Linear Time Construction of Cover Suffix Tree and Applications

Jakub Radoszewski
University of Warsaw, Poland
Samsung R&D, Warsaw, Poland

Abstract

The Cover Suffix Tree (CST) of a string T is the suffix tree of T with additional explicit nodes corresponding to halves of square substrings of T. In the CST an explicit node corresponding to a substring C of T is annotated with two numbers: the number of non-overlapping consecutive occurrences of C and the total number of positions in T that are covered by occurrences of C in T. Kociumaka et al. (*Algorithmica*, 2015) have shown how to compute the CST of a length-n string in $\mathcal{O}(n \log n)$ time. We give an algorithm that computes the same data structure in $\mathcal{O}(n)$ time assuming that T is over an integer alphabet and discuss its implications.

A string C is a cover of text T if occurrences of C in T cover all positions of T; C is a seed of T if occurrences and overhangs (i.e., prefix-suffix occurrences) of C in T cover all positions of T. An α-partial cover (α-partial seed) of text T is a string C whose occurrences in T (occurrences and overhangs in T, respectively) cover at least α positions of T. Kociumaka et al. (*Algorithmica*, 2015; *Theor. Comput. Sci.*, 2018) have shown that knowing the CST of a length-n string T, one can compute a linear-sized representation of all seeds of T as well as all shortest α-partial covers and seeds in T for a given α in $\mathcal{O}(n)$ time. Thus our result implies linear-time algorithms computing these notions of quasiperiodicity. The resulting algorithm computing seeds is substantially different from the previous one (Kociumaka et al., SODA 2012, *ACM Trans. Algorithms*, 2020); in particular, it is non-recursive. Kociumaka et al. (*Algorithmica*, 2015) proposed an $\mathcal{O}(n \log n)$-time algorithm for computing a shortest α-partial cover for each $\alpha = 1, \ldots, n$; we improve this complexity to $\mathcal{O}(n)$.

Our results are based on a new combinatorial characterization of consecutive overlapping occurrences of a substring S of T in terms of the set of runs (see Kolpakov and Kucherov, FOCS 1999) in T. This new insight also leads to an $\mathcal{O}(n)$-sized index for reporting overlapping consecutive occurrences of a given pattern P of length m in the optimal $\mathcal{O}(m + \mathsf{output})$ time, where output is the number of occurrences reported. In comparison, a general index for reporting bounded-gap consecutive occurrences of Navarro and Thankachan (*Theor. Comput. Sci.*, 2016) uses $\mathcal{O}(n \log n)$ space.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases cover (quasiperiod), seed, suffix tree, run (maximal repetition)

Digital Object Identifier 10.4230/LIPIcs.ESA.2023.89

Related Version *Full Version*: `https://arxiv.org/abs/2308.04289`

Funding *Jakub Radoszewski*: Supported by the Polish National Science Center, grant no. 2022/46/E/ST6/00463.

Acknowledgements The author thanks the anonymous reviewers for reading the manuscript carefully and providing several useful suggestions.

1 Introduction

The Cover Suffix Tree (CST, in short) of a string T, denoted as $CST(T)$, is the suffix tree of T ($ST(T)$) augmented with additional nodes and values. For every substring C of T, $CST(T)$ allows to efficiently compute the number of positions in T that are covered by occurrences of C, provided that the node representing C in $CST(T)$ is known. Thus the CST

is a generalization of string covers [3]. The CST of a string was introduced by Kociumaka et al. [29] for computing so-called *partial covers* of a string (see below). Other applications of the CST to the field of quasiperiodicity (see [2]) were discussed in [29, 30].

Let n denote the length of a string T. Kociumaka et al. [29] presented an algorithm computing $CST(T)$ in $\mathcal{O}(n \log n)$ time. Our main result is an algorithm that constructs $CST(T)$ in $\mathcal{O}(n)$ time. We assume that T is over an *integer alphabet* $\{0, \ldots, n^{\mathcal{O}(1)}\}$. This assumption has become a standard in suffix tree construction algorithms since the linear-time suffix tree construction algorithm of Farach [18].

In Section 1.1 we provide more details on the CST. Then in Section 1.2 we discuss applications of our result to computing various notions of quasiperiodicity and in Section 1.3 we present an application of our approach to a variant of text indexing.

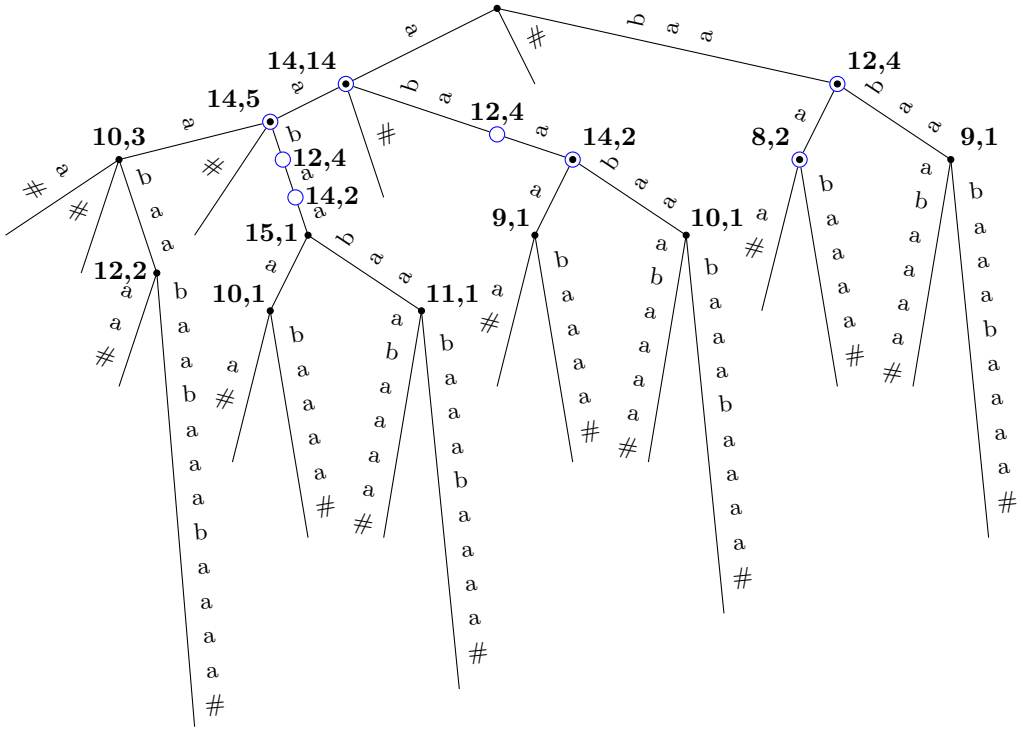

Figure 1 $CST(T)$ for $T = \mathrm{aaabaabaabaaabaaaa\#}$. Black circles denote explicit nodes of $ST(T)$ and blue circles represent nodes corresponding to halves of square substrings in T. The numbers next to nodes denote $cv(v)$ and $nov(v)$.

1.1 Cover Suffix Tree

In the suffix tree $ST(T)$, there is a 1-to-1 correspondence between substrings of T and (explicit and implicit) nodes. The same applies to $CST(T)$. The set of explicit nodes of the $CST(T)$ comprises of the explicit nodes of the suffix tree of T and of nodes corresponding to halves of square substrings of T. Here a *square* is a string of the form $X^2 = XX$, for some string X which is called the *square half*.

The CST has the same tree structure as the Maximal Augmented Suffix Tree (MAST) introduced by Apostolico and Preparata for the String Statistics Problem [4]. It was already observed by Brodal et al. [11] that the MAST of T uses only $\mathcal{O}(n)$ space; this is because

the suffix tree has $\mathcal{O}(n)$ nodes [32] and the number of different square substrings of T is $\mathcal{O}(n)$ [20]. By a recent result of Brlek and Li [9, 10] showing that a string of length n contains at most n different square substrings, it follows that the CST (and MAST) contains at most $3n$ explicit nodes.

