

Internal Dictionary Matching

Panagiotis Charalampopoulos¹, Tomasz Kociumaka^{2,3}, Manal Mohamed¹,
Jakub Radoszewski², Wojciech Rytter², and Tomasz Walen²

¹Department of Informatics, King’s College London, London, UK,
[panagiotis.charalampopoulos,manal.mohamed]@kcl.ac.uk

²Institute of Informatics, University of Warsaw, Warsaw, Poland,
[kociumaka,jrad,rytter,walen]@mimuw.edu.pl

³Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel

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 $T[i..j]$ and reporting *distinct* patterns from \mathcal{D} that occur in $T[i..j]$. 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 $\mathcal{O}(\log^{\mathcal{O}(1)}n + |\text{output}|)$.

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.

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 [10], as well as indexes for approximate dictionary matching [12] 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 [28], 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 [31]. 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 [29], 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,

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

Table 1: Our results.

Our techniques and a roadmap. First, in Section 3, we present straightforward solutions for queries EXISTS(i, j) and REPORT(i, j). In Section 4 we describe an involved solution for 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 COUNT(i, j) queries. In Section 6 we extend our solutions for the case of a dynamic dictionary and provide a matching conditional lower bound. Finally, in Appendix A we consider yet another type of queries, COUNTDISTINCT(i, j), and develop an approximate solution for them.

2 Preliminaries

We begin with basic definitions and notation generally following [13]. 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 $U = T[i..i+m-1]$.

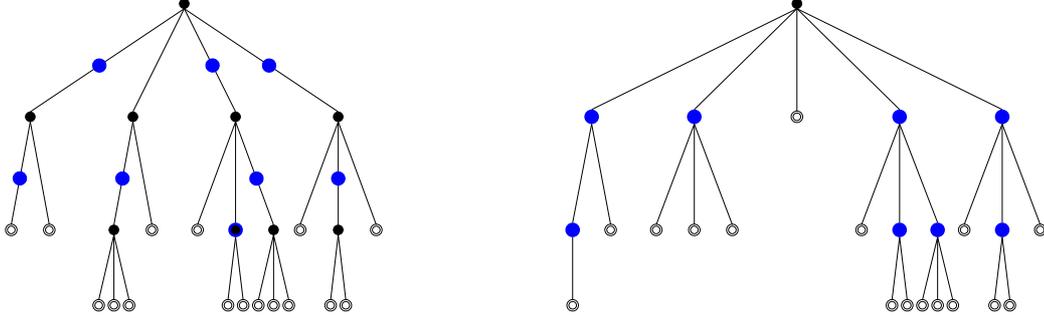
If a string U is both a proper prefix and a proper suffix of a string T of length n , then U is called a *border* of T . A positive integer p is called a *period* of T if $T[i] = T[i+p]$ for all $i = 1, \dots, n-p$. A string T has a period p if and only if it has a border of length $n-p$. We refer to 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 $\mathcal{O}(n)$ [15]. In the case of integer alphabets, in order to access the child

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). \square



(a) Seven implicit nodes of $\mathcal{T}(T)$ are made explicit. Each marked node (big circles in blue) represents a pattern $P \in \mathcal{D}$.

(b) \mathcal{D} -modified suffix tree.

Figure 3: The two-step construction of the \mathcal{D} -modified suffix tree.

We state the following simple lemma.

Lemma 3.3. *With the \mathcal{D} -modified suffix tree of T at hand, given positions a, j in T with $a \leq j$, we can compute all $P \in \mathcal{D}$ that occur at position a and are of length at most $j - a + 1$ in time $\mathcal{O}(1 + |\text{output}|)$.*

Proof. We start from the root of the \mathcal{D} -modified suffix tree and go down towards the terminal node with path-label $T_{(a)}$. We report all encountered nodes v as long as $\delta(v) \leq j - a + 1$ is satisfied. We stop when this inequality is not satisfied. \square

The \mathcal{D} -modified suffix tree enables us to answer $\text{EXISTS}(i, j)$ and $\text{REPORT}(i, j)$ queries.

Theorem 3.4.

- (a) $\text{EXISTS}(i, j)$ queries can be answered in $\mathcal{O}(1)$ time with a data structure of size $\mathcal{O}(n)$ that can be constructed in $\mathcal{O}(n + d)$ time.
- (b) $\text{REPORT}(i, j)$ queries can be answered in $\mathcal{O}(1 + |\text{output}|)$ time with a data structure of size $\mathcal{O}(n + d)$ that can be constructed in $\mathcal{O}(n + d)$ time.

