


Internal Dictionary Matching

Panagiotis Charalampopoulos 

Department of Informatics, King's College London, London, UK
Institute of Informatics, University of Warsaw, Warsaw, Poland
<https://nms.kcl.ac.uk/panagiotis.charalampopoulos/>
panagiotis.charalampopoulos@kcl.ac.uk

Tomasz Kociumaka 

Institute of Informatics, University of Warsaw, Warsaw, Poland
Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel
<https://www.mimuw.edu.pl/~kociumaka/>
kociumaka@mimuw.edu.pl

Manal Mohamed 

Department of Informatics, King's College London, London, UK
manal.mohamed@kcl.ac.uk

Jakub Radoszewski 

Institute of Informatics, University of Warsaw, Warsaw, Poland
Samsung R&D Institute, Warsaw, Poland
<https://www.mimuw.edu.pl/~jrad>
jrad@mimuw.edu.pl

Wojciech Rytter 

Institute of Informatics, University of Warsaw, Warsaw, Poland
<https://www.mimuw.edu.pl/~rytter>
rytter@mimuw.edu.pl

Tomasz Waleń 

Institute of Informatics, University of Warsaw, Warsaw, Poland
<https://www.mimuw.edu.pl/~walen>
walen@mimuw.edu.pl

Abstract

We introduce data structures answering queries concerning the occurrences of patterns from a given dictionary \mathcal{D} in *fragments* of a given string T of length n . The dictionary is *internal* in the sense that each pattern in \mathcal{D} is given as a fragment of T . This way, \mathcal{D} takes space proportional to the number of patterns $d = |\mathcal{D}|$ rather than their total length, which could be $\Theta(n \cdot d)$.

In particular, we consider the following types of queries: reporting and counting *all* occurrences of patterns from \mathcal{D} in a fragment $T[i..j]$ (operations $\text{REPORT}(i, j)$ and $\text{COUNT}(i, j)$ below, as well as operation $\text{EXISTS}(i, j)$ that returns true iff $\text{COUNT}(i, j) > 0$) and reporting *distinct* patterns from \mathcal{D} that occur in $T[i..j]$ (operation $\text{REPORTDISTINCT}(i, j)$). We show how to construct, in $\mathcal{O}((n + d) \log^{\mathcal{O}(1)} n)$ time, a data structure that answers each of these queries in time $\mathcal{O}(\log^{\mathcal{O}(1)} n + |\text{output}|)$ – see the table below for specific time and space complexities.

Query	Preprocessing time	Space	Query time
$\text{EXISTS}(i, j)$	$\mathcal{O}(n + d)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
$\text{REPORT}(i, j)$	$\mathcal{O}(n + d)$	$\mathcal{O}(n + d)$	$\mathcal{O}(1 + \text{output})$
$\text{REPORTDISTINCT}(i, j)$	$\mathcal{O}(n \log n + d)$	$\mathcal{O}(n + d)$	$\mathcal{O}(\log n + \text{output})$
$\text{COUNT}(i, j)$	$\mathcal{O}(\frac{n \log n}{\log \log n} + d \log^{3/2} n)$	$\mathcal{O}(n + d \log n)$	$\mathcal{O}(\frac{\log^2 n}{\log \log n})$

The case of counting patterns is much more involved and needs a combination of a locally consistent parsing with orthogonal range searching. Reporting distinct patterns, on the other hand, uses the structure of maximal repetitions in strings. Finally, we provide tight – up to subpolynomial factors – upper and lower bounds for the case of a dynamic dictionary.



© Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń;
licensed under Creative Commons License CC-BY

30th International Symposium on Algorithms and Computation (ISAAC 2019).

Editors: Pinyan Lu and Guochuan Zhang; Article No. 22; pp. 22:1–22:17



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases string algorithms, dictionary matching, internal pattern matching

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2019.22

Related Version A full version of the paper is available at <https://arxiv.org/abs/1909.11577>.

Funding *Panagiotis Charalampopoulos*: Supported in part by the ERC grant TOTAL agreement no. 677651.

Tomasz Kociumaka: Supported by ISF grants no. 824/17 and 1278/16 and by an ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064).

Jakub Radoszewski: Supported by the Polish National Science Center, grant number 2018/31/D/ST6/03991.

Tomasz Walen: Supported by the Polish National Science Center, grant number 2018/31/D/ST6/03991.

Acknowledgements Panagiotis Charalampopoulos and Manal Mohamed thank Solon Pissis for preliminary discussions.

1 Introduction

In the problem of dictionary matching, which has been studied for more than forty years, we are given a dictionary \mathcal{D} , consisting of d patterns, and the goal is to preprocess \mathcal{D} so that presented with a text T we are able to efficiently compute the occurrences of the patterns from \mathcal{D} in T . The Aho–Corasick automaton preprocesses the dictionary in linear time with respect to its total length and then processes T in time $\mathcal{O}(|T| + |\text{output}|)$ [1]. Compressed indexes for dictionary matching [9], as well as indexes for approximate dictionary matching [10] have been studied. Dynamic dictionary matching in its more general version consists in the problem where a dynamic dictionary is maintained, text strings are presented as input and for each such text all the occurrences of patterns from the dictionary in the text have to be reported; see [2, 3].

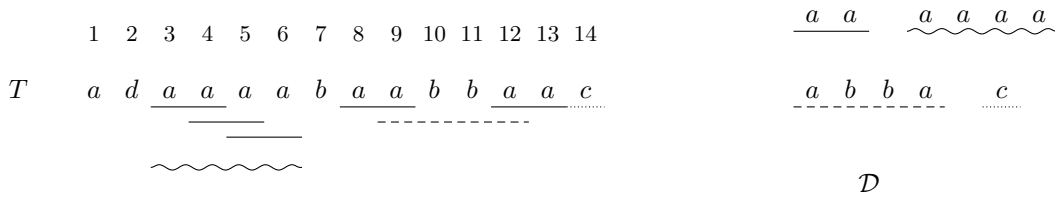
Internal queries in texts have received much attention in recent years. Among them, the *Internal Pattern Matching* (IPM) problem consists in preprocessing a text T of length n so that we can efficiently compute the occurrences of a substring of T in another substring of T . A nearly-linear sized data structure that allows for sublogarithmic-time IPM queries was presented in [22], while a linear sized data structure allowing for constant-time IPM queries in the case that the ratio between the lengths of the two substrings is constant was presented in [25]. Other types of internal queries include computing the longest common prefix of two substrings of T , computing the periods of a substring of T , etc. We refer the interested reader to [23], which contains an overview of the literature.

We introduce the problem of *Internal Dictionary Matching* (IDM) that consists in answering the following types of queries for an internal dictionary \mathcal{D} consisting of substrings of text T : given (i, j) , report/count all occurrences of patterns from \mathcal{D} in $T[i..j]$ and report the distinct patterns from \mathcal{D} that occur in $T[i..j]$.

Some interesting internal dictionaries \mathcal{D} are the ones comprising of palindromic, square, or non-primitive substrings of T . In each of these three cases, the total length of patterns might be quadratic, but the internal dictionary is of linear size and can be constructed in $\mathcal{O}(n)$ time [16, 12, 6]. Our data structure provides a general framework for solving problems related to the internal structure of the string. The case of palindromes has already been studied in [30], where authors proposed a data structure of size $\mathcal{O}(n \log n)$ that returns the number of all distinct palindromes in $T[i..j]$ in $\mathcal{O}(\log n)$ time.

Let us formally define the problem and the types of queries that we consider.

