Text Indexing: From Reporting to Counting

Ben Bals
CWI, Amsterdam, The Netherlands
Vrije Universiteit Amsterdam, The Netherlands

Panagiotis Charalampopoulos
King's College London, UK

Oded Lachish
Birkbeck, University of London, UK

Solon P. Pissis
The Cyprus Institute, Nicosia, Cyprus

Hilde Verbeek
CWI, Amsterdam, The Netherlands

Abstract

We prove an elementary yet powerful combinatorial lemma: in any rooted tree with L leaves, the number of nodes whose depth is smaller than the number of their leaf descendants is at most L. For any string T of length n, a direct application of this lemma to the suffix trie of T yields that the number of substrings of T whose length is smaller than their number of occurrences in T is at most n. This combinatorial insight leads to space-efficient data structures with optimal query times for string *counting* problems via the following algorithmic framework: store the counts for the at most n "frequent" substrings of T in a preprocessing step, and use a *reporting* query to count for the "infrequent" substrings. Our framework acts as a convenient black box, lifting indexes with reporting time $\mathcal{O}(|P| + |\mathsf{Occ}_T(P)|)$ to support counting queries in time $\mathcal{O}(|P|)$, where P is the queried pattern and $\mathsf{Occ}_T(P)$ is the set of occurrences of P in T. As applications, we show efficient indexes for consecutive occurrences, weighted sequences, strings with utilities, and non-overlapping occurrences.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases text indexing, data structures, string algorithms, string processing

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.68

Related Version *Full Version*: https://arxiv.org/abs/2607.24043

Funding *Hilde Verbeek*: Supported by a Constance van Eeden Fellowship.

1 Introduction

Text indexing is a classical problem in computer science with many applications, especially in bioinformatics [35] and information retrieval [3]. The goal of text indexing is to preprocess a string $T = T[1\,..\,n]$ of length n, called the *text*, into a data structure, called the *index*, supporting queries for a string $P = P[1\,..\,m]$ of length $m \leq n$, called the *pattern*. The most typical query types in text indexes are *reporting* and *counting*: the former asks for the set $\mathsf{Occ}_T(P)$ of all starting positions (occurrences) of P in T, while the latter asks for $|\mathsf{Occ}_T(P)|$.

The study of text indexing dates back to the early 1970s, when Weiner [52] introduced suffix trees. This foundational data structure occupies $\mathcal{O}(n)$ space, can be constructed in $\mathcal{O}(n)$ time [24], and answers reporting queries in $\mathcal{O}(m+|\mathsf{Occ}_T(P)|)$ time and counting queries in $\mathcal{O}(m)$ time. Subsequent work on text indexing focused mainly on reducing the space complexity – the main bottleneck in practice – often at the cost of increased query time. Key milestones in this direction include the suffix array [33, 44], the compressed suffix array [34], the FM-index [25], as well as the r-index [27] and other indexes for highly repetitive texts [46].

The *suffix tree* of T is the compacted trie of all the suffixes of T. Starting from the (uncompacted) suffix trie of T, the suffix tree is obtained by suppressing all internal nodes with exactly one child: these nodes become *implicit*, corresponding to points on edges of the suffix tree rather than explicit nodes. The remaining nodes – the root, all branching internal nodes, and all leaves – are called *explicit*. This compactification leads to $\mathcal{O}(n)$ explicit nodes, for a text T of length n, and therefore to an $\mathcal{O}(n)$-space index.

Despite the considerable attention given to suffix trees, the following intuition appears to have gone unnoticed. The structural property that allows suffix tries to be compacted from up to $\Theta(n^2)$ to $\Theta(n)$ nodes – namely, that many nodes of the trie share the exact same set of leaf descendants along unary paths – also bounds the number of nodes whose depth is smaller than their number of leaf descendants. Our central contribution is an elementary yet powerful combinatorial lemma formalizing this intuition: in any rooted tree on L leaves, the number of nodes whose depth is smaller than the number of their leaf descendants is at most L. Applied to the suffix trie of a given string, this yields the following interpretation.

▶ **Theorem 1.** *In any string T of length n, the number of non-empty substrings of T whose length is smaller than their number of occurrences in T is at most n.*

We prove that this bound is asymptotically tight for $T = (a_1 a_2 \dots a_r)^r$ over alphabet $\Sigma = \{a_1, a_2, \dots, a_r\}$. The algorithmic implications of Theorem 1 are far-reaching. Theorem 1 leads to space-efficient data structures with *optimal query times* for string *counting* problems via the following framework: store the counts for the at most n "frequent" substrings of T in a preprocessing step, and use a *reporting* query to count for the "infrequent" substrings. Our framework acts as a convenient black box, lifting indexes with reporting time $\mathcal{O}(m+|\mathsf{Occ}_T(P)|)$ to counting indexes with query time $\mathcal{O}(m)$. More formally, we prove the following theorem.

▶ **Theorem 2** (Algorithmic Framework)**.** *Let D be a data structure constructed over a text T of length n, so that, when queried for a pattern P of length m, it returns an answer of total size w in time $\mathcal{O}(m+|\mathsf{Occ}_T(P)|+w)$. Then, there is a data structure $\hat{D}$ of size $\mathcal{O}(|D|+w\cdot n)$ that answers the same queries for P in time $\mathcal{O}(m+w)$.*

Our framework can be easily extended for efficiently addressing the problem where each occurrence of a pattern has some associated value, and we wish to aggregate these values for all sought occurrences under an associative operator. See Corollary 8 for a formal statement.

As a first application, we consider the *consecutive occurrences indexing* problem [9, 11, 12, 13, 30, 38, 47, 48]. In its most basic variant, we are asked to preprocess T and an interval $[\alpha, \beta]$ into an index, such that, for any pattern P, we can report the consecutive occurrences of P in T with a distance in $[\alpha, \beta]$. While a more general $\mathcal{O}(n \log n)$-space index (that also gets the interval $[\alpha, \beta]$ at query time) with optimal reporting queries exists due to Navarro and Thankachan [47], the counting version with fixed bounded-gap constraints has no known $\mathcal{O}(n \cdot \mathrm{polylog}(n))$-space solution with $\mathcal{O}(m)$-time queries. A black-box application of Theorem 2 to the main result of [47] yields an $\mathcal{O}(n \log n)$-space index with optimal $\mathcal{O}(m)$-time counting queries. We improve the space complexity to $\mathcal{O}(n)$. We also present other implications of Theorem 2 in data structures for indexing problems, including problems involving weighted sequences [4, 17, 26] and strings with utilities [7, 8].

For a class of indexing problems in the framework of Corollary 8, including indexing strings with utilities, we further show that we can *construct* an $\mathcal{O}(n)$-space index in $\widetilde{\mathcal{O}}(n\sqrt{n})$ time by using Theorem 1 and exploiting the periodic structure of T. Our data structure improves over the state-of-the-art index for strings with utilities [8] *polynomially* in both the preprocessing time (from $\mathcal{O}(n^2)$ to $\widetilde{\mathcal{O}}(n\sqrt{n})$) as well as the space–query time trade-off (from $\mathcal{O}(n^2/\tau)$ space and $\mathcal{O}(m+\tau)$ query time to $\mathcal{O}(n)$ space and $\mathcal{O}(m)$ query time).

As a bonus, we unify the treatment of several indexing problems, recovering known results for their counting variants by applying our algorithmic framework to their reporting variants; we do so for indexing strings with properties [1, 4], for indexing for overlapping or non-overlapping occurrences, and for covers [15, 21, 48].

Notably, our combinatorial lemma on trees can be generalized to yield the following two corollaries on strings, which have further consequences.

▶ **Corollary 3.** *In any string T of length n, for any integer $\tau \in [1, n]$, the number of non-empty substrings U of T for which $|U| \cdot \tau < |\mathit{Occ}_T(U)|$ is at most n/τ.*

▶ **Corollary 4.** *In any string T of length n, for any integer $\tau \in [1, n]$, the number of non-empty substrings U of T for which $|U| < \tau \cdot |\mathit{Occ}_T(U)|$ is at most $n\tau$.*

Corollary 3 enables us to transform state-of-the-art reporting data structures (e.g., [29, 32]) that occupy less than $\mathcal{O}(n)$ space (say, they use $o(n \log n)$ bits) to support counting queries. Roughly speaking, given an s-space data structure, we set $\tau = n/s$, precompute and store the answers for at most $n/\tau = s$ substrings U of T (the ones for which $|U| \cdot \tau < |\mathsf{Occ}_T(U)|$), and spend $\mathcal{O}(m \cdot n/s)$ time at query time for any given length-m pattern. We use this idea to construct a compact data structure for counting non-overlapping occurrences in near-optimal time, which was mentioned in the literature (cf. [32]) as a "challenging" open problem.

Furthermore, Corollary 4 enables us to extend the application of our framework to the *internal pattern matching* (IPM) model [18, 43], where the pattern is specified as a fragment of T. We use this idea to construct an IPM data structure for indexing strings with utilities.

Paper organization. In Section 2, we prove the combinatorial lemma and Theorem 1. In Section 3, we prove Theorem 2 and present the algorithmic framework. In Section 4, we present the headline applications of our framework and demonstrate the versatility of our approach by recovering several known results under our framework. In Section 5, we prove Corollary 3 and showcase an application. In Section 6, we prove Corollary 4 and showcase the IPM index for strings with utilities as an application. In Section 7, we present a construction algorithm for a class of indexing problems, including indexing strings with utilities.

2 The Combinatorial Lemma

▶ **Lemma 5.** *For any rooted tree $\mathcal{T}$ with L leaves over a set $\mathcal{V}$ of nodes, we have*

$$|\{v \in \mathcal{V} \mid \mathit{depth}(v) < |\mathcal{L}(v)|\}| \leq L,$$

where $\mathit{depth}(v)$ is the depth of node v and $\mathcal{L}(v)$ is the set of leaf descendants of v.

Proof. Let $\mathcal{L}(\mathcal{T})$ be the set of leaves of $\mathcal{T}$. Further, let $\mathcal{S} = \{v \in \mathcal{V} \mid \mathsf{depth}(v) < |\mathcal{L}(v)|\}$. We show that $|\mathcal{S}| \leq L$ by constructing an *injective* mapping $\phi : \mathcal{S} \to \mathcal{L}(\mathcal{T})$.