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 $\mathcal{O}(n)$ [21, 9].

(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 3.2 and 3.3. \square

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 $T[a..a + L_k[a]]$ integer identifiers id (or colors) in $[1..n]$, and construct an array $I_k[a] = \text{id}(P)$, where $P = T[a..a + L_k[a]]$. We then rely on the following theorem.

Theorem 4.1 (Colored Range Reporting [36]). *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 4.1. We observe the following.

Observation 4.2. *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 *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 [31, 6, 29]. 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 aperiodic *k-dictionary*

We make use of the following sparsity property.

Fact 4.3 (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 $2^k/3 \geq \frac{1}{6}|P|$. \square

Lemma 4.4. *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 4.3. 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 3.4(b). \square

4.2 Processing 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 [32, 6].

Observation 4.5. *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 [29], 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 4.6 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 4.6 (see [29, 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 4.6. 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 4.7. *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 [29, 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 $|V| \geq 2 \cdot \text{per}(R)$. \square

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}_k^p -modified suffix tree (asking for a level-1 ancestor of the leaf corresponding to $T_{(i)}$, as in the proof of Theorem 3.4(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 4.8. *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 3.4(b), for the subarray of ℓ_k corresponding to the first $\text{per}(R)$ positions of $R \cap U$. This way, due to Observation 4.8, 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 3.3. There is no double-reporting while processing a single run, by Observation 4.8 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 $\mathcal{O}(1 + |\text{output}|)$.

4.3 Reducing the space

The space occupied by our data structure can be reduced to $\mathcal{O}(n + d)$. We store the \mathcal{D} -modified suffix tree and mark all nodes from each \mathcal{D}_k^i , for $k \in [0.. \lfloor \log n \rfloor]$ and $i \in \{a, p\}$, with a different color. For each dictionary $\mathcal{D}' = \mathcal{D}_k^i$, we further store, using $\mathcal{O}(|\mathcal{D}'|)$ space, the \mathcal{D}' -modified suffix tree without its unmarked leaves.

We will show below that the only additional operation we now need to support is determining the parent of a given unmarked leaf in the original \mathcal{D}' -modified suffix tree (before the leaves were chopped). This can be done using the nearest colored ancestor data structure of [19] over the \mathcal{D} -modified suffix

tree. For a tree of size N , it achieves $\mathcal{O}(\log \log N)$ time per query after $\mathcal{O}(N)$ -time preprocessing. We can, however, exploit the fact that we only have $\text{colors} = \mathcal{O}(\log n)$ colors to obtain constant-time queries within the same construction time.

In [19] it is shown that, in order to answer nearest colored ancestor queries in a tree with N nodes, it is enough to store some arrays of total size $\mathcal{O}(N)$ and predecessor data structures for $\mathcal{O}(\text{colors})$ subsets of $[1..2N]$ whose total size is $\mathcal{O}(N)$. The time needed to compute the sets for the predecessor data structures and the arrays is $\mathcal{O}(N)$. The time complexity of the query is proportional to the time required for answering a constant number of predecessor queries over the aforementioned sets. We implement a predecessor data structure for a set $S \subseteq [1..2N]$ using $\mathcal{O}(N)$ bits of space as follows. We store a bitmap that has the i th bit set if and only if $i \in S$ and augment it with a data structure that answers **rank** and **select** queries in $\mathcal{O}(1)$ time and requires $o(N)$ additional bits of space [25, 11]. Such a component can be constructed in $\mathcal{O}(N/\log N)$ time [5, 35]. Note that $\text{pred}_S(i) = \text{select}(\text{rank}(i))$. We thus use $\mathcal{O}((n+d)\log n)$ bits, i.e., $\mathcal{O}(n+d)$ machine words, in total for the part of the data structure responsible for reporting occurrences starting at given positions.

It was shown in [16] that we can implement an $\mathcal{O}(1)$ -query-time RMQ data structure for an array of size N using $\mathcal{O}(N)$ bits. This data structure only returns the index of the minimum value in the given range.