INTERNAL DICTIONARY MATCHING
Input: A text T of length n and a dictionary \mathcal{D} consisting of d patterns, each given as a substring $T[a..b]$ of T .
Queries:
EXISTS(i, j): Decide whether at least one pattern $P \in \mathcal{D}$ occurs in $T[i..j]$.
REPORT(i, j): Report all occurrences of all the patterns of \mathcal{D} in $T[i..j]$.
REPORTDISTINCT(i, j): Report all patterns $P \in \mathcal{D}$ that occur in $T[i..j]$.
COUNT(i, j): Count the number of all occurrences of all the patterns of \mathcal{D} in $T[i..j]$.



■ **Figure 1** Occurrences of patterns from the dictionary \mathcal{D} in the text T .

► **Example 1.** Let us consider the dictionary $\mathcal{D} = \{aa, aaaa, abba, c\}$ and the text $T = adaaaabaabbaac$; see Fig. 1. We then have:

$$\begin{aligned} \text{EXISTS}(2, 12) &= \mathbf{true} \\ \text{REPORT}(2, 12) &= \{(aa, 3), (aaaa, 3), (aa, 4), (aa, 5), (aa, 8), (abba, 9)\} \\ \text{COUNT}(2, 12) &= 6 \\ \text{REPORTDISTINCT}(2, 12) &= \{aa, aaaa, abba\} \\ \text{EXISTS}(1, 3) &= \mathbf{false} \end{aligned}$$

Let us consider **REPORT**(i, j) queries. One could answer them in time $\mathcal{O}(j - i + |\text{output}|)$ by running $T[i..j]$ over the Aho–Corasick automaton of \mathcal{D} [1] or in time $\tilde{\mathcal{O}}(d + |\text{output}|)$ ¹ by performing internal pattern matching [25] for each element of \mathcal{D} individually. None of these approaches is satisfactory as they can require $\Omega(n)$ time in the worst case.

Our results. A natural problem would be to consider a dynamic dictionary, in the sense that one would perform interleaved IDM queries and updates to \mathcal{D} (insertions/deletions of patterns). We show a conditional lower bound for this problem. In particular, we show that the product of the time to process an update and the time to answer whether any pattern from \mathcal{D} occurs in $T[i..j]$ cannot be $\mathcal{O}(n^{1-\epsilon})$ for any constant $\epsilon > 0$, unless the Online Boolean Matrix-Vector Multiplication conjecture [18] is false. Interestingly, in our lower bound construction we only add single-letter patterns to an initially empty dictionary.

We thus focus on the case of a static dictionary, as it was defined above. We propose an $\tilde{\mathcal{O}}(n + d)$ -sized data structure, which can be built in time $\tilde{\mathcal{O}}(n + d)$ and answers all IDM queries in time $\tilde{\mathcal{O}}(1 + |\text{output}|)$. The exact complexities are shown in the table on the front page.

¹ The $\tilde{\mathcal{O}}(\cdot)$ notation suppresses $\log^{O(1)} n$ factors.

By building upon our solutions for static dictionaries, we provide algorithms for the case of a dynamic dictionary, where patterns can be added to or removed from \mathcal{D} . We show how to process updates in $\tilde{O}(n^\alpha)$ time and answer queries $\text{EXISTS}(i, j)$, $\text{REPORT}(i, j)$ and $\text{REPORTDISTINCT}(i, j)$ in $\tilde{O}(n^{1-\alpha} + |\text{output}|)$ time for any $0 < \alpha < 1$, matching – up to subpolynomial factors – our conditional lower bound.

Our techniques and a roadmap. First, in Section 3, we present straightforward solutions for queries $\text{EXISTS}(i, j)$ and $\text{REPORT}(i, j)$. In Section 4 we describe an involved solution for $\text{REPORTDISTINCT}(i, j)$ queries, that heavily relies on the periodic structure of the input text and on tools that we borrow from computational geometry. In Section 5 we rely on locally consistent parsing and further computational geometry tools to obtain an efficient solution for $\text{COUNT}(i, j)$ queries. Finally, in Section 6 we extend our solutions for the case of a dynamic dictionary and provide a matching conditional lower bound.

2 Preliminaries

We begin with basic definitions and notation generally following [11]. Let $T = T[1]T[2] \cdots T[n]$ be a *string* of length $|T| = n$ over a linearly sortable alphabet Σ . The elements of Σ are called *letters*. By ε we denote an *empty string*. For two positions i and j on T , we denote by $T[i..j] = T[i] \cdots T[j]$ the *fragment* (sometimes called substring) of T that starts at position i and ends at position j (it equals ε if $j < i$). It is called *proper* if $i > 1$ or $j < n$. A fragment of T is represented in $\mathcal{O}(1)$ space by specifying the indices i and j . A *prefix* of T is a fragment that starts at position 1 ($T[1..j]$, notation: $T^{(j)}$) and a *suffix* is a fragment that ends at position n ($T[i..n]$, notation: $T_{(i)}$). We denote the *reverse string* of T by T^R , i.e. $T^R = T[n]T[n-1] \cdots T[1]$.

Let U be a string of length m with $0 < m \leq n$. We say that there exists an *occurrence* of U in T , or, more simply, that U *occurs in* T , when U is a fragment of T . We thus say that U occurs at the *starting position* i in T when $U = T[i..i+m-1]$.

If a string U is both a proper prefix and a proper suffix of a string T of length n , then U is called a *border* of T . A positive integer p is called a *period* of T if $T[i] = T[i+p]$ for all $i = 1, \dots, n-p$. A string T has a period p if and only if it has a border of length $n-p$. We refer to the smallest period as *the period* of the string, and denote it as $\text{per}(T)$, and, analogously, to the longest border as *the border* of the string. A string is called *periodic* if its period is no more than half of its length and *aperiodic* otherwise.

The elements of the dictionary \mathcal{D} are called *patterns*. Henceforth we assume that $\varepsilon \notin \mathcal{D}$, i.e. the length of each $P \in \mathcal{D}$ is at least 1. If ε was in \mathcal{D} , we could trivially treat it individually. We further assume that each pattern of \mathcal{D} is given by the starting and ending positions of its occurrence in T . Thus, the size of the dictionary $d = |\mathcal{D}|$ refers to the number of strings in \mathcal{D} and not their total length.

The *suffix tree* $\mathcal{T}(T)$ of a non-empty string T of length n is a compact trie representing all suffixes of T . The *branching* nodes of the trie as well as the *terminal* nodes, that correspond to suffixes of T , become *explicit* nodes of the suffix tree, while the other nodes are *implicit*. Each edge of the suffix tree can be viewed as an upward maximal path of implicit nodes starting with an explicit node. Moreover, each node belongs to a unique path of that kind. Thus, each node of the trie can be represented in the suffix tree by the edge it belongs to and an index within the corresponding path. We let $\mathcal{L}(v)$ denote the *path-label* of a node v , i.e., the concatenation of the edge labels along the path from the root to v . We say that v is path-labelled $\mathcal{L}(v)$. Additionally, $\delta(v) = |\mathcal{L}(v)|$ is used to denote the *string-depth* of node v .

A terminal node v such that $\mathcal{L}(v) = T_{(i)}$ for some $1 \leq i \leq n$ is also labelled with index i . Each fragment of T is uniquely represented by either an explicit or an implicit node of $\mathcal{T}(T)$, called its *locus*. Once $\mathcal{T}(T)$ is constructed, it can be traversed in a depth-first manner to compute the string-depth $\delta(v)$ for each explicit node v . The suffix tree of a string of length n , over an integer ordered alphabet, can be computed in time and space $\mathcal{O}(n)$ [13]. In the case of integer alphabets, in order to access the child of an explicit node by the first letter of its edge label in $\mathcal{O}(1)$ time, perfect hashing [14] can be used. Throughout the paper, when referring to the suffix tree $\mathcal{T}(T)$ of T , we mean the suffix tree of $T\$$, where $\$ \notin \Sigma$ is a sentinel letter that is lexicographically smaller than all the letters in Σ . This ensures that all terminal nodes are leaves.

