Algorithmica manuscript No.
(will be inserted by the editor)

Internal Dictionary Matching

Panagiotis Charalampopoulos* · **Tomasz Kociumaka** · **Manal Mohamed** · **Jakub Radoszewski** · **Wojciech Rytter** · **Tomasz Waleń**

Received: date / Accepted: date

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

A preliminary version of this paper was presented at the 30th International Symposium on Algorithms and Computation (ISAAC 2019) [14].

Panagiotis Charalampopoulos
The Interdisciplinary Center Herzliya, Israel
E-mail: pcharalampo@gmail.com
https://orcid.org/0000-0002-6024-1557
*Corresponding author

Tomasz Kociumaka
University of California, Berkeley, U.S.
E-mail: kociumaka@berkeley.edu
https://orcid.org/0000-0002-2477-1702

Manal Mohamed
London, UK
E-mail: manalabd@gmail.com
https://orcid.org/0000-0002-1435-5051

Jakub Radoszewski
University of Warsaw, Poland & Samsung R&D, Poland
E-mail: jrad@mimuw.edu.pl
https://orcid.org/0000-0002-0067-6401

Wojciech Rytter
University of Warsaw, Poland
E-mail: rytter@mimuw.edu.pl
https://orcid.org/0000-0002-9162-6724

Tomasz Waleń
University of Warsaw, Poland
E-mail: walen@mimuw.edu.pl
https://orcid.org/0000-0002-7369-3309

In particular, we consider the following types of queries: reporting and counting *all* occurrences of patterns from D in a fragment $T[i..j]$ and reporting *distinct* patterns from D that occur in $T[i..j]$. We show how to construct, in $O((n+d)\log^{O(1)} n)$ time, a data structure that answers each of these queries in time $O(\log^{O(1)} n + |output|)$.

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.

Keywords Dictionary matching · Internal pattern matching · Range searching · Dynamic dictionary

1 Introduction

In the problem of dictionary matching, which has been studied for more than forty years [1], we are given a dictionary D, consisting of d patterns, and the goal is to preprocess D so that presented with a text T we are able to efficiently compute the occurrences of the patterns from 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 $O(|T|+|output|)$ [1]. Compressed indexes for dictionary matching [11], as well as indexes for approximate dictionary matching [16] 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 [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 [31], while a linear sized data structure allowing for constant-time IPM queries in the case where the ratio between the lengths of the two substrings is constant was presented in [34]. 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 [32], 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 D consisting of substrings of text T: given (i, j), report/count all occurrences of patterns from D in $T[i..j]$ and report the distinct patterns from D that occur in $T[i..j]$.

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 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 $P \in \mathsf{D}$ occurs in $T[i..j]$.
REPORT(i,j): Report all occurrences of all the patterns of D in $T[i..j]$.
REPORTDISTINCT(i,j): Report all patterns $P \in \mathsf{D}$ that occur in $T[i..j]$.
COUNT(i,j): Count the number of all occurrences of all the patterns of D in $T[i..j]$.

Example 1.1 Consider a text $T = adaaaabaabbaac$ and a dictionary $\mathsf{D} = \{aa, aaaa, abba, c\}$; see Fig. 1. We then have:

$$\begin{aligned}
\textsc{Exists}(2,12) &= \textbf{true}\\
\textsc{Report}(2,12) &= \{(aa,3),(aaaa,3),(aa,4),(aa,5),(aa,8),(abba,9)\}\\
\textsc{Count}(2,12) &= 6\\
\textsc{ReportDistinct}(2,12) &= \{aa,aaaa,abba\}\\
\textsc{Exists}(1,3) &= \textbf{false}
\end{aligned}$$

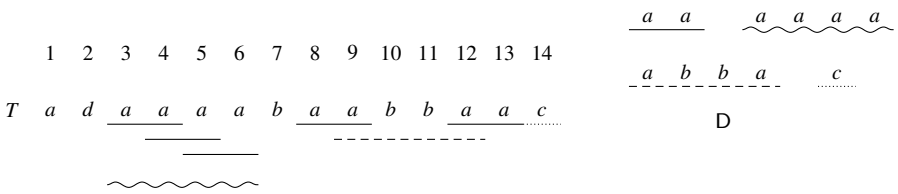

Fig. 1 Occurrences of patterns from the dictionary D in the text T.

Let us consider REPORT(i,j) queries. One could answer them in $O(j-i+|output|)$ time by running $T[i..j]$ over the Aho–Corasick automaton of D [1] or in time $\tilde{O}(d+|output|)$[1] by performing internal pattern matching [34] for each element of D individually. None of these approaches is satisfactory as they can require $\Omega(n)$ time in the worst case.

The IDM problem in the special case of the dictionary D being the set of all palindromes in T has already been studied in [41], where authors proposed a data structure of size $O(n\log n)$ that returns the number of all distinct palindromes in $T[i..j]$ in $O(\log n)$ time. Let us note that in this case the total length of patterns might be quadratic, but the internal dictionary is of linear size and can be constructed in $O(n)$ time [23]. Our general solution can be applied, in particular, to other such dictionaries that satisfy these requirements, say, for the dictionary of all squares in T [18,7].

Our results and techniques We focus on the case of a static dictionary, as it was defined above. We propose an $\tilde{O}(n+d)$-sized data structure, which can be built in time

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

$\tilde{O}(n+d)$ and answers all IDM queries in time $\tilde{O}(1+|output|)$. The exact complexities, shown in Table 1, are achieved using slightly different techniques for different query types. The solutions for queries $\textsc{Exists}(i,j)$ and $\textsc{Report}(i,j)$, presented in Section 3, are rather straightforward. The data structure for $\textsc{ReportDistinct}(i,j)$ queries, described in Section 4, heavily relies on the periodic structure of the input text and on tools that we borrow from computational geometry. An efficient solution for $\textsc{Count}(i,j)$ queries, provided in Section 5, is based on locally consistent parsing and further computational geometry tools.

Table 1 Our results.

Query	Preprocessing time	Space	Query time
$\textsc{Exists}(i,j)$	$O(n+d)$	$O(n)$	$O(1)$
$\textsc{Report}(i,j)$	$O(n+d)$	$O(n+d)$	$O(1+\|output\|)$
$\textsc{ReportDistinct}(i,j)$	$O(n\log n+d)$	$O(n+d)$	$O(\log n+\|output\|)$
$\textsc{Count}(i,j)$	$O(\frac{n\log n}{\log\log n}+d\log^{3/2} n)$	$O(n+d\log n)$	$O(\frac{\log^2 n}{\log\log n})$

Furthermore, in Section 6, we consider a variant of the problem where the dictionary D is dynamic, i.e. allowing for interleaved IDM queries and updates to D (insertions/deletions of patterns). We first show a conditional lower bound for this problem: Unless the Online Boolean Matrix-Vector Multiplication conjecture [26] is false, the product of the time to process an update and the time to answer an $\textsc{Exists}(i,j)$ query cannot be $O(n^{1-\varepsilon})$ for any constant $\varepsilon>0$. By building upon our solutions for static dictionaries, we also provide algorithms matching—up to sub-polynomial factors—our conditional lower bound: For any $0<\alpha<1$, we show how to process updates in $\tilde{O}(n^{\alpha})$ time and answer queries $\textsc{Exists}(i,j)$, $\textsc{Report}(i,j)$, $\textsc{ReportDistinct}(i,j)$, and $\textsc{Count}(i,j)$ in $\tilde{O}(n^{1-\alpha}+|output|)$ time.

Follow-up results $\textsc{CountDistinct}(i,j)$ is an equally interesting type of internal dictionary matching query, in which the number of distinct patterns from D that occur in $T[i..j]$ is to be returned. As has been already mentioned, a special case where D consists of all palindromes in T was considered in [41]. In a very recent follow-up of our work by a superset of the authors [13], another special case where D consists of all squares in T was considered. For general dictionaries, the following results were shown in [13].

- For any $m\in[1..n]$, there exists an $\tilde{O}(\min(nd/m, n^2/m^2)+d)$-size data structure that answers $\textsc{CountDistinct}(i,j)$ queries in $\tilde{O}(m)$ time.
- We can construct in $\tilde{O}(n+d)$ time a data structure that gives 2-approximate answers to $\textsc{CountDistinct}(i,j)$ queries in $\tilde{O}(1)$ time.

In an earlier version of this work, we presented an $\tilde{O}(n+d)$-size data structure that could answer $\textsc{CountDistinct}(i,j)$ queries $O(\log n)$-approximately in $\tilde{O}(1)$ time, by appropriately adapting our data structure for $\textsc{Count}(i,j)$ queries. As this data structure is now obsolete, we have not included its description in this version.

2 Preliminaries

We begin with basic definitions and notation generally following the conventions of [17]. 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 $O(1)$ space by specifying the indices i and j. A *prefix* of T is a fragment that starts at position 1 ($T[1\,..\,j]$) 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 $i = 1,\ldots,n-p$. 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 $\mathsf{per}(T)$, 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.

Consider a set of strings $\mathbf{S}$. The *trie* of $\mathbf{S}$ is a rooted tree with edges labeled by single letters so that edges going down from a node have distinct labels. For each node v of the trie, the *path-label* of v is the concatenation of the edge labels on the path from the root to v, and the *string-depth* of v, denoted $\delta(v)$, is the length of the path-label. In the trie of $\mathbf{S}$, the path-label of each node is a prefix of some string $S \in \mathbf{S}$. Moreover, each prefix of a string $S \in \mathbf{S}$ is the path-label of a unique node of the trie, called its *locus*. In the *compact trie* of $\mathbf{S}$, only *branching* nodes (with at least two children) and *terminal* nodes (loci of strings $S \in \mathbf{S}$) are stored *explicitly*. The remaining *implicit* nodes of the trie (non-terminal nodes with exactly one child) are dissolved so that each edge of the compact trie can be viewed as an upward maximal path of implicit nodes starting with an explicit node. Subsequent nodes on this path are indexed by consecutive integers so that the index of the starting explicit node is zero. Note that each node belongs to a unique path of that kind. Thus, each node of the trie can be represented in the compact trie by the edge it belongs to and an index within the corresponding path.

