Algorithmica (2015) 73:217–233
DOI 10.1007/s00453-014-9915-3

Fast Algorithm for Partial Covers in Words

Tomasz Kociumaka · Solon P. Pissis · Jakub Radoszewski · Wojciech Rytter · Tomasz Waleń

Received: 10 December 2013 / Accepted: 19 June 2014 / Published online: 16 July 2014

Abstract A factor u of a word w is a *cover* of w if every position in w lies within some occurrence of u in w. A word w covered by u thus generalizes the idea of a *repetition*, that is, a word composed of exact concatenations of u. In this article we introduce a new notion of *α-partial cover*, which can be viewed as a relaxed variant of cover, that is, a factor covering at least α positions in w. We develop a data structure of $\mathcal{O}(n)$ size (where $n = |w|$) that can be constructed in $\mathcal{O}(n \log n)$ time which we apply to compute all shortest α-partial covers for a given α. We also employ it for an $\mathcal{O}(n \log n)$-time algorithm computing a shortest α-partial cover for each $\alpha = 1, 2, \ldots, n$.

Keywords Cover of a word · Quasiperiodicity · Suffix tree

T. Kociumaka · J. Radoszewski (✉) · W. Rytter · T. Waleń
Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Warsaw, Poland
e-mail: jrad@mimuw.edu.pl

T. Kociumaka
e-mail: kociumaka@mimuw.edu.pl

W. Rytter
e-mail: rytter@mimuw.edu.pl

T. Waleń
e-mail: walen@mimuw.edu.pl

S. P. Pissis
Department of Informatics, King's College London, London WC2R 2LS, UK
e-mail: solon.pissis@kcl.ac.uk

W. Rytter
Faculty of Mathematics and Computer Science, Copernicus University, Toruń, Poland

1 Introduction

The notion of periodicity in words and its many variants have been well-studied in numerous fields like combinatorics on words, pattern matching, data compression, automata theory, formal language theory, and molecular biology (see [10]). However the classic notion of periodicity is too restrictive to provide a description of a word such as abaababaaba, which is covered by copies of aba, yet not exactly periodic. To fill this gap, the idea of *quasiperiodicity* was introduced [1]. In a periodic word, the occurrences of the period do not overlap. In contrast, the occurrences of a quasiperiod in a quasiperiodic word may overlap. Quasiperiodicity thus enables the detection of repetitive structures that would be ignored by the classic characterization of periods.

The most well-known formalization of quasiperiodicity is the cover of word. A factor u of a word w is said to be a *cover* of w if $u \neq w$, and every position in w lies within some occurrence of u in w. Equivalently, we say that u *covers* w. Note that a cover of w must also be a *border*—both prefix and suffix—of w. Thus, in the above example, aba is the shortest cover of abaababaaba.

A linear-time algorithm for computing the shortest cover of a word was proposed by Apostolico et al. [3], and a linear-time algorithm for computing all the covers of a word was proposed by Moore and Smyth [22]. Breslauer [4] gave an online linear-time algorithm computing the *minimal cover array* of a word—a data structure specifying the shortest cover of every prefix of the word. Li and Smyth [21] provided a linear-time algorithm for computing the *maximal cover array* of a word, and showed that, analogous to the border array [9], it actually determines the structure of *all* the covers of every prefix of the word.

A known extension of the notion of cover is the notion of *seed*. A seed is not necessarily aligned with the ends of the word being covered, but is allowed to overflow on either side. More formally, a word u is a seed of w if u is a factor of w and w is a factor of some word y covered by u. Seeds were first introduced by Iliopoulos et al. [17]. A linear-time algorithm for computing the shortest seed of a word was given by Kociumaka et al. [18].

Still it remains unlikely that an arbitrary word, even over the binary alphabet, has a cover (or even a seed). For example, abaaababaabaaaababaa is a word that not only has no cover, but whose every prefix also has no cover. In this article we provide a natural form of quasiperiodicity. We introduce the notion of *partial covers*, that is, factors covering at least a given number of positions in w. Recently, Flouri et al. [13] suggested a related notion of *enhanced covers* which are additionally required to be borders of the word.

Partial covers can be viewed as a relaxed variant of covers alternative to approximate covers [23]. The approximate covers require each position to lie within an approximate occurrence of the cover. This allows for small irregularities within each fragment of a word. On the other hand partial covers require exact occurrences but drop the condition that all positions need to be covered. This allows some fragments to be completely irregular as long as the total length of such fragments is small. Due to the requirement of exact occurrences in partial covers they enjoy a number of combinatorial properties thanks to which they can be computed more efficiently than approximate covers,

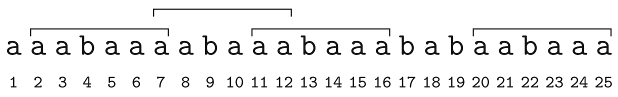

Fig. 1 For $u = \texttt{aabaaa}$ and $w = \texttt{aaabaaaabaaabaaababaabaaa}$ we have $\mathit{Covered}(u, w) = 21$

where the time complexity rarely drops below quadratic and some problems are even NP-hard.