For a node v of $CST(T)$, by $\bar{v}$ we denote the substring of T that corresponds to v. A *consecutive occurrence* of string S in T is a pair of indices (i, j) in T such that $j > i$, S has occurrences starting at positions i and j in T and S does not occur at any of the positions $i + 1, \ldots, j - 1$. A consecutive occurrence (i, j) of S is called *overlapping* if $j < i + |S|$ and *non-overlapping* otherwise. In $CST(T)$, each node v is annotated by two values (see Figure 1):

- $cv(v)$, equal to the total number of positions in T covered by occurrences of $\bar{v}$, and
- $nov(v)$, equal to one plus the number of non-overlapping consecutive occurrences of $\bar{v}$ in T. (Intuitively, the one corresponds to the rightmost occurrence of $\bar{v}$ in T.)

The key property of values $cv(v)$ and $nov(v)$ is that if u is an implicit node of $CST(T)$ that is located on an edge from an explicit node v to its parent, then $cv(u)$ can be expressed in terms of $cv(v)$ and $nov(v)$ as follows: $cv(u) = cv(v) - (|\bar{v}| - |\bar{u}|)nov(v)$; see Figure 2.

Figure 2 Left: the locus v of $C = \text{abaabaa}$ in $CST(T)$ is an explicit node and we have $cv(v) = 10$, $nov(v) = 1$ (see Figure 1). Right: the locus v' of the prefix $C' = \text{abaaba}$ of C is an implicit node one character above v. We then have $cv(v') = cv(v) - nov(v) = 9$.

We obtain the following result.

▶ **Theorem 1.** *The Cover Suffix Tree (CST) of a string of length n over an integer alphabet can be constructed in $\mathcal{O}(n)$ time.*

The $\mathcal{O}(n \log n)$-time algorithm from [29] for constructing $CST(T)$ processes the suffix tree of T bottom-up, storing for each explicit node v the set of occurrences of the corresponding substring of T in an AVL tree. Its time complexity follows by using an efficient algorithm for merging AVL trees [13] (cf. [12]). We use a completely different approach, based on a new combinatorial observation that links overlapping consecutive occurrences of substrings of T to runs in T [31]. We show that the (multi)set of substrings whose overlapping consecutive occurrences are implied by a run has a simple, "triangular" structure. The values $ov(v)$ and $cv(v)$ for explicit nodes v of $CST(v)$ are then computed in two bottom-up traversals, one in the tree of suffix links of $ST(T)$ and the other in the $CST(T)$.

1.2 Applications of CST to quasiperiodicity

Theorem 1 has several applications in the field of quasiperiodicity [2]. Basic notions of quasiperiodicity are covers [3], that were already mentioned before, and seeds [25]. A string C is a cover of a string T if each position of T is inside at least one occurrence of C. A string S is a seed of a string T if it is a cover of a superstring of T. In other words, all positions of T are covered by occurrences and overhangs of S, where an *overhang* is a prefix of T being a suffix of S or a suffix of T being a prefix of S. A substring C of T is called an *α-partial cover* of T if the occurrences of C in T cover at least α positions in T. A substring S of T is called an *α-partial seed* of T if the occurrences and overhangs of S in T cover at least α positions in T. See Figure 3 for examples.

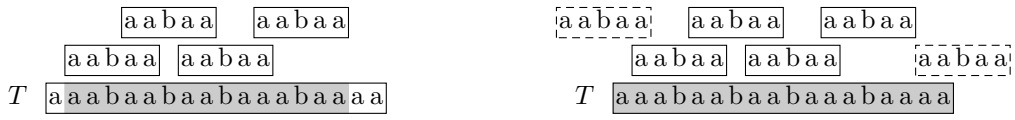

Figure 3 C = aabaa is a 15-partial cover of T (left). Indeed, the node v of $CST(T)$ corresponding to C has $cv(v) = 15$ (Figure 1). C is also an 18-partial seed of T, hence, a seed of T (right).

In [29, 30] it was noticed that given the $CST(T)$, all shortest α-partial covers and α-partial seeds of T for a specified value of the parameter α can be computed in $\mathcal{O}(n)$ time. Thus our Theorem 1 implies the following result.

▶ **Corollary 2.** *Let T be a string of length n over an integer alphabet and $\alpha \in [1\,..\,n]$. All shortest α-partial covers and seeds of T can be computed in $\mathcal{O}(n)$ time.*

In [29] also the following problem was considered.

AllPartialCovers
Input: a string T of length n
Output: for all $\alpha = 1, \ldots, n$, a shortest α-partial cover of T

▶ **Example 3.** For T from Figure 1, the solution to AllPartialCovers problem can be as follows: substring C = a for $\alpha \leq 14$, substring C = aabaa for $\alpha = 15$ (see Figure 3), and any length-α substring of T for $\alpha \geq 16$.

An $\mathcal{O}(n \log n)$-time solution for AllPartialCovers based on $CST(T)$ and on computing the upper envelope [24] of $\mathcal{O}(n)$ line segments was presented in [29]. We obtain the following result.

▶ **Theorem 4.** *AllPartialCovers problem can be solved in $\mathcal{O}(n)$ time for a length-n string over an integer alphabet.*

A linear-time algorithm computing all covers of a string was presented almost 30 years ago by Moore and Smyth [33, 34]. A rather involved linear-time algorithm computing a representation of all seeds in a string over an integer alphabet was given much more recently by Kociumaka et al. [27]. The representation (already introduced in the earlier, $\mathcal{O}(n \log n)$-time algorithm by Iliopoulos et al. [25]) consists of a set of paths in the suffix trees of T and of T reversed, at most one path on each edge of the suffix trees. Seeds of T are exactly $|T|$-partial seeds of T, so Corollary 2 immediately implies an alternative linear-time algorithm computing all shortest seeds of T. Moreover, in [29, Theorem 3] it was observed that having $CST(T)$, the aforementioned representation of all seeds in T can be computed in $\mathcal{O}(n)$ time. Thus Theorem 1 yields an alternative $\mathcal{O}(n)$-time algorithm computing the same representation of all seeds in T as in [27]. The resulting algorithm is substantially different from the algorithm of [27]; in particular, it is non-recursive and arguably simpler.

Recently, Kociumaka et al. [28] showed that there exists a different representation of all seeds of a string, consisting of $\mathcal{O}(n)$ disjoint paths on just the suffix tree of T, and that this representation can be computed in $\mathcal{O}(n)$ time assuming an integer alphabet. This representation, however, no longer satisfies the convenient property that at most one path on each edge of the suffix tree is in the representation (see [28, Fig. 2] for an example).

1.3 Reporting overlapping occurrences

Data stored in the CST can be used to compute, for any substring S of T, the number $ov(S)$ of overlapping consecutive occurrences of S in T. We show that the technique behind Theorem 1 can be further exploited to obtain a linear-space index for reporting overlapping consecutive occurrences of a query pattern.

Navarro and Thankachan [35] proposed an index that, given a length-m substring S of T and an interval $[\alpha, \beta]$, reports all consecutive occurrences (i, j) of S such that $\alpha \leq j - i \leq \beta$ in $\mathcal{O}(m + \mathsf{output})$ time, where output is the number of consecutive occurrences reported. The size of their index for a text T of length n is $\mathcal{O}(n \log n)$. We solve the following problem.