The *suffix tree* $\mathbf{T}(T)$ of a string T of length n is the compact trie of $\{T_{(i)}\$: 1 \leq i \leq n\}$, where $\$ \notin \Sigma$ is a sentinel character that is lexicographically smaller than all the letters in Σ; this ensures that all terminal nodes are leaves. The suffix tree of a string of length n over an integer ordered alphabet can be computed in time and space $O(n)$ [19]. Once the suffix tree is constructed, it can be traversed in a depth-first manner to compute the string-depth $\delta(v)$ for each explicit node v. For each $1 \leq i \leq n$, the terminal node with path-label $T_{(i)}\$$ is additionally labelled with index i.

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 $O(n)$-time

preprocessing, weighted ancestor queries for nodes of a weighted tree $\mathbf{T}$ of size n with integer weights from a universe $[1\,..\,U]$ can be answered in $O(\log\log U)$ time per query [4]. Moreover, any batch of k weighted ancestor queries can be answered in $O(n+k)$ time [33, Section 7]. 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 $O(1)$ time after $O(n)$-time preprocessing [8]; in this special case the values ω are called *levels* and the queries are called *level ancestor queries*. The suffix tree $\mathbf{T}(T)$ is a weighted tree with $\omega(v)=\delta(v)$ for all v. Hence, the locus of a fragment $T[i\,..\,j]$ in $\mathbf{T}(T)$ is the weighted ancestor of the terminal node with path-label $T_{(i)}\$$ at string-depth $j-i+1$.

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

3 EXISTS(i,j) and REPORT(i,j) queries

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

Definition 3.1 A D-*modified suffix tree* of a string T is a tree with leaves corresponding to non-empty suffixes of $T\$$ and internal nodes corresponding to $\{\varepsilon\}\cup\mathsf{D}$. 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 δ. Moreover, each leaf node stores its label (i.e., the leaf corresponding to $T_{(i)}\$$ stores the label i).

Lemma 3.2 *A D-modified suffix tree of T has size $O(n+d)$ and can be constructed in $O(n+d)$ time.*

Proof The D-modified suffix tree is obtained from the suffix tree $\mathbf{T}(T)$ in two steps. In the first step, we mark all nodes of $\mathbf{T}(T)$ with path-label equal to a pattern $P\in\mathsf{D}\cup\{\varepsilon\}$: if any of them are implicit, we also make them explicit; see the annotated nodes in Fig. 2 (top). Note that we can find the loci of the patterns in $\mathbf{T}(T)$ in $O(n+d)$ time by answering a batch of weighted ancestor queries [33, Section 7]. We are left with the task of making several implicit nodes explicit, possibly multiple nodes on some edges of the tree. For this, the implicit nodes on each edge need to be sorted by their string-depths. Sorting locally within each edge could add $\omega(n+d)$ to the overall time complexity, so we use a global bucket sort instead, which costs $O(n+d)$ time.

In the second step, we traverse the tree top-down and compute, for each marked node or leaf u, its nearest marked ancestor $v\neq u$; the node v becomes the parent of the node u in the D-modified suffix tree. The string-depths of nodes are retained from the original suffix tree. □

We state the following simple lemma.

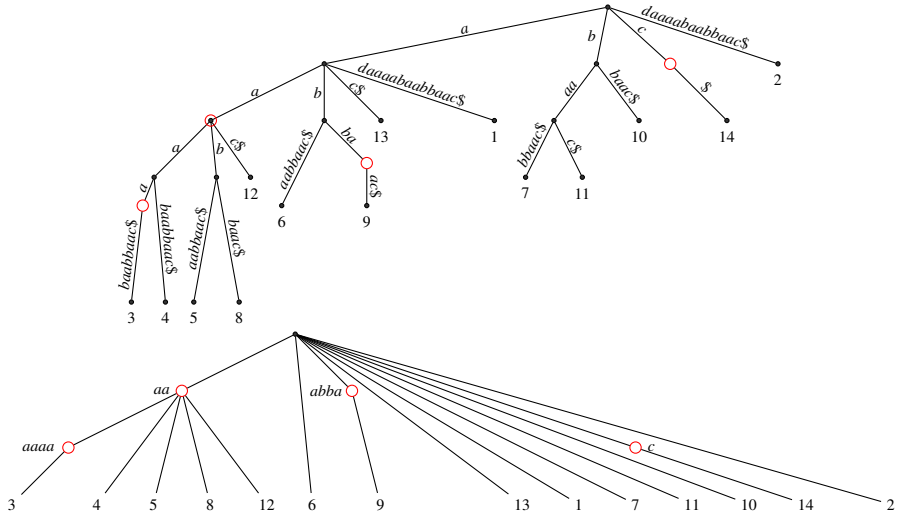

Fig. 2 Example of a D-modified suffix tree for dictionary $\mathsf{D} = \{aa, aaaa, abba, c\}$ and text $T = adaaaabaabbaac$ from Example 1.1. Top: the suffix tree $\mathbf{T}(T)$ with the nodes corresponding to elements of D annotated in red; bottom: the D-modified suffix tree of T.

Lemma 3.3 *With the* D*-modified suffix tree of T at hand and $O(n+d)$ additional space, given positions a, j in T with $a \leq j$, we can compute all $P \in \mathsf{D}$ that occur as prefixes of $T[a..j]$ in time $O(1+|output|)$.*

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

The path is traversed using level ancestor queries on the D-modified suffix tree asked from the leaf [8]. □

The D-modified suffix tree enables us to efficiently answer $\textsc{Exists}(i,j)$ and $\textsc{Report}(i,j)$ queries. Let us introduce two auxiliary length-n arrays that can be computed from the D-modified suffix tree; the first one will be used in the solution in this section, whereas the other one in the following sections:

$$\mathit{Shortest}(\mathsf{D})[a] = \min(\{b : T[a..b] \in \mathsf{D}\} \cup \{\infty\})$$
$$\mathit{Longest}(\mathsf{D})[a] = \max(\{b : T[a..b] \in \mathsf{D}\} \cup \{0\})$$

Example 3.4 Arrays *Shortest*, *Longest* for T and $\mathsf{D} = \{aa, aaaa, abba, c\}$ from Fig. 2.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	a	d	a	a	a	a	b	a	a	b	b	a	a	c
$\mathit{Shortest}(\mathsf{D})[i]$	∞	∞	4	5	6	∞	∞	9	12	∞	∞	13	∞	14
$\mathit{Longest}(\mathsf{D})[i]$	0	0	6	5	6	0	0	9	12	0	0	13	0	14

Lemma 3.5 *Given the* D*-modified suffix tree of a string* T *together with a data structure supporting* $O(1)$*-time level ancestor queries, each element of the arrays* $\mathit{Shortest}(\mathsf{D})$ *and* $\mathit{Longest}(\mathsf{D})$ *can be retrieved in* $O(1)$ *time.*

Proof For a leaf of the D-modified suffix tree with label a, we set $\mathit{Longest}(\mathsf{D})[a]$ to the string-depth of its parent. For a leaf with label a and level greater than 1, we set $\mathit{Shortest}(\mathsf{D})[a]$ to the string-depth of its ancestor at level 1, retrieved using a level ancestor query. If the leaf is itself at level 1, then $\mathit{Shortest}(\mathsf{D})[a] = \infty$. □

Theorem 3.6

(a) $\textsc{Exists}(i, j)$ *queries can be answered in* $O(1)$ *time with a data structure of size* $O(n)$ *that can be constructed in* $O(n+d)$ *time.*

(b) $\textsc{Report}(i, j)$ *queries can be answered in* $O(1+|\mathit{output}|)$ *time with a data structure of size* $O(n+d)$ *that can be constructed in* $O(n+d)$ *time.*

Proof (a) It can be readily verified that the answer to query $\textsc{Exists}(i, j)$ is **true** if and only if the minimum element in the subarray $\mathit{Shortest}(\mathsf{D})[i\,..\,j]$ is at most j. Thus, in order to answer $\textsc{Exists}(i, j)$ queries, it suffices to construct the array $\mathit{Shortest}(\mathsf{D})$ and a data structure that answers range minimum queries (RMQ) on $\mathit{Shortest}(\mathsf{D})$. The array $\mathit{Shortest}(\mathsf{D})$ can be constructed in $O(n+d)$ time using Lemmas 3.2 and 3.5, and the data structure answering range minimum queries in $O(1)$ time can be built in time $O(n)$ [24,9].

(b) We first identify all positions $a \in [i\,..\,j]$ that are starting positions of occurrences of some pattern $P \in \mathsf{D}$ in $T[i\,..\,j]$ using RMQs over array $\mathit{Shortest}(\mathsf{D})$ as follows. The first RMQ is over the range $[i\,..\,j]$ and identifies any such position a (if any such position exists). The range is then split into two parts, $[i\,..\,a-1]$ 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 D occurs, we report all the patterns occurring at each of these positions and being contained in $T[i\,..\,j]$ using Lemma 3.3. The complexities follow from Lemmas 3.2 and 3.3. □

4 $\textsc{ReportDistinct}(i, j)$ queries

Below, we present an algorithm that reports patterns from D occurring in $T[i\,..\,j]$, allowing for $O(1)$ 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 D into $\mathsf{D}_0, \ldots, \mathsf{D}_{\lfloor \log n \rfloor}$ such that $\mathsf{D}_k = \{P \in \mathsf{D} : \lfloor \log |P| \rfloor = k\}$. We call D_k a *k-dictionary*. We now show how to process a single k-dictionary D_k; the query procedure may clearly assume $k \leq \log |T[i\,..\,j]|$.

We first build the D_k-modified suffix tree of T. Then, we compute the array $L_k := \mathit{Longest}(\mathsf{D}_k)$, assign to all the patterns of D_k equal to some $T[a\,..\,L_k[a]]$ integer identifiers id (or colors) in $[1\,..\,n]$, and construct an array $I_k[a] = \mathsf{id}(P)$, where $P = T[a\,..\,L_k[a]]$. Technically, in order to compute the array I_k efficiently, for each index a, in addition to $L_k[a]$, we store the node of the D_k-modified suffix tree representing $T[a\,..\,L_k[a]]$; this node is obtained as in the proof of Lemma 3.5.

