Counting Distinct Patterns in Internal Dictionary Matching

Panagiotis Charalampopoulos
King's College London, UK
University of Warsaw, Poland
panagiotis.charalampopoulos@kcl.ac.uk

Tomasz Kociumaka
Bar-Ilan University, Ramat Gan, Israel
kociumaka@mimuw.edu.pl

Manal Mohamed
London, UK
manalabd@gmail.com

Jakub Radoszewski
University of Warsaw, Poland
Samsung R&D, Warsaw, Poland
jrad@mimuw.edu.pl

Wojciech Rytter
University of Warsaw, Poland
rytter@mimuw.edu.pl

Juliusz Straszyński
University of Warsaw, Poland
jks@mimuw.edu.pl

Tomasz Waleń
University of Warsaw, Poland
walen@mimuw.edu.pl

Wiktor Zuba
University of Warsaw, Poland
w.zuba@mimuw.edu.pl

Abstract

We consider the problem of preprocessing a text T of length n and a dictionary $\mathcal{D}$ in order to be able to efficiently answer queries CountDistinct(i, j), that is, given i and j return the number of patterns from $\mathcal{D}$ that occur in the *fragment* $T[i\,..\,j]$. The dictionary is *internal* in the sense that each pattern in $\mathcal{D}$ is given as a fragment of T. This way, the dictionary takes space proportional to the number of patterns $d = |\mathcal{D}|$ rather than their total length, which could be $\Theta(n \cdot d)$. An $\tilde{\mathcal{O}}(n+d)$-size [1] data structure that answers CountDistinct(i, j) queries $\mathcal{O}(\log n)$-approximately in $\tilde{\mathcal{O}}(1)$ time was recently proposed in a work that introduced internal dictionary matching [ISAAC 2019]. Here we present an $\tilde{\mathcal{O}}(n+d)$-size data structure that answers CountDistinct(i, j) queries 2-approximately in $\tilde{\mathcal{O}}(1)$ time. Using range queries, for any m, we give an $\tilde{\mathcal{O}}(\min(nd/m, n^2/m^2) + d)$-size data structure that answers CountDistinct(i, j) queries exactly in $\tilde{\mathcal{O}}(m)$ time. We also consider the special case when the dictionary consists of all square factors of the string. We design an $\mathcal{O}(n \log^2 n)$-size data structure that allows us to count distinct squares in a text fragment $T[i\,..\,j]$ in $\mathcal{O}(\log n)$ time.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases dictionary matching, internal pattern matching, squares

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.8

Related Version Full version at `https://arxiv.org/abs/2005.05681`.

Funding *Panagiotis Charalampopoulos*: Partially supported by ERC grant TOTAL under the EU's Horizon 2020 Research and Innovation Programme (agreement no. 677651).
Tomasz Kociumaka: 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: Supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.
Juliusz Straszyński: Supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.
Tomasz Waleń: Supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.
Wiktor Zuba: Supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.

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

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).
Editors: Inge Li Gørtz and Oren Weimann; Article No. 8; pp. 8:1–8:15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Internal Dictionary Matching was recently introduced in [5] as a generalization of Internal Pattern Matching. In the classical Dictionary Matching problem, 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 can efficiently compute the occurrences of the patterns from $\mathcal{D}$ in T. In Internal Dictionary Matching, the text T is given in advance, the dictionary $\mathcal{D}$ is a set of fragments of T, and the Dictionary Matching queries can be asked for any fragment of T.

The Internal Pattern Matching problem consists in preprocessing a text T of length n so that we can efficiently compute the occurrences of a fragment of T in another fragment of T. A data structure of nearly linear size that allows for sublogarithmic-time Internal Pattern Matching queries was presented in [15], while a linear-size data structure allowing for constant-time Internal Pattern Matching queries in the case that the ratio between the lengths of the two factors is constant was presented in [18]. Other types of internal queries have been also studied; we refer the interested reader to [17].

In [5], several types of Internal Dictionary Matching queries about fragments $T[i\,..\,j]$ in a string T were considered: $\textsc{Exists}(i,j)$, $\textsc{Report}(i,j)$, $\textsc{ReportDistinct}(i,j)$, $\textsc{Count}(i,j)$, $\textsc{CountDistinct}(i,j)$. Data structures of size $\tilde{\mathcal{O}}(n+d)$ and query time $\tilde{\mathcal{O}}(1+\mathsf{output})$ were shown for answering each of the first four queries, with $\textsc{Count}$ queries requiring most advanced techniques. For $\textsc{CountDistinct}$ queries, only a data structure answering these queries $\mathcal{O}(\log n)$-approximately was shown. In this work, we focus on more efficient data structures for such queries. $\textsc{CountDistinct}$ queries are formally defined as follows.

$\textsc{CountDistinct}$
Input: A text T of length n and a dictionary $\mathcal{D}$ consisting of d patterns, each given as a fragment $T[a\,..\,b]$ of T (represented only by integers a, b).
Query: $\textsc{CountDistinct}(i,j)$: Count all distinct patterns $P \in \mathcal{D}$ that occur in $T[i\,..\,j]$.

Observe that the input size is $n+d$, while the total length of strings in $\mathcal{D}$ could be $\Theta(n \cdot d)$.

We also consider a special case of this problem when the dictionary $\mathcal{D}$ is the set of all squares (i.e., strings of the form UU) in T. The case that $\mathcal{D}$ is the set of palindromes in T was considered by Rubinchik and Shur in [20].

▶ **Example 1.** Let us consider the following text:

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

For the dictionary $\mathcal{D} = \{\mathtt{aa}, \mathtt{aaaa}, \mathtt{abba}, \mathtt{c}\}$, we have:

$$\textsc{CountDistinct}(5,12) = 2,\ \textsc{CountDistinct}(2,6) = 2,\ \textsc{CountDistinct}(2,12) = 3.$$

In particular, $T[5\,..\,12]$ contains two distinct patterns from $\mathcal{D}$: `aa` (two occurrences) and `abba`. When the dictionary $\mathcal{D}$ represents all squares in T, we have

$$\textsc{CountDistinct}(5,12) = 3,\ \textsc{CountDistinct}(2,6) = 2,\ \textsc{CountDistinct}(2,12) = 4.$$

In particular, $T[5\,..\,12]$ contains three distinct squares: `aa` (two occurrences), `bb` and `aabaab`.

Let us note that one could answer $\textsc{CountDistinct}(i,j)$ queries in time $\mathcal{O}(j-i)$ by running $T[i\,..\,j]$ over the Aho–Corasick automaton of $\mathcal{D}$ [1] or in time $\tilde{\mathcal{O}}(d)$ by performing Internal Pattern Matching [18] for each element of $\mathcal{D}$ individually. Neither of these approaches is satisfactory as they can require $\Omega(n)$ time in the worst case.

Our results and a roadmap. We start with preliminaries in Section 2 and an algorithmic toolbox in Section 3. Our results for the case of a static dictionary are summarized in Table 1. Our solutions exploit string periodicity using runs and use data structures for variants of the (colored) orthogonal range counting problem and for auxiliary internal queries on strings.