Reporting bounded-gap overlapping consecutive occurrences
Input: A string T of length n
Query input: A substring S of T, $\lvert S\rvert = m$, and a positive integer β such that $\beta < m$
Query output: All consecutive occurrences (i, j) of S in T such that $j - i \leq \beta$

▶ **Theorem 5.** *There is an index of size $\mathcal{O}(n)$ that reports all bounded-gap overlapping consecutive occurrences of a length-m pattern in $\mathcal{O}(m + \mathsf{output})$ time, where output is the number of consecutive occurrences reported. If T is over a constant-sized alphabet, the index can be constructed in $\mathcal{O}(n)$ time. The construction time becomes expected if T is over an integer alphabet.*

The data structure of Theorem 5 is superior to the data structure of [35] if $\alpha = 0$ and $\beta < m$.

In the data structure we use the same combinatorial observation as in Theorem 1. With it, a query for a pattern S consists in finding the corresponding node v in $CST(T)$ and reporting all "triangular" structures implied by runs that contain v. To this end, range minimum query data structures [8] are used to store the "bottom sides" of the triangles. Thanks to this, it is actually sufficient to store $ST(T)$ instead of the $CST(T)$. The expected time in the construction algorithm stems from using perfect hashing [21] to store children of a node of the suffix tree if T is over a superconstant alphabet.

1.4 Structure of the paper

We start by recalling basic definitions related to strings and compact tries (including suffix trees). In Section 3 we present the proof of the main Theorem 1. Solution to AllPartialCovers (Theorem 4) is provided in Section 4. The data structure for reporting bounded-gap overlapping consecutive occurrences (proof of Theorem 5) is presented in the full version. We conclude in Section 5.

2 Preliminaries

2.1 Strings

By Σ we denote the finite alphabet of all the considered strings. We assume that characters of a string S are numbered 1 through $|S|$, with $S[i] \in \Sigma$ denoting the ith character. An integer $j \in [1\,..\,|S|]$ is called an index in S. A string $S[i]S[i+1]\cdots S[j]$ for any indices i, j such that $i \leq j$ is called a *substring* of S. By $S[i\,..\,j]$ we denote a *fragment* of S that can be viewed as a positioned substring $S[i]S[i+1]\cdots S[j]$ (formally, it is represented in $\mathcal{O}(1)$ space with a reference to S and the interval $[i\,..\,j]$). We also denote $S[i\,..\,j-1]$ as $S[i\,..\,j)$. Two fragments

$S[i\,..\,j]$ and $S[i'\,..\,j']$ *match* (notation: $S[i\,..\,j] = S[i'\,..\,j']$) if the underlying substrings are the same. Similarly we define matching of a fragment and a substring. Two fragments $S[i\,..\,j]$ and $S[i'\,..\,j']$ are *equivalent* (notation: $S[i\,..\,j] \equiv S[i'\,..\,j']$) if $i = i'$ and $j = j'$. A string U is called a prefix (suffix) of a string S if $U = S[1\,..\,|U|]$ ($U = S[|S| - |U| + 1\,..\,|S|]$, respectively) and a *border* of S if it is both a prefix and a suffix of S.

Henceforth by T we denote the text string and by n we denote $|T|$. We say that a string S occurs in the text T at position i if $S = T[i\,..\,i + |S|)$. A pair of indices (i, j) in T is called a *consecutive occurrence of substring* S if $i < j$, $T[i\,..\,i + |S|) = T[j\,..\,j + |S|)$ and $T[k\,..\,k + |S|) \neq S$ for all $k \in (i\,..\,j)$. A consecutive occurrence is called *overlapping* if $j < i + |S|$ and otherwise it is called *non-overlapping*. By $OvOcc(S)$ we denote the set of overlapping consecutive occurrences of S in T.

For a string U and $d \in \mathbb{Z}_{\geq 0}$, by U^d we denote the dth power of U, equal to a concatenation of d copies of U. A string V is *primitive* if $V = U^d$ for $d \in \mathbb{Z}_+$ implies that $d = 1$. The following property of primitive strings is a known consequence of Fine and Wilf's lemma [19].

▶ **Lemma 6** (Synchronization property, see [14]). *A string V is primitive if and only if V has exactly two occurrences in V^2.*

A string of the form U^2 is called a *square*.

▶ **Theorem 7** ([20] and [15, 6]). *The number of distinct square substrings in a length-n string is $\mathcal{O}(n)$ and they can all be computed in $\mathcal{O}(n)$ time assuming an integer alphabet.*

We say that a string S has a period p if $S[i] = S[i + p]$ holds for all $i \in [1\,..\,|S| - p]$; equivalently, if S has a border of length $|S| - p$. By $per(S)$ we denote the smallest period of S.

A *run* in a string T is a triad (a, b, p) such that (1) p is the smallest period of $T[a\,..\,b]$, (2) $2p \leq b - a + 1$, (3) $a = 1$ or $T[a-1] \neq T[a-1+p]$, and (4) $b = |T|$ or $T[b+1] \neq T[b+1-p]$. The *exponent* of a run $R = (a, b, p)$ is defined as $exp(R) = (b - a + 1)/p$. By $\mathcal{R}(T)$ we denote the set of all runs in T.

▶ **Theorem 8** ([5]). *A string T of length n has at most n runs and they can be computed in $\mathcal{O}(n)$ time if T is over an integer alphabet.*

An earlier bound $|\mathcal{R}(T)| = \mathcal{O}(n)$ together with an $\mathcal{O}(n)$-time algorithm for computing $\mathcal{R}(T)$ was proposed in [31]. All runs can be computed in $\mathcal{O}(n)$ time also for a string over a general ordered alphabet [17].

2.2 Compact tries

The *suffix trie* of a string T contains a node for every distinct substring of $T\#$, where $\# \notin \Sigma$ is a special end marker. The root node is the empty string. For each pair of substrings (S, Sc) of T, where $c \in \Sigma$, there is an edge from S to Sc labeled with the character c. Each suffix of $T\#$ corresponds to a leaf of the suffix trie.

A *compact suffix trie* of T contains the root, the branching nodes, the leaf nodes, and possibly some other nodes of the suffix trie as *explicit nodes*. Maximal paths that do not contain explicit nodes are replaced by single compact edges, and a fragment of T is used to represent the label of every such edge in $\mathcal{O}(1)$ space. The nodes that are dissolved due to compactification are called *implicit nodes*; an implicit node u can be referred to as a pair (v, d) where v is the nearest explicit descendant of u and d is the distance (number of characters) between u and v. The most common example of a compact suffix trie of T is the *suffix tree* of T, denoted here as $ST(T)$, in which each maximal branchless path from the suffix trie is replaced by a single compact edge.

▶ **Theorem 9** ([18, 26]). *The suffix tree of a string of length n over an integer alphabet can be constructed in $\mathcal{O}(n)$ time.*

For a node v of a compact suffix trie $\mathcal{T}$ of T, the corresponding substring $\bar{v}$ of T is called the *string label* of v. Conversely, for a substring (or fragment) S of T, its locus in $\mathcal{T}$ is the (explicit or implicit) node v of $\mathcal{T}$ such that $\bar{v} = S$.

The locus in $ST(T)$ of a substring S is denoted as $locus(S)$. For a non-root explicit node v of $ST(T)$, its *suffix link* leads from v to the node $suf(v) = locus(X)$, where $\bar{v} = cX$, $c \in \Sigma$; it is known that $suf(v)$ is then an explicit node. By $ST'(T)$ we denote tree of suffix links in $ST(T)$. The nodes of $ST'(T)$ are the explicit nodes of $ST(T)$ and for each non-root explicit node v of $ST(T)$, in $ST'(T)$ there is an edge connecting node v with node $suf(v)$.

Let $Sq(T) = \{S : S^2 \text{ is a substring of } T\}$. Then the *tree structure* of $CST(T)$ is a compact suffix trie of T that could be obtained from the suffix tree $ST(T)$ by making loci of substrings $Sq(T)$ explicit.

A *weighted ancestor query* on a compact suffix trie $\mathcal{T}$ is given a leaf ℓ of $\mathcal{T}$ and a non-negative integer d and asks for the topmost (explicit) ancestor w of ℓ such that $|\bar{w}| \geq d$. We denote such a query and its result as $w = WA(\ell, d)$. We use the following offline solution to the problem of answering WA queries.