We then construct the data structure specified in the following theorem for I_k; this data structure, due to Muthukrishnan [39], allows for efficient colored range reporting queries.

Theorem 4.1 (Colored Range Reporting [39]) *Given an array $A[1..N]$ of elements from $[1..U]$, we can construct a data structure of size $O(N)$ in $O(N+U)$ time, so that upon query $[i..j]$ all distinct elements in $A[i..j]$ can be reported in $O(1+|output|)$ time.*

Let $t = \max\{i, j-2^{k+1}+1\}$. First, we perform a colored range reporting query on the range $[i..t)$ of array I_k and obtain a set C_k of distinct patterns, employing Theorem 4.1. (An illustration is provided in Fig. 3.) We observe the following.

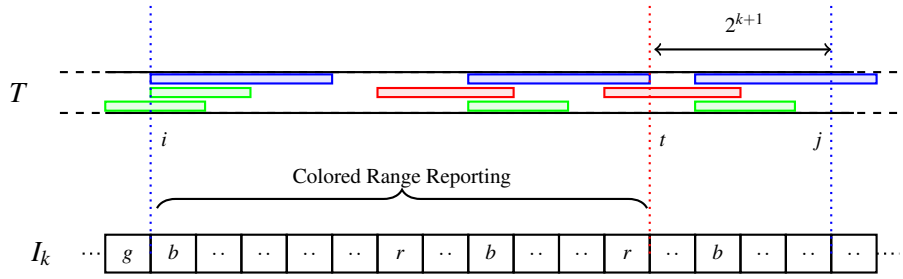

Fig. 3 An illustration of array I_k and the colored range reporting query that we perform. The k-dictionary consists of the blue, green and red patterns, whereas C_k consists of the blue and red patterns. Note that patterns that start in $[i..t)$ cannot end in a position to the right of j.

Observation 4.2 *Any pattern of a k-dictionary D_k occurring in T at position $p \in [i..t)$ is a prefix of a pattern $P \in \mathsf{C}_k$.*

Based on this observation, we report the remaining patterns that start in $[i..t)$ using the D_k-modified suffix tree, in $O(1+|output|)$ time, following parent pointers and temporarily marking the loci of reported patterns to avoid double-reporting. We have shown the following lemma.

Lemma 4.3 *All distinct patterns from dictionary D_k that start in an interval $[i..t)$ can be reported in $O(1+|output|)$ time with a data structure of size $O(n+|\mathsf{D}_k|)$ that can be constructed in $O(n+|\mathsf{D}_k|)$ time.*

We thus now only have to report the patterns from D_k that occur in $T[t..j]$. To this end, we further partition D_k to a *periodic k-dictionary* and an *aperiodic k-dictionary*:

$$\mathsf{D}_k^{\text{per}} = \{P \in \mathsf{D}_k : \mathsf{per}(P) \le 2^k/3\} \quad \text{and} \quad \mathsf{D}_k^{\text{aper}} = \{P \in \mathsf{D}_k : \mathsf{per}(P) > 2^k/3\}.$$

Note that we can partition D_k in $O(|\mathsf{D}_k|)$ time using the so-called 2-period queries of [34,6,32]. Such a query decides whether a given fragment of the text is periodic

and, if so, also returns its period. These queries can be answered in $O(1)$ time after an $O(n)$-time preprocessing of the text.

Let us provide some intuition behind this partition of D_k. As shown next, each pattern in $\mathsf{D}_k^{\text{aper}}$ can occur only a constant number of times in $T[t\,..\,j]$. (If two occurrences of a pattern have a "large" overlap, then this pattern must have a "small" period.) Patterns in $\mathsf{D}_k^{\text{per}}$ may have many occurrences; however, as we will show below, each of them occurs in one of a constant number of *runs* (formally defined in Section 4.2). Our algorithm processes each such run R, efficiently computing a single occurrence of each $P \in \mathsf{D}_k^{\text{per}}$ that occurs in R.

4.1 Processing an aperiodic k-dictionary

We make use of the following sparsity property.

Fact 4.4 (Sparsity of occurrences) *The occurrences of a pattern P of an aperiodic k-dictionary $\mathsf{D}_k^{\text{aper}}$ in T start more than $\frac{2^k}{3}$ 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 $P \in \mathsf{D}_k^{\text{aper}}$. □

Lemma 4.5 REPORTDISTINCT(t, j) *queries for the aperiodic k-dictionary $\mathsf{D}_k^{\text{aper}}$ and $j - t < 2^{k+1}$ can be answered in $O(1 + |output|)$ time with a data structure of size $O(n + |\mathsf{D}_k^{\text{aper}}|)$ that can be constructed in $O(n + |\mathsf{D}_k^{\text{aper}}|)$ time.*

Proof Since the fragment $T[t\,..\,j]$ is of length at most 2^{k+1}, by Fact 4.4, it may contain at most three occurrences of each pattern in $\mathsf{D}_k^{\text{aper}}$. We can thus simply use a REPORT(t, j) query for dictionary $\mathsf{D}_k^{\text{aper}}$ and then remove duplicates. The complexities follow from Theorem 3.6(b). □

4.2 Processing a periodic k-dictionary

Our solution for periodic patterns relies on the well-studied theory of *runs* (maximal repetitions) 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 $p = \mathsf{per}(R)$, that is, $T[a-1] \neq T[a+p-1]$ and $T[b-p+1] \neq T[b+1]$ provided that the respective positions exist. The number of runs in a string of length n is $O(n)$ and all the runs can be computed in $O(n)$ time [35,6]. The following property of runs is known [34,32].

Observation 4.6 *Let P be a periodic pattern. Every occurrence of P in T is contained within a unique run R in T with $\mathsf{per}(R) = \mathsf{per}(P)$. We say that this run R* extends *the occurrence.*

Let $\mathbf{R}$ be the set of all runs in T. We construct for all $k \in [0\,..\,\lfloor \log n \rfloor]$ the sets of runs $\mathbf{R}_k = \{R \in \mathbf{R} : \mathsf{per}(R) \leq \frac{2^k}{3}, |R| \geq 2^k\}$ in $O(n)$ time overall. Note that these sets might not be disjoint; however, $|\mathbf{R}_k| = O(\frac{n}{2^k})$ (cf. Lemma 4.8 below) and thus their total size is $O(n)$.

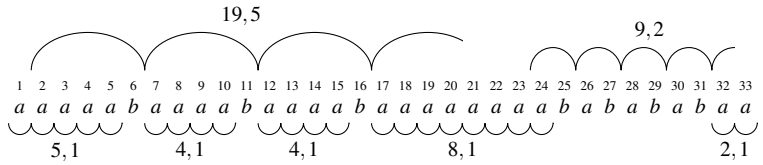

Fig. 4 All the runs in a sample string T, annotated with their lengths and periods.

Example 4.7 For the string T in Fig. 4, we have

$$\begin{aligned}\mathbf{R} &= \{T[1..5], T[2..20], T[7..10], T[12..15], T[17..24], T[24..32], T[32..33]\},\\ \mathbf{R}_2 &= \{T[1..5], T[7..10], T[12..15], T[17..24]\},\\ \mathbf{R}_3 &= \{T[17..24], T[24..32]\},\\ \mathbf{R}_4 &= \{T[2..20]\}.\end{aligned}$$

Note that $\mathbf{R}_2 \cap \mathbf{R}_3 \neq \emptyset$ and that the run $T[32..33]$ does not belong to any set $\mathbf{R}_k$.

If U is a fragment of T, by $\mathbf{R}_k(U) \subseteq \mathbf{R}_k$ we denote the set of all runs $R \in \mathbf{R}_k$ such that $|R \cap U| \geq 2^k$, that is, runs from $\mathbf{R}_k$ whose overlap with the fragment U contains at least 2^k positions. The following lemma is known, but we include a proof for the sake of completeness and to help the reader to develop some intuition.

Lemma 4.8 ([32, Lemma 4.4.7]) $|\mathbf{R}_k(U)| = O\left(\frac{1}{2^k}|U|\right)$.

Proof Since each of the runs in $\mathbf{R}_k$ is of length at least 2^k, it suffices to prove that any position i of T is contained in at most 3 runs from $\mathbf{R}_k$.

Let $m = \lfloor \frac{2^k}{3} \rfloor$. Let us suppose for the sake of contradiction that some position i is contained in at least four runs from $\mathbf{R}_k$. Observe that each of the runs must cover all positions of at least one of the fragments $T[i-2m+1..i]$, $T[i-m+1..i+m]$ and $T[i..i+2m-1]$. By the pigeonhole principle, two of these runs, say R_1 and R_2, have an overlap of at least $2m$ positions. Further, we have $|R_1 \cap R_2| \geq 2m \geq \mathsf{per}(R_1) + \mathsf{per}(R_2)$.

This contradicts the following known fact, being a consequence of the periodicity lemma [20]: If R_1 and R_2 are two different overlapping runs with periods $\mathsf{per}(R_1)$ and $\mathsf{per}(R_2)$, then $|R_1 \cap R_2| \leq \mathsf{per}(R_1) + \mathsf{per}(R_2) - 1$. □

Strategy Let us now provide a high-level description of our approach for processing $\mathsf{D}_k^{\mathrm{per}}$. Given a fragment $U = T[t..j]$, we will first identify the set $\mathbf{R}_k(U)$ of runs in $\mathbf{R}_k$ that have a sufficient overlap with U. There is a constant number of them by Lemma 4.8. For an occurrence in U of a pattern $P \in \mathsf{D}_k^{\mathrm{per}}$, the unique run R extending this occurrence of P must be in $\mathbf{R}_k(U)$. We will process each run $R \in \mathbf{R}_k(U)$ in order to compute a unique (the leftmost) occurrence in $R \cap U$ of each such pattern P.