Table 1 Our results for CountDistinct queries. Here, m is an arbitrary parameter.

Space	Preprocessing time	Query time	Variant	Section
$\tilde{\mathcal{O}}(n+d)$	$\tilde{\mathcal{O}}(n+d)$	$\tilde{\mathcal{O}}(1)$	2-approximation	4
$\tilde{\mathcal{O}}(n^2/m^2+d)$	$\tilde{\mathcal{O}}(n^2/m+d)$	$\tilde{\mathcal{O}}(m)$	exact	5.1
$\tilde{\mathcal{O}}(nd/m+d)$	$\tilde{\mathcal{O}}(nd/m+d)$	$\tilde{\mathcal{O}}(m)$	exact	5.2
$\mathcal{O}(n\log^2 n)$	$\mathcal{O}(n\log^2 n)$	$\mathcal{O}(\log n)$	$\mathcal{D}=$ squares, exact	6

For the case of a dynamic dictionary, where queries are interleaved with insertions and deletions of patterns in the dictionary, it was shown in [5] that the product of the time to process an update and the time to answer an Exists(i,j) query cannot be $\mathcal{O}(n^{1-\epsilon})$ for any constant $\epsilon>0$, unless the Online Boolean Matrix-Vector Multiplication conjecture [13] is false. In the full version of this paper, we outline a general scheme that adapts our data structures for the case of a dynamic dictionary. In particular, we show how to answer CountDistinct(i,j) queries 2-approximately in $\tilde{\mathcal{O}}(m)$ time and process each update in $\tilde{\mathcal{O}}(n/m)$ time, for any m.

2 Preliminaries

We begin with basic definitions and notation. 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* of T that starts at position i and ends at position j (the fragment is empty if $j<i$). A fragment 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 and a *suffix* is a fragment that ends at position n. By UV and U^k we denote the concatenation of strings U and V and k copies of the string U, respectively. A *cyclic rotation* of a string U is any string V such that $U=XY$ and $V=YX$ for some strings X and Y.

Let U be a string of length m with $0<m\leq n$. We say that U is a *factor* of T if there exists a fragment $T[i\,..\,i+m-1]$, called an *occurrence* of U in T, that is matches U. We then say that U occurs at the *starting position* i in T.

A positive integer p is called a *period* of T if $T[i]=T[i+p]$ for all $i=1,\ldots,n-p$. We refer to the smallest period as *the period* of the string, and denote it by $\mathsf{per}(T)$. A string is called *periodic* if its period is no more than half of its length and *aperiodic* otherwise. The weak version of the periodicity lemma [9] states that if p and q are periods of a string T and satisfy $p+q\leq|T|$, then $\gcd(p,q)$ is also a period of T. A string T is called *primitive* if it cannot be expressed as U^k for a string U and an integer $k>1$.

The elements of the dictionary $\mathcal{D}$ are called *patterns*. Henceforth, we assume that $\varepsilon\notin\mathcal{D}$, i.e., that the length of each $P\in\mathcal{D}$ is at least 1. We also 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 patterns in $\mathcal{D}$ and not their total length. A *compact trie* of $\mathcal{D}$ is the trie of $\mathcal{D}$ in which all non-terminal nodes with exactly one child become implicit. The path-label $\mathcal{L}(v)$ of a node v is defined as the path-ordered concatenation of the string-labels of the edges in the root-to-v path. We refer to $|\mathcal{L}(v)|$ as the *string-depth* of v.

3 Algorithmic Tools

3.1 Modified Suffix Trees

A $\mathcal{D}$-*modified suffix tree* [5], denoted as $\mathcal{T}_{T,\mathcal{D}}$, of a given text T of length n and a dictionary $\mathcal{D}$ is obtained from the trie of $\mathcal{D} \cup \{T[i\,..\,n] : 1 \le i \le n\}$ by contracting, for each non-terminal node u other than the root, the edge from u to the parent of u. As a result, all the nodes of $\mathcal{T}_{T,\mathcal{D}}$ (except for the root) correspond to patterns in $\mathcal{D}$ or to suffixes of T. For $1 \le i \le n$, the node representing $T[i\,..\,n]$ is labelled with i; see Figure 1. For a dictionary $\mathcal{D}$ whose patterns are given as fragments of a text T, we can construct $\mathcal{T}_{T,\mathcal{D}}$ in $\mathcal{O}(|\mathcal{D}| + |T|)$ time [5].

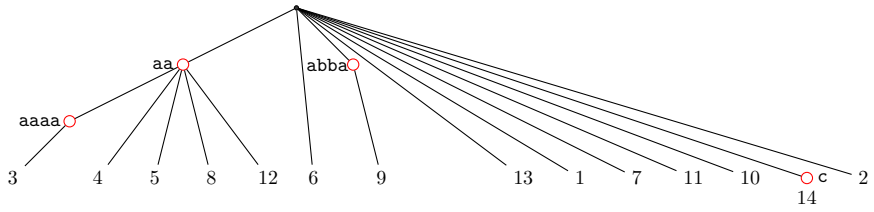

Figure 1 Example of a $\mathcal{D}$-modified suffix tree for text T = `adaaaabaabbaac` and dictionary $\mathcal{D} = \{\texttt{aa}, \texttt{aaaa}, \texttt{abba}, \texttt{c}\}$ (figure from [5]).

Let us denote by $\mathsf{Occ}(\mathcal{D})$ the set of all occurrences of dictionary patterns in T, that is, the set of all fragments of T that match a pattern in $\mathcal{D}$. Using $\mathcal{T}_{T,\mathcal{D}}$, the set $\mathsf{Occ}(\mathcal{D})$ can be computed in time $\mathcal{O}(n + d + |\mathsf{Occ}(\mathcal{D})|)$.

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 top-most ancestor of u with weight of at least ℓ.

▶ **Theorem 2** ([2, Section 6.2.1]). *After $\mathcal{O}(n)$-time preprocessing, weighted ancestor queries for nodes of a weighted tree $\mathcal{T}$ of size n can be answered in $\mathcal{O}(\log\log n)$ time per query.*

The $\mathcal{D}$-modified suffix tree $\mathcal{T}_{T,\mathcal{D}}$ is a weighted tree with the weight of each node defined as the length of the corresponding string. We define the *locus* of a fragment $T[i\,..\,j]$ in $\mathcal{T}_{T,\mathcal{D}}$ to be the weighted ancestor of the leaf i at string-depth $j - i + 1$.

3.2 Auxiliary Internal Queries

In a *Bounded LCP* query, one is given two fragments U and V of T and needs to return the longest prefix of U that occurs in V; we denote such a query by $\mathit{BoundedLCP}(U, V)$. Kociumaka et al. [18] presented several tradeoffs for this problem, including the following.