For colored range reporting, the main component of the data structure underlying Theorem 4.1 from [36] is an RMQ data structure over array $J[i] = \max\{j : j < i, A[i] = A[j]\}$. We build an $\mathcal{O}(N)$ -bits RMQ data structure over J . The query procedure however, needs access to A , i.e. the colors. We can retrieve the value of the i -th element in our array of colors using our representation of the \mathcal{D}' -modified suffix tree, since the its color corresponds to the respective leaf in the \mathcal{D}' -modified suffix tree or, if it is unmarked, to that of its parent.

Then, filtering $\mathcal{O}(|\text{output}|)$ starting positions in the periodic case, is based on RMQ queries over multiple arrays of total length $\mathcal{O}(n \log n)$. To construct them, we build the \mathcal{D}' -modified suffix trees one by one and build the relevant RMQ data structures that require $\mathcal{O}(n \log n)$ bits in total before chopping the unmarked leaves. The actual value at the indices returned by RMQ queries can, as above, be determined using our representation of the \mathcal{D}' -modified suffix tree.

We arrive at the main result of this section.

Theorem 4.9. *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 5.1. *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. [37]). 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 IDM (Lemma 5.4).

Theorem 5.2 ([35]). *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. [13]. 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 5.2.

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 5.3. *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 5.4. BREAKPOINTS-ANCHOR IPM queries can be answered in time $\mathcal{O}(\log n / \log \log n)$ 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.

Lemma 5.5. BREAKPOINTS-ANCHOR IDM queries can be answered in time $\mathcal{O}(\log n / \log \log n)$ 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 5.5 can be applied naively to answer COUNT(i, j) queries as follows. Let us set $B_P = \{p + 1/2 : p \in [1..|P| - 1]\}$ for each pattern $P \in \mathcal{D}$ and construct the data structure of Lemma 5.5. 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