Let us now focus on the structure of the problem that allows us to only compute such a unique occurrence. A string S is called *primitive* if it cannot be expressed as X^k for a string X and an integer $k > 1$. The *primitivity lemma* states that a non-empty string V is primitive if and only if it occurs only twice in VV: as a prefix and as a suffix. Moreover, any length-$|\mathsf{per}(U)|$ fragment of a string U is primitive [17]. The

following consequence of these two standard facts will be used for $P \in \mathsf{D}_k^{\mathrm{per}}$ and Q being (a fragment of) a run that extends its occurrence.

Lemma 4.9 *If a pattern P occurs in a text Q and satisfies $|P| \geq \mathsf{per}(Q)$, then P has exactly one occurrence starting within the first $\mathsf{per}(Q)$ positions of Q.*

Proof First, let us prove that P has at least one occurrence starting within the first $q := \mathsf{per}(Q)$ positions of Q. Suppose, for the sake of contradiction, that the leftmost occurrence of P in Q is at some position $j > q$. Then, by periodicity, we have $Q[j\,..\,j+|P|-1] = Q[j-q\,..\,j+|P|-1-q] = P$, a contradiction.

We now proceed to showing that P has exactly one occurrence starting within the first q positions of Q. Towards a contradiction, suppose that this is not the case and that P occurs at positions $i, j \in [1\,..\,q]$, with $i < j$. Now, let us consider $V := Q[i\,..\,i+q-1] = Q[j\,..\,j+q-1]$. Then, V occurs in VV at position $j-i+1$, neither as a prefix nor as a suffix. Hence, V is not primitive, a contradiction. □

We next present our data structure.

Construction of the data structure As discussed above, we compute $\mathbf{R}_k$ in $O(n)$ time. We also build the $\mathsf{D}_k^{\mathrm{per}}$-modified suffix tree of T and the array $\ell_k := \mathit{Shortest}(\mathsf{D}_k^{\mathrm{per}})$ in $O(n + |\mathsf{D}_k^{\mathrm{per}}|)$ time (using Lemma 3.2 and Lemma 3.5, respectively). We then preprocess the array ℓ_k for RMQ queries.

Answering a query Let us consider a query for $U = T[t\,..\,j]$. We first compute all runs in $\mathbf{R}_k(U)$ using the following lemma.

Lemma 4.10 *Let U be a fragment of T of length at most 2^{k+1}. Then $\mathbf{R}_k(U)$ can be retrieved in $O(1)$ time after an $O(n)$-time preprocessing.*

Proof A periodic extension query [32, Section 5.1], given a fragment V of the text T as input, checks if V is periodic and, if so, returns the run R extending V. Such queries can be answered in $O(1)$ time after $O(n)$-time preprocessing.

Let us cover all positions of U using $O(\frac{1}{2^k}|U|) = O(1)$ fragments of length $\lfloor \frac{2^{k+1}}{3} \rfloor$ with overlaps of at least $\lfloor \frac{2^k}{3} \rfloor$ positions and ask a periodic extension query for each fragment V in the cover. For each run $R \in \mathbf{R}_k(U)$, the overlap $R \cap U$ must contain a fragment V in the cover. Due to $|V| \geq 2 \cdot \mathsf{per}(R)$, the periodic extension of V must be R. □

Finally, let us show how to process each of the $O(1)$ runs returned by the procedure underlying Lemma 4.10.

We do the following for each run $R \in \mathbf{R}_k(U)$. We use RMQs repeatedly, as in the proof of Theorem 3.6(b), for the subarray of ℓ_k corresponding to the first $\mathsf{per}(R)$ positions of $R \cap U$. This way, due to Lemma 4.9, we compute precisely the positions where a pattern $P \in \mathsf{D}_k^{\mathrm{per}}$ has its leftmost occurrence in $R \cap U$. The number of positions identified for a single run $R \in \mathbf{R}_k(U)$ is therefore upper bounded by the number of distinct patterns occurring within $R \cap U$. For each such starting position a, we employ Lemma 3.3 to return all patterns in $\mathsf{D}_k^{\mathrm{per}}$ that are prefixes of $T[a\,..\,j]$. We thus report all

distinct patterns occurring within $R \cap U$. By Lemma 4.9, there is no double-reporting while processing a single run, and hence the time required to process this run is $O(1+|output|)$, where $|output|$ is the number of distinct patterns from $\mathsf{D}_k^{\mathrm{per}}$ occurring in the whole fragment U (rather than just $R \cap U$). See Fig. 5.

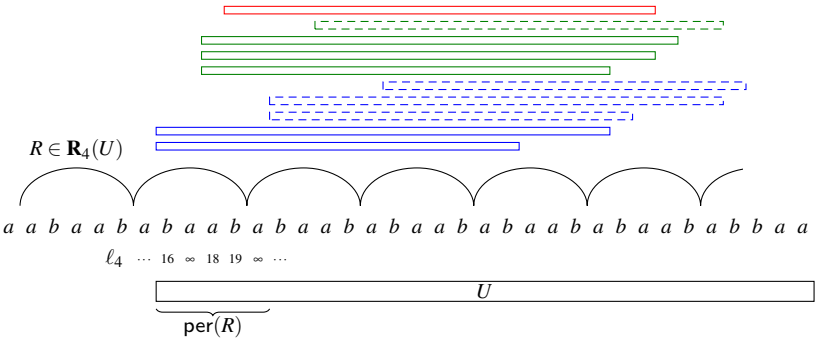

Fig. 5 Distinct patterns in $\mathsf{D}_4^{\mathrm{per}}$ that are generated by $R \in \mathbf{R}_4(U)$ and occur in $R \cap U$ are reported by their occurrences in the first $\mathsf{per}(R)$ positions of $R \cap U$. Dashed occurrences are not reported as the corresponding patterns occur earlier.

Since $|\mathbf{R}_k(U)| = O(1)$, we report each pattern a constant number of times and the overall time required is $O(1+|output|)$. We have thus proved the following lemma.

Lemma 4.11 REPORTDISTINCT(t, j) *queries for the periodic k-dictionary $\mathsf{D}_k^{\mathrm{per}}$ and $j - t < 2^{k+1}$ can be answered in $O(1+|output|)$ time with a data structure of size $O(n+|\mathsf{D}_k^{\mathrm{per}}|)$ that can be constructed in $O(n+|\mathsf{D}_k^{\mathrm{per}}|)$ time.*

Overall, by summing across all $k \in [0\,..\,\lfloor \log n \rfloor]$, we obtain a data structure of total size $O(n \log n + d)$ that can be constructed in time $O(n \log n + d)$ and answers REPORTDISTINCT(i, j) queries in $O(\log n + |output|)$ time. In the next subsection, we carefully reduce the space occupied by our data structure.

4.3 Reducing the space usage

The space occupied by our data structure can be reduced to $O(n+d)$. Let us notice that the $O(n \log n)$ term in the space complexity comes from several data structures that take $O(n+d')$ or $O(n)$ space for a given dictionary D' and are stored in our solution for some or all of the dictionaries D_k, $\mathsf{D}_k^{\mathrm{aper}}$, $\mathsf{D}_k^{\mathrm{per}}$ for $k \in [0\,..\,\lfloor \log n \rfloor]$:

(a) the D'-modified suffix tree with a data structure for level ancestor queries [8], which use $O(n+d')$ space,
(b) arrays *Shortest*(D') and *Longest*(D'), which use $O(n)$ space, and data structures for answering RMQs on these arrays,
(c) the data structure for colored range reporting on an auxiliary array I_k, which uses $O(n)$ space.

The remaining data structures—for answering 2-period queries (in the construction of $\mathsf{D}_k^{\text{aper}}$ and $\mathsf{D}_k^{\text{per}}$) and periodic extension queries (Lemma 4.10), and the sets of runs $\mathbf{R}_k$—take $O(n)$ space in total. Let us now show how to modify the components (a)–(c) to reduce the space usage to $O(n+d)$ without losing the desired functionality.

Part (a) We store the D-modified suffix tree and, for all $k \in [0\,..\,\lfloor \log n \rfloor]$ and $i \in \{\text{aper}, \text{per}\}$, we mark all nodes from each D_k^i with a different color. For each dictionary $\mathsf{D}' \in \{\mathsf{D}_k, \mathsf{D}_k^{\text{aper}}, \mathsf{D}_k^{\text{per}}\}$, we further store, using $O(|\mathsf{D}'|)$ space, the D'-modified suffix tree with its leaves trimmed, augmented with the level ancestor data structure.

The only additional operation we now need to support is determining the parent of a given leaf in the original D'-modified suffix tree (before the leaves were trimmed). This can be done using the nearest colored ancestor data structure of [22] over the D-modified suffix tree. In such queries, given a tree with colored nodes, we want to be able to compute the nearest ancestor of a query node v that has a color c specified in the query. For a tree of size N, the data structure of [22] achieves $O(\log\log N)$ time per query after $O(N)$-time preprocessing. We can, however, exploit the fact that we only have $\mathsf{palette} = O(\log n)$ colors to obtain $O(1)$-time queries within the same construction time.

It is shown in [22] 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 $O(N)$ and predecessor data structures for $O(\mathsf{palette})$ subsets of $[1\,..\,2N]$ whose total size is $O(N)$. Here, a predecessor query $\mathsf{pred}_S(i)$, for a set S and number i, returns the maximum element of S that does not exceed i. The time needed to compute the sets for the predecessor data structures and the arrays is $O(N)$. 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 $O(N)$ bits of space as follows. We store a bitmap that has the ith bit set if and only if $i \in S$ and augment it with a data structure that answers rank and select queries in $O(1)$ time and requires $o(N)$ additional bits of space [28, 15]. Such a component can be constructed in $O(N/\log N)$ time [5, 38]. Note that $\mathsf{pred}_S(i) = \mathsf{select}(\mathsf{rank}(i))$. We thus use $O((n+d)\log n)$ bits, i.e., $O(n+d)$ machine words, in total.