We say that a tree is a *weighted tree* if it is a rooted tree with an integer weight on each node v , denoted by $\omega(v)$, such that the weight of the root is zero and $\omega(u) < \omega(v)$ if u is the parent of v . We say that a node v is a *weighted ancestor at depth ℓ* of a node u if v is the highest ancestor of u with weight of at least ℓ . After $\mathcal{O}(n)$ -time preprocessing, weighted ancestor queries for nodes of a weighted tree \mathcal{T} of size n can be answered in $\mathcal{O}(\log \log n)$ time per query [4]. If ω has a property that the difference of weights of a child and its parent is always equal to 1, then the queries can be answered in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$ -time preprocessing [7]; in this special case the values ω are called *levels* and the queries are called *level ancestor queries*. The suffix tree $\mathcal{T}(T)$ is a weighted tree with $\omega = \delta$. Hence, the locus of a fragment $T[i..j]$ in $\mathcal{T}(T)$ is the weighted ancestor of the terminal node with path-label $T_{(i)}$ at string-depth $j - i + 1$.

3 Exists(i, j) and Report(i, j) queries

We first present a convenient modification to the suffix tree with respect to a dictionary \mathcal{D} ; see Fig. 2.

► **Definition 2.** A \mathcal{D} -modified suffix tree of string T is a tree with terminal nodes corresponding to non-empty suffixes of $T\$$ and branching nodes corresponding to $\{\varepsilon\} \cup \mathcal{D}$. A node corresponding to string U is an ancestor of a node corresponding to string V if and only if U is a prefix of V . Each node stores its level as well as its string-depth (i.e., the length of its corresponding string).

► **Lemma 3.** A \mathcal{D} -modified suffix tree of T has size $\mathcal{O}(n + d)$ and can be constructed in $\mathcal{O}(n + d)$ time.

Proof. The \mathcal{D} -modified suffix tree is obtained from the suffix tree $\mathcal{T}(T)$ in two steps.

In the first step, we mark all nodes of $\mathcal{T}(T)$ with path-label equal to a pattern $P \in \mathcal{D}$: if any of them are implicit, we first make them explicit; see Fig. 3(a). We can find the loci of the patterns in $\mathcal{T}(T)$ in $\mathcal{O}(n + d)$ time by answering the weighted ancestor queries as a batch [24], employing a data structure for a special case of Union-Find [15]. (If many implicit nodes along an edge are to become explicit, we can avoid the local sorting based on depth if we sort globally in time $\mathcal{O}(n + d)$ using bucket sort and then add the new explicit nodes in decreasing order with respect to depth.)

In the second step, we recursively contract any edge (u, v) , where u is the parent of v if:

1. both u and v are unmarked, or
2. u is marked and v is an unmarked internal node.

The resulting tree is the \mathcal{D} -modified suffix tree and has $\mathcal{O}(n)$ terminal nodes and $\mathcal{O}(d)$ internal nodes; see Fig. 3(b). ◀

Proof. (a) Let us define an array $B[a] = \min\{b : T[a..b] \in \mathcal{D}\}$. If there is no pattern from \mathcal{D} starting in T at position a , then $B[a] = \infty$. It can be readily verified that the answer to query $\text{EXISTS}(i, j)$ is yes if and only if the minimum element in the subarray $B[i..j]$ is at most j . Thus, in order to answer $\text{EXISTS}(i, j)$ queries, it suffices to construct the array B and a data structure that answers range minimum queries (RMQ) on B . Using the \mathcal{D} -modified suffix tree of T , whose construction time is the bottleneck, array B can be populated in $\mathcal{O}(n)$ time as follows. For each terminal node with path-label $T_{(a)}$ and level greater than 1, we set $B[a]$ to the string-depth of its ancestor at level 1 using a level ancestor query. If the terminal node is at level 1, then $B[a] = \infty$. A data structure answering range minimum queries in $\mathcal{O}(1)$ time can be built in time $\mathcal{O}(n)$ [17, 8].

(b) We first identify all positions $a \in [i..j]$ that are starting positions of occurrences of some pattern $P \in \mathcal{D}$ in $T[i..j]$ using RMQs over array B , which has been defined in the proof of part (a), as follows. The first RMQ, is over the range $[i..j]$ and identifies a position a (if any such position exists). The range is then split into two parts, namely $[i, a-1]$ and $[a+1, j]$. We recursively, use RMQs to identify the remaining positions in each part. Once we have found all the positions where at least one pattern from \mathcal{D} occurs, we report all the patterns occurring at each of these positions and being contained in $T[i..j]$. The complexities follow from Lemmas 3 and 4. \blacktriangleleft

4 ReportDistinct(i, j) queries

Below, we present an algorithm that reports patterns from \mathcal{D} occurring in $T[i..j]$, allowing for $\mathcal{O}(1)$ copies of each pattern on the output. We can then sort these patterns, remove duplicates, and report distinct ones using an additional global array of counters, one for each pattern.

Let us first partition \mathcal{D} into $\mathcal{D}_0, \dots, \mathcal{D}_{\lfloor \log n \rfloor}$ such that $\mathcal{D}_k = \{P \in \mathcal{D} : \lfloor \log |P| \rfloor = k\}$. We call \mathcal{D}_k a k -dictionary. We now show how to process a single k -dictionary \mathcal{D}_k ; the query procedure may clearly assume $k \leq \log |T[i..j]|$.

We precompute an array $L_k[1..n]$ such that $T[a..L_k[a]]$ is the longest pattern in \mathcal{D}_k is a prefix of $T_{(a)}$. We can do this in $\mathcal{O}(n)$ time by inspecting the parents of terminal nodes in the \mathcal{D}_k -modified suffix tree. Next, we assign to all the patterns of \mathcal{D}_k equal to some $T[a..L_k[a]]$ integer identifiers id (or colors) in $[1..n]$, and construct an array $I_k[a] = \text{id}(P)$, where $P = T[a..L_k[a]]$. We then rely on the following theorem.

► **Theorem 6** (Colored Range Reporting [28]). *Given an array $A[1..N]$ of elements from $[1..U]$, we can construct a data structure of size $\mathcal{O}(N)$ in $\mathcal{O}(N+U)$ time, so that upon query $[i..j]$ all distinct elements in $A[i..j]$ can be reported in $\mathcal{O}(1 + |\text{output}|)$ time.*

We first perform a colored range reporting query on the range $[i..j - 2^{k+1}]$ of array I_k and obtain a set of distinct patterns \mathcal{C}_k , employing Theorem 6. We observe the following.

► **Observation 7.** *Any pattern of a k -dictionary \mathcal{D}_k occurring in T at position $p \in [i..j - 2^{k+1}]$ is a prefix of a pattern $P \in \mathcal{C}_k$.*

Based on this observation, we will report the remaining patterns using the \mathcal{D}_k -modified suffix tree, following parent pointers and temporarily marking the loci of reported patterns to avoid double-reporting. We thus now only have to compute the patterns from \mathcal{D}_k that occur in $T[t..j]$, where $t = \max\{i, j - 2^{k+1} + 1\}$.

We further partition \mathcal{D}_k for $k > 1$ to a *periodic k -dictionary* and an *aperiodic k -dictionary*:

$$\mathcal{D}_k^p = \{P \in \mathcal{D}_k : \text{per}(P) \leq 2^k/3\} \quad \text{and} \quad \mathcal{D}_k^a = \{P \in \mathcal{D}_k : \text{per}(P) > 2^k/3\}.$$

Note that we can partition \mathcal{D}_k in $\mathcal{O}(|\mathcal{D}_k|)$ time using the so-called 2-PERIOD QUERIES of [25, 5, 23]. Such a query decides whether a given fragment of the text is periodic and, if so, it also returns its period. It can be answered in $\mathcal{O}(1)$ time after an $\mathcal{O}(n)$ -time preprocessing of the text.