▶ **Theorem 10** ([28]). *Any q weighted ancestor queries on a compact suffix trie with $\mathcal{O}(n)$ nodes of a length-n string can be answered in $\mathcal{O}(n+q)$ time.*

Data structures for answering weighted ancestor queries with different complexities are known [1, 22], also in the special case of the compact suffix trie being the suffix tree [7, 23].

Let v be a node of a compact suffix trie and $S = \bar{v}$. Then $cv(v)$ is formally defined as

$$cv(v) = \bigcup \{ [i \,..\, i + |S|) : T[i \,..\, i + |S|) = S \}.$$

Moreover, $nov(v)$ equals one plus the number of non-overlapping consecutive occurrences of S in T. $CST(T)$ stores the values $cv(v)$ and $nov(v)$ for each explicit node v. By $occ(v)$ we further denote the total number of occurrences of $\bar{v}$ in T. The values $occ(v)$ for all explicit nodes of a compact suffix trie can be computed bottom-up in linear time, as $occ(v)$ is the number of leaves in the subtree of node v. By $ov(v)$ we denote the number of overlapping consecutive occurrences of $\bar{v}$ in T. We have $ov(v) + nov(v) = occ(v)$. We use the notations $cv()$, $nov()$, $ov()$ and $occ()$ also for substrings of T.

3 Construction of the CST

3.1 Computing the tree structure

▶ **Lemma 11.** *The tree structure of the CST of a string T of length n over an integer alphabet can be computed in $\mathcal{O}(n)$ time.*

Proof. The suffix tree of a string over an integer alphabet can be constructed in $\mathcal{O}(n)$ time (Theorem 9). By Theorem 7, the set $Sq(T)$ of square substring halves, each represented as a fragment of T, can be computed in $\mathcal{O}(n)$ time.

The final step is to make all implicit nodes of the suffix tree that correspond to elements of $Sq(T)$ explicit. Let $T[i \,..\, i + 2d)$ be a square substring and ℓ be the leaf of the suffix tree of T corresponding to the suffix $T[i \,..\, n]$. Using a weighted ancestor query we can compute a pair (v, p) where $v = WA(\ell, d)$ is the nearest explicit descendant of the locus u of $T[i \,..\, i + d)$ and p is the distance between u and v. With Theorem 10 a batch of $\mathcal{O}(n)$ such queries

can be answered in $\mathcal{O}(n)$ time. Finally, we use Radix Sort to sort the pairs (v, p) (under an arbitrary, fixed order on nodes of the suffix tree) in $\mathcal{O}(n)$ time. As a result, the loci of substrings in $Sq(T)$ are grouped by their nearest explicit descendants, and each group is sorted by decreasing depths. This allows to make all the desired implicit nodes explicit in $\mathcal{O}(n)$ time. ◀

3.2 Properties of overlapping consecutive occurrences

If a substring S of T does not have overlapping occurrences in T, i.e., $ov(S) = 0$, then $cv(S) = nov(S) \cdot |S| = occ(S) \cdot |S|$ is easy to compute. Hence, below we characterize overlapping consecutive occurrences of substrings. To this end, we use runs.

For indices $1 \le i \le j_1 \le j_2 \le n$, we denote the set of fragments corresponding to a path in the suffix tree of T:

$$\mathsf{Path}(i, j_1, j_2) = \{T[i\,..\,j] \,:\, j \in [j_1\,..\,j_2]\}.$$

For a run $R = (a, b, p)$ in T, we denote

$$\mathsf{Triangle}(R) = \mathsf{Triangle}(a, b, p) = \bigcup_{i=a}^{b-2p} \mathsf{Path}(i, i+p, b-p).$$

Figure 4 gives a graphical motivation for the name of this set of fragments, whereas Figure 5 shows that in some cases the triangle is "wrapped".

The following key combinatorial lemma shows that sets $\mathsf{Triangle}(R)$ for $R \in \mathcal{R}(T)$ are sufficient for counting overlapping consecutive occurrences.

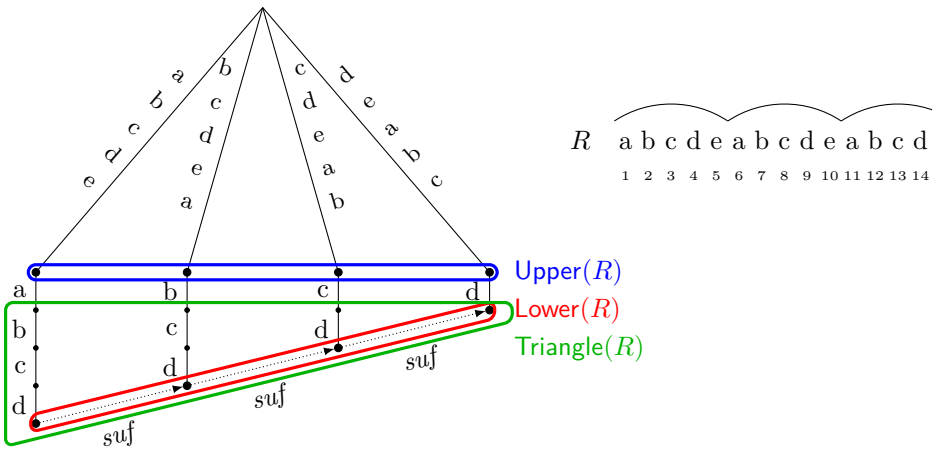

Figure 4 Illustration of the sets $\mathsf{Triangle}(R)$, $\mathsf{Upper}(R)$ and $\mathsf{Lower}(R)$ on paths in $CST(T)$ for an example run $R = (1, 14, 5)$. All nodes representing fragments from $\mathsf{Triangle}(R)$ are distinct because $exp(R) = 2.8 \le 3$. The nodes in $\mathsf{Upper}(R)$ and $\mathsf{Lower}(R)$ are explicit in $CST(T)$ (see Observation 13).

▶ **Lemma 12.** *Let S be a string of length d. Then S has an overlapping consecutive occurrence (i, j) in T for some indices i, j if and only if S matches a fragment $T[i\,..\,i+d) \in \mathsf{Triangle}(R)$ for some run R with period $j - i$ in T.*

Proof. ($\Rightarrow$) If S has an overlapping consecutive occurrence (i, j), then the substring $F = T[i\,..\,j+d)$ has a border S, so F has a period $p = j - i < d$.

We further have $|F| = j + d - i > 2(j - i) = 2p$. Period p is the smallest period of F; indeed, a period $q \in [1\,..\,p)$ would imply an occurrence $T[i + q\,..\,i + q + d)$ of S at position $i + q$ such that $i < i + q < i + p = j - i$, so (i, j) would not be a consecutive occurrence of S.

Finally, fragment F extends to a unique maximal periodic fragment with smallest period p; it is a run $R = (a, b, p)$ in T. We have $T[i\,..\,i + d) \in \mathsf{Path}(i, i + p, b - p) \subseteq \mathsf{Triangle}(R)$ as $i \in [a\,..\,b - 2p]$.

($\Leftarrow$) Assume that $T[i\,..\,i + d) \in \mathsf{Triangle}(R)$ holds for some run $R = (a, b, p)$ and $S = T[i\,..\,i + d)$. Then $[i\,..\,i + d + p) \subseteq [a\,..\,b]$, so the period of the run implies that $T[i + p\,..\,i + p + d) = T[i\,..\,i + d) = S$. Moreover, $d > p$ by the definition of $\mathsf{Triangle}(R)$, so the two occurrences of S overlap.