Part (b) The arrays $\mathit{Longest}(\mathsf{D}')$ and $\mathit{Shortest}(\mathsf{D}')$ do not need to be stored explicitly since their elements can be retrieved using the parent pointers and $O(1)$-time level ancestor queries in the D'-modified suffix tree (Lemma 3.5).

It was shown in [21] that an $O(1)$-query-time RMQ data structure for an array of size N can be implemented using $O(N)$ bits. This data structure only returns the index of the minimum value in the given range. To construct the data structures, we build the underlying arrays one by one and store the relevant RMQ data structures, which require $O(n\log n)$ bits, i.e. $O(n)$ words, in total.

Part (c) For colored range reporting over an array A, the main component of the data structure underlying Theorem 4.1 from [39] is an RMQ data structure over an array $J[i] = \max\{j : j < i, A[i] = A[j]\}$. We build an $O(N)$-bits RMQ data structure over J. The query procedure however, needs access to A, i.e. the colors. Let us recall that our array of colors I_k is defined by assigning to all the patterns of D_k equal to $T[a\,..\,L_k[a]]$

for some a, where $L_k := Longest(\mathsf{D}_k)$, colors id in $[1\,..\,n]$, and setting $I_k[a] = \mathsf{id}(P)$, where $P = T[a\,..\,L_k[a]]$. The array L_k can be retrieved from part (b). Let us recall that each entry $L_k[a]$ can be retrieved in $O(1)$ time along with the corresponding internal node of the D_k-modified suffix tree. All the internal nodes are stored explicitly, so their colors can simply be kept with them.

With Lemmas 4.3, 4.5, 4.11 and the above modifications to the space usage, we arrive at the main result of this section.

Theorem 4.12 REPORTDISTINCT(i,j) *queries can be answered in time* $O(\log n + |output|)$ *with a data structure of size* $O(n+d)$ *that can be constructed in time* $O(n\log n + d)$.

5 COUNT(i,j) queries

We first solve an auxiliary BREAKPOINTS-ANCHOR IPM problem and apply it to answer COUNT(i,j) queries in $O(\log^2 n/\log\log n)$ time with a data structure of size $O(nd)$. Then, we employ a locally consistent parsing to reduce the space usage to $O(n + d\log n)$.

5.1 An auxiliary problem

For a string S, by inter-position $i+1/2$ we refer to a location between positions i and $i+1$ in S. We also refer to inter-positions $1/2$ and $|S|+1/2$.

We consider a variant of the Internal Pattern Matching (IPM) 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 match P and 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, a length-m substring P of T, and a set B of inter-positions (breakpoints) of P.
Query: COUNT$_\beta(i,j)$: for a given inter-position (anchor) β in T, the number of fragments $T[r+1\,..\,r+m]$ contained in $T[i\,..\,j]$ that satisfy $T[r+1\,..\,\lfloor\beta\rfloor] = P[1\,..\,\lfloor b\rfloor]$ and $T[\lceil\beta\rceil\,..\,r+m] = P[\lceil b\rceil\,..\,m]$ for some $b\in B$. (Equivalently, $T[r+1\,..\,r+m] = P$ and $\beta - r \in B$.)

Before we proceed with the solution to the auxiliary problem, let us recall known results on 2D counting problems. In the 2D orthogonal range counting problem, one is to preprocess an $n\times n$ grid with $O(n)$ 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 $O(n)$ 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. [40]). For each rectangle $[x_1,y_1]\times[x_2,y_2]$ in grid $\mathscr{G}$, 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 $\mathscr{G}'$. Then the number of rectangles stabbed by point (a,b) in $\mathscr{G}$ is equal to the sum of weights of points in $(-\infty,a]\times(-\infty,b]$ in $\mathscr{G}'$. We will use the following result in our solution to BREAKPOINTS-ANCHOR IPM (Lemma 5.4).

Theorem 5.1 **([12])** *Range counting queries for n points in 2D (rank space) can be answered in time $O(\log n/\log\log n)$ with a data structure of size $O(n)$ that can be constructed in time $O(n\sqrt{\log n})$.*

Data structure Let $W_1 = \{P[\lceil b\rceil\,..\,m] : b \in B\}\cup\{\varepsilon\}$ 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; the trie W can be constructed efficiently using a rather standard approach.

Lemma 5.2 *W can be constructed in $O(n+|B|)$ time.*

Proof First we sort all elements of W_2 lexicographically. For this, (implicit or explicit) nodes of $\mathbf{T}(T)$ that correspond to strings in W_1 are marked using batch weighted ancestor queries. Implicit nodes on each edge are sorted top-down, off-line for all edges of $\mathbf{T}(T)$, using radix sort. Then we traverse $\mathbf{T}(T)$ using left-to-right depth first search. When a node corresponding to a string $U\in W_1$ is visited for the first/last time, we add $U\$/U\#$ to the sorted list of W_2, which we further denote by $\mathbf{L}$.

We can compute longest common prefixes of fragments of T in $O(1)$ time after $O(n)$-time preprocessing. This lets us compute longest common prefixes of all pairs of strings that are adjacent in $\mathbf{L}$ in $O(n+|B|)$ total time. Then the construction of W mimics the algorithm for constructing the suffix tree of a string from its suffix and LCP arrays; see e.g. [17]. □

We also build the W_1-modified suffix tree of T and preprocess it for weighted ancestor queries. Moreover, for each string $U\in W_1$, we store pointers to the leaves of W representing $U\$$ and $U\#$.

Similarly, let W^R be the compact trie for set Z_2 consisting of elements $V\$$ and $V\#$ for each $V\in Z_1 = \{(P[1\,..\,\lfloor b\rfloor])^R : b\in B\}\cup\{\varepsilon\}$. We preprocess the pair (W^R, Z_1) analogously to how we preprocess the pair (W, W_1).

Note that both tries W, W^R have $O(|B|)$ leaves. Let us now consider a 2D grid of size $O(|B|)\times O(|B|)$, whose x-coordinates (resp. y-coordinates) correspond to the leaves of W (resp. W^R). For each breakpoint $b\in B$ we do the following. Let x_1 and x_2 be the leaves with path-label $P[\lceil b\rceil\,..\,m]\$$ and $P[\lceil b\rceil\,..\,m]\#$ in W, respectively. Similarly, let y_1 and y_2 be the leaves with path-label $(P[1\,..\,\lfloor b\rfloor])^R\$$ and $(P[1\,..\,\lfloor b\rfloor])^R\#$ 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. 6. We then preprocess the grid for the counting version of 2D range stabbing queries, employing Theorem 5.1.

Query Without loss of generality, we may assume $i-\frac{1}{2}\le\beta\le j+\frac{1}{2}$; otherwise $\textsc{Count}_\beta(i,j)=0$. Let U be the longest string in W_1 that is a prefix of $T[\lceil\beta\rceil\,..\,j]$; this string can be determined in $O(\log\log n)$ time using a weighted ancestor query in

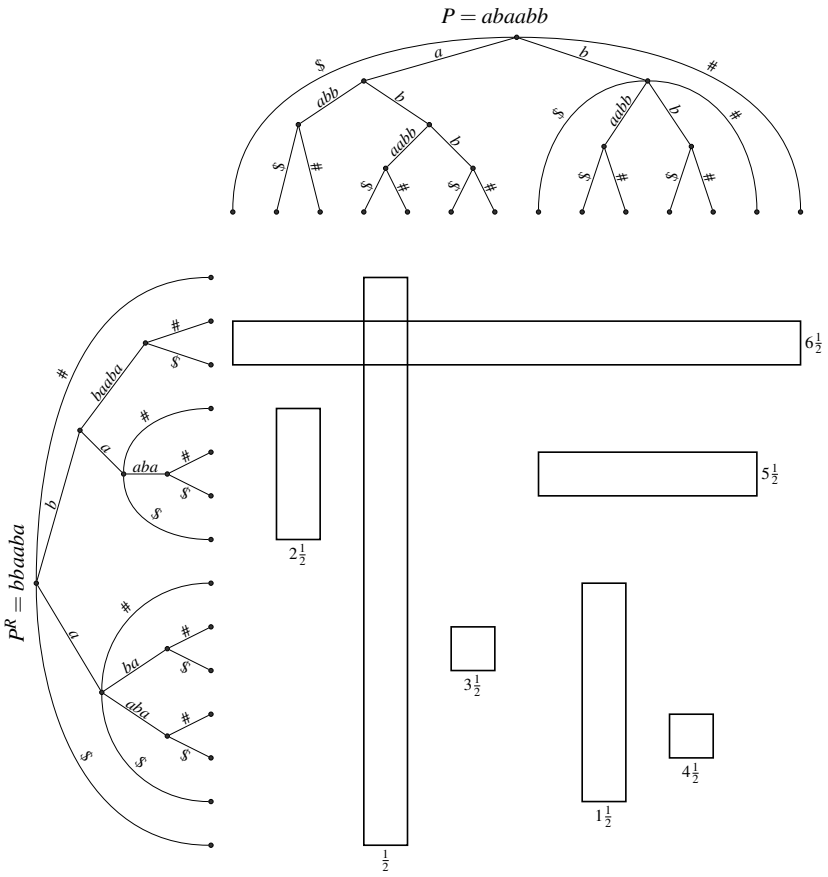

Fig. 6 Example of the construction of rectangles in the proof of Lemma 5.4 for $P = abaabb$ and breakpoints $i+1/2$ for $i = 0,1,2,3,4,5,6$. Each rectangle is annotated with its breakpoint.

the W_1-modified suffix tree of T. Moreover, let u' be the leaf of W representing $U\$$ (accessed through a pointer stored at U). We execute a symmetric procedure to find the longest string $V \in Z_1$ that is a prefix of $(T[i\,..\,\lfloor\beta\rfloor])^R$ and determine the leaf v' of W^R representing $V\$$.

Observation 5.3 *The number of fragments $T[r+1\,..\,t] = P$ satisfying $i \le r+1 \le t \le j$ and $\beta - r \in B$ 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 $P[\lceil b\rceil\,..\,m]$ is a prefix of $T[\lceil\beta\rceil\,..\,j]$ and $P[1\,..\,\lfloor b\rfloor]$ is a suffix of $T[i\,..\,\lfloor\beta\rfloor]$. This concludes the proof of the following result.