4.1 Processing an aperiodic k -dictionary

We make use of the following sparsity property.

► **Fact 8** (Sparsity of occurrences). *The occurrences of a pattern P of an aperiodic k -dictionary \mathcal{D}_k^a in T start over $\frac{1}{6}|P|$ positions apart.*

Proof. If two occurrences of P started $d \leq \frac{2^k}{3}$ positions apart, then d would be a period of P , contradicting $P \in \mathcal{D}_k^a$. Then, since $2^k \leq |P| < 2^{k+1}$, we have that $2^k/3 \geq \frac{1}{6}|P|$. ◀

► **Lemma 9.** *REPORTDISTINCT(t, j) queries for the aperiodic k -dictionary \mathcal{D}_k^a and $j - t \leq 2^{k+1}$ can be answered in $\mathcal{O}(1 + |\text{output}|)$ time with a data structure of size $\mathcal{O}(n + |\mathcal{D}_k^a|)$, that can be constructed in $\mathcal{O}(n + |\mathcal{D}_k^a|)$ time.*

Proof. Since the fragment $T[t..j]$ is of length at most 2^{k+1} , it may only contain a constant number of occurrences of each pattern in \mathcal{D}_k^a by Fact 8. We can thus simply use a REPORT(t, j) query for dictionary \mathcal{D}_k^a and then remove duplicates. The complexities follow from Theorem 5(b). ◀

4.2 Processing a periodic k -dictionary

Our solution for periodic patterns relies on the well-studied theory of maximal repetitions (*runs*) in strings. A run is a periodic fragment $R = T[a..b]$ which can be extended neither to the left nor to the right without increasing the period $p = \text{per}(R)$, that is, $T[a-1] \neq T[a+p-1]$ and $T[b-p+1] \neq T[b+1]$ provided that the respective letters exist. The number of runs in a string of length n is $\mathcal{O}(n)$ and all the runs can be computed in $\mathcal{O}(n)$ time [26, 5].

► **Observation 10.** *Let P be a periodic pattern. If P occurs in $T[t..j]$, then P is a fragment of a unique run R such that $\text{per}(R) = \text{per}(P)$. We say that this run R extends P .*

Let \mathcal{R} be the set of all runs in T . Following [23], we construct for all $k \in [0.. \lfloor \log n \rfloor]$ the sets of runs $\mathcal{R}_k = \{R \in \mathcal{R} : \text{per}(R) \leq \frac{2^k}{3}, |R| \geq 2^k\}$ in $\mathcal{O}(n)$ time overall. Note that these sets are not disjoint; however, $|\mathcal{R}_k| = \mathcal{O}(\frac{n}{2^k})$ (cf. Lemma 11 below) and thus their total size is $\mathcal{O}(n)$. If U is a fragment of T , by $\mathcal{R}_k(U) \subseteq \mathcal{R}_k$ we denote the set of all runs $R \in \mathcal{R}_k$ such that $|R \cap U| \geq 2^k$, that is, runs whose overlap with the fragment U is at least 2^k .

► **Lemma 11** (see [23, Lemma 4.4.7]). $|\mathcal{R}_k(U)| = \mathcal{O}(\frac{1}{2^k}|U|)$.

Strategy. Given a fragment $U = T[t..j]$, we will first identify all runs $\mathcal{R}_k(U)$ of \mathcal{R}_k that have a sufficient overlap with U . There is a constant number of them by Lemma 11. For an occurrence of a pattern $P \in \mathcal{D}_k^p$ in U , the unique run R extending this occurrence of P must be in $\mathcal{R}_k(U)$. We will preprocess the runs in order to be able to compute a unique (the leftmost) occurrence *induced* by run R for each such pattern P .

► **Lemma 12.** *Let U be a fragment of T of length at most 2^{k+1} . Then $\mathcal{R}_k(U)$ can be retrieved in $\mathcal{O}(1)$ time after an $\mathcal{O}(n)$ -time preprocessing.*

Proof. PERIODIC EXTENSION QUERIES [23, Section 5.1], given a fragment V of the text T as input, return the run R extending V . They can be answered in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$ -time preprocessing.

Let us cover U using $\mathcal{O}(\frac{1}{2^k}|U|)$ fragments of length $\frac{2^{k+1}}{3}$ with overlaps of at least $\frac{2^k}{3}$ and ask a PERIODIC EXTENSION QUERY for each fragment V in the cover. For each run $R \in \mathcal{R}_k(U)$ with sufficient overlap, $R \cap U$ must contain a fragment V in the cover and its periodic extension must be R since $|V| \geq 2 \cdot \text{per}(R)$. ◀

Preprocessing. We construct an array $\ell_k[1..n]$ such that $T[i.. \ell_k[i]]$ is the shortest pattern $P \in \mathcal{D}_k^p$ that occurs at position i . Note that $\ell_k[i]$ can be retrieved in $\mathcal{O}(1)$ time using a level ancestor query in the \mathcal{D}_p^k -modified suffix tree (asking for a level-1 ancestor of the leaf corresponding to $T_{(i)}$, as in the proof of Theorem 5(a)). We then preprocess the array ℓ_k for RMQ queries.

Processing a run at query. Let us begin with a consequence of the fact that the shortest period is primitive.

► **Observation 13.** *If a pattern P occurs in a text Q and satisfies $|P| \geq \text{per}(Q)$, then P has exactly one occurrence in the first $\text{per}(Q)$ positions of Q .*

We use RMQs repeatedly, as in the proof of Theorem 5(b), for the subarray of ℓ_k corresponding to the first $\text{per}(R)$ positions of $R \cap U$. This way, due to Observation 13, we compute exactly the positions where a pattern $P \in \mathcal{D}_k^p$ has its leftmost occurrence in $R \cap U$. The number of positions identified for a single run $R \in \mathcal{R}_k(U)$ is therefore upper bounded by the number of distinct patterns occurring within $R \cap U$. We then report all distinct patterns occurring within $R \cap U$ by processing each such starting position using Lemma 4. There is no double-reporting while processing a single run, by Observation 13 and hence the time required to process this run is $\mathcal{O}(1 + |\text{output}|) - |\text{output}|$ here refers to the number of distinct patterns from \mathcal{D}_k^p occurring within U . Since $|\mathcal{R}_k(U)| = \mathcal{O}(1)$, we report each pattern a constant number of times and the overall time required is $\mathcal{O}(1 + |\text{output}|)$.

The space occupied by our data structure can be reduced to $\mathcal{O}(n + d)$; details can be found in the full version of the paper.

► **Theorem 14.** *REPORTDISTINCT(i, j) queries can be answered in $\mathcal{O}(\log n + |\text{output}|)$ time with a data structure of size $\mathcal{O}(n + d)$ that can be constructed in $\mathcal{O}(n \log n + d)$ time.*

5 Count(i, j) queries

We first solve an auxiliary problem and show how it can be employed to give an unsatisfactory solution to COUNT(i, j). We then refine our approach using recompression and obtain the following.

► **Theorem 15.** *COUNT(i, j) queries can be answered in $\mathcal{O}(\log^2 n / \log \log n)$ time with a data structure of size $\mathcal{O}(n + d \log n)$ that can be constructed in $\mathcal{O}(n \log n / \log \log n + d \log^{3/2} n)$ time.*

5.1 An auxiliary problem

By inter-position $i + 1/2$ we refer to a location between positions i and $i + 1$ in T . We also refer to inter-positions $1/2$ and $n + 1/2$. We consider the following auxiliary problem, in which we are given a set of inter-positions (*breakpoints*) B of P and upon query we are to compute all fragments of $T[i..j]$ that align a specific inter-position (*anchor*) β of the text with some inter-position in B .