▶ **Lemma 3** ([18],[17, Corollary 7.3.4]). *Given a text T of length n, one can construct in $\mathcal{O}(n\sqrt{\log n})$ time an $\mathcal{O}(n)$-size data structure that answers Bounded LCP queries in $\mathcal{O}(\log^{\epsilon} n)$ time, for any constant $\epsilon > 0$.*

Recall that $\textsc{Count}(i, j)$ returns the number of all occurrences of all the patterns of $\mathcal{D}$ in $T[i\,..\,j]$. The following result was proved in [5].

▶ **Lemma 4** ([5]). *The $\textsc{Count}(i, j)$ queries can be answered in $\mathcal{O}(\log^2 n/\log\log n)$ time with an $\mathcal{O}(n + d\log n)$-size data structure, constructed in $\mathcal{O}(n\log n/\log\log n + d\log^{3/2} n)$ time.*

3.3 Geometric Toolbox

For a set of n points in 2D, a range counting query returns the number of points in a given rectangle.

▶ **Theorem 5** (Chan and Pătraşcu [4]). *Range counting queries for n integer points in 2D 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 $\mathcal{O}(n\sqrt{\log n})$.*

A quarterplane is a range of the form $(-\infty, x_1] \times (-\infty, x_2]$. By reversing coordinates we can also consider quarterplanes with some dimensions of the form $[x_i, \infty)$. Let us state the following result on orthant color range counting due to Kaplan et al. [14] in the special case of two dimensions.

▶ **Theorem 6** ([14, Theorem 2.3]). *Given n colored integer points in 2D, we can construct in $\mathcal{O}(n \log n)$ time an $\mathcal{O}(n \log n)$-size data structure that, given any quarterplane Q, counts the number of distinct colors with at least one point in Q in $\mathcal{O}(\log n)$ time.*

We show how to apply geometric methods to a special variant of the CountDistinct problem, where we are interested in a small subset of occurrences of each pattern.

Let $\mathcal{D} = \{P_1, P_2, \ldots, P_d\}$ and $\mathcal{S}$ be a family of sets $S_1, \ldots, S_d$ such that $S_k \subseteq \mathsf{Occ}(P_k)$, where $\mathsf{Occ}(P_k)$ is the set of positions of T where P_k occurs. Let $\|\mathcal{S}\| = \sum_k |S_k|$. For each pattern P_k, we call the positions in the set S_k the *special positions of* P_k. Counting distinct patterns occurring at their special positions in $T[i\,..\,j]$ is called $\textsc{CountDistinct}_{\mathcal{S}}(i, j)$.

▶ **Lemma 7.** *The $\textsc{CountDistinct}_{\mathcal{S}}(i, j)$ queries can be answered in $\mathcal{O}(\log n)$ time with a data structure of size $\mathcal{O}(n + \|\mathcal{S}\| \log n)$ that can be constructed in $\mathcal{O}(n + \|\mathcal{S}\| \log n)$ time.*

Proof. We assign a different integer color c_k to every pattern $P_k \in \mathcal{D}$. Then, for each fragment $T[a\,..\,b] = P_k$ such that $a \in S_k$, we add point (a, b) with color c_k in an initially empty 2D grid $\mathcal{G}$. A $\textsc{CountDistinct}_{\mathcal{S}}(i, j)$ query reduces to counting different colors in the range $[i, \infty) \times (-\infty, j]$ of $\mathcal{G}$. The complexities follow from Theorem 6. ◀

3.4 Runs

A *run* (also known as a *maximal repetition*) is a periodic fragment $R = T[a\,..\,b]$ which can be extended neither to the left nor to the right without increasing the period $p = \mathsf{per}(R)$, i.e., $T[a-1] \neq T[a+p-1]$ and $T[b-p+1] \neq T[b+1]$ provided that the respective positions exist. If $\mathcal{R}$ is the set of all runs in a string T of length n, then $|\mathcal{R}| \leq n$ [3] and $\mathcal{R}$ can be computed in $\mathcal{O}(n)$ time [19]. The *exponent* $\mathsf{exp}(R)$ of a run R with period p is $|R|/p$. The sum of exponents of runs in a string of length n is $\mathcal{O}(n)$ [3, 19].

The *Lyndon root* of a periodic string U is the lexicographically smallest rotation of its $\mathsf{per}(U)$-length prefix. If L is the Lyndon root of a periodic string U, then U may be represented as (L, r, a, b); here $U = L[|L| - a + 1\,..\,|L|]L^r L[1\,..\,b]$, and r is called the *rank* of U. Note that the minimal rotation of a fragment of a text can be computed in $\mathcal{O}(1)$ time after an $\mathcal{O}(n)$-time preprocessing [16].

For a periodic fragment U, let $\mathsf{run}(U)$ be the run with the same period that contains U.

▶ **Lemma 8** ([3, 7, 17]). *For a periodic fragment U, $\mathsf{run}(U)$ and its Lyndon root are uniquely determined and can be computed in constant time after linear-time preprocessing.*

We use runs in 2-approximate $\textsc{CountDistinct}(i, j)$ queries and in counting squares.

4 Answering CountDistinct 2-Approximately

4.1 CountDistinct for Extended or Contracted Fragments

For two positions ℓ and r, we define $\mathsf{Pref}_{\mathcal{D}}(\ell, r)$ as the longest prefix of $T[\ell\,..\,r]$ that matches some pattern $P \in \mathcal{D}$; the length of such prefix is at most $r - \ell + 1$. Let us show how to compute the locus of $\mathsf{Pref}_{\mathcal{D}}(\ell, r)$ in the $\mathcal{D}$-modified suffix tree $\mathcal{T}_{T,\mathcal{D}}$. To this end, we preprocess $\mathcal{T}_{T,\mathcal{D}}$ for weighted ancestor queries and store at every node v of $\mathcal{T}_{T,\mathcal{D}}$ a pointer $p(v)$ to the nearest ancestor u (including v) of v such that $\mathcal{L}(u) \in \mathcal{D}$. To return $\mathsf{Pref}_{\mathcal{D}}(\ell, r)$, we find the locus u of $T[\ell\,..\,r]$ in the $\mathcal{D}$-modified suffix tree. We return $p(u)$ if $|\mathcal{L}(u)| = |T[\ell\,..\,r]|$ and $p(v)$, where v is the parent of u, otherwise.

Lemma 9 applies the $\mathcal{D}$-modified suffix tree to the problem of maintaining the count of distinct patterns occurring in a fragment subject to extending or shrinking the fragment.

▶ **Lemma 9.** *For any constant $\epsilon > 0$, given $\textsc{CountDistinct}(i, j)$, one can compute $\textsc{CountDistinct}(i \pm 1, j)$ and $\textsc{CountDistinct}(i, j \pm 1)$ in $\mathcal{O}(\log^{\epsilon} n)$ time with an $\mathcal{O}(n+d)$-size data structure that can be constructed in $\mathcal{O}(n\sqrt{\log n} + d)$ time.*

Proof. We only present a data structure for $\textsc{CountDistinct}(i \pm 1, j)$ queries. Queries $\textsc{CountDistinct}(i, j \pm 1)$ can be handled analogously by building the same data structure for the reverses of all the strings in scope.