Lemma 5.4 BREAKPOINTS-ANCHOR IPM *queries can be answered in $O(\frac{\log n}{\log\log n})$ time with a data structure of size $O(n+|B|)$ that can be constructed in time $O(n+|B|\sqrt{\log|B|})$.*

Next, we define the analogous BREAKPOINTS-ANCHOR IDM problem, a variant of the Internal Dictionary Matching (IDM) problem.

BREAKPOINTS-ANCHOR IDM
Input: A length-n text T, a dictionary D of patterns, and for each pattern $P \in \mathsf{D}$, a set B_P of inter-positions (breakpoints) of P.
Query: $\textsc{Count}_\beta(i,j)$: for a given inter-position (anchor) β in T, the number of fragments $T[r+1\,..\,r+|P|]$ contained in $T[i\,..\,j]$ that satisfy $T[r+1\,..\,\lfloor\beta\rfloor] = P[1\,..\,\lfloor b\rfloor]$ and $T[\lceil\beta\rceil\,..\,r+|P|] = P[\lceil b\rceil\,..\,|P|]$ for some $P \in \mathsf{D}$ and $b \in B_P$. (Equivalently, $T[r+1\,..\,r+|P|] = P$ and $\beta - r \in B_P$.)

The following lemma is obtained by building a 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* $O(\frac{\log n}{\log\log n})$ *time with a data structure of size* $O(n+\sum_{P\in\mathsf{D}}|B_P|)$. *The data structure can be constructed in time* $O(n+\sqrt{\log n}\sum_{P\in\mathsf{D}}|B_P|)$.

A warm-up solution for $\textsc{Count}(i,j)$ Naively, Lemma 5.5 can be applied as follows to answer $\textsc{Count}(i,j)$ queries. Let us set $B_P = \{p+1/2 : p \in [1\,..\,|P|-1]\}$ for each pattern $P \in \mathsf{D}$ 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 $\mathsf{val}(v)$ 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 $|\mathsf{val}(v)| = 1$; otherwise, $\mathsf{val}(v) = \mathsf{val}(u_\ell)\mathsf{val}(u_r)$, where u_ℓ and u_r are the children of v. For each node v in BT, we precompute and store the count for $\mathsf{val}(v)$, defined as the number of occurrences of patterns from D in $\mathsf{val}(v)$. If v is a leaf, this count equals $|\{P \in \mathsf{D} : P = \mathsf{val}(v)\}|$ and can be determined easily due to $|\mathsf{val}(v)| = 1$. Otherwise, each occurrence is contained in $\mathsf{val}(u_\ell)$, is contained in $\mathsf{val}(u_r)$, or spans both $\mathsf{val}(u_\ell)$ and $\mathsf{val}(u_r)$. 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 $\mathsf{val}(v)$ with the anchor between $\mathsf{val}(u_\ell)$ and $\mathsf{val}(u_r)$.

To answer a query concerning $T[i\,..\,j]$, we recursively count the occurrences in the intersection of $\mathsf{val}(v)$ with $T[i\,..\,j]$, starting from the root r of BT as it satisfies $\mathsf{val}(r) = T[1\,..\,n]$. If the intersection is empty, the result is 0, and if $\mathsf{val}(v)$ 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 $\mathsf{val}(u_\ell)$ and $\mathsf{val}(u_r)$. 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 $\mathsf{val}(v)$ and $T[i\,..\,j]$ with the anchor between $\mathsf{val}(u_\ell)$ and $\mathsf{val}(u_r)$.

The query time is $O(\log^2 n/\log\log n)$ 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 $O(\log n)$.

5.2 Recompression

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

Each symbol A is also associated with its *parse tree* $\mathsf{PT}(A)$ consisting of a root with a *label* A to which zero or more subtrees are attached: if A is a terminal, there are no subtrees; if $A \to BC$ is a concatenation symbol, then $\mathsf{PT}(B)$ and $\mathsf{PT}(C)$ are attached; if $A \to B^k$ is a power symbol, then k copies of $\mathsf{PT}(B)$ are attached. Note that if we traverse the leaves of $\mathsf{PT}(A)$ from left to right, spelling out the corresponding non-terminals, then we obtain $\mathsf{g}(A)$. The parse tree PT of the whole RLSLP generating a string T is defined as $\mathsf{PT}(S)$ for the starting symbol S. We define the *value* $\mathsf{val}(v)$ of a node v in PT to be the fragment $T[a..b]$ corresponding to the leaves $T[a], \ldots, T[b]$ in the subtree of v. Note that $\mathsf{val}(v)$ is an occurrence of $\mathsf{g}(A)$, 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 Jeż [29,30] consists in the construction of a particular RLSLP generating the input text T. The underlying parse tree PT is of depth $O(\log n)$ and it can be constructed in $O(n)$ time. As observed by I [27], 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 ([27]) *If two fragments match, then their popped sequences are equal. Moreover, each popped sequence consists of $O(\log n)$ runs (maximal powers of a single symbol) and can be constructed in $O(\log n)$ 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 us now provide some more intuition. For any occurrence $T[a..b]$ of a string S in T, we have a chain of nodes whose labels can be represented compactly, whose values constitute $T[a..b]$, and whose labels are independent of what precedes or succeeds $T[a..b]$. This yields a uniform handle for all such occurrences.

Let $F_1^{p_1} \cdots F_t^{p_t}$ be the run-length encoding of the popped sequence of a substring S of T. If $|S| = 1$, then we set $L(S) = \emptyset$; otherwise, we define

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

By Theorem 5.6, the set $L(S)$ can be constructed in $O(\log n)$ time given an arbitrary occurrence of S in T.

Lemma 5.7 *Let v be a non-leaf node of PT and let $T[a..b]$ be an occurrence of S contained in $\mathsf{val}(v)$, but not contained in $\mathsf{val}(u)$ for any child u of v. If $T[a..c]$ is the longest prefix of $T[a..b]$ contained in $\mathsf{val}(u)$ 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 $\mathsf{val}(u)$ for a child u of v, then $|T[a..c']| \in L(S)$.*

Proof Consider the chain $v_1,\ldots,v_p$ of nodes in PT whose values constitute $T[a\,..\,b]$ and whose labels form the popped sequence of S. These nodes are descendants of v and, since $|S|>1$ guarantees $p>1$, they are proper descendants of v (i.e., descendants of children of v). Consequently, $T[a\,..\,c]=\mathsf{val}(v_1)\cdots\mathsf{val}(v_q)$, where $v_1,\ldots,v_q$ is the longest prefix of $v_1,\ldots,v_p$ consisting of descendants of the same child of v.

If the labels of v_q and v_{q+1} are distinct, then their labels belong to distinct runs in the popped sequence, and thus $|T[a\,..\,c]|\in L(U)$. (See Fig. 7 for an illustration of this case.)

Otherwise, Theorem 5.6 guarantees that v_q and v_{q+1} share the same parent. As they are descendants of different children of v, their parent must be v. Due to this, and since $v_1,\ldots,v_q$ form a chain consisting of descendants of the same child of v, we have $q=1$. Hence, $|T[a\,..\,c]|=|\mathsf{val}(v_1)|\in L(S)$.

The proof of the second claim is symmetric. □

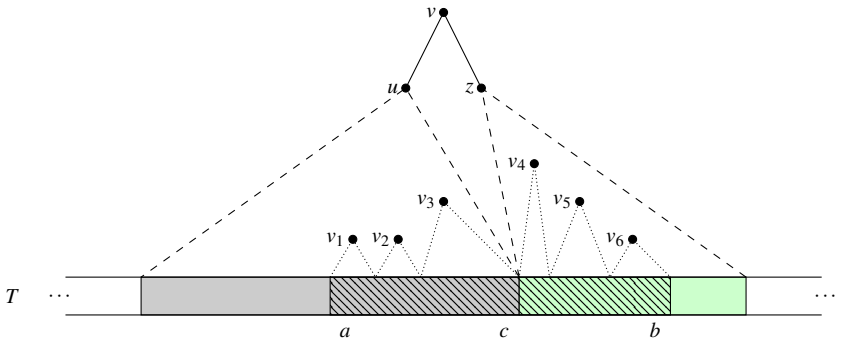

Fig. 7 Node v has two children, u and z. We denote $val(u)$ by a grey rectangle, and $\mathsf{val}(z)$ by a green rectangle. $T[a\,..\,c]$ is the longest prefix of $S=T[a\,..\,b]$ that is contained in $\mathsf{val}(u)$. Note that the labels of v_3 and v_4 must be distinct as these nodes do not share a single parent—one is a proper descendant of u, while the other is a proper descendant of z. Hence, $|T[a\,..\,c]|\in L(S)$.

Data structure We use recompression to build an RLSLP generating T and the underlying parse tree PT. We also construct the data structure for BREAKPOINTS-ANCHOR IDM queries of Lemma 5.5 with breakpoints $B_P=\{i+\frac{1}{2}:i\in L(P)\}$ for each pattern $P\in\mathsf{D}$. Moreover, for every symbol A we store the number of occurrences of patterns from D in $\mathsf{g}(A)$. Additionally, if $A\to B^k$ is a power, we also store the number of occurrences in $\mathsf{g}(B^i)$ for $i\in[1\,..\,k]$. The space consumption is $O(n+d\log n)$ since $|B_P|=O(\log n)$ for each $P\in\mathsf{D}$.

Efficient preprocessing The RLSLP and the parse tree are built in $O(n)$ time, and the sets B_P are determined in $O(d\log n)$ time using Theorem 5.6. The data structure of Lemma 5.5 is then constructed in $O(n+d\log^{3/2}n)$ time.