BREAKPOINTS-ANCHOR IPM

Input: A length- n text T , its length- m substring P , and a set B of inter-positions (breakpoints) of P .

Query: $\text{COUNT}_\beta(i, j)$: the number of fragments $T[r..r+m-1]$ of $T[i..j]$ that match P such that $\beta - r + 1 \in B$ (β is an anchor).

In the 2D orthogonal range counting problem, one is to preprocess an $n \times n$ grid with $\mathcal{O}(n)$ marked points so that upon query $[x_1, y_1] \times [x_2, y_2]$, the number of points in this rectangle can be computed efficiently. In the (dual) 2D range stabbing counting problem, one is to preprocess the grid with $\mathcal{O}(n)$ rectangles so that upon query (x, y) the number of (stabbed) rectangles that contain (x, y) can be retrieved efficiently. The counting version of range stabbing queries in 2D reduces to two-sided range counting queries in 2D as follows (cf. [29]). For each rectangle $[x_1, y_1] \times [x_2, y_2]$ in grid G , we add points (x_1, y_1) and $(x_2 + 1, y_2 + 1)$ with weight 1 and points $(x_1, y_2 + 1)$ and $(x_2, y_1 + 1)$ with weight -1 in a grid G' . Then the number of rectangles stabbed by point (a, b) in G is equal to the sum of weights of points in $(-\infty, a] \times (-\infty, b]$ in G' . We will use the following result in our solution to BREAKPOINTS-ANCHOR IPM (Lemma 18).

► **Theorem 16** ([27]). *Range counting queries for n points in 2D (rank space) can be answered in time $\mathcal{O}(\log n / \log \log n)$ with a data structure of size $\mathcal{O}(n)$ that can be constructed in time $\mathcal{O}(n\sqrt{\log n})$.*

Data structure. Let $W_1 = \{P[[b]..m] : b \in B\}$ and consider the set W_2 obtained by adding $U\$$ and $U\#$ for each element U of W_1 to an initially empty set, where $\$$ is a letter smaller (resp. $\#$ is larger) than all the letters in Σ . Let W be the compact trie for the set of strings W_2 . For each internal node v of W that does not have an outgoing edge with label $\$$, we add such a (leftmost) edge with a leaf attached to its endpoint. W can be constructed in $\mathcal{O}(|B|)$ time after an $\mathcal{O}(n)$ -time preprocessing of T , allowing for constant-time longest common prefix queries; cf. [11]. We also build the W_1 -modified suffix tree of T and preprocess it for weighted ancestor queries. We keep two-sided pointers between nodes of W and of the W_1 -modified suffix tree of T that have the same path-label. Similarly, let W^R be the compact trie for set Z_2 consisting of elements $U\$$ and $U\#$ for each $U \in Z_1 = \{(P[1..[b]])^R : b \in B\}$. We preprocess W^R analogously. Each of the tries has at most $k = \mathcal{O}(|B|)$ leaves.

Let us now consider a 2D grid of size $k \times k$, whose x -coordinates (resp. y -coordinates) correspond to the leaves of W (resp. W^R). For each $b \in B$ we do the following. Let x_1 and x_2 be the leaves with path-label $P[[b]..m]\$$ and $P[[b]..m]\#$ in W , respectively. Similarly, let y_1 and y_2 be the leaves with path-label $(P[1..[b]])^R\$$ and $(P[1..[b]])^R\#$ in W^R , respectively. We add the rectangle $R_b = [x_1, y_1] \times [x_2, y_2]$ in the grid. An illustration is provided in Fig. 4. We then preprocess the grid for the counting version of 2D range stabbing queries, employing Theorem 16.

Query. Let the longest prefix of $T[[\beta]..j]$ that is a prefix of an element of W_1 be U and its locus in W be u . This can be computed in $\mathcal{O}(\log \log n)$ time using a weighted ancestor query in the W_1 -modified suffix tree of T and following the pointer to W . If u is an explicit

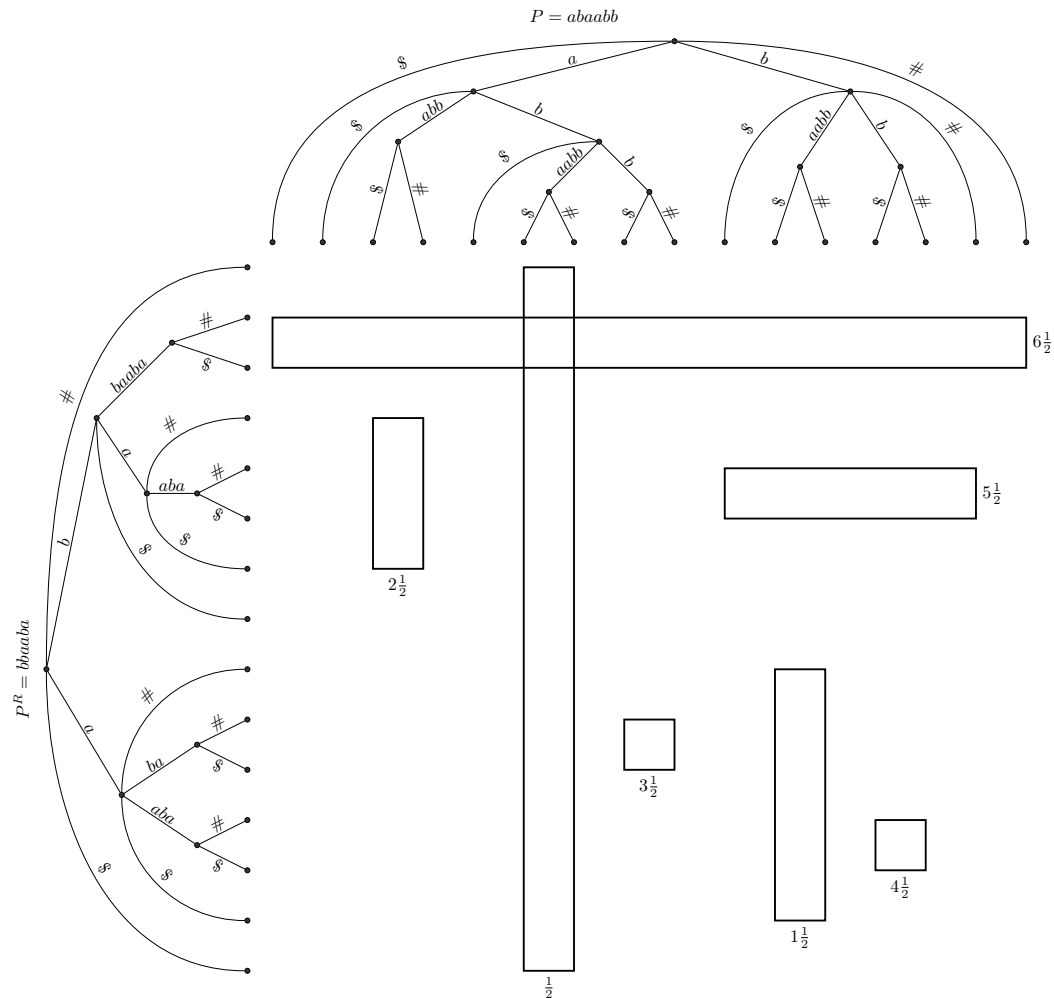
node, we follow the edge with label \$, while if it is implicit along edge (p, q) , we follow the edge with label \$ from p . In either case, we reach a leaf u' . We do the symmetric procedure with $(T[i..[\beta]])^R$ in W^R and obtain a leaf v' .

► **Observation 17.** *The number of fragments $T[r..t] = P$ with $r, t \in [i..j]$ and $\beta - r + 1 \in B$ is equal to the number of rectangles stabbed by the point of the grid defined by u' and v' .*

The observation holds because this point is inside rectangle R_b for $b \in B$ if and only if $P[[b]..m]$ is a prefix of $T[[\beta]..j]$ and $P[1..[b]]$ is a suffix of $T[i..[\beta]]$. This concludes the proof of the following result.