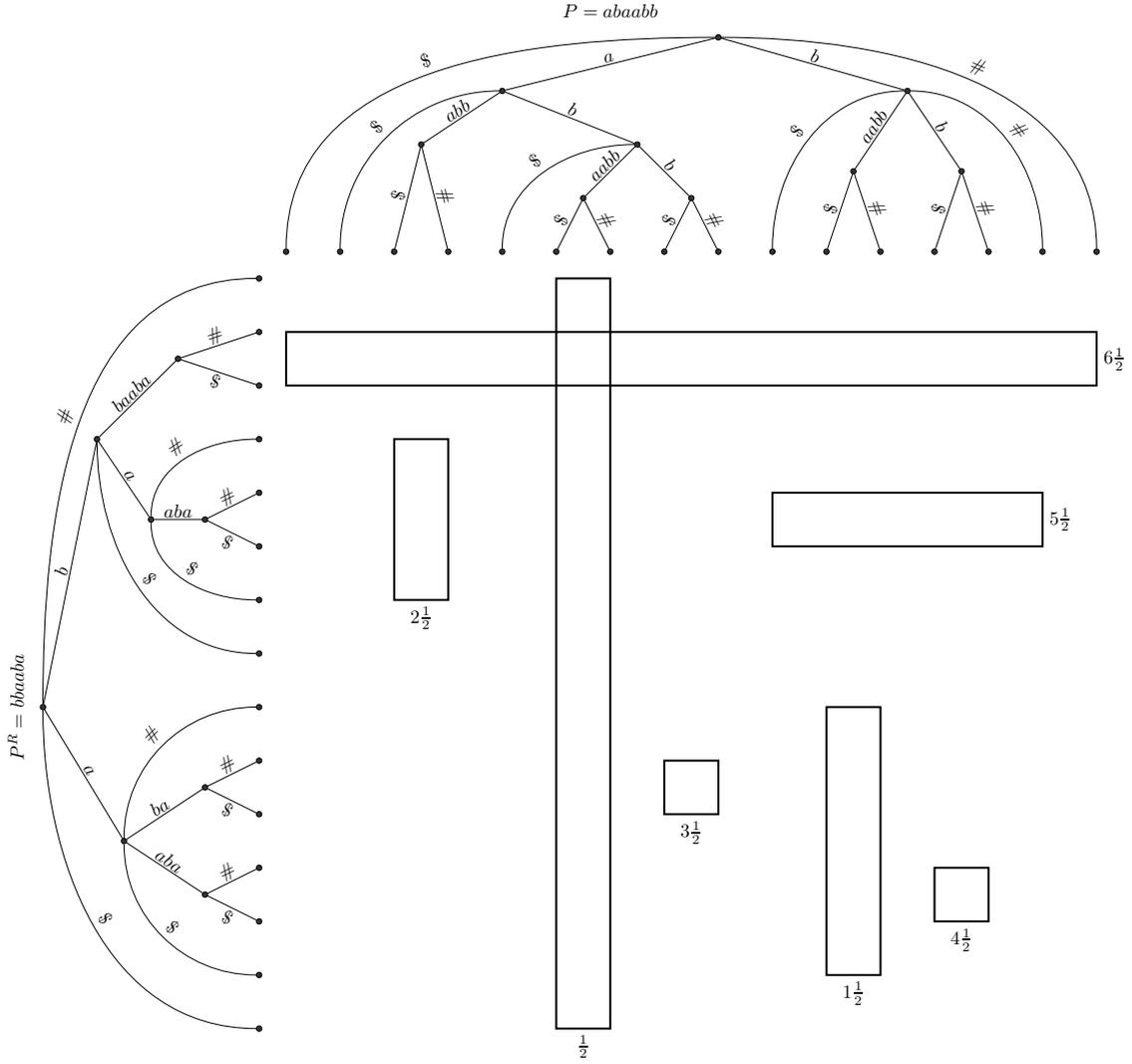


Figure 4: Example of the construction of rectangles in the proof of Lemma 5.4 for $P = \text{abaabb}$ and breakpoints $i + 1/2$ for $i = 0, 1, 2, 3, 4, 5, 6$. Each rectangle is annotated with its breakpoint.

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 $T[i..j]$, we recursively count the occurrences in the intersection of $\text{val}(v)$ with $T[i..j]$, starting from the root r of BT as it satisfies $\text{val}(r) = T[1..n]$. If the intersection is empty, the result is 0, and if $\text{val}(v)$ is contained in $T[i..j]$, 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 $T[i..j]$ spans both these fragments, and it can be determined from a BREAKPOINTS-ANCHOR IDM query in the intersection of $\text{val}(v)$ and $T[i..j]$ 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 $T[a..b]$ 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 [26, 27] 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 [24], this parse tree PT is *locally consistent* in a certain sense. To formalize this property, he introduced the *popped sequence* of every fragment $T[a..b]$, which is a sequence of symbols labelling a certain chain of nodes whose values constitute $T[a..b]$.

Theorem 5.6 ([24]). *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

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

By Theorem 5.6, the set $L(S)$ can be constructed in $\mathcal{O}(\log n)$ time given the occurrence $T[a..b] = S$.

Lemma 5.7. *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. \square

Data Structure. We use recompression to build an RSLP generating T and the underlying parse tree PT . We also construct the component of Lemma 5.5 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 5.6. The data structure of Lemma 5.5 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 5.7, 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 $\text{val}(u_1) \cdots \text{val}(u_i)$ with the anchor between $\text{val}(u_1)$ and $\text{val}(u_2)$. The correctness of this step follows from Lemma 5.7. 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 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 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 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 5.7, 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 6.1 (OMv Conjecture [23]). 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 [23, Theorem 2.2].

Theorem 6.2 ([23]). *Conjecture 6.1 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 6.3. *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 6.1 would be disproved due to Theorem 6.2. \square

Example 6.4. 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 $\text{EXISTS}(1, 4)$, $\text{EXISTS}(5, 8)$, $\text{EXISTS}(9, 12)$ are Yes, No, Yes, respectively, which corresponds to $Mv = [1 \ 0 \ 1]^T$.

In the remainder of this section we focus on providing algorithms matching this lower bound.

We first summarize what is known for different kinds of internal pattern matching queries, in which at query time we are given a fragment $T[i..j]$ and a substring P of the text and ask queries analogous to those that we have defined for internal dictionary matching. Note that we assume that the pattern P is a substring of T and is given by one of its occurrences. We answer $\text{COUNT}(P, i, j)$ queries as follows, similar to [33]. We construct the suffix tree $\mathcal{T}(T)$ and preprocess it so that each node stores the lexicographic range of suffixes of which its path-label is a prefix. We also construct a 2D orthogonal range counting data structure over an $n \times n$ grid \mathcal{G} , in which, for each suffix $T[a..n]$, we insert a point (a, b) , where b is the lexicographic rank of this suffix among all suffixes. We answer $\text{COUNT}(P, i, j)$ as follows. We first locate the locus of P in $\mathcal{T}(T)$ using a weighted ancestor query in $\tilde{O}(1)$ time and retrieve the associated lexicographic range $[l, r]$. Next, we perform a counting query for the range $[i, j - |P| + 1] \times [l, r]$ of \mathcal{G} , which returns the desired count; see also [34].