Next, we process the RLSLP in a bottom-up fashion. If A is a terminal, its count is easily determined. If $A\to BC$ is a concatenation, we sum the counts for B and

C and the number of occurrences spanning both $\mathsf{g}(B)$ and $\mathsf{g}(C)$. 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 $\mathsf{val}(u_\ell)$ and $\mathsf{val}(u_r)$ 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 $\mathsf{val}(v)$ with the anchor between $\mathsf{val}(u_\ell)$ and $\mathsf{val}(u_r)$. Finally, if $A \to B^k$, then to determine the count in $\mathsf{g}(B^i)$, we add the count for B, the count in $\mathsf{g}(B^{i-1})$, 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 $u_1, \ldots, u_k$, and make a BREAKPOINTS-ANCHOR IDM query in $\mathsf{val}(u_1)\cdots\mathsf{val}(u_i)$ with the anchor between $\mathsf{val}(u_1)$ and $\mathsf{val}(u_2)$. The correctness of this step follows from Lemma 5.7. The running time of this processing is $O(n\log n/\log\log n)$, so the overall construction time is $O(n\log n/\log\log n + d\log^{3/2} n)$.

Query Upon a query $\textsc{Count}(i,j)$, we proceed essentially as in the warm-up solution: we recursively count the occurrences contained in the intersection of $T[i..j]$ with $\mathsf{val}(v)$ for nodes v in PT, starting from the root of PT. If the two fragments are disjoint, the result is 0, and if $\mathsf{val}(v)$ 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 $\mathsf{val}(u_\ell)$ and $\mathsf{val}(u_r)$, we also add the result of a BREAKPOINTS-ANCHOR IDM query in the intersection of $T[i..j]$ with $\mathsf{val}(v)$ and the anchor between $\mathsf{val}(u_\ell)$ and $\mathsf{val}(u_r)$. If the label is a power symbol $A \to B^k$, we determine which of the children $u_1, \ldots, u_k$ of v are spanned by $T[i..j]$. We denote these children by $u_\ell, \ldots, u_r$ 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 $\mathsf{val}(u_\ell)\cdots\mathsf{val}(u_r)$ and anchor between $\mathsf{val}(u_\ell)$ and $\mathsf{val}(u_{\ell+1})$. If $r > \ell+1$, we further add the precomputed value for $\mathsf{g}(B^{r-\ell-1})$ to account for the occurrences contained in $\mathsf{val}(u_{\ell+1})\cdots\mathsf{val}(u_{r-1})$ and make a BREAKPOINTS-ANCHOR IDM query in the intersection of $T[i..j]$ with $\mathsf{val}(u_{\ell+1})\cdots\mathsf{val}(u_r)$ 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 $O(\log^2 n/\log\log n)$ since we make $O(\log n)$ non-trivial recursive calls and each of them is processed in $O(\log n/\log\log n)$ time.

We arrive at the main result of this section.

Theorem 5.8 $\textsc{Count}(i,j)$ *queries can be answered in* $O(\log^2 n/\log\log n)$ *time with a data structure of size* $O(n + d\log n)$ *that can be constructed in* $O(n\log n/\log\log n + d\log^{3/2} n)$ *time.*

6 Dynamic dictionaries

In this section, we study the IDM problem in the case that insertions to and deletions from the dictionary are allowed. First, we present a conditional lower bound for this problem that is based on the conjectured hardness of a known problem in the online setting. In the remainder, we focus on providing algorithms matching this conditional lower bound.

6.1 Conditional lower bound

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 $r_1, \ldots, r_n$, 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 [26]) For any constant $\varepsilon > 0$, there is no $O(n^{3-\varepsilon})$-time algorithm that solves OMv correctly with probability at least 2/3.

We now present a restricted version of [26, Theorem 2.2] which is sufficient for our purposes.

Theorem 6.2 ([26]) *For all constants $\gamma, \varepsilon > 0$, the OMv conjecture implies that there is no algorithm 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 $O(r_2^{1+\gamma-\varepsilon})$ with error probability at most $1/3$.*

We proceed to obtain an OMv-based conditional lower bound for IDM in the case of a dynamic dictionary. Our lower bound is stated for $\textsc{Exists}(i, j)$ queries, but it clearly carries over to the remaining query types considered in this work.

Theorem 6.3 *For all constants $\alpha, \beta > 0$ with $\alpha + \beta < 1$, the OMv conjecture implies that there is no algorithm that preprocesses T and D in time polynomial in n, performs insertions to D in time $O(n^{\alpha})$, and answers $\textsc{Exists}(i, j)$ queries in time $O(n^{\beta})$ in an online manner with error probability at most $1/3$.*

Proof Let us suppose that there is such an algorithm and set $\gamma = (\alpha + \varepsilon/2)/(\beta + \varepsilon/2)$, where $\varepsilon = 1 - \alpha - \beta$. Given an $r_1 \times r_2$ matrix M satisfying $r_1 = \lfloor r_2^{\gamma} \rfloor$, we construct a text T of length $n = r_1 r_2 + r_2$ as follows. Let T' be a text created by concatenating the rows of M in increasing order. Then, T is obtained by assigning to each non-zero element of T' the column index of the matrix entry it originates from, and appending one by one the integers in $[1\,..\,r_2]$ in increasing order. Formally, for $i \in [1\,..\,r_1 r_2]$, let $a[i] = \lceil i/r_2 \rceil$ and $b[i] = 1 + (i-1) \bmod r_2$, and set $T[i] = b[i] \cdot M[a[i], b[i]]$; for $i \in [r_1 r_2 + 1\,..\,r_1 r_2 + r_2]$, set $T[i] = i - r_1 r_2$. (We append these letters in order to ensure that the dictionary that we construct below is internal.)

We compute Mv as follows. We add the indices of v's non-zero entries into an initially empty dictionary. We then perform queries $\textsc{Exists}(1 + (t-1)r_2, tr_2)$ for $t \in [1\,..\,r_1]$. The answer to the query $\textsc{Exists}(1 + (t-1)r_2, tr_2)$ is equal to the boolean dot product of the tth row of M with v. We thus obtain Mv, with each entry correct with probability at least $2/3$. We can guarantee that the whole vector Mv is correct with probability at least $1 - n^{-\Omega(1)} \geq 2/3$ by maintaining $\Theta(\log n)$ independent instances of the algorithm and taking the dominant answer to each $\textsc{Exists}$ query. In total, we perform $\tilde{O}(r_2)$ insertions to D and $\tilde{O}(r_1)$ $\textsc{Exists}$ queries. Thus, the total time required is $\tilde{O}(r_2 n^{\alpha} + r_1 n^{\beta}) = \tilde{O}(n^{\beta+\varepsilon/2} n^{\alpha} + n^{\alpha+\varepsilon/2} n^{\beta}) = \tilde{O}(n^{1-\varepsilon/2}) = O(r_2^{1+\gamma-\varepsilon'})$ for $\varepsilon' > 0$. Conjecture 6.1 would be disproved due to Theorem 6.2. □

Example 6.4 For the matrix

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

we construct the text $T = 1030003002001234$. For the vector $v = \begin{bmatrix} 1\ 1\ 0\ 1 \end{bmatrix}^T$, the dictionary is $\mathsf{D} = \{1,2,4\}$. The answers to $\textsc{Exists}(1,4)$, $\textsc{Exists}(5,8)$, $\textsc{Exists}(9,12)$ queries are **true**, **false**, **true**, respectively, which corresponds to $Mv = \begin{bmatrix} 1\ 0\ 1 \end{bmatrix}^T$.

Remark 6.5 The proof of the above theorem requires a large alphabet. Let us describe a straightforward modification to this proof that yields the same lower bound for ternary alphabets. We add a \$ between every pair of consecutive letters in T and then replace each integer in T with its binary representation. Similarly, the patterns we add to or remove from D are the same integers as above, but represented in binary.

6.2 Upper bound

Before we proceed solving the dynamic IDM problem, let us recall known results for internal pattern matching queries. In this setting, each query specifies a fragment $T[i..j]$ and a substring P of the text and asks questions 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 $\textsc{Count}(P,i,j)$ queries as follows, similar to [36]. We construct the suffix tree $\mathbf{T}(T)$ 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 $\mathscr{G}$, 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 $\textsc{Count}(P,i,j)$ as follows. We first locate the locus of P in $\mathbf{T}(T)$ using a weighted ancestor query in $\tilde{O}(1)$ time and retrieve the associated lexicographic range $[\ell..r]$. Next, we perform a counting query for the range $[i..j-|P|+1] \times [\ell..r]$ of $\mathscr{G}$, which returns the desired count; see also [37]. The same approach—with range reporting instead of range counting—can be used for $\textsc{Report}(P,i,j)$ queries.

Theorem 6.6 ([31,36]) *Given a text T of length n, we can construct in $\tilde{O}(n)$ time an $\tilde{O}(n)$-size data structure that, given a pattern P specified as a fragment of T and positions i, j, can answer queries of the form $\textsc{Exists}(P,i,j)$, $\textsc{Report}(P,i,j)$, and $\textsc{Count}(P,i,j)$ in time $\tilde{O}(1+|output|)$.*

We are now ready to solve the dynamic IDM problem. Let us denote the initial dictionary by D^0. Further, let $u_1, u_2, \ldots$ be the sequence of dictionary updates and D^r be the dictionary after update u_r. Each update is an insertion or a deletion of a pattern in D. We first discuss how to answer $\textsc{ReportDistinct}$ 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 a dictionary D^r for $r = t - O(m)$. 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 keep two copies of our data structure, switching their roles after (roughly) every $m/2$ updates. One copy is for handling at most $m/2$ updates and answering queries, while the other one is reinitialized in chunks in the background. We also store updates $u_{r+1},\ldots,u_t$ (or the differences between D^r and D^t).

To answer a REPORTDISTINCT query, we do the following:

1. use the static data structure to answer the REPORTDISTINCT query for D^r;
2. filter out the $O(m)$ reported patterns that are in $\mathsf{D}^r \setminus \mathsf{D}^t$;
3. search for the $O(m)$ patterns in $\mathsf{D}^t \setminus \mathsf{D}^r$ individually in $\tilde{O}(1)$ time per pattern by performing internal pattern matching queries, employing Theorem 6.6.