We show how to compute the number of patterns $P \in \mathcal{D}$ whose only occurrence in some fragment $T[\ell\,..\,r]$ starts at position ℓ. The computation of $\textsc{CountDistinct}(i \pm 1, j)$ follows directly by setting $j = r$ and ℓ equal to $i - 1$ or i.

Data structure. We preprocess T for Bounded LCP queries (Lemma 3) and construct the $\mathcal{D}$-modified suffix tree $\mathcal{T}_{T,\mathcal{D}}$ of text T and dictionary $\mathcal{D}$. In addition, we preprocess $\mathcal{T}_{T,\mathcal{D}}$ for weighted ancestor queries and store at every node v of $\mathcal{T}_{T,\mathcal{D}}$ the number $\#(v)$ of the ancestors u (including v) of v such that $\mathcal{L}(u) \in \mathcal{D}$.

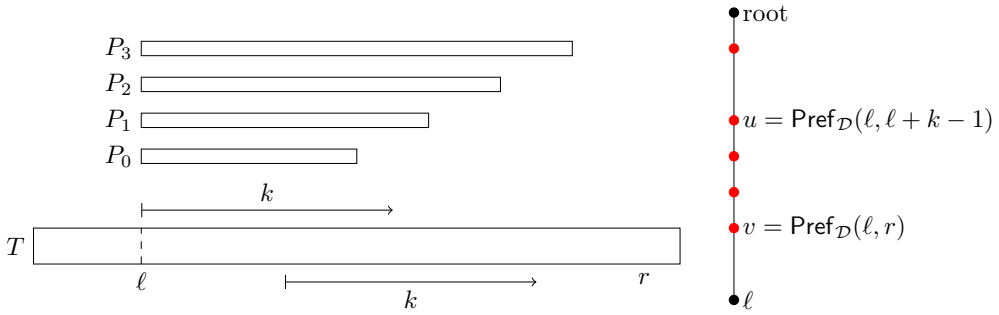

Figure 2 The setting of Lemma 9. Left: text T. Right: the path from the root of $\mathcal{T}_{T,\mathcal{D}}$ to the leaf with path-label $T[\ell\,..\,n]$. The nodes of the path whose path-labels match some patterns from $\mathcal{D}$ are drawn in red. Here, P_0 is the longest pattern that occurs at ℓ and also has an occurrence in $T[\ell + 1\,..\,r]$; its locus in $\mathcal{T}_{T,\mathcal{D}}$ is $u = \mathsf{Pref}_{\mathcal{D}}(\ell, \ell + k - 1)$. The patterns that occur in $T[\ell\,..\,r]$ only at position ℓ are P_1, P_2 and P_3. The locus of P_3 is $v = \mathsf{Pref}_{\mathcal{D}}(\ell, r)$. Then, $\#(v) - \#(u) = 5 - 2 = 3$.

Query. We want to count patterns longer than $k = |\mathit{BoundedLCP}(T[\ell\,..\,r], T[\ell + 1\,..\,r])|$. Let $u = \mathsf{Pref}_{\mathcal{D}}(\ell, \ell + k - 1)$ and $v = \mathsf{Pref}_{\mathcal{D}}(\ell, r)$. The desired number of patterns is equal to $\#(v) - \#(u)$. See Figure 2 for a visualization. ◀

4.2 Auxiliary Operation

Two fragments $U = T[i_1 \,..\, j_1]$ and $V = T[i_2 \,..\, j_2]$ are called *consecutive* if $i_2 = j_1 + 1$. We denote the overlap $T[\max\{i_1, i_2\} \,..\, \min\{j_1, j_2\}]$ of U and V by $U \cap V$.

3-Fragments-Counting
Input: A text T of length n and a dictionary $\mathcal{D}$ consisting of d patterns
Query: Given three consecutive fragments F_1, F_2, F_3 in T such that $|F_1| = |F_3|$ and $|F_2| \geq 8 \cdot |F_1|$, count distinct patterns P from $\mathcal{D}$ that have an occurrence starting in F_1 and ending in F_3 and do not occur in either F_1F_2 or F_2F_3

Let us fix $|F_1| = |F_3| = x$ and $|F_2| = y \geq 8x$. Additionally, let us call an occurrence of $P \in \mathcal{D}$ that starts in fragment F_a and ends in fragment F_b an (F_a, F_b)-occurrence. We will call an (F_1, F_3)-occurrence an *essential occurrence*.

We say that a string S is *highly periodic* if $\mathsf{per}(S) \leq \frac{1}{4}|S|$. We first consider the case that all patterns in $\mathcal{D}$ are not highly periodic.

▶ **Lemma 10.** *If each $P \in \mathcal{D}$ is not highly periodic, then*

$$\textit{3-Fragments-Counting}(F_1, F_2, F_3) = \textit{Count}(F_1F_2F_3) - \textit{Count}(F_1F_2) - \textit{Count}(F_2F_3) + \textit{Count}(F_2).$$

Proof. Let us start with the following claim.

▷ Claim 11. Any $P \in \mathcal{D}$ that has an essential occurrence occurs exactly once in $F_1F_2F_3$.

Proof. We have $|F_1F_2F_3| = x + y + x = 2x + y$. String P has an essential occurrence, so $|P| \geq y$. Therefore, if there are two occurrences of P in $F_1F_2F_3$, then they overlap in

$$2|P| - (2x + y) \geq 2|P| - (\tfrac{1}{4}|P| + |P|) = \tfrac{3}{4}|P|$$

positions. This implies that P is highly periodic, which is a contradiction. ◁

Claim 11 shows that 3-Fragments-Counting(F_1, F_2, F_3) is equal to the number of essential occurrences. Let us prove that the stated formula does not count any (F_a, F_b)-occurrences other than (F_1, F_3)-occurrences.

- Each (F_1, F_2)-occurrence is registered when we add Count$(F_1F_2F_3)$ and unregistered when we subtract Count(F_1F_2). Similarly for (F_2, F_3)-occurrences.
- Each (F_2, F_2)-occurrence is registered when we add Count$(F_1F_2F_3)$, Count(F_2) and unregistered when we subtract Count(F_1F_2), Count(F_2F_3).
- Each (F_1, F_1)-occurrence is registered when we add Count$(F_1F_2F_3)$ and unregistered when we subtract Count(F_1F_2). Similarly for (F_3, F_3)-occurrences. ◀

We now proceed with answering 3-Fragments-Counting queries for the dictionary of highly periodic patterns.

▶ **Lemma 12.** *If F_2 is aperiodic, then there are no essential occurrences of highly periodic patterns. Otherwise, all essential occurrences of highly periodic patterns are generated by the same run, that is,* $\mathsf{run}(F_2)$.

Proof. The first claim follows from the fact that such an occurrence of a pattern $P \in \mathcal{D}$ has an overlap of length at least $2\mathsf{per}(P)$ with F_2 and hence $\mathsf{per}(P) \leq \frac{1}{2}|F_2|$ is a period of F_2.