Theorem 6.5 ([28, 33]). *$\text{EXISTS}(P, i, j)$, $\text{REPORT}(P, i, j)$ and $\text{COUNT}(P, i, j)$ queries can be answered in time $\tilde{O}(1 + |\text{output}|)$ with an $\tilde{O}(n)$ -sized data structure that can be constructed in $\tilde{O}(n)$ time.*

Let us denote the dictionary we start with by \mathcal{D}^0 . Further, let u_1, u_2, \dots be the sequence of dictionary updates and \mathcal{D}^r be the dictionary after update u_r . Each update is an insertion or a deletion of a pattern in \mathcal{D} . We first discuss how to answer REPORTDISTINCT queries.

ReportDistinct(i, j). We maintain the invariant that after update u_t we have access to the static data structure of Section 4 for answering REPORTDISTINCT queries in T with respect to dictionary \mathcal{D}^r , for some $r = t - \mathcal{O}(m)$; note that m will depend on $n + d$. This can be achieved by rebuilding the data structure of Section 4 every m updates in $\tilde{O}(n + d)$ time, which amortizes to $\tilde{O}((n + d)/m)$ time per update. (The time complexity can be made worst-case by application of the standard time-slicing technique.) We also store updates u_{r+1}, \dots, u_t (or the differences between \mathcal{D}^r and \mathcal{D}^t).

To answer a REPORTDISTINCT query, we do the following:

1. use the static data structure to answer the REPORTDISTINCT query for \mathcal{D}^r ;
2. filter out the $\mathcal{O}(m)$ reported patterns that are in $\mathcal{D}^r \setminus \mathcal{D}^t$;
3. search for the $\mathcal{O}(m)$ patterns in $\mathcal{D}^t \setminus \mathcal{D}^r$ individually in $\tilde{O}(1)$ time per pattern by performing internal pattern matching queries, employing Theorem 6.5.

Each query thus requires time $\tilde{O}(m + |\text{output}|)$. We arrive at the following proposition.

Proposition 6.6. *The $\text{REPORTDISTINCT}(i, j)$ queries for a dynamic dictionary can be answered in $\tilde{O}(m + |\text{output}|)$ time per query and $\tilde{O}((n + d)/m)$ time per update for any $m \in [1..n + d]$ using $\tilde{O}(n + d)$ space.*

We next show how to attain $\tilde{O}(n^\alpha)$ update time and $\tilde{O}(n^{1-\alpha} + |\text{output}|)$ query time for any $0 < \alpha < 1$. In other words, we show how to avoid the direct dependency on the size of the dictionary.

We store \mathcal{D} as an array D of collections so that a pattern $P \in \mathcal{D}$ is stored in $D[p]$, where p is the starting position of the leftmost occurrence of P in T . We can find the desired position p for a pattern P in $\mathcal{O}(\log \log n)$ time by locating its locus on $\mathcal{T}(T)$ using a weighted ancestor query; we can have precomputed the leftmost occurrence of the path-label of each explicit node in a DFS traversal of $\mathcal{T}(T)$. Let each collection store its elements in a min heap with respect to their lengths.

The dictionary $\mathcal{D}' = \{\min D[p] : 1 \leq p \leq n\}$ is of size $\mathcal{O}(n)$. An insertion in \mathcal{D} corresponds to a possible insertion followed by a possible deletion in \mathcal{D}' , while a deletion in \mathcal{D} corresponds to a possible deletion in \mathcal{D}' , followed by an insertion if the collection in D where the deletion occurs is non-empty. We

observe that if some $P \in \mathcal{D} \setminus \mathcal{D}'$ occurs in $T[i..j]$, then the minimum element in the collection in which P belongs also occurs in $T[i..j]$.

We thus use the solution for $\text{REPORTDISTINCT}(i, j)$ from Proposition 6.6 for \mathcal{D}' . We then iterate over the elements of each collection from which an element has been reported in the order of increasing length, while they occur in $T[i..j]$; we check whether this is the case using Theorem 6.5.

