longest common prefix of P_2' and $w^{\lceil |P_2'|/|w| \rceil}$ in $O(|P_2'|) = O(|P_1| + |P_2|)$ time. Further, as mentioned, the period of $|P'|$ can be found in $O(|P'|) = O(|P|)$ time. Other than that, the transformation consists of modulo calculations and cyclic shifts, which clearly can be done in linear time.

4.1 When one of S_1 and S_2 is the Empty String

In the transformation above, it might happen that S_1 or S_2 or both are empty, in which case the data structure from Lemma 9 cannot be used. In this and the next subsection, we give additional data structures to handle these cases. Let us first consider the case where $S_2 = \epsilon$ and $S_1 \neq \epsilon$. The general idea is that to answer a query $S_1 w^{aq+b} w[1..d]$, $q \geq c$, where S_1 and w do not share a suffix, we need to find all occurrences of S_1 followed by a long enough run of w. Note that each one of these occurrences can contain multiple occurrences of our pattern, for different choices of q.

Data structure Let $T[i..j+r] = w^t w[1..r]$ be a run in T. For each run in T, we insert a point into a two-dimensional grid G_w. Each point stores the positions i, j and r of the occurrence of the run. The coordinates x, y of the point in G_w are defined as follows:

- x is the lexicographic rank of $T[1..i-1]^{rev}$ among all reversed prefixes of T.
- $y = t|w| + r$.

In terms of space complexity, as before, by the runs theorem, the sum of exponents of all runs in T is $O(n)$ [4, 25]. Thus, the total number of points in G_w is $O(n)$. Further, we store a compact trie of all w's together with a dictionary for finding t and d using linear space. The two dimensional points can be processed into a data structure allowing 3-sided range queries in linear space and $O((\text{occ} + 1) \log^\epsilon n)$ running time [38], where occ is the number of reported points.

Query To answer a query $S_1 w^{aq+b} w[1..d]$, as before, we find the lexicographical range $[x_1, x_2]$ for S_1 using the suffix tree $\mathcal{T}$. Then, we query the grid G_w for $[x_1, x_2] \times [(ac+b)|w| + d, \infty]$. For a point (x, y) with (i, j, r) obtained this way, we report $T[i - |S_1| + 1, i + |w|(aq+b) + d]$ for all q such that $c \leq q$ and $i + |w|(aq+b) + d \leq j + r$, which is equivalent to $q \leq \lfloor \frac{(y-d)/|w| - b}{a} \rfloor$.

The querying of the grid reports occ points in $O((\text{occ}+1)\log^\epsilon n)$ running time, and each reported point gives at least one occurrence. The additional occurrences can be found in constant time per occurrence. Thus, the total query time is $O(|S_1S_2w| + (1+\text{occ})\log^\epsilon n)$.

We can deal with the case where $S_1 = \epsilon$ analogously, by building the same structure on T^{rev} and reversing the pattern.

4.2 When both S_1 and S_2 are the Empty String

If both S_1 and S_2 are the empty string, then we cannot "anchor" our occurrences at the start of a run—i.e., $w^{aq+b}w[1..d]$ may occur in runs whose period is a shift of w. To deal with this, we characterize all runs by their Lyndon root, and write $w^{aq+b}w[1..d]$ as a query of the form $w'[|w|-e+1]w'^{a'q+b'}w'[1..d']$, where w' is a Lyndon word. In the following, we show how to answer these kinds of queries.

We create a structure that given a primitive string w that is a Lyndon word, numbers a, b, c, $d < |w|$, and $e < |w|$, finds all occurrences of patterns of the form $w[|w|-e+1]w^{aq+b}w[1..d]$ in T, where $q \geq c$ and $q \in \mathbb{N}$.

Data structure For a run $T[i'..j'+r'] = u^{t'}u[1..r']$ in T, let w be the Lyndon root of the run, and let $r < |w|$, $l < |w|$ and t be such that $T[i'..j'+r'] = T[i-l+1..j+r] = w[|w|-l+1]w^tw[1..r]$. We build a three-dimensional grid G_w. For each run, we store i, j and the point $(x, y, z) = (l, t, r)$. We store G_w in a linear space data structure which supports five-sided range queries in time $O((\text{occ}+1)\log^\epsilon n)$, where occ is the number of reported points, given in [37]. By the runs theorem, the total number of points in all G_ws is bounded by $O(n)$, and thus so is the space of our data structure.

Query Assume we are given a query w, a, b, c, d, e. In the following, we have to again find runs of w which are long enough, but with an extra caveat: we need to treat the runs $w[|w|-l+1]w^tw[1..r]$ differently depending on i) if $e \leq l$ and ii) if $d \leq r$, since depending on those, the leftmost and rightmost occurrences in the run have different positions. This gives us four cases to investigate.