For any node $v \in \mathcal{S}$, the condition $\mathsf{depth}(v) < |\mathcal{L}(v)|$ means that the number of leaf descendants of v is strictly greater than the depth of v. We say that a node v has in-order rank i if it is the ith node visited in an in-order traversal of $\mathcal{T}$. Let the leaf descendants of v be $\ell_{v,1}, \ell_{v,2}, \ldots, \ell_{v,|\mathcal{L}(v)|}$, in increasing order with respect to their in-order ranks. The function $\phi : \mathcal{S} \to \mathcal{L}(\mathcal{T})$ maps each node $v \in \mathcal{S}$ to its $(\mathsf{depth}(v) + 1)$th leaf descendant according to the above order; i.e., $\phi(v) = \ell_{v,\mathsf{depth}(v)+1}$. Since $v \in \mathcal{S}$, we have $|\mathcal{L}(v)| \geq \mathsf{depth}(v) + 1$, ensuring that $\ell_{v,\mathsf{depth}(v)+1}$ exists within the subtree rooted at v. We must show that for any $u, v \in \mathcal{S}$ with $u \neq v$, we have $\phi(u) \neq \phi(v)$. We consider the following two cases:

Case 1: Neither u nor v is an ancestor of the other. In this case, the subtrees rooted at u and v are disjoint. Consequently, their sets of leaf descendants $\mathcal{L}(u)$ and $\mathcal{L}(v)$ are disjoint. Since $\phi(u) \in \mathcal{L}(u)$ and $\phi(v) \in \mathcal{L}(v)$, it immediately follows that $\phi(u) \neq \phi(v)$.

Case 2: u is an ancestor of v or vice versa. We assume, without loss of generality, that u is an ancestor of v – the other case is symmetric. Let $d_u \coloneqq \mathsf{depth}(u)$ and $d_v \coloneqq \mathsf{depth}(v)$. Since u is a strict ancestor of v, we have $d_u < d_v$. The leaf $\phi(u)$ is the $(d_u + 1)$th leaf descendant of u. The leaf $\phi(v)$ is the $(d_v + 1)$th leaf descendant of v. Because v is in the subtree of u, the leaf descendants of v form a contiguous segment within the leaf descendants of u (ordered by in-order ranks). Let $k \geq 0$ be the number of leaf descendants of u with smaller in-order rank than v. The index of $\phi(v)$ among the leaf descendants of u is $k + d_v + 1$. Since $k \geq 0$ and $d_u < d_v$, we must have $d_u + 1 < d_v + 1 \leq k + d_v + 1 \Rightarrow \phi(u) \neq \phi(v)$.

Since the mapping $\phi : \mathcal{S} \to \mathcal{L}(\mathcal{T})$ is injective, the cardinality of the set $\mathcal{S}$ must be less than or equal to the cardinality of the set $\mathcal{L}(\mathcal{T})$, and hence $|\mathcal{S}| \leq |\mathcal{L}(\mathcal{T})| = L$. ◀

The following consequence of the above lemma for strings is immediate.

▶ **Theorem 1.** *In any string T of length n, the number of non-empty substrings of T whose length is smaller than their number of occurrences in T is at most n.*

Proof. Apply Lemma 5 to the trie of $\{T[1\,..\,n]\$, T[2\,..\,n]\$, \ldots, T[n]\$\}$, for a unique letter \$ that does not occur in T. Since, by construction, there is a bijective mapping between the internal nodes of the trie and the substrings of T, the claim follows. ◀

We next show that the bounds in both Lemma 5 and Theorem 1 are asymptotically tight.

▶ **Proposition 6.** *For every integer $r \geq 2$, there exists a string T of length $n = r^2$ over an alphabet of size r that has $n - 2\sqrt{n} + 2$ non-empty substrings U satisfying $|U| < |\mathsf{Occ}_T(U)|$.*

Proof. Let $T = X^r$, with $X = a_1 a_2 \ldots a_r$ over $\Sigma = \{a_1, a_2, \ldots, a_r\}$. Any substring U of T of length $\ell \leq r$ is uniquely determined by its length and its starting position i modulo r; such a substring U occurs r times if $(i-1) \bmod r + \ell - 1 \leq r$, and $r-1$ times otherwise. For any $\ell \leq r-2$, all r length-ℓ substrings of $XX[1\,..\,\ell-1]$ are distinct and have at least $r - 1 > \ell$ occurrences in T. The number of these substrings is $r(r-2)$. Further, both $X[1\,..\,|X|-1]$ and $X[2\,..\,|X|]$ have r occurrences and length $r-1$, while all other substrings of T of length $r-1$ have $r-1$ occurrences. Finally, note that any substring U of length $\ell \geq r$ has at most r occurrences, and thus $\ell \geq |\mathsf{Occ}_T(U)|$. We conclude that the total number of substrings U of T that satisfy $|U| < |\mathsf{Occ}_T(U)|$ is $r(r-2) + 2 = n - 2\sqrt{n} + 2$. ◀

3 Algorithmic Framework

Theorem 1 enables efficient solutions for a number of interesting data structure problems on strings. Theorem 2 captures a wide class of such data structure problems where we obtain improvements in a black-box manner.

Below, we use perfect hashing to efficiently navigate the suffix tree.

▶ **Lemma 7** ([50])**.** *A static $\mathcal{O}(n)$-space dictionary on a set of n keys can be deterministically constructed in time $\mathcal{O}(n \log^2 \log n)$, so that look-up queries to the dictionary take time $\mathcal{O}(1)$.*

▶ **Theorem 2** (Algorithmic Framework)**.** *Let D be a data structure constructed over a text T of length n, so that, when queried for a pattern P of length m, it returns an answer of total size w in time $\mathcal{O}(m + |\mathsf{Occ}_T(P)| + w)$. Then, there is a data structure $\hat{D}$ of size $\mathcal{O}(|D| + w \cdot n)$ that answers the same queries for P in time $\mathcal{O}(m + w)$.*

There are many text indexing problems for which there is a data structure D that reports some specific subset of $\mathsf{Occ}_T(P)$ in $\mathcal{O}(m + |\text{output}|)$ time. For any such problem, the counting variant (that is, when we want to find the size of this subset) naturally satisfies the requirements of Theorem 2. See Sections 4.1, 4.3, and 4.4 for applications.

Proof. We construct $\hat{D}$ as follows. We first perform the preprocessing of D and then construct the suffix tree of T. The suffix tree takes $\mathcal{O}(n)$ space [52]. Each branching node of the suffix tree is augmented with a dictionary that supports $\mathcal{O}(1)$-time access of the edges based on the first letter (key) of their label using Lemma 7. The total size of the dictionaries is $\mathcal{O}(n)$.

In the suffix tree, we find the implicit or explicit nodes that correspond to "frequent" substrings (i.e., those that occur more often than they are long) as follows. The *length* of a substring is the string-depth of its associated node. The *number of occurrences* is the number of leaf descendants. Both quantities can be computed simultaneously for every node using a bottom-up traversal. For every node corresponding to a frequent substring, we proceed as follows. If the node is implicit, we make it explicit. By Theorem 1, this adds only $\mathcal{O}(n)$ extra nodes to the tree. Then, we query D using this substring to compute the w-size answer and store it with the corresponding node in the suffix tree. This requires $\mathcal{O}(|D| + w \cdot n)$ space.

Upon a query P, we first locate the corresponding node in our suffix tree in $\mathcal{O}(m)$ time [52]. If the node is annotated with a precomputed answer, we output the answer in $\mathcal{O}(w)$ time. If not, we offload the query to D, which outputs the answer in $\mathcal{O}(m + |\mathsf{Occ}_T(P)| + w)$ time (by the hypothesis). As P was not frequent (otherwise, there would have been a precomputed answer), we know that $m \geq |\mathsf{Occ}_T(P)|$ and thus this case also takes $\mathcal{O}(m + w)$ time. ◀

Another large category of problems where Theorem 2 can be applied is where the desired output of the query is a summary statistic over some or all of the occurrences of the pattern P in T. The following corollary shows that for many of these cases, we can achieve optimal query time, given that a few mild conditions are met. Yet, it does not exhaustively characterize these use cases. In some cases, a more careful, specialized application can yield good results, even if the meta statement below is not applicable (we discuss some such cases in Section 4.4).

▶ **Corollary 8** (Summary Statistics). *Let T be a text of length n and $\mathsf{Stat}(T, i, j)$ be an $\mathcal{O}(1)$-size statistic computable, for any fragment $T[i \mathinner{..} j]$, in $\mathcal{O}(1)$ time after preprocessing using space s_{Stat}. Let $\bigodot$ be an associative and commutative aggregation function for the output of Stat that can combine two values in $\mathcal{O}(1)$ time. There is an $\mathcal{O}(n + s_{\mathsf{Stat}})$-space data structure that, given a pattern P of length m, computes $\bigodot_{i \in \mathsf{Occ}_T(P)} \mathsf{Stat}(T, i, i + |P| - 1)$ in $\mathcal{O}(m)$ time.*

We see an application of this corollary in Section 4.2 and many more in Section 4.4. Furthermore, in Section 7, we show how to construct the data structure in $\widetilde{\mathcal{O}}(n\sqrt{n})$ time for a class of problems in this framework.

Proof. We will construct a data structure D that fulfills the requirements from Theorem 2. The result then follows from Theorem 2.

As preprocessing, we construct the suffix tree of T (exactly as we do in Theorem 2) and, in addition, do the preprocessing for Stat. The data structure has size $|D| = \mathcal{O}(n + s_{\mathsf{Stat}})$.