As for the second claim, it suffices to show that, for any pattern $P \in \mathcal{D}$ that has an essential occurrence, we have $\mathsf{per}(P) = \mathsf{per}(F_2)$. The inequalities $|F_2| \geq 2\mathsf{per}(F_2)$ and $|F_2| \geq 2\mathsf{per}(P)$ imply $|F_2| \geq \mathsf{per}(F_2) + \mathsf{per}(P)$. Hence, by the periodicity lemma, $q = \gcd(\mathsf{per}(P), \mathsf{per}(F_2))$ is a period of F_2. As $q \leq \mathsf{per}(F_2)$, we conclude that $q = \mathsf{per}(F_2)$. Thus, $\mathsf{per}(F_2)$ divides $\mathsf{per}(P)$, and therefore $\mathsf{per}(P) = \mathsf{per}(F_2)$. This concludes the proof. ◀

For a periodic factor U of T, let $\textsc{Periodic}(U)$ denote the set of distinct patterns from $\mathcal{D}$ that occur in U and have the same shortest period. Let us make the following observation.

▶ **Observation 13.** *If all $P \in \mathcal{D}$ are highly periodic, F_2 is periodic, and $R = \mathsf{run}(F_2)$, then*

$$\textit{3-Fragments-Counting}(F_1, F_2, F_3) = |\textsc{Periodic}(F_1F_2F_3 \cap R)| - |\textsc{Periodic}(F_1F_2 \cap R) \cup \textsc{Periodic}(F_2F_3 \cap R)|.$$

Next we now show how to efficiently evaluate the right-hand side of the formula in the observation above, using Theorem 5 for efficiently answering range counting queries in 2D.

We group all highly periodic patterns by Lyndon root and rank; for a Lyndon root L and a rank r, we denote by $\mathcal{D}^p_{L,r}$ the corresponding set of patterns. Then, we build the data structure of Theorem 5 for the set of points obtained by adding the point (a, b) for each $(L, r, a, b) \in \mathcal{D}^p_{L,r}$. We refer to the 2D grid underlying this data structure as $\mathcal{G}_{L,r}$. Note that the total number of points in the data structures over all Lyndon roots and ranks is $\mathcal{O}(d)$.

Each occurrence of a pattern (L, r, a, b) lies within some run in $\mathcal{R}$ with Lyndon root L. Let us state a simple fact.

▶ **Fact 14.** *A periodic string (L, r, a, b) occurs in a periodic string (L, r', a', b') if and only if at least one of the following conditions is met:*
(1) *$r = r'$, $a \leq a'$, and $b \leq b'$;*
(2) *$r = r' - 1$ and $a \leq a'$;*
(3) *$r = r' - 1$ and $b \leq b'$;*
(4) *$r \leq r' - 2$.*

▶ **Lemma 15.** *One can compute $|\textsc{Periodic}(U)|$ for any periodic fragment U in time $\mathcal{O}(\log n / \log\log n)$ using a data structure of size $\mathcal{O}(n + d)$ that can be constructed in time $\mathcal{O}(n + d\sqrt{\log n})$.*

Proof. For $U = (L, r, a, b)$, we count points contained in at least one of the rectangles
(1) $(-\infty, a] \times (-\infty, b]$ in $\mathcal{G}_{L,r}$,
(2) $(-\infty, a] \times (-\infty, |L|]$ in $\mathcal{G}_{L,r-1}$,
(3) $(-\infty, |L|] \times (-\infty, b]$ in $\mathcal{G}_{L,r-1}$,
and we add to the count the number of patterns of the form (L, r', a, b) with $r' < r - 1$. For the latter term, it suffices to store an array $X_L[1\,..\,t]$ such that $X_L[r] = \sum_{i=1}^{r} |\mathcal{D}^p_{L,i}|$, where t is the maximum rank of a pattern with Lyndon root L. The total size of these arrays is $\mathcal{O}(n)$ by the linearity of the sum of exponents of runs in a string [3, 19]. ◀

▶ Remark 16. In particular, in the proof of the above lemma, we count points that are contained within at least one out of a constant number of rectangles. Therefore, not only we can easily compute $|\textsc{Periodic}(U)|$, but similarly we are able to compute $|\textsc{Periodic}(U_1) \cup \textsc{Periodic}(U_2)|$ for some periodic factors U_1, U_2 of T.

We are now ready to prove the main result of this subsection.

▶ **Lemma 17.** *The 3-Fragments-Counting(F_1, F_2, F_3) queries can be answered in time $\mathcal{O}(\log^2 n/\log\log n)$ 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.*

Proof. By Lemma 10, in order to count the patterns that are not highly periodic, it suffices to perform three Count queries. To this end, we employ the data structure of Lemma 4 which answers Count queries in $\mathcal{O}(\log^2 n/\log\log n)$ time, occupies space $\mathcal{O}(n + d\log n)$, and can be constructed in time $\mathcal{O}(n\log n/\log\log n + d\log^{3/2} n)$.

We now proceed to counting highly periodic patterns. First, we check whether F_2 is periodic; this can be done in $\mathcal{O}(1)$ time after an $\mathcal{O}(n)$-time preprocessing of T [18, 17]. If F_2 is not periodic, then by Lemma 12 no highly periodic pattern has an essential occurrence, and we are thus done. If F_2 is periodic, three $|\textsc{Periodic}(U)|$ queries suffice to obtain the answer due to Observation 13. They can be efficiently answered due to Lemma 15 and Remark 16; the complexities are dominated by those for building the data structure for Count queries. ◀

4.3 Approximation Algorithm

Let us fix $\delta = \frac{1}{9}$. A fragment of length $\lfloor(1+\delta)^p\rfloor$ for any positive integer p will be called a *p-basic fragment*. Our data structure stores CountDistinct(i, j) for every basic fragment $T[i\,..\,j]$. Using Lemma 9, these values can be computed in $\mathcal{O}(n\log^{1+\epsilon} n + d)$ time with a sliding window approach. The space requirement is $\mathcal{O}(n\log n + d)$.

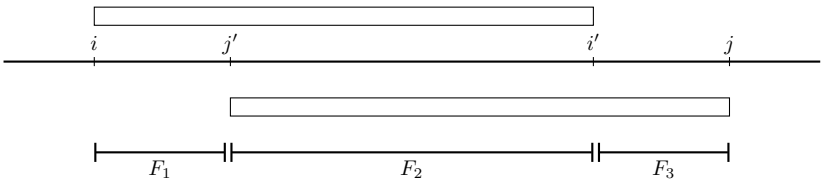

Figure 3 A 2-approximation of CountDistinct(i, j) is achieved using precomputed counts for basic factors $T[i\,..\,i']$ and $T[j'\,..\,j]$.