1. We find all points in $[e,\infty] \times [ac+b,\infty] \times [d,\infty]$. For each such, we output the following occurrences: $T[i-e+k\cdot|w|, i+(k+aq+b)|w|+d]$, where $k \leq t-ac-b$ and $c \leq q \leq \lfloor\frac{t-b-k}{a}\rfloor$.
2. We find all points in $[e,\infty] \times [ac+b+1,\infty] \times [0,d-1]$. For each such, we output all occurrences of the form $T[i-e+k\cdot|w|, i+(k+aq+b)|w|+d]$, where $k \leq t-1-ac-b$ and $c \leq q \leq \lfloor\frac{t-1-b-k}{a}\rfloor$.
3. We find all points in $[0,e-1] \times [ac+b+1,\infty] \times [d,\infty]$ and output the occurrences of the form $T[i+|w|-e+k\cdot|w|, i+|w|+(k+aq+b)|w|+d]$, where $k \leq t-ac-b-1$ and $c \leq q \leq \lfloor\frac{t-b-k-1}{a}\rfloor$.
4. We find all points in $[0,e-1] \times [ac+b+2,\infty] \times [0,d-1]$ and output all occurrences of the form $T[i+|w|-e+k\cdot|w|, i+|w|+(k+aq+b)|w|+d]$, where $k \leq t-ac-b-2$ and $c \leq q \leq \lfloor\frac{t-b-k-2}{a}\rfloor$.

Each range query uses $O((\text{occ}+1)\log^\epsilon n)$ time, where occ is the number of reported points, and each reported point gives at least one occurrence. Additional occurrences within the same run can be found in constant time per occurrence. Thus, the total time is $O((occ+1)\log^\epsilon n)$.

In summary, we have proved Theorem 3.

References

1. Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 427–436, 1995.
2. Arturs Backurs and Piotr Indyk. Which regular expression patterns are hard to match? In *Proc. 57th FOCS*, pages 457–466, 2016.
3. Arturs Backurs and Piotr Indyk. Which regular expression patterns are hard to match? In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 457–466. IEEE, 2016.
4. Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "runs" theorem. *SIAM J. Comput.*, 46(5):1501–1514, 2017.
5. Philip Bille. New algorithms for regular expression matching. In *Proc. 33rd ICALP*, pages 643–654, 2006.
6. Philip Bille and Martin Farach-Colton. Fast and compact regular expression matching. *Theoret. Comput. Sci.*, 409:486 – 496, 2008.
7. Philip Bille and Inge Li Gørtz. Substring range reporting. *Algorithmica*, 69:384–396, 2014.
8. Philip Bille and Inge Li Gørtz. Sparse regular expression matching. In *Proc. 35th SODA*, pages 3354–3375, 2024.
9. Philip Bille, Inge Li Gørtz, Max Rishøj Pedersen, and Teresa Anna Steiner. Gapped indexing for consecutive occurrences. *Algorithmica*, 85(4):879–901, 2023.
10. Philip Bille, Inge Li Gørtz, Max Rishøj Pedersen, and Teresa Anna Steiner. Gapped indexing for consecutive occurrences. *Algorithmica*, 85(4):879–901, 2023. URL: `https://doi.org/10.1007/s00453-022-01051-6`, `doi:10.1007/S00453-022-01051-6`.
11. Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and Søren Vind. String indexing for patterns with wildcards. *Theory Comput. Syst.*, 55(1):41–60, 2014.
12. Philip Bille and Mikkel Thorup. Faster regular expression matching. In *Proc. 36th ICALP*, pages 171–182, 2009.
13. Philip Bille and Mikkel Thorup. Regular expression matching with multi-strings and intervals. In *Proc. 21st SODA*, 2010.
14. Karl Bringmann, Allan Grønlund, and Kasper Green Larsen. A dichotomy for regular expression membership testing. In *Proc. 58th FOCS*, pages 307–318, 2017.
15. Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proc. 36th STOC*, pages 91–100, 2004.
16. Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *International Conference on Automata, Languages and Programming*, pages 6–19, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.
17. Minos N Garofalakis, Rajeev Rastogi, and Kyuseok Shim. SPIRIT: Sequential pattern mining with regular expression constraints. In *Proc. 25th VLDB*, pages 223–234, 1999.
18. Daniel Gibney and Sharma V. Thankachan. Text indexing for regular expression matching. *Algorithms*, 14(5), 2021. URL: `https://www.mdpi.com/1999-4893/14/5/133`, `doi:10.3390/a14050133`.
19. Isaac Goldstein, Tsvi Kopelowitz, Moshe Lewenstein, and Ely Porat. Conditional lower bounds for space/time tradeoffs. In *Proc. 15th WADS*, pages 421–436, 2017. `doi:10.1007/978-3-319-62127-2\_36`.
20. Costas S. Iliopoulos and M. Sohel Rahman. Indexing factors with gaps. *Algorithmica*, 55(1):60–70, 2009.
21. Theodore Johnson, S. Muthukrishnan, and Irina Rozenbaum. Monitoring regular expressions on out-of-order streams. In *Proc. 23nd ICDE*, pages 1315–1319, 2007.
22. Kenrick Kin, Björn Hartmann, Tony DeRose, and Maneesh Agrawala. Proton: multitouch gestures as regular expressions. In *Proc. SIGCHI*, pages 2885–2894, 2012.