Upon a query P, we find all occurrences of P in T in $\mathcal{O}(m + |\mathsf{Occ}_T(P)|)$ time using the suffix tree of T, and, for each occurrence $i \in \mathsf{Occ}_T(P)$, we compute $\mathsf{Stat}(T, i, i+m-1)$ in $\mathcal{O}(1)$ time each. We next combine these $|\mathsf{Occ}_T(P)|$ values in $\mathcal{O}(|\mathsf{Occ}_T(P)|)$ time using repeated applications of $\bigodot$, obtaining the desired output in total query time $\mathcal{O}(m + |\mathsf{Occ}_T(P)|)$. ◀

▶ Remark 9. If the querying algorithm of D from Theorem 2 or the functions Stat or $\odot$ from Corollary 8 have worse running times, the resulting data structure after applying our results degrades gracefully. In particular, if they require logarithmic time, then we only get an extra logarithmic factor in our query time as well. This is easy enough to verify by adding this overhead to the calculations in the proofs of Theorem 2 and Corollary 8. We omitted this to simplify the presentation (most importantly, to avoid overly complex statements).

4 Applications

We first present three applications of our framework, where it either yields a novel counting structure for a widely-studied reporting problem, improves upon the best known data structure for a problem, or enriches an existing index with natural non-trivial query types. Then, we recover several known results, thus unifying the treatment of a number of problems.

4.1 Counting Consecutive Occurrences

Given a string T, an integer interval $[\alpha, \beta]$, and a pattern P, a pair (i, j) is called an (α, β)-*consecutive occurrence* of P in T if $i, j \in \mathsf{Occ}_T(P)$, there is no other occurrence of P between i and j, and $\alpha \leq j - i \leq \beta$. We denote this set of pairs by $\mathsf{Occ}_T^{\alpha\beta}(P)$. Many variants of consecutive occurrences have been studied in the literature [9, 10, 11, 12, 13, 16, 30, 38, 40, 45, 47, 48]. Here, we consider the counting version of the most basic variant.

Consecutive Occurrences Counting (COC)
Preprocess: A text T of length n and an integer interval $[\alpha, \beta]$.
Query: Given a pattern P of length m, return $\lvert\mathsf{Occ}_T^{\alpha\beta}(P)\rvert$.

Applying Theorem 2 to the $\mathcal{O}(n \log n)$-space data structure of Navarro and Thankachan [47] for optimal reporting of $\mathsf{Occ}_T^{\alpha\beta}(P)$ yields an $\mathcal{O}(n \log n)$-space data structure with $\mathcal{O}(m)$-time queries. However, we can do better. We rely on the following auxiliary data structure.

▶ **Lemma 10** ([14]). *Given an array A of n elements, there exists an $\mathcal{O}(n)$-space data structure, so that given integers $i, k > 0$, it reports the elements of $A[i \,..\, i + k - 1]$ in sorted order in the optimal $\mathcal{O}(k)$ time.*

▶ **Theorem 11.** *There exists an $\mathcal{O}(n)$-space data structure for COC with $\mathcal{O}(m)$ query time.*

Proof. We construct the suffix tree of T, storing the labels of outgoing edges at each node using Lemma 7, and its suffix array preprocessed using Lemma 10. We store the answers for the at most n substrings U of T with $|\mathsf{Occ}_T(U)| > |U|$ (Theorem 1) in the suffix tree. The total space is $\mathcal{O}(n)$. Upon a query P, if the answer is stored, we return it in $\mathcal{O}(m)$ time by spelling P in the suffix tree, thus reaching the node where it is stored. Otherwise, by using Lemma 10, we obtain the *sorted* array A of the at most m occurrences of P in T using the corresponding suffix array range in $\mathcal{O}(|A|)$ time. Consecutive occurrences of P in T correspond to adjacent elements in A; we iterate over A in $\mathcal{O}(|A|)$ time, incrementing our count whenever $\alpha \leq A[k+1] - A[k] \leq \beta$. Since $|A| \leq m$, the query time is $\mathcal{O}(m)$. ◀

Interestingly, we can lift the restriction that *there is no in-between occurrence*, and count *all pairs* (i, j), such that $i, j \in \mathsf{Occ}_T(P)$ and $\alpha \leq j - i \leq \beta$. These are known as (α, β)-*gapped occurrences* [10]. For a fixed j, the value $(A[j] - A[i])$ decreases as i increases, so the valid indices $i < j$ form a contiguous block. That lets us maintain two pointers ("fingers") across the whole scan instead of restarting a search for every j. The query time is thus still $\mathcal{O}(m)$.

4.2 Strings with Utilities

A *string with utilities* (u-string, in short) is a pair (T, w), consisting of a string $T \in \Sigma^n$ and a function w mapping each $i \in [1, n]$ to a real number called the *utility* of $T[i]$. The *local utility* of a fragment $T[i\,..\,j]$ of T is defined as $\sum_{k=i}^{j} w[k]$. The *global utility* of a pattern $P \in \Sigma^m$ in T is defined as $\sum_{i \in \mathsf{Occ}_T(P)} \sum_{j=i}^{i+m-1} w[j]$. See [28] for a survey. The following problem of indexing u-strings was introduced and studied by Bernardini et al. [8].

Useful String Indexing (USI)
Preprocess: A u-string (T, w) of length n.
Query: Given a pattern P of length m, return its global utility in T.

Bernardini et al. [8] presented an $\mathcal{O}(n^2/\tau)$-space data structure with $\mathcal{O}(m + \tau)$-time queries. We present a substantial improvement:

▶ **Theorem 12.** *There exists an $\mathcal{O}(n)$-space data structure for USI with $\mathcal{O}(m)$ query time.*

Proof. We use Corollary 8 setting $\mathsf{Stat}(T, i, j) \coloneqq \sum_{k=i}^{j} w[k]$ and $\odot \coloneqq \sum$. As adding two numbers takes constant time, it suffices to show how to preprocess T and w to enable $\mathcal{O}(1)$-time access to Stat. This is possible using the prefix sums of w. That is, in preprocessing, we construct $\Pi_w[i] \coloneqq \sum_{j \le i} w[j]$, for all $i \in [1, n]$. Then, for an occurrence $P = T[i\,..\,j]$, we can compute $\sum_{k=i}^{j} w[k] = \Pi_w[j] - \Pi_w[i-1]$ in $\mathcal{O}(1)$ time. ◀

For simplicity of presentation, we have used the *addition* operator for both the local and the global utility in the above definitions. We will generalize Theorem 12 in Section 7.

4.3 Weighted Sequences

In a *weighted sequence* T of length n over an alphabet Σ, in every position $i \in [1, n]$, every letter of the alphabet is associated with a probability of occurrence such that the sum of probabilities at each position equals 1. We denote by $p_i^{(T)}(c)$ the occurrence probability of letter c at position i of T. The classical pattern matching problem extends naturally to weighted sequences. Given a string P called a pattern, a weighted sequence T called a text, both over an alphabet Σ, and a threshold probability $\frac{1}{z}$, the task is to find all positions i in T where $\mathrm{Prob}(P, i) \coloneqq \prod_{j=0}^{|P|-1} p_{i+j}^{(T)}(P[j]) \ge \frac{1}{z}$; $\mathrm{Prob}(P, i)$ is called the occurrence probability of P at position i of T. Barton et al. [4] showed that we can construct an index of $\mathcal{O}(nz)$ words of space supporting pattern matching queries in the optimal time $\mathcal{O}(m + |\mathsf{Occ}_T^z(P)|)$, where $\mathsf{Occ}_T^z(P)$ is the set of occurrences of P in T. To the best of our knowledge, there is no known index that can find the heaviest occurrence of P in T among the ones in $\mathsf{Occ}_T^z(P)$ or the average probability of the occurrences in $\mathsf{Occ}_T^z(P)$ in optimal time. The two corresponding problems are formally defined as follows.

Heaviest z-Occurrence (HzO)
Preprocess: A weighted sequence T of length n and a threshold $\frac{1}{z}$.
Query: Given a pattern P of length m, return $\max\{\mathrm{Prob}(P, i) \mid i \in \mathsf{Occ}_T^z(P)\}$.

Average z-Occurrence Probability (AzOP)
Preprocess: A weighted sequence T of length n and a threshold $\frac{1}{z}$.
Query: Given a pattern P of length m, return $\sum_{i \in \mathsf{Occ}_T^z(P)} \mathrm{Prob}(P, i)/|\mathsf{Occ}_T^z(P)|$.

▶ **Theorem 13.** *There exists an $\mathcal{O}(nz)$-space data structure that can answer HzO or AzOP queries in $\mathcal{O}(m)$ time.*

Proof. Barton et al. [4] showed the following: for every weighted sequence T of length n and every $\frac{1}{z}$, we can construct a family $\mathcal{F}(T) = (S_j, w_j)_{j=1}^{\lfloor z \rfloor}$ of $\lfloor z \rfloor$ u-strings, each of length n, that carries all the information about the strings occurring in T with probability $p \geq \frac{1}{z}$. Formally, P occurs at position i of T with occurrence probability $p \geq \frac{1}{z}$ if and only if there exists a string $S_j \in \mathcal{F}(T)$ such that $P = S_j[i\,..\,i+m-1]$, $\prod_{k=i}^{i+m-1} w_j[k] = p$, and $p \geq \frac{1}{z}$.

We first construct the suffix tree of $Y = S_1\$ \dots \$S_{\lfloor z \rfloor}\$$, where \$ is a unique letter that does not occur in $S_1, \dots, S_{\lfloor z \rfloor}$. We also store the labels of outgoing edges at each node using Lemma 7. This takes $\mathcal{O}(nz)$ space. Using this, upon a query, we can compute $\mathsf{Occ}_Y(P)$ in $\mathcal{O}(m+|\mathsf{Occ}_Y(P)|)$ time. Similarly to the proof of Theorem 12, we can also, after preprocessing, access the probability of each of these occurrences in $\mathcal{O}(1)$ time. From this, we directly get an $\mathcal{O}(nz)$-space data structure to answer HzO or AzOP queries in $\mathcal{O}(m + |\mathsf{Occ}_Y(P)|)$ time. Applying Theorem 2 to this yields the claimed trade-off. ◀

4.4 More Applications

Beyond the three headline applications described above, the combinatorial insight (Theorem 1) leads to efficient data structures for a wide range of summary statistics queries. Many of them have a known efficient data structure, so our main contribution here is that Theorem 2 and Corollary 8 provide a *simple* and *unified* framework for this set of problems.