Report(i, j). We first perform a $\text{REPORTDISTINCT}(i, j)$ query and then find all occurrences of each returned pattern in $T[i..j]$ in time $\tilde{O}(1 + |\text{output}|)$ by Theorem 6.5.

Exists(i, j). We again use \mathcal{D}' . We first use the static version of $\text{COUNT}(i, j)$, presented in Section 5 and then the counting version of internal pattern matching for removed/added patterns using Theorem 6.5, incrementing/decrementing the counter appropriately. We rebuild the data structure every m updates.

Count(i, j). We first build the data structure of Section 5 for $\text{COUNT}(i, j)$ queries for dictionary \mathcal{D}^0 . For the subsequent m updates, we answer $\text{COUNT}(i, j)$ queries, using this data structure and treating individually the added/removed patterns using Theorem 6.5. These queries are thus answered in $\tilde{O}(m)$ time.

After every m updates, we update our data structure to refer to the current dictionary as follows. (We focus on \mathcal{D}^0 and \mathcal{D}^m for notational simplicity.) We update the counts of occurrences for all nodes of PT by computing the counts for the set of added and the set of removed patterns in $\tilde{O}(n)$ time and updating the previously stored counts accordingly.

As for $\text{BREAKPOINTS-ANCHOR IDM}$, we also have to do something smarter than simply recompute the whole data structure from scratch, as we do not want to spend $\Omega(d)$ time. At preprocessing, we set our grid \mathcal{G} to be of size $K \times K$ for $K = \mathcal{O}(n^2)$ and identify x -coordinate i with the i th smallest element of the set $W = \{Ux : U \text{ a substring of } T \text{ and } x \in \{\$, \#\}\}$. (Similarly for y -coordinates and T^R .)

We can preprocess the suffix tree $\mathcal{T}(T)$ in $\mathcal{O}(n)$ time so that the rank of a given $T[a..b]\$$ or $T[a..b]\#$ in W can be computed in $\tilde{O}(1)$ time. Let us assume that $\mathcal{T}(T)$ has been built for T , without $\$$ appended to it. We make a DFS traversal of $\mathcal{T}(T)$, maintaining a global counter cr , which is initialized to zero at the root. The DFS visits the children of a node in a left-to-right order. When traversing an edge, we increment cr by the size of the path-label of this edge. When an explicit node v is visited for the *first* time we set the rank of $\mathcal{L}(v)\$$ equal to cr ; if v is a leaf then cr is incremented by one. The rank of $\mathcal{L}(v)\#$ is set to cr when v is visited for the *last* time. Let q be the locus of $T[a..b]$ in $\mathcal{T}(T)$, which can be computed in $\mathcal{O}(\log \log n)$ time using a weighted ancestor query. If q is an explicit node, the ranks of $T[a..b]\$$ and $T[a..b]\#$ are already stored at q . Otherwise, these ranks can be inferred from the ranks of $\mathcal{L}(v)\$$ and $\mathcal{L}(v)\#$ stored at the nearest explicit descendant v of q by, respectively, subtracting and adding the distance between v and q .

Thus, instead of explicitly building trees W and W^R as in the proof of Lemma 5.4, we use $\mathcal{T}(T)$ and $\mathcal{T}(T^R)$ and maintain rectangles in \mathcal{G} . After m updates, we remove (resp. add) the $\tilde{O}(m)$ rectangles corresponding to patterns in $\mathcal{D}^0 \setminus \mathcal{D}^m$ (resp. $\mathcal{D}^m \setminus \mathcal{D}^0$). We can maintain a data structure of size $\mathcal{O}(r)$ for the counting version of range stabbing in a 2D grid of size $K \times K$ with r rectangles with $\mathcal{O}(\log K \log r / (\log \log r)^2)$ time per update and query [22].

To wrap up, updating the data structure every m updates requires $\tilde{O}(1 + n/m)$ amortized time. We can deamortize the time complexities using the time-slicing technique. This concludes the proof of the following theorem.

Theorem 6.7. *EXISTS(i, j), REPORT(i, j), REPORTDISTINCT(i, j), and COUNT(i, j) queries for a dynamic dictionary can be answered in $\tilde{O}(m + |\text{output}|)$ time per query and $\tilde{O}(n/m)$ time per update for any parameter $m \in [1..n]$ using $\tilde{O}(n + d)$ space.*

7 Final Remarks

The question of whether answering queries of the type $\text{COUNTDISTINCT}(i, j)$, i.e. returning 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 $\tilde{O}(1)$ with a data structure of size $\tilde{O}(n + d)$ 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 can be found in Appendix A.