Finally, we need to show that (i, j), for $j = i + p$, is a consecutive occurrence of S. If there was an occurrence $T[k\,..\,k + d) = S$ with $i < k < j$, then the string $X = T[i\,..\,i + p)$ would have an occurrence in $T[i\,..\,i + 2p) = X^2$ being neither a prefix nor a suffix of X^2. String X is primitive, as otherwise the run R would have a period smaller than p. Therefore this situation is impossible by the synchronization property (Lemma 6). ◀

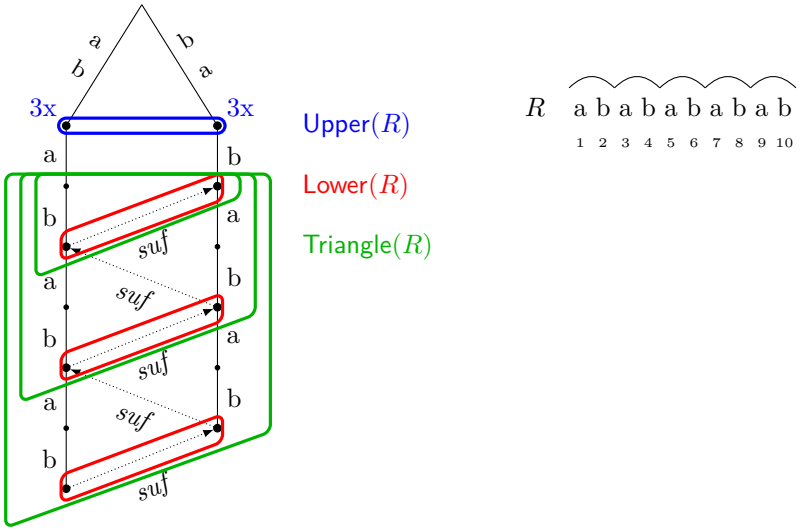

Figure 5 Illustration of the sets $\mathsf{Triangle}(R)$, $\mathsf{Upper}(R)$ and $\mathsf{Lower}(R)$ on paths in $CST(T)$ for a run $R = (1, 10, 2)$ with exponent 5. The substrings in the set $\mathsf{Triangle}(R)$ form a multiset being the sum of the two trapezia and a triangle. The set $\mathsf{Upper}(R)$ contains six fragments; three of them match substring ab, and the remaining three match ba.

For a run $R = (a, b, p)$, we further denote:

$$\mathsf{Upper}(a, b, p) = \{T[i\,..\,i + p) \,:\, i \in [a\,..\,b - 2p]\}$$
$$\mathsf{Lower}(a, b, p) = \{T[i\,..\,b - p] \,:\, i \in [a\,..\,b - 2p]\}$$

Intuitively, $\mathsf{Lower}(R)$ consists of bottommost endpoints of paths Path from $\mathsf{Triangle}(R)$ and $\mathsf{Upper}(R)$ consists of parents of topmost endpoints of these paths. Informally, they are the "lower side" and the "excluded upper side" of the triangle; see also Figures 4 and 5. Below we show basic properties of these sets.

▶ **Observation 13.** *Let R be a run in T.*

(a) *All fragments in $\mathsf{Upper}(R)$ are square halves in T.*

(b) *The loci of fragments in $\mathsf{Lower}(R)$ are explicit nodes in $ST(T)$.*

Proof. (a) By the periodicity of run R, each fragment $T[i\,..\,i+p) \in \mathsf{Upper}(R)$ is followed by a matching fragment $T[i+p\,..\,i+2p)$. This is because $i \le b-2p$. Hence, $T[i\,..\,i+2p)$ is indeed a square in T.

(b) Let $T[i\,..\,b-p] \in \mathsf{Lower}(R)$ and $c = T[b+1-p]$. The period of the run implies that $T[i\,..\,b-p] = T[i+p\,..\,b]$.

If $T[i\,..\,b-p]$ is a suffix of T, its locus in $ST(T)$ is explicit as the locus has children along the characters c and $\#$.

Otherwise, character $c' = T[b+1]$ is different from c by the right maximality of the run R. Hence, $T[i\,..\,b-p]c$ and $T[i\,..\,b-p]c'$ are different substrings of T, as claimed. ◀

Let us note that if $exp(R) > 3$, then each of the sets $\mathsf{Upper}(R)$, $\mathsf{Triangle}(R)$ may contain matching fragments; see Figure 5.

3.3 Counting overlapping consecutive occurrences

For each explicit node v of $CST(T)$, instead of $nov(v)$, we will compute the number $ov(v)$ of overlapping consecutive occurrences of the substring $\bar{v}$ in T.

For a set $\mathcal{F}$ of fragments of T, we denote by $\#_v(\mathcal{F}) = |\{T[i\,..\,j] \in \mathcal{F} : \bar{v} = T[i\,..\,j]\}|$ the number of fragments in $\mathcal{F}$ that match $\bar{v}$. Lemma 12 implies the following formula for $ov(v)$.

▶ **Observation 14.** *For a node v of $CST(T)$, $ov(v) = \sum_{R\in\mathcal{R}(T)} \#_v(\mathsf{Triangle}(R))$.*

We will show how to efficiently evaluate these formulas for all explicit nodes v simultaneously.

For each explicit node v of $CST(T)$ we will compute two counters:

$$C_{upper}[v] = \sum_{R\in\mathcal{R}(T)} \#_v(\mathsf{Upper}(R)), \quad C_{lower}[v] = \sum_{R\in\mathcal{R}(T)} \#_v(\mathsf{Lower}(R)).$$

That is, $C_{upper}[v]$ ($C_{lower}[v]$) stores the number of times fragments matching the substring $\bar{v}$ occur in $\mathsf{Upper}(R)$ ($\mathsf{Lower}(R)$, respectively) over all runs $R \in \mathcal{R}(T)$.

For an explicit node v of $CST(T)$, by $subtree(v)$ we denote the set of explicit descendants of v in the tree (including v). The following lemma shows how to compute ov from the counters C_{upper} and C_{lower}. The lemma follows by Observation 14.

▶ **Lemma 15.** *For an explicit node v of the $CST(T)$, we have*

$$ov(v) = \sum_{w\in subtree(v)} (C_{lower}[w] - C_{upper}[w]).$$

Proof. If node x is an ancestor of node y, by $x \rightsquigarrow y$ we denote the set of explicit nodes on the path from x to y. By *root* we denote the root of $CST(T)$. By the definitions of $\mathsf{Triangle}(R)$, $\mathsf{Upper}(R)$ and $\mathsf{Lower}(R)$ and Observation 14, we have:

$$\begin{aligned} ov(v) &= \sum_{R\in\mathcal{R}(T)} \#_v(\mathsf{Triangle}(R)) \\ &= \sum_{(a,p,b)\in\mathcal{R}(T)} \sum_{i=a}^{b-2p} \#_v(\mathsf{Path}(i, i+p, b-p)) \\ &= \sum_{(a,p,b)\in\mathcal{R}(T)} |\{i \in [a\,..\,b-2p] : v \in (locus(T[i\,..\,i+p]) \rightsquigarrow locus(T[i\,..\,b-p]))\}| \end{aligned}$$

$$
\begin{aligned}
&= \sum_{(a,p,b)\in\mathcal{R}(T)} |\{i \in [a\,..\,b-2p] \,:\, v \in (root \rightsquigarrow locus(T[i\,..\,b-p]))\}| \\
&\quad - \sum_{(a,p,b)\in\mathcal{R}(T)} |\{i \in [a\,..\,b-2p] \,:\, v \in (root \rightsquigarrow locus(T[i\,..\,i+p))\}| \\
&= \sum_{R\in\mathcal{R}(T)} \sum_{w\in subtree(v)} \#_w(\mathsf{Lower}(R)) - \sum_{R\in\mathcal{R}(T)} \sum_{w\in subtree(v)} \#_w(\mathsf{Upper}(R)) \\
&= \sum_{w\in subtree(v)} (C_{lower}[w] - C_{upper}[w]).
\end{aligned}
$$

◀

3.3.1 Computing C_{lower} and C_{upper}

Let us recall that $ST'(T)$ is the tree of suffix links of $ST(T)$.