For instance, for $\mathsf{Stat}(T, i, j) \coloneqq 1$ and $\bigodot \coloneqq \sum$, we get the classical counting of occurrences of P in T; for $\mathsf{Stat}(T, i, j) \coloneqq \sum_{k=i}^{j} w[k]$ (for a weight array w of length $n = |T|$) and $\bigodot \coloneqq \sum$, we solve the USI problem. This directly extends to combinations of these statistics (e.g., the average weight of the occurrences of the pattern).

Let us give a few more examples. Assume that w is a length-n weight array given at preprocessing time and denote by m the length of P.

▶ **Example 14** (Heaviest Occurrence)**.** Set $\mathsf{Stat}(T, i, j) \coloneqq \sum_{k=i}^{j} w[k]$ and $\bigodot \coloneqq \max$. Corollary 8 yields an $\mathcal{O}(n)$-space data structure answering such queries in $\mathcal{O}(m)$ time. This is a natural follow-up to the USI problem where the heaviest occurrence might then be a particularly important one for a given application. Also note that the data structure can be straightforwardly extended to output the position of the heaviest occurrence instead of only the weight. Other statistics that can be obtained in a similar fashion include the classical problems of finding the first (leftmost) or (rightmost) last occurrence of a given pattern.

▶ **Example 15** (Higher Moments)**.** Let $r \geq 0$. Set $\mathsf{Stat}(T, i, j) \coloneqq (\sum_{k=i}^{j} w[k])^r$ and $\bigodot \coloneqq \sum$. Corollary 8 yields an $\mathcal{O}(n)$-space data structure answering such queries in $\mathcal{O}(m)$ time. Observe that for $r = 0$, we recover counting of occurrences; for $r = 1$, we recover USI. For $r = 2$, we get the sum of squares of occurrence weights. Since the variance of a random variable X is $\mathbb{E}[X^2] - \mathbb{E}[X]^2$, we can use the data structure for $r = 1$ and $r = 2$ to deduce the variance of the occurrence weights of a pattern. This is particularly well-motivated within the framework of strings with utilities [7, 8], where w quantifies the profit, risk, or confidence score of each position in T. Higher moments $r \geq 3$ serve as fundamental building blocks for more complex statistics [49] and we thus note that they are supported by the framework.

▶ **Example 16** (Pattern Matching with Properties)**.** In this problem, in addition to T we are given a *hereditary property* Π, which is a family of integer intervals contained in $[1, n]$ (hereditary means that it is closed under subintervals). Our goal is to preprocess T so that, for a pattern P, we can report all occurrences of P in T which, interpreted as intervals,

belong to Π. The property Π can be represented in $\mathcal{O}(n)$ space using an array L such that the longest interval in Π starting at position i is $[i, i + L[i] - 1]$. The problem was studied in [1, 4, 17, 37, 39]; it is closely related to indexing weighted sequences. We obtain a general analogue to Corollary 8 by wrapping any Stat function in a check to only consider occurrences that obey the property restriction (e.g., [4] shows how to perform this check optimally).

▶ **Example 17** (Non-Overlapping Occurrences)**.** Given a string S and two fragments of S, $S[i\,..\,i']$ and $S[j\,..\,j']$ with $i \leq j$, we say that the fragments are *non-overlapping* if $j > i'$. The set of non-overlapping occurrences of a string P in a string T is a *largest* set of occurrences of P in T that are pairwise non-overlapping. We denote this set by $\mathsf{Occ}_T^{\mathrm{NO}}(P)$. There exists a well-known $\mathcal{O}(n)$-space data structure with optimal $\mathcal{O}(m + |\mathsf{Occ}_T^{\mathrm{NO}}(P)|)$ query time for the reporting version [21]. Thus, applying Theorem 2 to the reporting data structure directly yields an $\mathcal{O}(n)$-space data structure with $\mathcal{O}(m)$-time queries for the counting version.

The counting version can alternatively be solved via the minimal augmented suffix tree (MAST) [2, 15]. Note that the solution of [21] is not based on MAST, so together with Theorem 2 this yields an alternative construction for the counting version.

▶ **Example 18** (Covered Positions)**.** Given a pattern P, we want to count the positions in T that are in some occurrence of P. We cannot deduce this from the number of occurrences, as we do not want to double-count positions that are in more than one occurrence. It is easy to obtain an $\mathcal{O}(n)$-space data structure with $\mathcal{O}(m + |\mathsf{Occ}_T(P)|)$-time queries using the suffix tree and the suffix array preprocessed using Lemma 10: on a query P, we iterate over all *sorted* occurrences and for two consecutive occurrences $i < j$, we add $\min(m, j - i)$ to the count. Applying Theorem 2 yields an $\mathcal{O}(n)$-space data structure with $\mathcal{O}(m)$-time queries.

Note that for this task, an $\mathcal{O}(n)$-space data structure with optimal query time is known based on the cover suffix tree [42, 48]. This data structure is more intricate (and has other uses, too), so we think this is a cute simplification for this task.

▶ **Example 19** (Overlapping Occurrences)**.** It is also natural to ask for the maximum number of occurrences that *do* overlap. Two ways to make this precise are to ask for the size of a largest set of occurrences such that either: (a) they all mutually overlap; or (b) they all transitively overlap (i.e., two occurrences do not need to overlap if they are connected by some sequence of overlaps with other occurrences from the set). For both of these flavors, we obtain $\mathcal{O}(n)$-space data structures with optimal query times using Theorem 2. For $\mathcal{O}(m+|\mathsf{Occ}_T(P)|)$ query time, simply find the sorted set of occurrences of P as in Example 18. Then perform a scanline algorithm, incrementing a counter every time an occurrence starts and decrementing it when one ends. The maximum value of this counter yields (a), and the maximum number of occurrences processed between two consecutive points where the counter is zero yields (b), both of which are simple enough to track in $\mathcal{O}(|\mathsf{Occ}_T(P)|)$ time.

5 Generalized Combinatorial Lemma and Applications

▶ **Lemma 20.** *For any rooted tree $\mathcal{T}$ with L leaves over a set $\mathcal{V}$ of nodes, and $\tau \in [1, L]$, we have*

$$|\{v \in \mathcal{V} \mid \tau \cdot \mathit{depth}(v) < |\mathcal{L}(v)|,\ \mathit{depth}(v) > 0\}| \leq L/\tau,$$

where $\mathit{depth}(v)$ is the depth of node v and $\mathcal{L}(v)$ is the set of leaf descendants of v.

Proof. Let $\mathcal{S} = \{v \in \mathcal{V} \mid \tau \cdot \mathsf{depth}(v) < |\mathcal{L}(v)|,\ \mathsf{depth}(v) > 0\}$. By definition, we have $|\mathcal{L}(v)| \geq \tau$ for every $v \in \mathcal{S}$. We will define a function ϕ that maps each node in $\mathcal{S}$ to a set of exactly τ of its leaf descendants such that $\phi(u) \cap \phi(v) = \emptyset$ for every distinct $u, v \in \mathcal{S}$. Since there are only L leaves in total, this immediately leads to a bound of $|\mathcal{S}| \leq L/\tau$.

For each node v, let $\ell_{v,1}, \ell_{v,2}, \ldots, \ell_{v,|\mathcal{L}(v)|}$ be the leaves in $\mathcal{L}(v)$ ordered by their rank in an in-order traversal of $\mathcal{T}$. For each node $v \in \mathcal{S}$, we set

$$\phi(v) := \{\ell_{v,i} \mid i = (\mathsf{depth}(v) - 1) \cdot \tau + j, \text{ for } j \in [1, \tau]\}.$$

We consider the following two cases:

Case 1: Neither u nor v is an ancestor of the other. Because the subtrees rooted in u and v are disjoint, we have $\mathcal{L}(u) \cap \mathcal{L}(v) = \emptyset$. Since $\phi(u) \subseteq \mathcal{L}(u)$ and $\phi(v) \subseteq \mathcal{L}(v)$, it immediately follows that $\phi(u) \cap \phi(v) = \emptyset$.

Case 2: u is an ancestor of v or vice versa. Assume without loss of generality that u is an ancestor of v. In this case, the leaves $\ell_{v,1}, \ldots, \ell_{v,|\mathcal{L}(v)|}$ appear as some contiguous interval within $\ell_{u,1}, \ldots, \ell_{u,|\mathcal{L}(u)|}$. In other words, there exists some $k \geq 0$ such that $\ell_{v,i} = \ell_{u,i+k}$ for all $i \in [1, |\mathcal{L}(v)|]$. Suppose that there exists some leaf $w \in \phi(u) \cap \phi(v)$. By the definition of ϕ, this means $\ell_{v,(\mathsf{depth}(v)-1)\cdot\tau+j_1} = \ell_{u,(\mathsf{depth}(u)-1)\cdot\tau+j_2}$ for some $j_1, j_2 \in [1, \tau]$. From this, we get $k = ((\mathsf{depth}(u) - 1) \cdot \tau + j_2) - ((\mathsf{depth}(v) - 1) \cdot \tau + j_1) = \tau \cdot (\mathsf{depth}(u) - \mathsf{depth}(v)) + j_2 - j_1$. Because $\mathsf{depth}(v) > \mathsf{depth}(u)$ and $j_2 - j_1 < \tau$, this leads to $k < 0$, a contradiction. As a consequence, the leaf w cannot exist. This means $\phi(u) \cap \phi(v) = \emptyset$.