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

Proposition 6.7 *For every text T of length n and integer parameter $m \geq 1$, a dynamic internal dictionary D can be maintained in $O(n+|\mathsf{D}|)$ space, supporting $\tilde{O}(n+|\mathsf{D}|)$-time initialization, $\tilde{O}(1+(n+|\mathsf{D}|)/m)$-time updates, and $\tilde{O}(m+|output|)$-time queries* REPORTDISTINCT(i,j).

We next show how to improve the update time to $O(1+n/m)$ matching, up to subpolynomial terms, the lower bound of Theorem 6.3.

We store D in an array D of size n, which consists in collections of total size d. $D[p]$ stores the elements of D whose leftmost occurrence in T is at position p in a min heap with respect to their lengths. In fact, as all elements stored in $D[p]$ are prefixes of $T[p..n]$, it suffices to store the length of each element. We can find the desired position p for a pattern $P \in \mathsf{D}$ in $O(\log\log n)$ time by locating its locus in $\mathbf{T}(T)$ using a weighted ancestor query; we can have precomputed the leftmost occurrence of the path-label of each explicit node of $\mathbf{T}(T)$ in a DFS traversal. We can initialize D in $O(n+|\mathsf{D}^0|)$ time by answering all weighted ancestor queries as a batch [33, Section 7].

The dictionary $\mathsf{D}' = \{\min D[p] : 1 \leq p \leq n\}$, where $\min D[p]$ is the shortest pattern whose length is stored in $D[p]$, is of size $O(n)$. Let P be a pattern inserted to or deleted from D, and let p be the position of the leftmost occurrence of P in T. For such an update in D, we update $D[p]$ and (possibly) update D' as follows.

- *Case I: P is inserted to* D. If P is shorter than $\min D[p]$ or $D[p]$ is an empty collection, then P is inserted to D' and $\min D[p]$ (if it exists) is deleted from D'. Finally, P is also inserted to the collection $D[p]$.
- *Case II: P is deleted from* D. If $P = \min D[p]$, then P is deleted from D', P is deleted from $D[p]$, and the new $\min D[p]$ (if any) is inserted to D'. Otherwise (if $P \neq \min D[p]$), P is only deleted from $D[p]$.

Observe that if $P \in D[p]$ occurs in $T[i..j]$, then $\min D[p]$ also occurs in $T[i..j]$. We thus maintain D' using the solution of Proposition 6.7. In order to answer a REPORTDISTINCT(i,j) query for D, we first answer an analogous query for D'. For

each reported pattern $\min D[p] \in \mathsf{D}'$, we iterate over patterns $P \in D[p]$ in the order of increasing lengths, as long as P occurs $T[i..j]$; we check whether this is the case using Theorem 6.6. Note that the relevant elements of each collection $D[p]$ are retrieved efficiently in the right order since $D[p]$ is implemented as a min heap.

REPORT(i,j) We first perform a REPORTDISTINCT(i,j) query and then find all occurrences of each returned pattern in $T[i..j]$ in time $\tilde{O}(1+|output|)$ by Theorem 6.6.

EXISTS(i,j) We observe that the answer for D is the same as for D'. For the latter dictionary, we use the static version of COUNT(i,j), presented in Section 5, and the counting version of internal pattern matching (Theorem 6.6) for the removed/inserted patterns, 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 COUNT(i,j) queries for dictionary D^0. For the subsequent m updates, we use this data structure to answer COUNT(i,j) queries, treating individually the inserted/removed patterns using Theorem 6.6. These queries are thus answered in $\tilde{O}(m)$ time.

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

As for 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 $\mathscr{G}$ to be of size $K \times K$ for $K = O(n^2)$ and identify x-coordinate i with the ith smallest element of the set $W = \{Ux : U \text{ is a substring of } T \text{ and } x \in \{\$,\#\}\}$. (Similarly for y-coordinates and T^R.)

In order to compute the rectangles in $\mathscr{G}$ and transform queries to points as in the original solution, we need to be able to efficiently transform strings in W to their ranks. Let us show how to preprocess the suffix tree $\mathbf{T}(T)$ in $O(n)$ time so that the rank of a given string $T[a..b]\$$ or $T[a..b]\#$ in W can be computed in $\tilde{O}(1)$ time. Let us assume that $\mathbf{T}(T)$ has been built for T, without the \$ appended to it. We make a DFS traversal of $\mathbf{T}(T)$, maintaining a global counter cr, which is initialized to zero at the root. The DFS visits the children of a node in the left-to-right order. When traversing an edge, we increment cr by the length of the path-label of this edge. When an explicit node v with path-label S is visited for the *first* time, we set the rank of $S\$$ equal to cr, and when v is visited for the *last* time, we set the rank of $S\#$ to $\mathsf{cr}+1$. Let q be the locus of $T[a..b]$ in $\mathbf{T}(T)$, which can be computed in $O(\log\log n)$ 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 $S\$$ and $S\#$, where S is the path-label of the nearest explicit descendant v of q by, respectively, subtracting and adding the distance between v and q.

Thus, rectangles and points from the proof of Lemma 5.4 are maintained in the $K \times K$ grid $\mathscr{G}$. After m updates, we remove (resp. add) the $\tilde{O}(m)$ rectangles corresponding to patterns in $\mathsf{D}^0 \setminus D^m$ (resp. $\mathsf{D}^m \setminus D^0$). Finally, instead of using Theorem 5.1, we can

maintain a data structure of size $O(r)$ for the counting version of range stabbing in a 2D grid of size $K \times K$ with r rectangles with $O(\log K \log r/(\log\log r)^2)$ time per update and query [25]. Since we are not optimizing $\tilde{O}(1)$ factors in the complexity, range trees (cf. [10]) can also be used for 2D range queries to avoid randomization.

To wrap up, updating the data structure every m updates to D 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 6.8 *For every text T of length n and parameter $m \in [1\,..\,n]$, a dynamic internal dictionary D can be maintained in $\tilde{O}(n+|\mathsf{D}|)$ space, supporting $\tilde{O}(n+|\mathsf{D}|)$-time initialization, $\tilde{O}(n/m)$-time updates, and $\tilde{O}(m+|output|)$-time queries* $\textsc{Exists}(i,j)$, $\textsc{Report}(i,j)$, $\textsc{ReportDistinct}(i,j)$, *and* $\textsc{Count}(i,j)$.

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

Funding

Panagiotis Charalampopoulos was partially supported by ERC grant TOTAL (no. 677651) under the EU's Horizon 2020 Research and Innovation Programme. Tomasz Kociumaka was supported by ISF grants no. 1278/16 and 1926/19, a BSF grant no. 2018364, and an ERC grant MPM (no. 683064) under the EU's Horizon 2020 Research and Innovation Programme. Jakub Radoszewski and Tomasz Waleń are supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.

Conflict of interest

Not applicable.

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 *26th 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 *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. Jon Louis Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23(4):214–229, 1980. doi:10.1145/358841.358850.
11. Ho-Leung Chan, Wing-Kai Hon, Tak Wah Lam, and Kunihiko Sadakane. Dynamic dictionary matching and compressed suffix trees. In *16th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005*, pages 13–22. SIAM, 2005. URL: http://dl.acm.org/citation.cfm?id=1070432.1070436.
12. Timothy M. Chan and Mihai Pătraşcu. Counting inversions, offline orthogonal range counting, and related problems. In *21st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010*, pages 161–173, 2010. doi:10.1137/1.9781611973075.15.
13. Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszyński, Tomasz Waleń, and Wiktor Zuba. Counting distinct patterns in internal dictionary matching. In *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020*, volume 161 of *LIPIcs*, pages 8:1–8:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.CPM.2020.8.
14. Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal Dictionary Matching. In *30th International Symposium on Algorithms and Computation (ISAAC 2019)*, volume 149 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:17. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ISAAC.2019.22.
15. David Clark. *Compact Pat trees*. PhD thesis, University of Waterloo, 1996. URL: http://hdl.handle.net/10012/64.
16. Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *36th Annual ACM Symposium on Theory of Computing, STOC 2004*, pages 91–100. ACM, 2004. doi:10.1145/1007352.1007374.
17. Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007. doi:10.1017/cbo9780511546853.
18. 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.
19. 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.
20. N. J. Fine and H. S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16:109–114, 1965.
21. 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.
22. 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.
23. Richard Groult, Élise Prieur, and Gwénaël 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.
24. 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.
25. 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.
26. Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *47th Annual ACM on Symposium on Theory of Computing, STOC 2015*, pages 21–30. ACM, 2015. doi:10.1145/2746539.2746609.
27. Tomohiro I. Longest common extensions with recompression. In *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.

28. 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.
29. Artur Jeż. Faster fully compressed pattern matching by recompression. *ACM Transactions on Algorithms*, 11(3):20:1–20:43, 2015. doi:10.1145/2631920.
30. 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.
31. 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.
32. 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.
33. Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A linear-time algorithm for seeds computation. *ACM Transactions on Algorithms*, 16(2):27:1–27:23, 2020. doi:10.1145/3386369.
34. Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal pattern matching queries in a text and applications. In *26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 532–551. SIAM, 2015. doi:10.1137/1.9781611973730.36.
35. 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.
36. Tsvi Kopelowitz, Moshe Lewenstein, and Ely Porat. Persistency in suffix trees with applications to string interval problems. In *18th International Symposium on String Processing and Information Retrieval, SPIRE 2011*, volume 7024 of *LNCS*, pages 67–80. Springer, 2011. doi:10.1007/978-3-642-24583-1_8.
37. 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.
38. 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.
39. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *13th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2002*, pages 657–666. SIAM, 2002. URL: http://dl.acm.org/citation.cfm?id=545381.545469.
40. Mihai Pătraşcu. Unifying the landscape of cell-probe lower bounds. *SIAM Journal on Computing*, 40(3):827–847, 2011. doi:10.1137/09075336X.
41. Mikhail Rubinchik and Arseny M. Shur. Counting palindromes in substrings. In *24th International Symposium on String Processing and Information Retrieval, SPIRE 2017*, volume 10508 of *LNCS*, pages 290–303. Springer, 2017. doi:10.1007/978-3-319-67428-5_25.