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

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] Maxim Babenko, Paweł Gawrychowski, Tomasz Kociumaka, and Tatiana Starikovskaya. Wavelet trees meet suffix trees. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 572–591. SIAM, 2015. doi:10.1137/1.9781611973730.39.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] 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>.
- [11] David Clark. *Compact Pat trees*. PhD thesis, University of Waterloo, 1996. URL: <http://hdl.handle.net/10012/64>.
- [12] 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.
- [13] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007. doi:10.1017/cbo9780511546853.
- [14] 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.

- [15] 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.
- [16] Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011. doi:10.1137/090779759.
- [17] 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.
- [18] 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.
- [19] Paweł Gawrychowski, Gad M. Landau, Shay Mozes, and Oren Weimann. The nearest colored node in a tree. *Theoretical Computer Science*, 710:66–73, 2018. doi:10.1016/j.tcs.2017.08.021.
- [20] 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.
- [21] 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.
- [22] Meng He and J. Ian Munro. Space efficient data structures for dynamic orthogonal range counting. *Computational Geometry*, 47(2):268–281, 2014. doi:10.1016/j.comgeo.2013.08.007.
- [23] Monika Henzinger, Sebastian Krinninger, 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.
- [24] 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.
- [25] Guy Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science, FOCS 1989*, pages 549–554. IEEE Computer Society, 1989. doi:10.1109/SFCS.1989.63533.
- [26] Artur Jeż. Faster fully compressed pattern matching by recompression. *ACM Transactions on Algorithms*, 11(3):20:1–20:43, 2015. doi:10.1145/2631920.
- [27] 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.
- [28] 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.
- [29] 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>.
- [30] Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A linear time algorithm for seeds computation, 2019. arXiv:1107.2422.
- [31] 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.

- [32] 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.
- [33] Tsvi Kopelowitz, Moshe Lewenstein, and Ely Porat. Persistency in suffix trees with applications to string interval problems. In Roberto Grossi, Fabrizio Sebastiani, and Fabrizio Silvestri, editors, *String Processing and Information Retrieval, 18th International Symposium, SPIRE 2011*, volume 7024 of *Lecture Notes in Computer Science*, pages 67–80. Springer, 2011. doi:10.1007/978-3-642-24583-1_8.
- [34] Veli Mäkinen and Gonzalo Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007. doi:10.1016/j.tcs.2007.07.013.
- [35] 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.
- [36] 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>.
- [37] Mihai Pătraşcu. Unifying the landscape of cell-probe lower bounds. *SIAM Journal on Computing*, 40(3):827–847, 2011. doi:10.1137/09075336X.
- [38] 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.

A An approximation for CountDistinct(i, j)

In this section we consider queries of the type $\text{COUNTDISTINCT}(i, j)$, i.e. returning the number of patterns from \mathcal{D} that occur in $T[i..j]$. It turns out that our techniques can be used to efficiently answer such queries $\mathcal{O}(\log n)$ -approximately.

Let us first note that the hardness of the considered problem partially stems from the fact that it is not decomposable. In this section we adapt the data structure presented in Section 5 for answering $\text{COUNT}(i, j)$ queries and show the following result.

Theorem A.1. *Queries of the form $\text{COUNTDISTINCT}(i, j)$ can be answered $\mathcal{O}(\log n)$ -approximately in $\mathcal{O}(\log^2 n / \log \log n)$ time with a data structure of size $\mathcal{O}(n + d \log^2 n)$ that can be constructed in $\mathcal{O}(n \log n + d \log^{5/2} n)$ time.*

A.1 An $\mathcal{O}(\log^2 n)$ -approximation

We modify the data structure of Section 5 for $\text{COUNT}(i, j)$ as follows. For each symbol A in PT , we store the number of patterns from \mathcal{D} that occur in $\mathbf{g}(A)$. Additionally, if $A \rightarrow B^k$ is a power, we also store the number of patterns from \mathcal{D} that occur in $\mathbf{g}(B^i)$ for $i \in [1..k]$.