Because $\phi(u) \cap \phi(v) = \emptyset$ for all distinct $u, v \in \mathcal{S}$, we have that $|\bigcup_{v\in\mathcal{S}} \phi(v)| = \sum_{v\in\mathcal{S}} |\phi(v)| = |\mathcal{S}| \cdot \tau$. Moreover, since there are L leaves in total, we have $|\bigcup_{v\in\mathcal{S}} \phi(v)| \leq L$ which leads to the bound $|\mathcal{S}| \cdot \tau \leq L$ and thus $|\mathcal{S}| \leq L/\tau$. ◀

▶ **Corollary 3.** *In any string T of length n, for any integer $\tau \in [1, n]$, the number of non-empty substrings U of T for which $|U| \cdot \tau < |\mathit{Occ}_T(U)|$ is at most n/τ.*

Proof. In the trie $\{T[1\,..\,n]\$, T[2\,..\,n]\$, \ldots, T[n]\$\}$, for a unique letter \$ that does not occur in T, each internal node v corresponds one-to-one to a substring U of T, with $|U| = \mathsf{depth}(v)$. Moreover, the leaves $\mathcal{L}(v)$ correspond to occurrences of U. As there are n leaves in total, the statement follows directly by applying Lemma 20 to the trie. ◀

Analogously to the applications of Theorem 1, Corollary 3 can be applied for designing space-efficient data structures for counting queries. We next showcase how we can obtain a *compact* data structure for counting non-overlapping occurrences in near-optimal time, which was mentioned as a "challenging" open problem by Gibney et al. [32].

Ganguly et al. [29] showed that all we need for reporting non-overlapping occurrences is a suffix tree (or any of its space-efficient counterparts) of text T. The time complexity of their query algorithm is $\mathcal{O}(\text{search}(P) + t_{\mathsf{SA}}|\mathsf{Occ}_T^{\mathrm{NO}}(P)| + \text{sort}_n(|\mathsf{Occ}_T^{\mathrm{NO}}(P)|))$, where $\text{search}(P)$ denotes the time for computing the suffix array range of pattern P, t_{SA} denotes the time for accessing a given entry in the suffix array (or inverse suffix array), and $\text{sort}_n(x)$ denotes the time for sorting a subset of $\{1, 2, \ldots, n\}$ of size $\mathcal{O}(x)$.

▶ **Theorem 21.** *Given a text T of length n over an integer alphabet $[0, \sigma)$, we can construct a data structure of $\mathcal{O}(n \log \sigma)$ bits so that, given a pattern P of length m, it outputs $|\mathit{Occ}_T^{NO}(P)|$ in $\mathcal{O}(m \log n \log_\sigma^\epsilon n)$ time for an arbitrarily small constant $\epsilon > 0$.*

Proof. Let us set $\tau := \lceil \log_\sigma n \rceil$. By Corollary 3, the number of substrings U of T that satisfy $|U| \cdot \tau < |\mathsf{Occ}_T(U)|$ is at most $\lfloor n/\tau \rfloor$. We store the answer for each such substring using the deterministic static dictionary from Lemma 7, keyed by the pattern's suffix array range start index and its length. Since each key-value pair requires $\mathcal{O}(\log n)$ bits, the dictionary takes $\mathcal{O}(n \log \sigma)$ bits of space. Then, for any pattern P of length m, either the answer is stored

or we have at most $m\lceil\log_\sigma n\rceil$ occurrences of P in T. Using the reporting data structure of Ganguly et al. [29], we achieve a query time of $\mathcal{O}(\text{search}(P) + t_{\mathsf{SA}} m \log_\sigma n + \text{sort}_n(m \log_\sigma n))$. Using the compressed suffix array [34, 51], we attain $t_{\mathsf{SA}} = \mathcal{O}(\log^\epsilon_\sigma n)$ using $\mathcal{O}(n \log \sigma)$ extra bits of space. Using binary search as in [44] over the suffix array [34] with the help of the LCP array [51], we get $\text{search}(P) = \mathcal{O}(m + \log n \log^\epsilon_\sigma n) \in \mathcal{O}(m \log n \log^\epsilon_\sigma n)$. Finally, using fast integer sorting [36], we get $\text{sort}_n(m \log_\sigma n) = \mathcal{O}(m \log_\sigma n \log\log n)$. The total space usage is $\mathcal{O}(n \log \sigma)$ bits and the query time is $\mathcal{O}(m \log n \log^\epsilon_\sigma n)$. ◀

6 Another Generalization of the Combinatorial Lemma with Applications in the IPM Model

▶ **Corollary 4.** *In any string T of length n, for any integer $\tau \in [1, n]$, the number of non-empty substrings U of T for which $|U| < \tau \cdot |\mathsf{Occ}_T(U)|$ is at most $n\tau$.*

Proof. Consider the string S formed by concatenating τ copies of T separated by a unique letter \$ that does not occur in T. In particular, consider the string:

$$S := (T\$)^\tau = \underbrace{T\$T\$\dots\$T\$}_{\tau \text{ copies of } T\$}.$$

The total length of S is $|S| = \tau(n+1)$.

Let U be a non-empty substring of T that satisfies the condition in the statement, namely $|U| < \tau \cdot |\mathsf{Occ}_T(U)|$. Because the string U does not contain \$, it cannot overlap the boundaries between the concatenated copies of T in S. Therefore, every occurrence of U in S must be fully contained within exactly one of the τ individual copies of T. This yields:

$$|\mathsf{Occ}_S(U)| = \tau \cdot |\mathsf{Occ}_T(U)|.$$

Substituting this relation into our inequality, we conclude that U must satisfy:

$$|U| < |\mathsf{Occ}_S(U)|.$$

Consider the suffix trie of S (excluding the τ suffixes of S starting with \$). This trie has exactly $n\tau$ leaves. By applying Lemma 5 to this trie, we conclude that the number of non-empty substrings U satisfying the statement's condition is at most $n\tau$. ◀

We apply Corollary 4 to USI in the IPM model. Namely, the pattern P is now specified in $\mathcal{O}(1)$ time and space using two integers i and j such that $P = T[i\,..\,j]$ and $m = |P| = j - i + 1$. We term the corresponding queries *internal* USI queries, and show the following result.

▶ **Theorem 22.** *For every string T of length n and every integer $\tau \in [1, n]$, there exists an $\mathcal{O}(n\tau)$-space data structure for USI that can answer internal USI queries in time $\mathcal{O}\big(1 + m/\tau\big)$.*

Proof. We construct the suffix tree and the suffix array of T. We also preprocess the suffix tree for answering weighted ancestor queries in $\mathcal{O}(1)$ time using $\mathcal{O}(n)$ space [5, 31]. A *weighted ancestor query* (WAQ) locates the locus of a substring $T[p\,..\,q]$ in the suffix tree of T: given the node u of the tree corresponding to $T[p\,..\,n]$, the query locates the highest ancestor v of u, such that the string written on the root-to-v path has length at least $q - p + 1$.

For every substring U with $|U| < \tau \cdot |\mathsf{Occ}_T(U)|$, we also store the answer using the deterministic static dictionary from Lemma 7, keyed by the pattern's suffix array range start index and its length. By Corollary 4, there are $\mathcal{O}(n\tau)$ such patterns; thus these answers take $\mathcal{O}(n\tau)$ space. Finally, we compute $\Pi_w[i] := \sum_{j \le i} w[j]$, for all $i \in [1, n]$, taking $\mathcal{O}(n)$ space.

Given a query (i, j), with $m = j - i + 1$, we find the locus of $P = T[i\,..\,i + m - 1]$ in the suffix tree of T in $\mathcal{O}(1)$ time using a WAQ, and distinguish two cases:

1. If $|\mathsf{Occ}_T(P)| > m/\tau$, the pattern's exact answer is explicitly stored in the dictionary. We return it in $\mathcal{O}(1)$ additional time.
2. If $|\mathsf{Occ}_T(P)| \leq m/\tau$, we traverse the subtree of its locus to locate all $|\mathsf{Occ}_T(P)|$ occurrences. For each occurrence, we compute the required local utility using prefix sums in $\mathcal{O}(1)$ time. The additional time spent is $\mathcal{O}(|\mathsf{Occ}_T(P)|) = \mathcal{O}(m/\tau)$.

In both cases, the extra work is bounded by $\mathcal{O}(1 + m/\tau)$. Thus, the total query time is $\mathcal{O}(1 + m/\tau)$ using an index of $\mathcal{O}(n\tau)$ space as claimed. ◀

7 Efficient Construction Algorithm

In this section, we show how to efficiently construct the indexing data structure for a class of problems in the framework of Corollary 8. Let us fix a text T of length n. The challenge is to compute statistics for every substring U of T with $|U| < |\mathsf{Occ}_T(U)|$. We have at most n such strings U by Theorem 1. Naïvely querying each of these strings may take $\Theta(n^2)$ time, given that each may have $\Theta(n)$ occurrences. We show how the associated statistics $\bigodot_{i \in \mathsf{Occ}_T(U)} \mathsf{Stat}(T, i, i + |U| - 1)$ can be computed in total $\widetilde{\mathcal{O}}(n\sqrt{n})$ time using $\mathcal{O}(n)$ space by bounding the total number of occurrences of short substrings with many occurrences and exploiting the implied periodicity of long ones.

We decompose the problem based on the quantities $|U|$ and $|\mathsf{Occ}_T(U)|$, by considering:

- short substrings with many occurrences, satisfying $|U| \leq 5\lceil\sqrt{n}\rceil$ and $|U| < |\mathsf{Occ}_T(U)|$;
- long substrings with many occurrences, satisfying $|U| > 5\lceil\sqrt{n}\rceil$ and $|U| < |\mathsf{Occ}_T(U)|$.

We next describe the class of problems in the framework of Corollary 8.

The class of problems. We follow the framework of Corollary 8, with the following additional properties required to make the construction work:

- $\mathsf{Stat}(T, i, j)$ can be queried in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$-time (and thus $\mathcal{O}(n)$-space) preprocessing, for all i, j (e.g., using prefix sums as in Theorem 12 if the operator is addition, or using range minimum queries [6] if the operator is the minimum);
- $\mathsf{Stat}(T, i, j) = \bigodot_{k=i}^{j} \mathsf{Stat}(T, k, k)$, for all i, j. (For simplicity of presentation, we use the same operator as in Corollary 8.)

Additionally, let us denote by $\otimes$ the repeated application of $\odot$ using a single operand; i.e., $k \otimes x = \underbrace{x \odot \cdots \odot x}_{k}$, for some integer $k > 0$. For some operators, such as addition or the minimum, this can be trivially evaluated in $\mathcal{O}(1)$ time. In general, it can be evaluated in $\mathcal{O}(\log k)$ time by repeatedly "squaring" smaller values. Thus, our bounds generally carry logarithmic factors, which can be shaved if the operator is addition or the minimum.