In order to answer an arbitrary CountDistinct(i, j) query, let $T[i\,..\,i']$ and $T[j'\,..\,j]$ be the longest prefix and suffix of $T[i\,..\,j]$ being a basic factor; see Figure 3. We sum up CountDistinct(i, i') and CountDistinct(j', j) and the result of a 3-Fragments-Counting query for $F_1 = T[i\,..\,j'-1]$, $F_2 = T[j'\,..\,i']$, $F_3 = T[i'+1\,..\,j]$. (Note that $(|F_1|+|F_2|)\cdot(1+\delta) > |F_1|+|F_2|+|F_3|$ implies $\delta(|F_1|+|F_2|) > |F_3|$, and since $|F_1| = |F_3|$, we have that $|F_1| = |F_3| \le \frac{1}{8}|F_2|$.) Now, a pattern $P \in \mathcal{D}$ is counted at least once if and only if it occurs in $T[i\,..\,j]$. Also, a pattern $P \in \mathcal{D}$ is counted at most twice (exactly twice if and only if it occurs in both F_1F_2 and F_2F_3). The above discussion and Lemma 17 yield the following result.

▶ **Theorem 18.** *The CountDistinct(i, j) queries can be answered 2-approximately in time $\mathcal{O}(\log^2 n/\log\log n)$ with a data structure of size $\mathcal{O}((n + d)\log n)$ that can be constructed in time $\mathcal{O}(n\log^{1+\epsilon} n + d\log^{3/2} n)$ for any constant $\epsilon > 0$.*

5 Time-Space Tradeoffs for Exact Counting

5.1 Tradeoff for Large Dictionaries

The following result is yet another application of Lemma 9.

▶ **Theorem 19.** *For any $m \in [1, n]$ and any constant $\epsilon > 0$, the* CountDistinct(i, j) *queries can be answered in $\mathcal{O}(m \log^{\epsilon} n)$ time using an $\mathcal{O}(n^2/m^2 + n + d)$-size data structure that can be constructed in $\mathcal{O}((n^2 \log^{\epsilon} n)/m + n\sqrt{\log n} + d)$ time.*

Proof. A fragment of the form $T[c_1 m{+}1 \,..\, c_2 m]$ for integers c_1 and c_2 will be called a *canonical fragment*. Our data structure stores CountDistinct(i', j') for every canonical fragment $T[i' \,..\, j']$ and the data structure of Lemma 9. Hence the space complexity $\mathcal{O}(n^2/m^2 + n + d)$.

We can compute in $\mathcal{O}(n \log^{\epsilon} n)$ time CountDistinct(i', j) for a given i' and all j using Lemma 9. There are $\mathcal{O}(n/m)$ starting positions of canonical fragments and hence the counts for all canonical fragments can be computed in $\mathcal{O}((n^2 \log^{\epsilon} n)/m)$ time. Additional preprocessing time $\mathcal{O}(n\sqrt{\log n} + d)$ originates from Lemma 9.

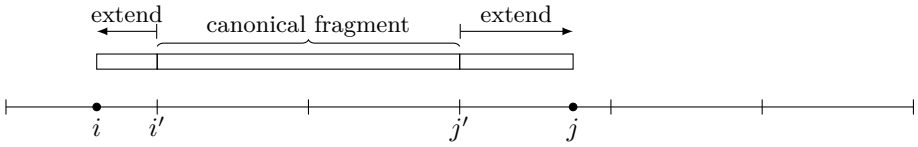

Figure 4 An illustration of the setting in the query algorithm underlying Theorem 19.

We can answer a CountDistinct(i, j) query in $\mathcal{O}(m \log^{\epsilon} n)$ time as follows. Let $T[i' \,..\, j']$ be the maximal canonical fragment contained in $T[i \,..\, j]$. We retrieve CountDistinct(i', j') for $T[i' \,..\, j']$. Then, we apply Lemma 9 $\mathcal{O}(m)$ times; each time we extend the fragment for which we count, until we obtain CountDistinct(i, j). See Figure 4. ◀

5.2 Tradeoff for Small Dictionaries

We call a set of strings $\mathcal{H}$ a *path-set* if all elements of $\mathcal{H}$ are prefixes of its longest element. We now show how to efficiently handle dictionaries that do not contain large path-sets.

▶ **Lemma 20.** *If $\mathcal{D}$ does not contain any path-set of size greater than k, then we can construct in $\mathcal{O}(kn \log n)$ time an $\mathcal{O}(kn \log n)$-size data structure that answers* CountDistinct(i, j) *queries in $\mathcal{O}(\log n)$ time.*

Proof. Let $\mathcal{D} = \{P_1, \ldots, P_d\}$ and $\mathcal{S} = \{\mathsf{Occ}(P_1), \ldots, \mathsf{Occ}(P_d)\}$. Every position of T contains at most k occurrences of patterns from $\mathcal{D}$. This implies that $\|\mathcal{S}\| \leq kn$. We can obviously treat a CountDistinct(i, j) query as a CountDistinct$_{\mathcal{S}}(i, j)$ query. The complexities follow from Lemma 7. ◀

A proof of the following lemma is rather standard and is included in the full version of the paper.

▶ **Lemma 21.** *For any $k \in [1, n]$, we can compute a maximal family $\mathcal{F}$ of pairwise-disjoint path-sets in $\mathcal{D}$, each consisting of at least k elements, in $\mathcal{O}(n + d)$ time.*

We now combine Lemmas 3, 20 and 21 to get the main result of this section.

▶ **Theorem 22.** *For any $m \in [1, n]$ and any constant $\epsilon > 0$, the* CountDistinct(i, j) *queries can be answered in $\mathcal{O}(m \log^{\epsilon} n + \log n)$ time using an $\mathcal{O}((nd \log n)/m + d)$-size data structure that can be constructed in $\mathcal{O}((nd \log n)/m + d)$ time.*

Proof. We first apply Lemma 21 for $k = \lceil d/m \rceil$. We then have a decomposition of $\mathcal{D}$ to a family $\mathcal{F}$ of at most m path-sets and a set $\mathcal{D}'$ with no path-set of size greater than $\lfloor d/m \rfloor$. We directly apply Lemma 20 for $\mathcal{D}'$. In order to handle path-sets, we build the data

structure of Lemma 3. Then, upon a CountDistinct(i, j) query, for each path-set $\mathcal{H} \in \mathcal{F}$, we compute the longest pattern in $\mathcal{H}$ that occurs in $T[i\,..\,j]$ using a Bounded LCP query followed by a predecessor query [24] in a structure that stores the lengths of the elements of $\mathcal{H}$, with the lexicographic rank in $\mathcal{H}$ stored as satellite information. The data structure of [24] is randomized, but it can be combined with deterministic dictionaries [21] using a simple two-level approach (see [23]), resulting in a deterministic *static* data structure. ◀

▶ Remark 23. Let us fix the query time to be $\mathcal{O}(m \log^{\epsilon} n)$ for $m = \Omega(\log n)$. Then, Theorem 22 outperforms Theorem 19 in terms of the required space for $d = o(n/(m \log n))$. For example, for $m = d = n^{1/4}$, the data structure of Theorem 22 requires space $\tilde{\mathcal{O}}(n)$ while the one of Theorem 19 requires space $\tilde{\mathcal{O}}(n\sqrt{n})$.