▶ **Observation 16.** *For each run R in T, $\mathsf{Lower}(R)$ forms a path in $ST'(T)$.*

▶ **Lemma 17.** *The counters $C_{lower}[v]$ for all explicit nodes v of $CST(T)$ can be computed in $\mathcal{O}(n)$ time.*

Proof. By the observation, $C_{lower}[v]$ is simply the number of paths $\mathsf{Lower}(R)$ that cover node v in $ST'(T)$ (in particular, no two fragments in a single set $\mathsf{Lower}(R)$ match).

To count paths covering each node in a rooted tree we apply a standard approach using ±1 counters. Initially all counters $C_{lower}[v]$ are equal to 0. For each run $R = (a, b, p) \in \mathcal{R}(T)$, we increment $C_{lower}[v]$ for the bottom endpoint v of the path $\mathsf{Lower}(R)$ ($v = locus(T[a\,..\,b-p])$) and decrement $C_{lower}[u]$ for the parent u in $ST'(T)$ of the top endpoint of $\mathsf{Lower}(R)$ ($u = locus(T[b-2p+1\,..\,b-p])$). In the end for each node u of $ST'(T)$ in a bottom-up order, we add $C_{lower}[v]$ to $C_{lower}[u]$ for all children v of u in $ST'(T)$.

Let us summarize and analyze the complexity of the algorithm. Tree $ST'(T)$ has $\mathcal{O}(n)$ nodes. By Theorem 8, there are at most n paths $\mathsf{Lower}(R)$ and all runs R can be computed in $\mathcal{O}(n)$ time. The endpoints of all paths $\mathsf{Lower}(R)$ can be located in $ST'(T)$ in $\mathcal{O}(n)$ time using weighted ancestor queries in $ST(T)$ (Theorem 10). Finally, the bottom-up traversal of the tree $ST'(T)$ takes $\mathcal{O}(n)$ time. ◀

We proceed to computing counters C_{upper}. Let us define an operation *rot* such that $rot(cX) = Xc$ for a string X and character $c \in \Sigma$. For $k \in \mathbb{Z}_{\geq 0}$, by $rot^k(S)$ we denote the composition of *rot* k times. If $S' = rot^k(S)$ for some strings S, S' and $k \in \mathbb{Z}_{\geq 0}$, we say that S' is a *cyclic rotation* of S. We also say that S and S' are *cyclically equivalent*.

For each run R in T, the strings in $\mathsf{Upper}(R)$ are cyclic rotations of each other. This motivates introduction of the following directed graph $G = (V, E)$. The set of vertices is $V = Sq(T)$ and the arcs are defined as follows: $(S, S') \in E$ if and only if $S, S' \in V$ and $S' = rot(S)$. Instead of addressing vertices of G by substrings of T, we will address them by their loci in $CST(T)$ which are explicit nodes of $CST(V)$.

▶ **Observation 18.** *For each run R in T, $\mathsf{Upper}(R)$ corresponds to a (directed) walk in G.*

Let us note that the vertices (and arcs) on the walk $\mathsf{Upper}(R)$ may repeat if $exp(R) > 3$ (see Figure 5 again). In particular, in this case $\mathsf{Upper}(R)$ is contained in a cycle in G.

We proceed to the construction of graph G. More precisely, a sufficient subset of arcs of G is constructed.

▶ **Lemma 19.** *A subset E' of E containing all arcs that belong to any walk $\mathsf{Upper}(R)$, for $R \in \mathcal{R}(T)$, can be constructed in $\mathcal{O}(n)$ time.*

Proof. For each distinct square substring $T[i\,..\,i+2d)$ of T, we insert into E' an arc from $\mathit{locus}(T[i\,..\,i+d))$ to $\mathit{locus}(T[i+1\,..\,i+d])$ if $T[i+1\,..\,i+d] \in Sq(T)$; the latter condition can be checked from the construction of the tree structure of $CST(T)$ (Lemma 11). By Theorem 7, square substrings of T can be enumerated in $\mathcal{O}(n)$ time. Then we use off-line weighted ancestor queries (Theorem 10) on $CST(T)$ to find the desired loci. This concludes that the time complexity is $\mathcal{O}(n)$. Now let us argue for the correctness of this algorithm in two steps.

Why $E' \subseteq E$: When adding an arc from $\mathit{locus}(T[i\,..\,i+d))$ to $\mathit{locus}(T[i+1\,..\,i+d])$, we know that $T[i\,..\,i+d) \in Sq(T)$ and we check if $T[i+1\,..\,i+d] \in Sq(T)$. Hence, an arc connects two vertices of $V = Sq(T)$. Finally, we have $\mathit{rot}(T[i\,..\,i+d)) = T[i+1\,..\,i+d]$ because $T[i\,..\,i+2d)$ is a square. Consequently, $E' \subseteq E$.

Why all arcs that belong to any walk $\mathsf{Upper}(R)$ are in E': Let $T[i\,..\,i+p), T[i+1\,..\,i+p] \in \mathsf{Upper}(a,b,p)$ be two consecutive elements. Then $a \le i < b-2p$, so $T[i\,..\,i+2p)$ is a square. Therefore, $(\mathit{locus}(T[i\,..\,i+p)), \mathit{locus}(T[i+1\,..\,i+p])) \in E'$ by definition. ◀

In the lemma above, it can be the case that $E' \subsetneq E$ if there are two substrings $S, S' \in Sq(T)$ such that $S' = \mathit{rot}(S)$ but there are no two fragments $T[i\,..\,i+|S|)$, $T[i+1\,..\,i+1+|S|)$ matching S and S', respectively. Moreover, it can happen that E' contains an arc that does not belong to any walk $\mathsf{Upper}(R)$. Indeed, when an arc from $\mathit{locus}(T[i\,..\,i+d))$ to $\mathit{locus}(T[i+1\,..\,i+d])$ is added to E', we avoid the unnecessary check if $T[i\,..\,i+d), T[i+1\,..\,i+d]$ belong to a set $\mathsf{Upper}(R)$ for any run R.

▶ **Lemma 20.** *The counters $C_{upper}[v]$ for all explicit nodes v of $CST(T)$ can be computed in $\mathcal{O}(n)$ time.*

Proof. By Observation 18, $C_{upper}[v]$ is the total number of times that walks $\mathsf{Upper}(R)$ visit the node $v \in V$. We will be able to compute these counters efficiently using the fact that graph G has a particularly simple structure: it is a collection of disjoint cycles and paths. The same applies to the graph $G' = (V, E')$ that is computed in Lemma 19. For a node u, by $\mathit{next}(u)$ we denote the unique node v such that $(u, v) \in E'$, and $\perp$ if no such node exists.

We can find all cycles in G' using the DFS. Then we apply an algorithm using ± 1 counters (as in the proof of Lemma 17) and additional counters C' assigned to cycles. Initially all counters are equal to 0. For each cycle Q, let us order the nodes $v_1, \ldots, v_{|Q|} \in Q$ along the cycle (arbitrarily) and assign them consecutive id numbers $\mathit{id}(v_i) = i$.