Preprocessing. We construct the suffix tree of T in $\mathcal{O}(n \log n)$ time [52] using $\mathcal{O}(n)$ space. Dictionaries storing the labels of outgoing edges to support efficient navigation can be constructed within the same complexities using Lemma 7.

The short substrings are handled using Lemma 23.

▶ **Lemma 23.** *Given Stat, the suffix tree of T, and an integer $\alpha > 0$, we can compute the associated statistic of each substring U of T, with $|U| \leq \alpha$ and $|U| < |\mathsf{Occ}_T(U)|$, in $\mathcal{O}(n\alpha)$ total time using $\mathcal{O}(n)$ space.*

Proof. We start with a straightforward claim.

▷ Claim 24. The total number of occurrences of substrings U of T, with $|U| \leq \alpha$, is $\mathcal{O}(n\alpha)$.

Proof. The number of possible starting positions for a substring of length m in T is $n-m+1$.

$$\sum_{m=1}^{\alpha}(n-m+1)=\sum_{m=1}^{\alpha}(n+1)-\sum_{m=1}^{\alpha}m<(n+1)\alpha=\mathcal{O}(n\alpha).$$ ◁

For every node v in a suffix tree, we denote by $\mathsf{str}(v)$ the concatenation of the edge labels from the root to v. Using a DFS traversal of the suffix tree of T, we store $|\mathsf{Occ}_T(U)|$, for every explicit node v, with $U \coloneqq \mathsf{str}(v)$. This takes $\mathcal{O}(n)$ time. Then, for every explicit or implicit node v of the suffix tree, such that $|U| \leq \alpha$ and $|U| < |\mathsf{Occ}_T(U)|$, with $U \coloneqq \mathsf{str}(v)$, we compute the statistic $\bigodot_{i\in\mathsf{Occ}_T(U)} \mathsf{Stat}(T, i, i+|U|-1)$ as follows. For every occurrence $U = T[i\,..\,j]$ in T, we compute its value in $\mathcal{O}(1)$ time by the assumption of the framework. (If v is implicit, we have $\mathsf{Occ}_T(U) = \mathsf{Occ}_T(U')$, with $U' \coloneqq \mathsf{str}(v')$ and v' the closest explicit descendant of v.) The construction time is bounded by $\mathcal{O}(n\alpha)$ by Claim 24. By Theorem 1, we consider at most n nodes, and so we store $\mathcal{O}(n)$ values. ◀

For long substrings, we will exploit the underlying periodic structure.

▶ **Definition 25** (Period). *The* period *of a string $S = S[1]S[2]\dots S[|S|]$, denoted by $\mathsf{per}(S)$, is the smallest positive integer p such that $S[i] = S[i+p]$ for all $i \in [1, |S|-p]$.*

A string S is called *periodic* if $\mathsf{per}(S) \leq |S|/2$. Lemma 26 shows that long substrings with many occurrences are in fact highly-periodic. As a consequence, given a long frequent substring, it suffices to query its Stat value only within similar periodic fragments of the text.

▶ **Lemma 26.** *Let $\beta \in \mathbb{Z}_{>0}$. For any string U with $|U| \cdot |\mathsf{Occ}_T(U)| > \beta \cdot n$, we have $\mathsf{per}(U) \leq \frac{1}{\beta}|U|$.*

Proof. Let $k \coloneqq |\mathsf{Occ}_T(U)|$ and $m \coloneqq |U|$. We have $k \cdot m > \beta \cdot n$. By the hypothesis $(k > \beta \cdot n/m)$ and the pigeonhole principle, there exists an index i such that $|\mathsf{Occ}_T(U) \cap [i, i+m-1]| \geq \beta+1$. Thus, there exist two occurrences $j_1 < j_2$ of U in T, with $j_1, j_2 \in [i, i+m-1]$ and $d = j_2 - j_1 \leq \frac{m}{\beta}$. These two occurrences of U at distance d have an overlap of length $m-d$. Thus, $\mathsf{per}(U) \leq d \leq \frac{m}{\beta} = \frac{1}{\beta}|U|$. ◀

We make use of the following standard definitions.

▶ **Definition 27** (Lyndon root). *Given a periodic string R with period $\mathsf{per}(R)$, the* Lyndon root *of R is the lexicographically smallest rotation of $R[1\,..\,\mathsf{per}(R)]$.*

▶ **Definition 28** (Lyndon representation). *Given a periodic string R with Lyndon root L, the* Lyndon representation *of R is the concatenation $L_1L^kL_2$, where L_1 is a proper suffix of L, L_2 is a proper prefix of L, and k is an integer referred to as the* exponent*. Further, we refer to L^k as the* body *of R, and L_1 and L_2 as the front and back* extensions *of R, respectively.*

▶ **Definition 29** (Run). *A fragment $R = S[i\,..\,j]$ of a string S is called a* run *of S if $\mathsf{per}(R) \leq |R|/2$ (i.e., R is a periodic string) and extending R either to the left or to the right (if possible) would result in an increase of its period, that is, $S[i-1] \neq S[i-1+\mathsf{per}(R)]$ (or $i = 1$) and $S[j+1] \neq S[j+1-\mathsf{per}(R)]$ (or $j = |S|$).*

We denote by $\mathcal{R}_L(T)$ the set of all runs in a string T that have Lyndon root L and length at least $5\lceil\sqrt{n}\rceil$. By Lemma 26, every occurrence of a long frequent substring U must be a fragment of a run from $\mathcal{R}_L(T)$, where L is the Lyndon root of U.

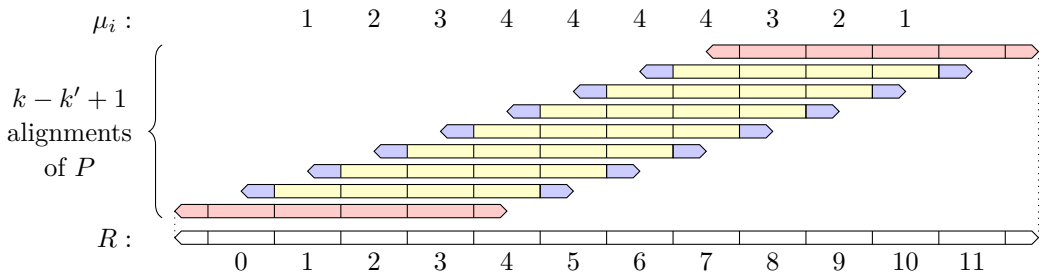

Figure 1 Occurrences of a pattern P with a run R, where $k = 12$ is the exponent of R and $k' = 4$ is the exponent of P. The first occurrence and the last occurrence, both colored red, exist conditionally depending on the extensions of the two runs; the inner occurrences always exist. The bodies of the inner occurrences (yellow) are computed separately from the extensions (blue). In this example, we have $\mu_\star = 4$.

▶ **Lemma 30.** *We are given a periodic fragment R of T by its Lyndon representation. Let L and k denote the Lyndon root and exponent of R, respectively. We can preprocess R in $\mathcal{O}(|R| \log k)$ time using $\mathcal{O}(|R|)$ space so that, given any periodic pattern P with Lyndon root L by its Lyndon representation, we can return P's statistic within R in $\mathcal{O}(\log k)$ time.*

Proof. Let $R = L_1 L^k L_2$ and $P = L'_1 L^{k'} L'_2$ be the Lyndon representations of R and P. We hence assume that $k \geq k'$, since otherwise P does not have any occurrences within R.

Every occurrence of P in R starts at a position aligning the occurrences of L in P with occurrences of L in R. Note that L is a primitive string (that is, if $L = U^k$ for a string U and an integer k, then $U = L$ and $k = 1$) and hence it has exactly two occurrences in string LL [22]. An illustration is provided in Figure 1.

It then follows that P occurs in R at least $k - k' - 1$ times and at most $k - k' + 1$ times. Note that we can align the first occurrence of L in P with the first (resp. the $(k - k' + 1)$th) occurrence of L in R to get a valid occurrence only if $|L'_1| \leq |L_1|$ (resp. $|L'_2| \leq |L_2|$). The other occurrences, obtained by aligning the first occurrence of L in P with the jth occurrence of L in R for $j \in [2, k - k']$, which we call *inner occurrences*, exist unconditionally. We compute the statistic of P in R in three steps:

1. the statistic of the bodies (i.e., $L^{k'}$) of P's inner occurrences;
2. the statistic of the extensions (i.e., L'_1 and L'_2) of P's inner occurrences;
3. the statistic of the at most two occurrences of P that are not inner, if they exist.

Inner occurrences body. We assume that $k \geq k' + 2$, as otherwise there are no inner occurrences. We denote by $X_0, \ldots, X_{k-1}$ the Stat values of the k occurrences of L in R. Moreover, we denote by $U_1, \ldots, U_{k-k'-1}$ the Stat values of the bodies of P's inner occurrences, where $U_i = X_i \odot \cdots \odot X_{i+k'-1}$. Combining the values of the inner occurrences, we get:

$$U_1 \odot \cdots \odot U_{k-k'-1} = (X_1 \odot \cdots \odot X_{k'}) \odot \cdots \odot (X_{k-k'-1} \odot \cdots \odot X_{k-2}) = \bigodot_{i=1}^{k-k'-1} \bigodot_{j=0}^{k'-1} X_{i+j}.$$

We denote by μ_i the multiplicity of X_i in the above aggregation. As illustrated in Figure 1, μ_i depends on k, k', and how close the ith occurrence of L in R is to an endpoint of the run. In particular, we have $\mu_1 = \mu_{k-2} = 1$. Then, as we move toward the middle of R (from either direction), the multiplicity might increase, by at most one per occurrence of L, but as each occurrence of L in R belongs to at most k' inner occurrences of P and we have a total of

$k-k'-1$ inner occurrences, we have $\mu_i = \min\{k-k'-1, k', i, k-i-1\}$. Let $\mu_\star := \mu_{\lfloor (k-1)/2 \rfloor}$, $y := \arg\min_j \{j \in [1, k-2] \mid \mu_j = \mu_\star\}$ and $z := \arg\max_j \{j \in [1, k-2] \mid \mu_j = \mu_\star\}$. Note that $y = \min\{k-k'-1, k'\}$, as for $i \geq y$, $\mu_i = \mu_\star$, while for $i < y$, $\mu_i = i$. By symmetry, $z = k-1-\min\{k-k'-1, k'\}$. Using the above formula for the multiplicities, we obtain:

$$\bigodot_{i=1}^{k-2} (\mu_i \otimes X_i) = \bigodot_{i=1}^{y-1} (i \otimes X_i) + (\mu_\star \otimes \bigodot_{i=y}^{z} X_i) + \bigodot_{i=z+1}^{k-2} ((k-i-1) \otimes X_i).$$

The values of the first and the last aggregations can be precomputed for all possible values of y and z, respectively, in $\mathcal{O}(k \log k)$ time and stored in $\mathcal{O}(k)$ space. For the middle aggregation, we can also precompute and store all $\mathcal{O}(k)$ relevant values since $z = k-1-y$. We can then evaluate $\mu_\star \otimes \bigodot_{i=y}^{z} X_i$ at query time in $\mathcal{O}(\log k)$ time obtaining the desired value. Thus, the values can be computed in $\mathcal{O}(\log k)$ time after $\mathcal{O}(|R| \log k)$-time preprocessing using $\mathcal{O}(|R|)$ space.