6 Internal Counting of Distinct Squares

The number of occurrences of squares could be quadratic, but we can construct a much smaller $\mathcal{O}(n \log n)$-size subset of these occurrences (called *boundary occurrences*) that, from the point of view of CountDistinct queries, gives almost the same answers. This is the main trick in this section. Distinct squares with a boundary occurrence in a given fragment can be counted in $\mathcal{O}(\log n)$ time due to Lemma 7. The remaining squares can be counted based on their structure: we show that they are all generated by the same run.

Now, the dictionary $\mathcal{D}$ is the set of all squares in T. By the following fact, $d = \mathcal{O}(n)$ and $\mathcal{D}$ can be computed in $\mathcal{O}(n)$ time.

▶ **Fact 24** ([7, 8, 10, 12]). *A string T of length n contains $\mathcal{O}(n)$ distinct square factors and they can all be computed in $\mathcal{O}(n)$ time.*

We say that an occurrence of a square U^2 is *induced* by a run R if it is contained in R and the shortest periods of U and R are the same. Every occurrence of a square is induced by exactly one run.

We need the following fact (note that it is false for the set of *all* runs; see [11]).

▶ **Fact 25.** *The sum of the lengths of all highly periodic runs is $\mathcal{O}(n \log n)$.*

Proof. We will prove that each position in T is contained in $\mathcal{O}(\log n)$ highly periodic runs. Let us consider all highly periodic runs R containing some position i, such that $m \leq \mathsf{per}(R) < \frac{3}{2}m$ for some even integer m. Suppose for the sake of contradiction that there are at least 5 such runs. Note that each such run fully contains one of the fragments $T[i - 3m + 1 + t\,..\,i + t]$ for $t \in \{0, m, 2m, 3m\}$. By the pigeonhole principle, one of these four fragments is contained in at least two runs, say R_1 and R_2. In particular, the overlap of these runs is at least $3m \geq \mathsf{per}(R_1) + \mathsf{per}(R_2)$, which is a contradiction by the periodicity lemma. ◀

We define a family of occurrences $\mathcal{B} = B_1, \ldots, B_d$ such that, for each square U_i^2, the set B_i contains the leftmost and the rightmost occurrence of U_i^2 in every run. We call these *boundary occurrences*. Boundary occurrences of squares have the following property.

▶ **Lemma 26.** *$\|\mathcal{B}\| = \mathcal{O}(n \log n)$ and the set family $\mathcal{B}$ can be computed in $\mathcal{O}(n \log n)$ time.*

Proof. Let us define the *root* of a square U^2 to be U. A square is primitively rooted if its root is a primitive string. Let *p-squares* be primitively rooted squares, *np-squares* be the remaining ones. The number of occurrences of p-squares in a string of length n is $\mathcal{O}(n \log n)$ and they can all be computed in $\mathcal{O}(n \log n)$ time; see [6, 22].

We now proceed to np-squares. Note that for any highly periodic run R, the leftmost occurrence of each np-square induced by R starts in one of the first $\mathsf{per}(R)$ positions of R; a symmetric property holds for rightmost occurrences and last $\mathsf{per}(R)$ positions. In addition, it can be readily verified that such a position is the starting (resp. ending) position of at most $\mathsf{exp}(R)$ squares induced by R. It thus suffices to bound the sum of $\mathsf{exp}(R) \cdot \mathsf{per}(R)$ over all highly periodic runs R. The fact that $\mathsf{exp}(R) \cdot \mathsf{per}(R) = |R|$ concludes the proof of the combinatorial part by Fact 25.

For the algorithmic part, it suffices to iterate over the $\mathcal{O}(n)$ runs of T. ◀

▶ **Lemma 27.** *If $T[i\,..\,j]$ is non-periodic, $\textsc{CountDistinct}(i,j) = \textsc{CountDistinct}_{\mathcal{B}}(i,j)$.*

Proof. Let us consider an occurrence of a square U^2 inside $T[i\,..\,j]$. Let R be the run that induces this occurrence. By the assumption of the lemma, R does not contain $T[i\,..\,j]$. Then at least one of the boundary occurrences of U^2 in R is contained in $T[i\,..\,j]$. ◀

For a periodic fragment F of T, by $\mathit{RunSquares}(F)$ we denote the number of distinct squares that are induced by F (being a run if interpreted as a standalone string). The value $\mathit{RunSquares}(F)$ can be computed in $\mathcal{O}(1)$ time, as it was shown in e.g. [7].

Let F_1 be a prefix and F_2 be a suffix of a periodic fragment F, such that each of F_1 and F_2 is of length at most $\mathsf{per}(F)$ – and hence they are disjoint. By $\mathit{BSq}(F, F_1, F_2)$ ("bounded squares") we denote the number of distinct squares induced by F which have an occurrence starting in F_1 or ending in F_2.

▶ **Lemma 28.** *Given $\mathsf{per}(F)$, the $\mathit{BSq}(F, F_1, F_2)$ queries can be answered in $\mathcal{O}(1)$ time.*

Proof. We are to count distinct squares induced by F that start in F_1 or end in F_2.

We introduce an easier version of BSq queries. Let $\mathit{BSq}'(F, F_1) = \mathit{BSq}(F, F_1, \varepsilon)$ be the number of squares induced by F which start in its prefix F_1 of length at most $p := \mathsf{per}(F)$.

Reduction of BSq to BSq'. First, observe that the set of squares induced by F starting at some position $q \in [1, p]$ and the set of squares induced by F ending at some position $q' \in [|F| - p + 1, |F|]$ are equal if $q \equiv q' + 1 \pmod{p}$ and disjoint otherwise. Also note that $F_2 = UV$ for some prefix V and some suffix U of $F[p]F[1\,..\,p-1]$; we consider this rotation of $F[1\,..\,p]$ to offset the $+1$ factor in the above modular equation. Let $|U| = a$ and $|V| = b$.

Then, by the aforementioned observation, we are to count distinct squares that start in some position in the set $[1, |F_1|] \cup [1, b] \cup [p - a + 1, p]$; see Figure 5.

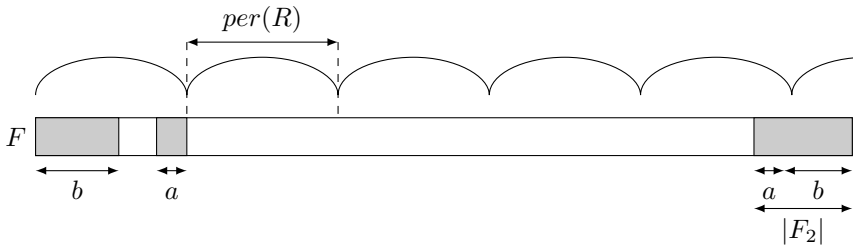

Figure 5 Reduction of BSq to BSq'; the case that $|F_1| \leq b$.

Hence the computation of $\mathit{BSq}(F, F_1, F_2)$ is reduced to at most two instances of the special case when F_2 is the empty string.