For each walk $\mathsf{Upper}(R)$, $R = (a,b,p) \in \mathcal{R}(T)$, we check if its endpoints v_1 and v_2 ($v_1 = \mathit{locus}(T[a\,..\,a+p))$ and $v_2 = \mathit{locus}(T[b-2p\,..\,b-p))$) belong to a cycle. If not, we increment $C_{upper}[v_1]$; we also decrement $C_{upper}[\mathit{next}(v_2)]$ if $\mathit{next}(v_2) \neq \perp$. Otherwise, if v_1, v_2 belong to a cycle Q, we increase the cycle counter $C'[Q]$ by $\lfloor\, |\mathsf{Upper}(R)| \,/\, |Q| \,\rfloor$. Moreover (for $v_1, v_2 \in Q$), if $\mathit{id}(v_1) \le \mathit{id}(v_2)$, we increment $C_{upper}[v_1]$ and decrement $C_{upper}[\mathit{next}(v_2)]$ if $\mathit{id}(v_2) < |Q|$. If, however, $\mathit{id}(v_1) > \mathit{id}(v_2)$, we increment $C_{upper}[v_1]$ and $C_{upper}(u)$ for the node $u \in Q$ with $\mathit{id}(u) = 1$ and decrement $C_{upper}[\mathit{next}(v_2)]$, thus "breaking the cyclicity".

For each cycle Q, let us remove the arc $(v, \mathit{next}(v))$ for vertex $v \in Q$ such that $\mathit{id}(v) = |Q|$. This way G' becomes acyclic; it can be viewed as a forest in which each tree is a path. For each node v of the modified graph G' in topological order, we add $C_{upper}[v]$ to $C_{upper}[\mathit{next}(v)]$ if $\mathit{next}(v) \neq \perp$. Finally, for each original cycle Q in G', we increase $C_{upper}[v]$ for all vertices $v \in Q$ by the counter $C'[Q]$. This way we have computed all the counters C_{upper} as desired.

Let us analyze the complexity. By Lemma 19, graph $G' = (V, E')$ can be constructed in $\mathcal{O}(n)$ time. By Theorem 8, there are at most n paths $\mathsf{Upper}(R)$ and all runs R can be computed in $\mathcal{O}(n)$ time. The endpoints of walks $\mathsf{Upper}(R)$ can be located in G' in $\mathcal{O}(n)$ time using weighted ancestor queries in $CST(T)$ (Theorem 10). Finally, the computation of counters via DFS and topological ordering takes $\mathcal{O}(n)$ time. ◀

This concludes efficient computation of the numbers of overlapping and non-overlapping consecutive occurrences.

▶ **Lemma 21.** *Values $ov(v)$ and $nov(v)$ for all explicit nodes v of $CST(T)$ can be computed in $\mathcal{O}(n)$ time.*

Proof. We compute the counters C_{lower} and C_{upper} using Lemmas 17 and 20, respectively. By the formula from Lemma 15, for each node v of $CST(T)$ in the bottom-up order, $ov(v)$ can be computed as a sum of $C_{lower}[v] - C_{upper}[v]$ and the sum of values $ov(w)$ for all explicit children w of v. Such values can be computed via a bottom-up traversal in $\mathcal{O}(n)$ time.

Finally we recall that $nov(v) = occ(v) - ov(v)$ and that $occ(v)$ for all explicit nodes of $CST(T)$ can be easily computed in $\mathcal{O}(n)$ time bottom-up. ◀

3.4 Computing coverage

For a substring S of T, we introduce the following notations:

$$cv\_ov(S) = \sum_{(i,j)\in OvOcc(S)} (j-i), \quad cv\_nov(S) = nov(S)\cdot|S|.$$

As before, we denote $cv\_ov(v) = cv\_ov(\bar{v})$ and $cv\_nov(v) = cv\_nov(\bar{v})$ for nodes v of $CST(T)$. The proof of the following observation provides intuition on these definitions.

▶ **Observation 22.** *For every substring S of T, $cv(S) = cv\_ov(S) + cv\_nov(S)$.*

Proof. Let us assign each position k of T that is covered by an occurrence of S to the rightmost occurrence $T[i\,..\,i+|S|)$ of S with $i < k$. Let j be the next occurrence of S to the right of position i (then $j > k$), if any. If j exists and $(i,j) \in OvOcc(S)$, then position k is counted in $cv\_ov(S)$. Otherwise position k is counted in $cv\_nov(S)$. ◀

Values $cv\_nov(v)$ for explicit nodes v of $CST(T)$ can be easily computed using values $nov(v)$ computed in Lemma 21. Lemma 12 yields the following formula for $cv\_ov(v)$.

▶ **Observation 23.** *For a node v of $CST(T)$, $cv\_ov(v) = \sum_{R\in\mathcal{R}(T)} \#_v(\mathsf{Triangle}(R))\cdot per(R)$.*

Now cv_ov values can be computed similarly as ov values were computed in Section 3.3. We just need to multiply counter updates by periods of respective runs.

▶ **Lemma 24.** *The values $cv\_ov(v)$ for all explicit nodes v of $CST(T)$ can be computed in $\mathcal{O}(n)$ time.*

Proof. Let

$$C'_{upper}[v] = \sum_{R\in\mathcal{R}(T)} \#_v(\mathsf{Upper}(R))\cdot per(R), \quad C'_{lower}[v] = \sum_{R\in\mathcal{R}(T)} \#_v(\mathsf{Lower}(R))\cdot per(R).$$

Following the proof of Lemma 15 it can be readily verified that for every explicit node v,

$$cv\_ov(v) = \sum_{w\in subtree(v)} (C'_{lower}[w] - C'_{upper}[w]).$$

The counters $C'_{lower}[v]$ ($C'_{upper}[v]$) for all explicit nodes can be computed as in Lemma 17 (Lemma 20, respectively), where instead of ±1 counters, for each path $\mathsf{Lower}(R)$ (walk $\mathsf{Upper}(R)$, respectively), $R \in \mathcal{R}(T)$, we add and subtract $per(R)$ in the respective nodes (and increase cycle counters $C'[Q]$ by amounts $\lfloor\, |\mathsf{Upper}(R)|\,/\,|Q|\,\rfloor \cdot per(R)$). ◀

This concludes the construction of $CST(T)$.

▶ **Theorem 1.** *The Cover Suffix Tree (CST) of a string of length n over an integer alphabet can be constructed in $\mathcal{O}(n)$ time.*

Proof. Lemma 11 can be used to construct the tree structure of $CST(T)$. We compute $occ(v)$ for all explicit nodes of $CST(T)$ in a bottom-up traversal and $ov(v)$ using Lemma 21, which lets us compute $nov(v) = occ(v) - ov(v)$ for all explicit nodes. Then for all explicit nodes we compute $cv\_ov(v)$ using Lemma 24, which lets us compute $cv(v)$ for all explicit nodes using values $cv\_nov(v)$ that, in turn, depend on $nov(v)$. Each of the lemmas, as well as the bottom-up processing, requires $\mathcal{O}(n)$ time. ◀

4 Solution to AllPartialCovers

An $\mathcal{O}(n \log n)$-time solution to AllPartialCovers from [29] is based on computing the upper envelope of $\mathcal{O}(n)$ line segments, each connecting points $(|v|, cv(v))$ and $(|v| - k, cv(v) - k \cdot nov(v))$ constructed for an edge of $CST(T)$ from v to its parent v' containing k implicit nodes. An upper envelope of $\mathcal{O}(n)$ line segments can be computed in $\mathcal{O}(n \log n)$ time [24].

We show that the AllPartialCovers problem can be solved in $\mathcal{O}(n)$ time using the following observation. A substring C of T is called *branching* if the locus of C in $ST(T)$ is a branching node.

▶ **Lemma 25.** *If C is a substring of T, then there is a substring C' of T such that $|C'| = |C|$, $cv(C') \geq cv(C)$ and C' is branching or a suffix of T.*

Proof. Let $C_0 = C$. If C_0 is branching or C_0 is a suffix of T, we are done. Otherwise, all occurrences of C_0 in T are followed by the same character. Let a be this character, X be C without its first character, and $C_1 = Xa$. We have $|C_1| = |C_0|$ and $cv(C_1) \geq cv(C_0)$. We use this construction to obtain substrings $C_1, C_2, \ldots$ The sequence ends at the first substring that is branching or a suffix of T; such a substring exists since the rightmost occurrence of C_i in T, for $i \geq 1$, is located to the right of the rightmost occurrence of C_{i-1} in T. Let C_k, for $k \geq 1$, be the last substring in this sequence. By the construction, $|C_k| = |C_0| = |C|$, $cv(C_k) \geq cv(C)$ and C_k is branching or a suffix of T. We choose $C' = C_k$. ◀

By the lemma, in the solution to AllPartialCovers it suffices to iterate over all suffixes of T and branching nodes of $ST(T)$. We obtain the following result that was already stated in Section 1.