Extensions of inner occurrences. We describe how to compute the values of the front extensions (i.e., L'_1) of inner occurrences; the back extensions (L'_2) can be handled symmetrically. There are exactly $k-k'-1$ inner occurrences, whose front extensions are the length-$|L'_1|$ suffixes of the first $k-k'-1$ occurrences of L in R. We denote by X_i^F the value of the length-$|L'_1|$ suffix of the ith occurrence of L. We have

$$\bigodot_{i=0}^{k-k'-2} X_i^F = \bigodot_{i=1}^{k-k'-1} \mathsf{Stat}(R, s+pi-|L'_1|, s+pi-1),$$

where $p = \mathsf{per}(R) = |L|$ and s is the starting position of the first occurrence of L in R (meaning that $s+pi$ is the starting position of the ith occurrence of L). We precompute the values for all relevant k' and $|L'_1|$ using a recurrence on k'. There are $k \cdot p$ values to compute in total, so this pre-processing step takes $\mathcal{O}(kp) = \mathcal{O}(|R|)$ time.

Non-inner occurrences. The at most two non-inner occurrences of P can either align P's first occurrence of L with R's first occurrence of L or align P's last occurrence of L with R's last occurrence of L. This entails aligning the front (resp. back) extensions of the two runs, meaning that they only exist conditionally depending on the lengths of the extensions and the exponents k and k':

- if $k = k'$, then we only have a single non-inner occurrence, which exists if and only if $|L'_1| \leq |L_1|$ and $|L'_2| \leq |L_2|$;
- if $k > k'$, a non-inner occurrence aligning the front extensions exists if and only if $|L'_1| \leq |L_1|$ and a non-inner occurrence aligning the back extensions exists if and only if $|L'_2| \leq |L_2|$.

These conditions can be tested in constant time. If the occurrences exist, their values can be retrieved in constant time as well from Stat.

Wrap-up. After performing the described precomputations in $\mathcal{O}(|R| \log k)$ time using $\mathcal{O}(|R|)$ space, the statistic of any periodic pattern P can be queried in $\mathcal{O}(\log k)$ time. ◀

We make use of the following variant of runs.

▶ **Definition 31** (τ-Run). *A* τ-run *is a run of length at least* $3\tau - 1$ *with period at most* $\frac{1}{3}\tau$.

▶ **Lemma 32** ([19]). *For any* $\tau \in \mathbb{Z}_{>0}$, *the number of* τ*-runs in* $T \in \Sigma^n$ *is* $\mathcal{O}(n/\tau)$.

The following result can be used to bound the total length of all highly-periodic runs.

▶ **Lemma 33** ([20]). *For any $T \in \Sigma^n$, $\rho(T) \coloneqq \sum_{R\in\mathcal{R}(T)}(|R| - 2\cdot \mathsf{per}(R) + 1) = \mathcal{O}(n\log n)$.*

We are now ready to describe the algorithm for long substrings.

▶ **Lemma 34.** *Given Stat, the suffix tree of T, and an integer $\alpha \geq 5\lceil\sqrt{n}\rceil$, we can compute the statistic of each substring U of T with $|U| > \alpha$ and $|U| < |\mathsf{Occ}_T(U)|$, in $\widetilde{\mathcal{O}}(n^2/\alpha)$ total time using $\mathcal{O}(n)$ space.*

Proof. For any U with $|\mathsf{Occ}_T(U)| > |U| > \alpha$, we have $|U|\cdot|\mathsf{Occ}_T(U)| > \alpha^2$. By Lemma 26 and the fact that $\alpha \geq 5\lceil\sqrt{n}\rceil$, we know that $\mathsf{per}(U) \leq \frac{1}{25}|U|$. Let us consider the set $\mathcal{R}$ of runs R with $\mathsf{per}(R) \leq \frac{1}{25}|R|$ and $|R| \geq \alpha$. For each run $R \in \mathcal{R}$, there exists some non-negative integer i such that $|R| \geq 3\cdot 2^i\lfloor\alpha/3\rfloor$ and $\mathsf{per}(R) \leq 2^i\lfloor\alpha/3\rfloor/3$. By Lemma 32, the total number of such runs is $\mathcal{O}(n/\alpha\cdot\sum_{i=0}^{\infty}1/2^i) = \mathcal{O}(n/\alpha)$. Furthermore, by Lemma 33, the total length of all runs $R \in \mathcal{R}$ is in $\mathcal{O}(n\log n)$. We henceforth consider only runs in $\mathcal{R}$. We preprocess the suffix tree of T in $\mathcal{O}(n)$ time for answering WAQs in $\mathcal{O}(1)$ time [5].

We compute all runs in $\mathcal{R}$ in $\mathcal{O}(n)$ time [23]. This algorithm outputs a canonical representation of every run that includes a constant-size representation of its Lyndon root L. Using a DFS traversal of the suffix tree of T, we compute $|\mathsf{Occ}_T(U)|$, for every explicit node v of the suffix tree with $U \coloneqq \mathsf{str}(v)$. This takes $\mathcal{O}(n)$ time. Then, during another traversal, we find each explicit or implicit node v of the suffix tree with path-label U such that $|U| > \alpha$ and $|U| < |\mathsf{Occ}_T(U)|$, and insert U to an initially empty set $\mathcal{U}$; we have at most n such nodes by Theorem 1. Note that we can compute the Lyndon representation of a periodic substring U of T in $\mathcal{O}(1)$ time after an $\mathcal{O}(n)$-time preprocessing of T using a so-called 2-period query [43] that returns the period of U, followed by a minimum rotation query [41] that returns the lexicographically smallest rotation of $U[1\,..\,\mathsf{per}(U)]$.

We group all elements of the multiset $\mathcal{R}\cup\mathcal{U}$ in $\mathcal{O}(n)$ time using WAQs to compute the loci of their Lyndon roots in the suffix tree (we process the Lyndon roots in the order of increasing length after an $\mathcal{O}(n)$-time sorting); then two elements are in the same group if their Lyndon roots have the same locus (and the same length).

We preprocess each $R \in \mathcal{R}$ according to Lemma 30 and then query the obtained data structure with every $U \in \mathcal{U}$ that has the same Lyndon root as R. Over all invocations of Lemma 30, we have: (i) $\widetilde{\mathcal{O}}(n)$ total construction time due to the total length of the runs; (ii) $\mathcal{O}(n)$ space if we process every run separately; and (iii) $\widetilde{\mathcal{O}}(n^2/\alpha)$ total query time since $|\mathcal{U}| \leq n$ and $|\mathcal{R}| = \mathcal{O}(n/\alpha)$. ◀

▶ **Theorem 35.** *There exists an $\mathcal{O}(n)$-space data structure for problems in the framework of Corollary 8 with $\mathcal{O}(m)$ query time, given that:*

- *$\mathsf{Stat}(T,i,j)$ can be queried in $\mathcal{O}(1)$ time, for all i, j, after $\mathcal{O}(n)$-time preprocessing;*
- *$\mathsf{Stat}(T,i,j) = \bigodot_{k=i}^{j}\mathsf{Stat}(T,k,k)$, for all i, j.*

We can construct this data structure in $\widetilde{\mathcal{O}}(n\sqrt{n})$ time using $\mathcal{O}(n)$ space.

Proof. For the first part, we simply follow the proof of Theorem 12, plugging in the more general operators. For the second part, preprocessing (i.e., constructing the suffix tree of T) takes $\mathcal{O}(n\log n)$ time using $\mathcal{O}(n)$ space. By setting $\alpha \coloneqq 5\lceil\sqrt{n}\rceil$, Lemma 23 takes $\mathcal{O}(n\sqrt{n})$ time and $\mathcal{O}(n)$ space, and Lemma 34 takes $\widetilde{\mathcal{O}}(n\sqrt{n})$ time and $\mathcal{O}(n)$ space. ◀

One can now easily verify that by applying Theorem 35 we can construct the data structure underlying Theorem 12 within the claimed bounds.

References

1 Amihood Amir, Eran Chencinski, Costas S. Iliopoulos, Tsvi Kopelowitz, and Hui Zhang. Property matching and weighted matching. *Theor. Comput. Sci.*, 395(2-3):298–310, 2008. doi:10.1016/J.TCS.2008.01.006.

2 Alberto Apostolico and Franco P. Preparata. Data structures and algorithms for the string statistics problem. *Algorithmica*, 15(5):481–494, 1996. doi:10.1007/BF01955046.

3 Ricardo Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval - the concepts and technology behind search, Second edition.* Pearson Education Ltd., Harlow, England, 2011. URL: http://www.mir2ed.org/.

4 Carl Barton, Tomasz Kociumaka, Chang Liu, Solon P. Pissis, and Jakub Radoszewski. Indexing weighted sequences: Neat and efficient. *Inf. Comput.*, 270, 2020. doi:10.1016/J.IC.2019.104462.