Computation of $BSq'(F, F_1)$. The number of squares induced by F starting at $F[i]$ is $\lfloor(|F| - i + 1)/(2p)\rfloor$. Consequently, $BSq'(F, F_1) = \sum_{i=1}^{|F_1|} \lfloor(|F| - i + 1)/(2p)\rfloor = |F_1| \cdot t - \max\{0, |F_1| - k - 1\}$, where $t = \lfloor|F|/(2p)\rfloor$ and $k = |F| \bmod (2p)$. ◀

▶ **Lemma 29.** *Assume that $F = T[i\,..\,j]$ is periodic and $R = T[a\,..\,b] = \mathsf{run}(T[i\,..\,j])$. Let $F_1 = T[i\,..\,a + p - 1]$ and $F_2 = T[b - p + 1\,..\,j]$, where $\mathsf{per}(R) = p$. Then:*

$$\textsc{CountDistinct}(i, j) = \textsc{CountDistinct}_{\mathcal{B}}(i, j) + \mathit{RunSquares}(F) - \mathit{BSq}(F, F_1, F_2). \quad (1)$$

Proof. In the sum $\textsc{CountDistinct}_{\mathcal{B}}(i, j) + \mathit{RunSquares}(F)$, all squares are counted once except for squares whose boundary occurrences are induced by R, which are counted twice. They are exactly counted in the term $\mathit{BSq}(F, F_1, F_2)$; see Figure 6. ◀

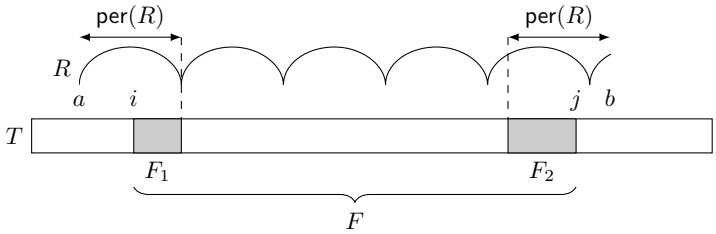

Figure 6 The setting in Lemma 29. Note that F_1 is empty if $i \geq a + \mathsf{per}(R)$; similarly for F_2.

▶ **Theorem 30.** *If $\mathcal{D}$ is the set of all square factors of T, then $\textsc{CountDistinct}(i, j)$ queries can be answered in $\mathcal{O}(\log n)$ time using a data structure of size $\mathcal{O}(n \log^2 n)$ that can be constructed in $\mathcal{O}(n \log^2 n)$ time.*

Proof. We precompute the set $\mathcal{B}$ in $\mathcal{O}(n \log n)$ time using Lemma 26 and perform $\mathcal{O}(n \log^2 n)$ time and space preprocessing for $\textsc{CountDistinct}_{\mathcal{B}}(i, j)$ queries.

In order to answer a $\textsc{CountDistinct}(i, j)$ query, first we ask a $\mathsf{run}(T[i\,..\,j])$ query of Lemma 8 to check if $T[i\,..\,j]$ is periodic.

We compute $\textsc{CountDistinct}_{\mathcal{B}}(i, j)$ which takes $\mathcal{O}(\log n)$ time due to Lemma 7. If $T[i\,..\,j]$ is non-periodic, then it is the final result due to Lemma 27.

Otherwise $T[i\,..\,j]$ is periodic. Let F, F_1, F_2 be as in Lemma 29. We answer $\mathit{RunSquares}(F)$ and $\mathit{BSq}(F, F_1, F_2)$ queries in $\mathcal{O}(1)$ time using the algorithm from [7] and Lemma 28, respectively. Finally, $\textsc{CountDistinct}(i, j)$ is computed using (1). ◀

7 Final Remarks

The general framework for dynamic dictionaries, presented in the full version of this paper, essentially consists in rebuilding a static data structure after every k updates. We return correct answers by performing individual queries for the patterns inserted or deleted from the dictionary since the data structure was built. In particular, we show that an application of this framework – with some tweaks – to the data structure of Section 4 yields the following.

▶ **Theorem 31.** *For any $k \in [1, n]$, we can construct a data structure in $\tilde{\mathcal{O}}(n + d)$ time, which processes each update to the dictionary in $\tilde{\mathcal{O}}(n/k)$ time and answers $\textsc{CountDistinct}(i, j)$ queries 2-approximately in $\tilde{\mathcal{O}}(k)$ time.*

We leave open the problem of whether an $\tilde{\mathcal{O}}(n + d)$-size data structure answering $\textsc{CountDistinct}(i, j)$ queries exactly in time $\tilde{\mathcal{O}}(1)$ exists.

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, 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.
3. 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.
4. 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. SIAM, 2010. doi:10.1137/1.9781611973075.15.
5. 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 *LIPIcs*, pages 22:1–22:17. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ISAAC.2019.22.
6. Maxime Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12(5):244–250, 1981. doi:10.1016/0020-0190(81)90024-7.
7. 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.
8. Antoine Deza, Frantisek Franek, and Adrien Thierry. How many double squares can a string contain? *Discrete Applied Mathematics*, 180:52–69, 2015. doi:10.1016/j.dam.2014.08.016.
9. Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. doi:10.2307/2034009.
10. Aviezri S. Fraenkel and Jamie Simpson. How many squares can a string contain? *Journal of Combinatorial Theory, Series A*, 82(1):112–120, 1998. doi:10.1006/jcta.1997.2843.
11. Amy Glen and Jamie Simpson. The total run length of a word. *Theoretical Computer Science*, 501:41–48, 2013. doi:10.1016/j.tcs.2013.06.004.
12. Dan Gusfield and Jens Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *Journal of Computer and System Sciences*, 69(4):525–546, 2004. doi:10.1016/j.jcss.2004.03.004.
13. 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.
14. Haim Kaplan, Natan Rubin, Micha Sharir, and Elad Verbin. Efficient colored orthogonal range counting. *SIAM Journal on Computing*, 38(3):982–1011, 2008. doi:10.1137/070684483.
15. 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.
16. Tomasz Kociumaka. Minimal suffix and rotation of a substring in optimal time. In *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016*, volume 54 of *LIPIcs*, pages 28:1–28:12. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CPM.2016.28.
17. 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.
18. 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.

19 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.

20 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 *Lecture Notes in Computer Science*, pages 290–303. Springer, 2017. doi:10.1007/978-3-319-67428-5_25.

21 Milan Ružić. Constructing efficient dictionaries in close to sorting time. In *Automata, Languages and Programming, ICALP 2008, Part I*, volume 5125 of *Lecture Notes in Computer Science*, pages 84–95. Springer, 2008. doi:10.1007/978-3-540-70575-8_8.

22 Jens Stoye and Dan Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree. *Theoretical Computer Science*, 270(1-2):843–856, 2002. doi:10.1016/S0304-3975(01)00121-9.

23 Mikkel Thorup. Space efficient dynamic stabbing with fast queries. In *35th Annual ACM Symposium on Theory of Computing, STOC 2003*, pages 649–658. ACM, 2003. doi:10.1145/780542.780636.

24 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.