Algorithm 1 Solution to AllPartialCovers.

```
for i := 1 to n do shortest[n − i + 1] := T[i . . n];
foreach branching node v of ST(T) do
    if |v̄| < |shortest[cv(v)]| then
        shortest[cv(v)] := v̄;
for i := n − 1 down to 1 do
    if |shortest[i]| > |shortest[i + 1]| then
        shortest[i] := shortest[i + 1];
```

▶ **Theorem 4.** *AllPartialCovers problem can be solved in $\mathcal{O}(n)$ time for a length-n string over an integer alphabet.*

Proof. We apply Algorithm 1. Clearly, the algorithm works in $\mathcal{O}(n)$ time. Let us argue for its correctness.

In the algorithm an auxiliary array *shortest* is used that stores fragments of T represented in $\mathcal{O}(1)$ space each. By Lemma 25, after the foreach-loop, $shortest[\alpha] = C$ for $\alpha \in [1\,..\,n]$ if C is a shortest substring of T such that $cv(C) = \alpha$. At the end, $shortest[\alpha] = C$ if C is a shortest substring of T such that $cv(C) \geq \alpha$. Hence, $shortest[\alpha]$ is a shortest α-partial cover of T by definition. ◀

5 Conclusions

We have designed the first linear-time algorithm computing the Cover Suffix Tree. We have shown several applications of this result, some of which follow directly from previous work. Experimental comparison of our algorithms for computing the Cover Suffix Tree and the set of seeds in a string with implementations of existing methods from [16] is left as future work.

It remains an open problem if our approach can help to improve upon the $\mathcal{O}(n \log n)$-time algorithm of Brodal et al. [11] for constructing MAST.

References

1. 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.
2. Alberto Apostolico and Andrzej Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theoretical Computer Science*, 119(2):247–265, 1993. doi:10.1016/0304-3975(93)90159-Q.
3. Alberto Apostolico, Martin Farach, and Costas S. Iliopoulos. Optimal superprimitivity testing for strings. *Information Processing Letters*, 39(1):17–20, 1991. doi:10.1016/0020-0190(91)90056-N.
4. 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.
5. Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "runs" theorem. *SIAM Journal on Computing*, 46(5):1501–1514, 2017. doi:10.1137/15M1011032.
6. Hideo Bannai, Shunsuke Inenaga, and Dominik Köppl. Computing all distinct squares in linear time for integer alphabets. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, volume 78 of *LIPIcs*, pages 22:1–22:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.CPM.2017.22.
7. Djamal Belazzougui, Dmitry Kosolobov, Simon J. Puglisi, and Rajeev Raman. Weighted ancestors in suffix trees revisited. In Paweł Gawrychowski and Tatiana Starikovskaya, editors, *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.
8. Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *4th Latin American Symposium on Theoretical Informatics, LATIN 2000*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000. doi:10.1007/10719839_9.
9. Srečko Brlek and Shuo Li. On the number of squares in a finite word, 2022. arXiv:2204.10204.
10. Srečko Brlek and Shuo Li. On the number of distinct squares in finite sequences: Some old and new results. In Anna E. Frid and Robert Mercas, editors, *14th International Conference on Combinatorics on Words, WORDS 2023*, volume 13899 of *Lecture Notes in Computer Science*, pages 35–44. Springer, 2023. doi:10.1007/978-3-031-33180-0_3.

11 Gerth Stolting Brodal, Rune B. Lyngso, Anna Ostlin, and Christian N. S. Pedersen. Solving the string statistics problem in time O(n log n). In Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan J. Eidenbenz, and Ricardo Conejo, editors, *29th International Colloquium on Automata, Languages and Programming, ICALP 2002*, volume 2380 of *Lecture Notes in Computer Science*, pages 728–739. Springer, 2002. doi:10.1007/3-540-45465-9_62.

12 Gerth Stolting Brodal and Christian N. S. Pedersen. Finding maximal quasiperiodicities in strings. In Raffaele Giancarlo and David Sankoff, editors, *11th Annual Symposium on Combinatorial Pattern Matching, CPM 2000*, volume 1848 of *Lecture Notes in Computer Science*, pages 397–411. Springer, 2000. doi:10.1007/3-540-45123-4_33.

13 Mark R. Brown and Robert Endre Tarjan. A fast merging algorithm. *Journal of the ACM*, 26(2):211–226, 1979. doi:10.1145/322123.322127.

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

15 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.

16 Patryk Czajka and Jakub Radoszewski. Experimental evaluation of algorithms for computing quasiperiods. *Theoretical Computer Science*, 854:17–29, 2021. doi:10.1016/j.tcs.2020.11.033.

17 Jonas Ellert and Johannes Fischer. Linear time runs over general ordered alphabets. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *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.

18 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.

19 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.

20 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.

21 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with O(1) worst case access time. *Journal of the ACM*, 31(3):538–544, 1984. doi:10.1145/828.1884.

22 Moses Ganardi and Paweł Gawrychowski. Pattern matching on grammar-compressed strings in linear time. In Joseph (Seffi) Naor and Niv Buchbinder, editors, *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022*, pages 2833–2846. SIAM, 2022. doi:10.1137/1.9781611977073.110.

23 Paweł Gawrychowski, Moshe Lewenstein, and Patrick K. Nicholson. Weighted ancestors in suffix trees. In Andreas S. Schulz and Dorothea Wagner, editors, *22th Annual European Symposium on Algorithms, Wrocław, Poland, ESA 2014*, volume 8737 of *Lecture Notes in Computer Science*, pages 455–466. Springer, 2014. doi:10.1007/978-3-662-44777-2_38.

24 John Hershberger. Finding the upper envelope of n line segments in O(n log n) time. *Information Processing Letters*, 33(4):169–174, 1989. doi:10.1016/0020-0190(89)90136-1.

25 Costas S. Iliopoulos, Dennis W. G. Moore, and Kunsoo Park. Covering a string. *Algorithmica*, 16(3):288–297, 1996. doi:10.1007/BF01955677.

26 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.

27 Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A linear time algorithm for seeds computation. In Yuval Rabani, editor, *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012*, pages 1095–1112. SIAM, 2012. doi:10.1137/1.9781611973099.86.

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

29 Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Fast algorithm for partial covers in words. *Algorithmica*, 73(1):217–233, 2015. doi:10.1007/s00453-014-9915-3.

30 Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Efficient algorithms for shortest partial seeds in words. *Theoretical Computer Science*, 710:139–147, 2018. doi:10.1016/j.tcs.2016.11.035.

31 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.

32 Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976. doi:10.1145/321941.321946.

33 Dennis W. G. Moore and William F. Smyth. Computing the covers of a string in linear time. In Daniel Dominic Sleator, editor, *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms. SODA 1994*, pages 511–515. ACM/SIAM, 1994. URL: http://dl.acm.org/citation.cfm?id=314464.314636.

34 Dennis W. G. Moore and William F. Smyth. A correction to "An optimal algorithm to compute all the covers of a string". *Information Processing Letters*, 54(2):101–103, 1995. doi:10.1016/0020-0190(94)00235-Q.

35 Gonzalo Navarro and Sharma V. Thankachan. Reporting consecutive substring occurrences under bounded gap constraints. *Theoretical Computer Science*, 638:108–111, 2016. doi:10.1016/j.tcs.2016.02.005.