At query, we imitate the algorithm for $\text{COUNT}(i, j)$. We count each distinct pattern that occurs in $T[i..j]$ at least once and at most $\mathcal{O}(\log^2 n)$ times—at most once per visited node v such that $\text{val}(v)$ is contained in $T[i..j]$ and at most $\mathcal{O}(\log n)$ times for each considered anchor. The latter follows from the fact that the grid \mathcal{G} that is used to answer $\text{BREAKPOINT-ANCHOR IDM}$ has $\mathcal{O}(\log n)$ rectangles for each pattern.

Efficient preprocessing. We first show how to compute the desired counts for all nodes of the parse tree in time $\mathcal{O}(d + \sum_{v \in \text{PT}} |\text{val}(v)|) = \mathcal{O}(d + n \log n)$ in total. For a symbol A , with $\mathbf{g}(A) = T[a..b]$, for each $i \in [a..b]$, we compute the locus of the longest prefix of $T[i..b]$ that is in \mathcal{D} in the \mathcal{D} -modified suffix tree of T using a weighted ancestor query. (We answer all weighted ancestor queries for all nodes as a batch, similar to before, in the stated time complexity.) Let the respective loci be $u_1, u_2, \dots, u_{b-a+1}$,

sorted in the lexicographical order of their path-labels. We process the loci in this order. We process u_i by incrementing the count by the number of nodes on the path $(\text{LCA}(u_{i-1}, u_i), u_i]$, where u_0 is the root of the tree.² (We assume that we have precomputed the number of ancestors of each node of the tree.) The correctness of this algorithm follows from a straightforward inductive argument and the fact that the deepest node among $\text{LCA}(u_j, u_i)$, for $j < i$, is $\text{LCA}(u_{i-1}, u_i)$.

Finally, we argue that the time required to compute the number of patterns from \mathcal{D} that occur in $\mathbf{g}(B^i)$ for $i \in [1 \dots k]$ for a symbol $A \rightarrow B^k$ is also $\mathcal{O}(d + n \log n)$. This follows from the fact that any pattern occurring in $\mathbf{g}(B^i)$ has an occurrence in the first $|\mathbf{g}(B)|$ positions. Hence, it suffices to compute the required loci only for these for each i . We thus consider $k \cdot |\mathbf{g}(B)| = |\mathbf{g}(A)|$ such loci in total for this step of the computation as well.

A.2 Refinement through Colored Range Stabbing Counting

We improve the approximation ratio as follows. We color all rectangles corresponding to the same pattern with the same color in our data structure for BREAKPOINT-ANCHOR IDM and upon query ask a color rectangle stabbing query, i.e. ask for the number of distinct colors that the rectangles stabbed by the point have.

Lemma A.2. *Given k 4-sided colored rectangles in $2D$, such that the number of rectangles colored with each color are $\mathcal{O}(c)$, we can answer colored range stabbing counting queries in $\mathcal{O}(\log k / \log \log k)$ time with a data structure of size $\mathcal{O}(c \cdot k)$ that can be constructed in $\mathcal{O}(c \cdot k \sqrt{\log k})$ time.*

Proof. We first make the following claim.

Claim A.3. *Given a collection \mathcal{R} of $\mathcal{O}(c)$ axes-parallel rectangles in $2D$, we can construct a collection \mathcal{R}' of $\mathcal{O}(c^2)$ non-overlapping rectangles in $2D$ in $\mathcal{O}(c^2)$ time such that for every $(a, b) \in \mathbb{Z}^2$, (a, b) stabs some rectangle in \mathcal{R} if and only if it stabs some rectangle in \mathcal{R}' .*

Proof. We run a textbook line-sweeping algorithm, efficiently maintaining information about points contained in some rectangle using a segment tree. \square

We apply the above claim to the rectangles of each color. Let us now note that a colored stabbing counting query for our original set of rectangles corresponds to a simple stabbing counting query for the new set and we can thus employ Theorem 5.2. \square

Recall that there are $\mathcal{O}(d \log n)$ rectangles in total, $\mathcal{O}(\log n)$ with each color. We plug in the data structure of Lemma A.2 in our solution for BREAKPOINT-ANCHOR IDM and obtain Theorem A.1.

²A lowest common ancestor (LCA) query takes as input two nodes of a rooted tree and returns the deepest node of the tree that is an ancestor of both. Such queries can be answered in constant time after a linear-time preprocessing of the tree [9].