► **Lemma 18.** *BREAKPOINTS-ANCHOR IPM queries can be answered in $\mathcal{O}(\log n / \log \log n)$ time with a data structure of size $\mathcal{O}(n + |B|)$ that can be constructed in time $\mathcal{O}(n + |B|\sqrt{\log |B|})$.*

For the analogously defined problem BREAKPOINTS-ANCHOR IDM, we obtain the following lemma by building trie W for the union of the sets W_2 defined in the above proof for each pattern (similarly for W^R) and adding all rectangles to a single grid.



■ **Figure 4** Example of the construction of rectangles in the proof of Lemma 18 for $P = abaabb$ and breakpoints $i + 1/2$ for $i = 0, 1, 2, 3, 4, 5, 6$. Each rectangle is annotated with its breakpoint.

► **Lemma 19.** BREAKPOINTS-ANCHOR IDM queries can be answered in $\mathcal{O}(\log n / \log \log n)$ time with a data structure of size $\mathcal{O}(n + \sum_{P \in \mathcal{D}} |B_P|)$. The data structure can be constructed in time $\mathcal{O}(n + \sqrt{\log n} \sum_{P \in \mathcal{D}} |B_P|)$.

A warm-up solution for Count(i, j). Lemma 19 can be applied somewhat naively to answer COUNT(i, j) queries as follows. Let us set $B_P = \{p + 1/2 : p \in [1 \dots |P| - 1]\}$ for each pattern $P \in \mathcal{D}$ and construct the data structure of Lemma 19. We build a balanced binary tree BT on top of the text and for each node v in BT define $\text{val}(v)$ to be the fragment consisting of the characters corresponding to the leaves in the subtree of v . Note that if v is a leaf, then $|\text{val}(v)| = 1$; otherwise, $\text{val}(v) = \text{val}(u_\ell)\text{val}(u_r)$, where u_ℓ and u_r are the children of v . For each node v in BT, we precompute and store the count for $\text{val}(v)$. If v is a leaf, this count can be determined easily. Otherwise, each occurrence is contained in $\text{val}(u_\ell)$, is contained in $\text{val}(u_r)$, or spans both $\text{val}(u_\ell)$ and $\text{val}(u_r)$. Hence, we sum the answers for the children u_ℓ and u_r of v and add the result of a BREAKPOINTS-ANCHOR IDM query in $\text{val}(v)$ with the anchor between $\text{val}(u_\ell)$ and $\text{val}(u_r)$.

To answer a query concerning $T[i \dots j]$, we recursively count the occurrences in the intersection of $\text{val}(v)$ with $T[i \dots j]$, starting from the root r of BT as it satisfies $\text{val}(r) = T[1 \dots n]$. If the intersection is empty, the result is 0, and if $\text{val}(v)$ is contained in $T[i \dots j]$, we can use the precomputed count. Otherwise, we recurse on the children u_ℓ and u_r of v and sum the resulting counts. It remains to add the number of occurrences spanning across both $\text{val}(u_\ell)$ and $\text{val}(u_r)$. This value is non-zero only if $T[i \dots j]$ spans both these fragments, and it can be determined from a BREAKPOINTS-ANCHOR IDM query in the intersection of $\text{val}(v)$ and $T[i \dots j]$ with the anchor between $\text{val}(u_\ell)$ and $\text{val}(u_r)$.

The query-time is $\mathcal{O}(\log^2 n / \log \log n)$ since non-trivial recursive calls are made only for nodes on the paths from the root r to the leaves representing $T[i]$ and $T[j]$. Nevertheless, the space required for this “solution” can be $\Omega(nd)$, which is unacceptable. Below, we refine this technique using a locally consistent parsing; our goal is to decrease the size of each set B_P from $\Theta(|P|)$ to $\mathcal{O}(\log n)$.

5.2 Recompression

A *run-length straight line program* (RSLP) is a context-free grammar which generates exactly one string and contains two kinds of non-terminals: *concatenations* with production of the form $A \rightarrow BC$ (for symbols B, C) and *powers* with production of the form $A \rightarrow B^k$ (for a symbol B and an integer $k \geq 2$). Every symbol A generates a unique string denoted $\mathbf{g}(A)$.

Each symbol A is also associated with its *parse tree* $\text{PT}(A)$ consisting of a root labeled with A to which zero or more subtrees are attached: if A is a terminal, there are no subtrees; if $A \rightarrow BC$ is a concatenation symbol, then $\text{PT}(B)$ and $\text{PT}(C)$ are attached; if $A \rightarrow B^k$ is a power symbol, then k copies of $\text{PT}(B)$ are attached. Note that if we traverse the leaves of $\text{PT}(A)$ from left to right, spelling out the corresponding non-terminals, then we obtain $\mathbf{g}(A)$. The parse tree PT of the whole RSLP generating T is defined as $\text{PT}(S)$ for the starting symbol S . We define the *value* $\text{val}(v)$ of a node v in PT to be the fragment $T[a \dots b]$ corresponding to the leaves $T[a], \dots, T[b]$ in the subtree of v . Note that $\text{val}(v)$ is an occurrence of $\mathbf{g}(A)$, where A is the label of v . A sequence of nodes in PT is a *chain* if their values are consecutive fragments in T .

The *recompression* technique by Jež [20, 21] consists in the construction of a particular RSLP generating the input text T . The underlying parse tree PT is of depth $\mathcal{O}(\log n)$ and it can be constructed in $\mathcal{O}(n)$ time. As observed by I [19], this parse tree PT is *locally consistent* in a certain sense. To formalize this property, he introduced the *popped sequence* of every fragment $T[a \dots b]$, which is a sequence of symbols labelling a certain chain of nodes whose values constitute $T[a \dots b]$.

► **Theorem 20** ([19]). *If two fragments are equal, then their popped sequences are equal. Moreover, each popped sequence consists of $\mathcal{O}(\log n)$ runs (maximal powers of a single symbol) and can be constructed in $\mathcal{O}(\log n)$ time. The nodes corresponding to symbols in a run share a single parent. Furthermore, the popped sequence consists of a single symbol only for fragments of length 1.*

Let $F_1^{p_1} \dots F_t^{p_t}$ be the run-length encoding of the popped sequence of a substring S of T . We define

$$L(S) = \{|\mathbf{g}(F_1)|, |\mathbf{g}(F_1^{p_1})|, |\mathbf{g}(F_1^{p_1} F_2^{p_2})|, \dots, |\mathbf{g}(F_1^{p_1} \dots F_{t-1}^{p_{t-1}})|, |\mathbf{g}(F_1^{p_1} \dots F_{t-1}^{p_{t-1}} F_t^{p_t})|\}.$$

By Theorem 20, the set $L(S)$ can be constructed in $\mathcal{O}(\log n)$ time given the occurrence $T[a..b] = S$.