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

24 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.

25 Roman M. Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 596–604. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814634.

26 Tsvi Kopelowitz and Robert Krauthgamer. Color-distance oracles and snippets. In *Proc. 27th CPM*, pages 24:1–24:10, 2016.

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

28 Moshe Lewenstein. Indexing with gaps. In *Proc. 18th SPIRE*, pages 135–143, 2011.

29 Moshe Lewenstein, J. Ian Munro, Venkatesh Raman, and Sharma V. Thankachan. Less space: Indexing for queries with wildcards. *Theor. Comput. Sci.*, 557:120–127, 2014.

30 Moshe Lewenstein, Yakov Nekrich, and Jeffrey Scott Vitter. Space-efficient string indexing for wildcard pattern matching. In *Proc. 31st STACS*, pages 506–517, 2014.

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

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

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

34 G. Navarro and Y. Nekrich. Top-k document retrieval in compressed space. In *Proc. 36th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 4009–4030, 2025.

35 Gonzalo Navarro and Mathieu Raffinot. Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. *J. Comput. Bio.*, 10(6):903–923, 2003.

36 Yakov Nekrich. New data structures for orthogonal range reporting and range minima queries. *arXiv preprint arXiv:2007.11094*, 2020.

37 Yakov Nekrich. New data structures for orthogonal range reporting and range minima queries. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1191–1205. SIAM, 2021.

38 Yakov Nekrich and Gonzalo Navarro. Sorted range reporting. In *Proc. 13th SWAT*, pages 271–282, 2012. doi:10.1007/978-3-642-31155-0_24.

39 Pierre Peterlongo, Julien Allali, and Marie-France Sagot. Indexing gapped-factors using a tree. *Int. J. Found. Comput. Sci.*, 19(1):71–87, 2008.

40 Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, page 233–242, USA, 2002. Society for Industrial and Applied Mathematics.

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

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

43 Larry Wall. *The Perl Programming Language.* Prentice Hall Software Series, 1994.

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

A Conditional Lower Bound for Character-class Kleene-star Patterns without an Anchor

We now show Theorem 2. The conditional lower bound is based on the Strong Set Disjointness Conjecture formulated in [19] and stated in the following.

▶ **Definition 10** (The Set Disjointness Problem). *In the Set Disjointness problem, the goal is to preprocess sets $S_1, \ldots, S_m$ of elements from a universe U into a data structure, to answer the following kind of query: For a pair of sets S_i and S_j, is $S_i \cap S_j$ empty or not?*

▶ **Conjecture 11** (The Strong Set Disjointness Conjecture). *For an instance $S_1, \ldots, S_m$ satisfying $\sum_{i=1}^{m} |S_i| = N$, any solution to the Set Disjointness problem answering queries in $O(t)$ time must use $\tilde{\Omega}\left(\frac{N^2}{t^2}\right)$ space.*

The lower bound example in [10], Section 5.2, specifically shows that, assuming Conjecture 11, indexing $T[1..n]$ to solve queries of the form $P_1\Sigma^{\leq r}P_2$ requires $\tilde{\Omega}(n^{2-2\delta-o(1)})$ space, assuming one desires to answer queries in $O(n^\delta)$ time, for any $\delta \in [0, 1/2]$. The alphabet size in their lower bound example is 3. To extend this lower bound to queries of the form $P_1D^*P_2$, we have to slightly adapt this lower bound and increase the alphabet size to 4 ($k_{\max}$ will equal 3 in the example).

When reducing from Set Disjointness, as a first step, [10] shows that we can assume that every universe element appears in the same number of sets (Lemma 6 in [10]). Call this number f. Then, they construct a string of length $2N \log m + 2N$ from alphabet $\{0, 1, \$\}$ as follows: For each element $e \in U$, they build a gadget consisting of the concatenation of the binary encodings of the sets e is contained in, each encoding followed by a \$. Such a gadget has length $B = f \log m + f$. To each gadget, they append a block of B many \$, and then append the resulting strings of length $2B$ in an arbitrary order.

We adapt this reduction as follows: the gadgets are defined in the same way as before, only each gadget is followed by a symbol #, where $\# \notin \{0, 1, \$\}$, instead of a block $\$^B$. The rest of the construction is the same. Now, if we want to answer a query S_i, S_j to the Set Disjointness problem, we set P_1 to the binary encoding of i, P_2 to the binary encoding of j, and $D = \{0, 1, \$\}$. It will find an occurrence if and only if there is a gadget corresponding to an element e which contains both the encoding of i and j, which means that e is contained in both S_i and S_j. The rest of the proof proceeds as in [10].