5 Djamal Belazzougui, Dmitry Kosolobov, Simon J. Puglisi, and Rajeev Raman. Weighted ancestors in suffix trees revisited. In *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021*, volume 191 of *LIPIcs*, pages 8:1–8:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CPM.2021.8.

6 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000. doi:10.1007/10719839_9.

7 Giulia Bernardini, Huiping Chen, Alessio Conte, Roberto Grossi, Veronica Guerrini, Grigorios Loukides, Nadia Pisanti, and Solon P. Pissis. Utility-oriented string mining. In *Proceedings of the 2024 SIAM International Conference on Data Mining, SDM 2024*, pages 190–198. SIAM, 2024. doi:10.1137/1.9781611978032.22.

8 Giulia Bernardini, Huiping Chen, Alessio Conte, Roberto Grossi, Veronica Guerrini, Grigorios Loukides, Nadia Pisanti, and Solon P. Pissis. Indexing strings with utilities. In *41st IEEE International Conference on Data Engineering, ICDE 2025*, pages 2782–2795. IEEE, 2025. doi:10.1109/ICDE65448.2025.00209.

9 Philip Bille and Inge Li Gørtz. Substring range reporting. *Algorithmica*, 69(2):384–396, 2014. doi:10.1007/s00453-012-9733-4.

10 Philip Bille, Inge Li Gørtz, Moshe Lewenstein, Solon P. Pissis, Eva Rotenberg, and Teresa Anna Steiner. Gapped string indexing in subquadratic space and sublinear query time. In *41st International Symposium on Theoretical Aspects of Computer Science, STACS 2024*, LIPIcs, pages 16:1–16:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.STACS.2024.16.

11 Philip Bille, Inge Li Gørtz, Max Rishøj Pedersen, Eva Rotenberg, and Teresa Anna Steiner. String indexing for top-k close consecutive occurrences. *Theor. Comput. Sci.*, 927:133–147, 2022. doi:10.1016/J.TCS.2022.06.004.

12 Philip Bille, Inge Li Gørtz, Max Rishøj Pedersen, and Teresa Anna Steiner. Gapped indexing for consecutive occurrences. *Algorithmica*, 85(4):879–901, 2023. doi:10.1007/S00453-022-01051-6.

13 Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and Søren Vind. String indexing for patterns with wildcards. *Theory Comput. Syst.*, 55(1):41–60, 2014. doi:10.1007/S00224-013-9498-4.

14 Gerth Stølting Brodal, Rolf Fagerberg, Mark Greve, and Alejandro López-Ortiz. Online sorted range reporting. In *Algorithms and Computation, 20th International Symposium, ISAAC 2009*, volume 5878 of *Lecture Notes in Computer Science*, pages 173–182. Springer, 2009. doi:10.1007/978-3-642-10631-6_19.

15 Gerth Stølting Brodal, Rune B. Lyngsø, Anna Östlin, and Christian N. S. Pedersen. Solving the string statistics problem in time $O(n \log n)$. In *Automata, Languages and Programming, 29th International Colloquium, ICALP 2002*, Lecture Notes in Computer Science, pages 728–739. Springer, 2002. doi:10.1007/3-540-45465-9_62.

16 Gerth Stølting Brodal, Rune B. Lyngsø, Christian N. S. Pedersen, and Jens Stoye. Finding maximal pairs with bounded gap. In *Combinatorial Pattern Matching, 10th Annual Symposium, CPM 99*, Lecture Notes in Computer Science, pages 134–149. Springer, 1999. doi:10.1007/3-540-48452-3_11.

17 Panagiotis Charalampopoulos, Costas S. Iliopoulos, Chang Liu, and Solon P. Pissis. Property suffix array with applications in indexing weighted sequences. *ACM J. Exp. Algorithmics*, 25:1–16, 2020. doi:10.1145/3385898.

18 Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal dictionary matching. *Algorithmica*, 83(7):2142–2169, 2021. doi:10.1007/S00453-021-00821-Y.

19 Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. Faster algorithms for longest common substring. In *29th Annual European Symposium on Algorithms, ESA 2021*, volume 204 of *LIPIcs*, pages 30:1–30:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ESA.2021.30.

20 Panagiotis Charalampopoulos, Jakub Radoszewski, Wojciech Rytter, Tomasz Walen, and Wiktor Zuba. The number of repetitions in 2D-strings. In *28th Annual European Symposium on Algorithms, ESA 2020*, volume 173 of *LIPIcs*, pages 32:1–32:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ESA.2020.32.

21 Hagai Cohen and Ely Porat. Range non-overlapping indexing. In *Algorithms and Computation, 20th International Symposium, ISAAC 2009*, volume 5878 of *Lecture Notes in Computer Science*, pages 1044–1053. Springer, 2009. doi:10.1007/978-3-642-10631-6_105.

22 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.

23 Jonas Ellert and Johannes Fischer. Linear time runs over general ordered alphabets. In *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021*, volume 198 of *LIPIcs*, pages 63:1–63:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ICALP.2021.63.

24 Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS 1997*, pages 137–143. IEEE Computer Society, 1997. doi:10.1109/SFCS.1997.646102.

25 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.

26 Estéban Gabory, Chang Liu, Grigorios Loukides, Solon P. Pissis, and Wiktor Zuba. Space-efficient indexes for uncertain strings. In *40th IEEE International Conference on Data Engineering, ICDE 2024*, pages 4828–4842. IEEE, 2024. doi:10.1109/ICDE60146.2024.00367.

27 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1459–1477. SIAM, 2018. doi:10.1137/1.9781611975031.96.

28 Wensheng Gan, Jerry Chun-Wei Lin, Philippe Fournier-Viger, Han-Chieh Chao, Vincent S. Tseng, and Philip S. Yu. A survey of utility-oriented pattern mining. *IEEE Trans. Knowl. Data Eng.*, 33(4):1306–1327, 2021. doi:10.1109/TKDE.2019.2942594.

29 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. Succinct non-overlapping indexing. *Algorithmica*, 82(1):107–117, 2020. doi:10.1007/S00453-019-00605-5.

30 Pawel Gawrychowski, Garance Gourdel, Tatiana Starikovskaya, and Teresa Anna Steiner. Compressed consecutive pattern matching. *Inf. Syst.*, 136:102607, 2026. doi:10.1016/J.IS.2025.102607.

31 Pawel Gawrychowski, Moshe Lewenstein, and Patrick K. Nicholson. Weighted ancestors in suffix trees. In *Algorithms - ESA 2014 - 22th Annual European Symposium*, volume 8737 of *Lecture Notes in Computer Science*, pages 455–466. Springer, 2014. doi:10.1007/978-3-662-44777-2_38.

32 Daniel Gibney, Paul Macnichol, and Sharma V. Thankachan. Non-overlapping indexing in BWT-runs bounded space. *Theor. Comput. Sci.*, 1056:115512, 2025. doi:10.1016/J.TCS.2025.115512.

33 Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New indices for text: Pat trees and Pat arrays. In *Information Retrieval: Data Structures and Algorithms*, pages 66–82. Prentice-Hall, 1992.

34 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005. doi:10.1137/S0097539702402354.

35 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/CBO9780511574931.

36 Yijie Han. Deterministic sorting in $O(n \log\log n)$ time and linear space. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing,*, pages 602–608. ACM, 2002. doi:10.1145/509907.509993.

37 Costas S. Iliopoulos and M. Sohel Rahman. Faster index for property matching. *Inf. Process. Lett.*, 105(6):218–223, 2008. doi:10.1016/j.ipl.2007.09.004.

38 Costas S. Iliopoulos and M. Sohel Rahman. Indexing factors with gaps. *Algorithmica*, 55(1):60–70, 2009. doi:10.1007/S00453-007-9141-3.

39 M. T. Juan, Jia Jie Liu, and Yue-Li Wang. Errata for "faster index for property matching". *Inf. Process. Lett.*, 109(18):1027–1029, 2009. doi:10.1016/J.IPL.2009.06.009.

40 Orgad Keller, Tsvi Kopelowitz, and Moshe Lewenstein. Range non-overlapping indexing and successive list indexing. In *Algorithms and Data Structures, 10th International Workshop, WADS 2007*, Lecture Notes in Computer Science, pages 625–636. Springer, 2007. doi:10.1007/978-3-540-73951-7_54.

41 Tomasz Kociumaka. Minimal suffix and rotation of a substring in optimal time. In *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016*, LIPIcs, pages 28:1–28:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CPM.2016.28.

42 Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. A linear-time algorithm for seeds computation. *ACM Trans. Algorithms*, 16(2):27:1–27:23, 2020. doi:10.1145/3386369.

43 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal pattern matching queries in a text and applications. *SIAM J. Comput.*, 53(5):1524–1577, 2024. doi:10.1137/23M1567618.

44 Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi:10.1137/0222058.

45 S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2002*, pages 657–666. ACM/SIAM, 2002. URL: http://dl.acm.org/citation.cfm?id=545381.545469.

46 Gonzalo Navarro. Indexing highly repetitive string collections, part I: Repetitiveness measures. *ACM Comput. Surv.*, 54(2):29:1–29:31, 2021. doi:10.1145/3434399.

47 Gonzalo Navarro and Sharma V. Thankachan. Reporting consecutive substring occurrences under bounded gap constraints. *Theor. Comput. Sci.*, 638:108–111, 2016. doi:10.1016/J.TCS.2016.02.005.

48 Jakub Radoszewski. Linear time construction of cover suffix tree and applications. In *31st Annual European Symposium on Algorithms, ESA 2023*, volume 274 of *LIPIcs*, pages 89:1–89:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.ESA.2023.89.

49 Mireille Régnier. A unified approach to word occurrence probabilities. *Discret. Appl. Math.*, 104(1-3):259–280, 2000. doi:10.1016/S0166-218X(00)00195-5.

50 Milan Ruzic. Constructing efficient dictionaries in close to sorting time. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008*, volume 5125 of *Lecture Notes in Computer Science*, pages 84–95. Springer, 2008. doi:10.1007/978-3-540-70575-8_8.

51 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007. doi:10.1007/S00224-006-1198-X.

52 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (SWAT 1973)*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.