► **Lemma 21.** *Let v be a non-leaf node of PT and let $T[a..b]$ be an occurrence of S contained in $\text{val}(v)$, but not contained in $\text{val}(u)$ for any child u of v . If $T[a..c]$ is the longest prefix of $T[a..b]$ contained in $\text{val}(u)$ for a child u of v , then $|T[a..c]| \in L(S)$. Symmetrically, if $T[c'+1..b]$ is the longest suffix of $T[a..b]$ contained in $\text{val}(u)$ for a child u of v , then $|T[a..c']| \in L(S)$.*

Proof. Consider a sequence v_1, \dots, v_p of nodes in the chain corresponding to the popped sequence of $S = T[a..b]$. Each of these nodes is a descendant of a child of v . Note that $T[a..c] = \text{val}(v_1) \dots \text{val}(v_q)$, where v_1, \dots, v_q is the longest prefix consisting of descendants of the same child. If the labels of v_q and v_{q+1} are distinct, then they belong to distinct runs and $|T[a..c]| \in L(S)$. Otherwise, v_q and v_{q+1} share the same parent: v . Thus, $q = 1$ and $|T[a..c]| = |\text{val}(v_1)| \in L(S)$. The proof of the second claim is symmetric. ◀

Data Structure. We use recompression to build an RSLP generating T and the underlying parse tree PT. We also construct the component of Lemma 19 with $B_P = \{i + \frac{1}{2} : i \in L(P)\}$ for each pattern $P \in \mathcal{D}$. Moreover, for every symbol A we store the number of occurrences of patterns from \mathcal{D} in $\mathbf{g}(A)$. Additionally, if $A \rightarrow B^k$ is a power, we also store the number of occurrences in $\mathbf{g}(B^i)$ for $i \in [1..k]$. The space consumption is $\mathcal{O}(n + d \log n)$ since $|B_P| = \mathcal{O}(\log n)$ for each $P \in \mathcal{D}$.

Efficient preprocessing. The RSLP and the parse tree are built in $\mathcal{O}(n)$ time, and the sets B_P are determined in $\mathcal{O}(d \log n)$ time using Theorem 20. The data structure of Lemma 19 is then constructed in $\mathcal{O}(n + d \log^{3/2} n)$ time. Next, we process the RSLP in a bottom-up fashion. If A is a terminal, its count is easily determined. If $A \rightarrow BC$ is a concatenation, we sum the counts for B and C and the number of occurrences spanning both $\mathbf{g}(B)$ and $\mathbf{g}(C)$. To determine the latter value, we fix an arbitrary node v with label A and denote its children u_ℓ, u_r . By Lemma 21, any occurrence of P intersecting both $\text{val}(u_\ell)$ and $\text{val}(u_r)$ has a breakpoint aligned to the inter-position between the two fragments. Hence, the third summand is the result of a BREAKPOINTS-ANCHOR IDM query in $\text{val}(v)$ with the anchor between $\text{val}(u_\ell)$ and $\text{val}(u_r)$. Finally, if $A \rightarrow B^k$, then to determine the count in $\mathbf{g}(B^i)$, we add the count for B , the count in $\mathbf{g}(B^{i-1})$, and the number of occurrences in B^i spanning both the prefix B and the suffix B^{i-1} . To find the latter value, we fix an arbitrary node v with label A , denote its children u_1, \dots, u_k , and make a BREAKPOINTS-ANCHOR IDM query in $\text{val}(u_1) \dots \text{val}(u_i)$ with the anchor between $\text{val}(u_1)$ and $\text{val}(u_2)$. The correctness of this step follows from Lemma 21. The running time of the last phase is $\mathcal{O}(n \log n / \log \log n)$, so the overall construction time is $\mathcal{O}(n \log n / \log \log n + d \log^{3/2} n)$.

Query. Upon a query $\text{COUNT}(i, j)$, we proceed essentially as in the warm-up solution: we recursively count the occurrences contained in the intersection of $T[i..j]$ with $\text{val}(v)$ for nodes v in PT , starting from the root of PT . If the two fragments are disjoint, the result is 0, and if $\text{val}(v)$ is contained in $T[i..j]$, it is the count precomputed for the label of v . Otherwise, the label of v is a non-terminal. If it is a concatenation symbol, we recurse on both children u_ℓ, u_r of v and sum the obtained counts. If $T[i..j]$ spans both $\text{val}(u_\ell)$ and $\text{val}(u_r)$, we also add the result of a $\text{BREAKPOINTS-ANCHOR IDM}$ query in the intersection of $T[i..j]$ with $\text{val}(v)$ and the anchor between $\text{val}(u_\ell)$ and $\text{val}(u_r)$. If the label is a power symbol $A \rightarrow B^k$, we determine which of the children u_1, \dots, u_k of v are spanned by $T[i..j]$. We denote these children by u_ℓ, \dots, u_r and recurse on u_ℓ and on u_r . If $r > \ell$, we also make a $\text{BREAKPOINTS-ANCHOR IDM}$ query in the intersection of $T[i..j]$ with $\text{val}(u_\ell) \cdots \text{val}(u_r)$ and anchor between $\text{val}(u_\ell)$ and $\text{val}(u_{\ell+1})$. If $r > \ell + 1$, we further add the precomputed value for $\mathbf{g}(B^{r-\ell-1})$ to account for the occurrences contained in $\text{val}(u_{\ell+1}) \cdots \text{val}(u_{r-1})$ and make a $\text{BREAKPOINTS-ANCHOR IDM}$ query in the intersection of $T[i..j]$ with $\text{val}(u_{\ell+1}) \cdots \text{val}(u_r)$ and anchor between u_{r-1} and u_r . By Lemma 21, the answer is the sum of the up to five values computed. The overall query time is $\mathcal{O}(\log^2 n / \log \log n)$, since we make $\mathcal{O}(\log n)$ non-trivial recursive calls and each of them is processed in $\mathcal{O}(\log n / \log \log n)$ time.

6 Dynamic dictionaries

In the Online Boolean Matrix-Vector Multiplication (OMv) problem, we are given as input an $n \times n$ boolean matrix M . Then, we are given in an online fashion a sequence of n vectors r_1, \dots, r_n , each of size n . For each such vector r_i , we are required to output Mr_i before receiving r_{i+1} .

► **Conjecture 22** (OMv Conjecture [18]). *For any constant $\epsilon > 0$, there is no $\mathcal{O}(n^{3-\epsilon})$ -time algorithm that solves OMv correctly with probability at least $2/3$.*

We now present a restricted, but sufficient for our purposes, version of [18, Theorem 2.2].

► **Theorem 23** ([18]). *Conjecture 22 implies that there is no algorithm, for a fixed $\gamma > 0$, that given as input an $r_1 \times r_2$ matrix M , with $r_1 = \lfloor r_2^\gamma \rfloor$, preprocesses M in time polynomial in $r_1 + r_2$ and, then, presented with a vector v of size r_2 , computes Mv in time $\mathcal{O}(r_2^{1+\gamma-\epsilon})$ for $\epsilon > 0$, and has error probability of at most $1/3$.*

We proceed to obtain a conditional lower bound for IDM in the case of a dynamic dictionary. This lower bound clearly carries over to the other problems we considered.

► **Theorem 24.** *The OMv conjecture implies that there is no algorithm that preprocesses T and \mathcal{D} in time polynomial in n , performs insertions to \mathcal{D} in time $\mathcal{O}(n^\alpha)$, answers $\text{EXISTS}(i, j)$ queries in time $\mathcal{O}(n^\beta)$, in an online manner, such that $\alpha + \beta = 1 - \epsilon$ for $\epsilon > 0$, and has error probability of at most $1/3$.*

Proof. Let us suppose that there is such an algorithm and set $\gamma = (\alpha + \epsilon/2)/(\beta + \epsilon/2)$. Given an $r_1 \times r_2$ matrix M , satisfying $r_1 = \lfloor r_2^\gamma \rfloor$, we construct a text T of length $n = r_1 r_2$ as follows. Let T' be a text created by concatenating the rows of M in increasing order. Then obtain T by assigning to each non-zero element of T' the column index of the matrix entry it originates from. Formally, for $i \in [1..r_1 r_2]$, let $a[i] = \lfloor (i-1)/r_2 \rfloor$ and $b[i] = i - a[i]r_2$ and set $T[i] = b[i] \cdot M[a[i] + 1, b[i]]$.

We compute Mv as follows. We add the indices of its at most r_2 non-zero entries in an initially empty dictionary. We then perform queries $\text{EXISTS}(1 + tr_2, (t+1)r_2)$ for $t = 0, \dots, r_1 - 1$. The answer to query $\text{EXISTS}(1 + tr_2, (t+1)r_2)$ is equal to the product of the

t th row of M with v . We thus obtain Mv . In total we perform $\mathcal{O}(r_2)$ insertions to \mathcal{D} and $\mathcal{O}(r_1)$ EXISTS queries. Thus, the total time required is $\mathcal{O}(r_2 n^\alpha + r_1 n^\beta) = \mathcal{O}(n^{\beta+\epsilon/2} n^\alpha + n^{\alpha+\epsilon/2} n^\beta) = \mathcal{O}(n^{1-\epsilon/2}) = \mathcal{O}(r_2^{1+\gamma-\epsilon'})$ for $\epsilon' > 0$. Conjecture 22 would be disproved due to Theorem 23. ◀

► **Example 25.** For the matrix

$$M = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

we construct the text $T = 103000340204$. For the vector $v = [1 \ 1 \ 0 \ 0]^T$, the dictionary is $\mathcal{D} = \{1, 2\}$. The answers to EXISTS(1, 4), EXISTS(5, 8), EXISTS(9, 12) are Yes, No, Yes, respectively, which corresponds to $Mv = [1 \ 0 \ 1]^T$.

A proof of the following theorem, in which we provide algorithms that essentially match this lower bound, can be found in the full version of the paper.

► **Theorem 26.** EXISTS(i, j), REPORT(i, j), REPORTDISTINCT(i, j), and COUNT(i, j) queries for a dynamic dictionary can be answered in $\tilde{\mathcal{O}}(m + |\text{output}|)$ time per query and $\tilde{\mathcal{O}}(n/m)$ time per update for any parameter $m \in [1..n]$ using $\tilde{\mathcal{O}}(n + d)$ space.

7 Final Remarks

The question of whether queries of the type COUNTDISTINCT(i, j), which ask for the number c of patterns from \mathcal{D} that occur in $T[i..j]$, can be answered in time $o(\min\{c, |j-i|\})$ or even $\tilde{\mathcal{O}}(1)$ with a data structure of size $\tilde{\mathcal{O}}(n + d)$ is left open for further investigation. It turns out that our techniques can be used to efficiently answer such queries $\mathcal{O}(\log n)$ -approximately; details are provided in the full version of the paper.

References

- 1 Alfred V. Aho and Margaret J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340, 1975. doi:10.1145/360825.360855.
- 2 Amihood Amir, Martin Farach, Zvi Galil, Raffaele Giancarlo, and Kunsoo Park. Dynamic Dictionary Matching. *Journal of Computer and System Sciences*, 49(2):208–222, 1994. doi:10.1016/S0022-0000(05)80047-9.
- 3 Amihood Amir, Martin Farach, Ramana M. Idury, Johannes A. La Poutré, and Alejandro A. Schäffer. Improved Dynamic Dictionary Matching. *Information and Computation*, 119(2):258–282, 1995. doi:10.1006/inco.1995.1090.
- 4 Amihood Amir, Gad M. Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *ACM Transactions on Algorithms*, 3(2):19, 2007. doi:10.1145/1240233.1240242.
- 5 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The “Runs” Theorem. *SIAM Journal on Computing*, 46(5):1501–1514, 2017. doi:10.1137/15M1011032.
- 6 Hideo Bannai, Shunsuke Inenaga, and Dominik Köppl. Computing All Distinct Squares in Linear Time for Integer Alphabets. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, volume 78 of *LIPICs*, pages 22:1–22:18. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.CPM.2017.22.
- 7 Michael A. Bender and Martin Farach-Colton. The Level Ancestor Problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004. doi:10.1016/j.tcs.2003.05.002.

- 8 Michael A. Bender, Martin Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005. doi:10.1016/j.jalgor.2005.08.001.
- 9 Ho-Leung Chan, Wing-Kai Hon, Tak Wah Lam, and Kunihiko Sadakane. Dynamic dictionary matching and compressed suffix trees. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005*, pages 13–22. SIAM, 2005. URL: <http://dl.acm.org/citation.cfm?id=1070432.1070436>.
- 10 Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In László Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, STOC 2004*, pages 91–100. ACM, 2004. doi:10.1145/1007352.1007374.
- 11 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007. doi:10.1017/cbo9780511546853.
- 12 Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Extracting powers and periods in a word from its runs structure. *Theoretical Computer Science*, 521:29–41, 2014. doi:10.1016/j.tcs.2013.11.018.
- 13 Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the Sorting-complexity of Suffix Tree Construction. *Journal of the ACM*, 47(6):987–1011, November 2000. doi:10.1145/355541.355547.
- 14 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a Sparse Table with $O(1)$ Worst Case Access Time. *Journal of the ACM*, 31(3):538–544, 1984. doi:10.1145/828.1884.
- 15 Harold N. Gabow and Robert Endre Tarjan. A Linear-Time Algorithm for a Special Case of Disjoint Set Union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985. doi:10.1016/0022-0000(85)90014-5.
- 16 Richard Groult, Élise Prieur, and Gwénaél Richomme. Counting distinct palindromes in a word in linear time. *Information Processing Letters*, 110(20):908–912, 2010. doi:10.1016/j.ipl.2010.07.018.
- 17 Dov Harel and Robert Endre Tarjan. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984. doi:10.1137/0213024.
- 18 Monika Henzinger, Sebastian Krininger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015*, pages 21–30. ACM, 2015. doi:10.1145/2746539.2746609.
- 19 Tomohiro I. Longest Common Extensions with Recompression. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, volume 78 of *LIPICs*, pages 18:1–18:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.CPM.2017.18.
- 20 Artur Jeż. Faster Fully Compressed Pattern Matching by Recompression. *ACM Transactions on Algorithms*, 11(3):20:1–20:43, 2015. doi:10.1145/2631920.
- 21 Artur Jeż. Recompression: A Simple and Powerful Technique for Word Equations. *Journal of the ACM*, 63(1):4:1–4:51, 2016. doi:10.1145/2743014.
- 22 Orgad Keller, Tsvi Kopelowitz, Shir Landau Feibish, and Moshe Lewenstein. Generalized substring compression. *Theoretical Computer Science*, 525:42–54, 2014. doi:10.1016/j.tcs.2013.10.010.
- 23 Tomasz Kociumaka. *Efficient Data Structures for Internal Queries in Texts*. PhD thesis, University of Warsaw, 2018. URL: <https://mimuw.edu.pl/~kociumaka/files/phd.pdf>.
- 24 Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A Linear Time Algorithm for Seeds Computation, 2019. arXiv:1107.2422.
- 25 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal Pattern Matching Queries in a Text and Applications. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 532–551. SIAM, 2015. doi:10.1137/1.9781611973730.36.

- 26 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 1999*, pages 596–604. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814634.
- 27 J. Ian Munro, Yakov Nekrich, and Jeffrey Scott Vitter. Fast construction of wavelet trees. *Theoretical Computer Science*, 638:91–97, 2016. doi:10.1016/j.tcs.2015.11.011.
- 28 S. Muthukrishnan. Efficient algorithms for document retrieval problems. In David Eppstein, editor, *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2002*, pages 657–666. SIAM, 2002. URL: <http://dl.acm.org/citation.cfm?id=545381.545469>.
- 29 Mihai Pătraşcu. Unifying the Landscape of Cell-Probe Lower Bounds. *SIAM Journal on Computing*, 40(3):827–847, 2011. doi:10.1137/09075336X.
- 30 Mikhail Rubinchik and Arseny M. Shur. Counting Palindromes in Substrings. In Gabriele Fici, Marinella Sciortino, and Rossano Venturini, editors, *String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017*, volume 10508 of *Lecture Notes in Computer Science*, pages 290–303. Springer, 2017. doi:10.1007/978-3-319-67428-5_25.