Let $\mathit{Covered}(u, w)$ denote the number of positions in w covered by occurrences of the word u in w; we call this value the *cover index* of u within w (see Fig. 1). We primarily focus on the following two problems, but the tools we develop can be used to answer a number of questions concerning partial covers, some of which are discussed in the conclusions.

PartialCovers **problem**
Input: a word w of length n and a positive integer $\alpha \le n$.
Output: all shortest factors u such that $\mathit{Covered}(u, w) \ge \alpha$.

Each factor given in the output is represented by the starting and ending position of its occurrence in w.

Example 1 Let w = $\texttt{bcccacccaccaccb}$ and $\alpha = 11$. Then the only shortest α-partial covers are $\texttt{ccac}$ and $\texttt{cacc}$.

AllPartialCovers **problem**
Input: a word w of length n.
Output: for all $\alpha = 1, \ldots, n$, a shortest factor u such that $\mathit{Covered}(u, w) \ge \alpha$.

Our contribution. The following summarizes our main result.

Theorem 1 *The PartialCovers and AllPartialCovers problems can be solved in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space.*

We extensively use suffix trees, for an exposition see [8,9]. A suffix tree of a word is a compact trie of its suffixes, the nodes of the trie which become nodes of the suffix tree are called *explicit* nodes, while the other nodes are called *implicit*. Each edge of the suffix tree can be viewed as an *upward* maximal path of implicit nodes starting with an explicit node. Moreover, each node belongs to a unique path of that kind. Then, each node of the trie can be represented in the suffix tree by the edge it belongs to and an index within the corresponding path. Each factor of the word corresponds to an explicit or implicit node of the suffix tree. A representation of this node is called the *locus* of the factor. Our algorithm finds the loci of the shortest partial covers, it is then straightforward to locate an occurrence for each of them.

1.1 A Sketch of the Algorithm

The algorithm first augments the suffix tree of w, that is, a linear number of implicit extra nodes become explicit. Then, each node of the augmented tree is annotated with two integer values. They allow for determining the size of the covered area for each implicit node by a simple formula, since limited to a single edge of the augmented suffix tree, these values form an arithmetic progression. This yields a solution to the *PartialCovers* problem. For an efficient solution to the *AllPartialCovers* problem, we additionally find the upper envelope of a number of line segments constructed from the arithmetic progressions.

1.2 Structure of the Paper

In Sect. 2 we formally introduce the augmented and annotated suffix tree that we call *Cover Suffix Tree*. We show its basic properties and present its application for *PartialCovers* and *AllPartialCovers* problems. Section 4 is dedicated to the construction of the Cover Suffix Tree. Before that, Sect. 3 presents an auxiliary data structure being an extension of the classical Union/Find data structure; its implementation is given later, in Sect. 5. Additional applications of the Cover Suffix Tree are given in Sects. 6 and 7. The former presents how the data structure can be used to compute all distinct primitively rooted squares in a word and a linear-sized representation of all the seeds in a word. The latter contains a short discussion of variants of the *PartialCovers* problem that can be solved in a similar way.

A preliminary version of this work appeared in the Proceedings of the Twenty-Fourth Annual Symposium on Combinatorial Pattern Matching, pp. 177–188, 2013.

2 Augmented and Annotated Suffix Trees

Let w be a word of length n over a totally ordered alphabet Σ. The suffix tree T of w can be constructed in $\mathcal{O}(n \log |\Sigma|)$ time [12,24]. For an explicit or implicit node v of T, we denote by $\hat{v}$ the word obtained by spelling the characters on a path from the root to v. We also denote $|v| = |\hat{v}|$. As in most applications of the suffix tree, the leaves of T play an auxiliary role and do not correspond to factors (actually they are suffixes of $w\#$, where $\# \notin \Sigma$). They are labeled with the starting positions of the suffixes of w.

We introduce the *Cover Suffix Tree* of w, denoted by $\mathit{CST}(w)$, as an *augmented*—new nodes are added—suffix tree in which the nodes are *annotated* with information relevant to covers. $\mathit{CST}(w)$ is similar to the data structure named *Minimal Augmented Suffix Tree* (see [2,6]).

For a set X of integers and $x \in X$, we define

$$\mathit{next}_X(x) = \min\{y \in X, y > x\},$$

and we assume $\mathit{next}_X(x) = \infty$ if $x = \max X$. By $\mathit{Occ}(v, w)$ we denote the set of starting positions of occurrences of $\hat{v}$ in w. For any $i \in \mathit{Occ}(v, w)$, we define:

$$\delta(i, v) = \mathit{next}_{\mathit{Occ}(v,w)}(i) - i.$$

Note that $\delta(i, v) = \infty$ if i is the last occurrence of $\hat{v}$. Additionally, we define:

$$cv(v) = \mathit{Covered}(\hat{v}, w),\ \ \Delta(v) = \left|\{i \in \mathit{Occ}(v, w) : \delta(i, v) \geq |v|\}\right|;$$

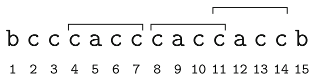

Fig. 2 Let $w =$ bcccacccaccaccb and let v be the node corresponding to cacc. We have $Occ(v, w) = \{4, 8, 11\}$, $cv(v) = 11$, $\Delta(v) = 2$

see, for example, Fig. 2.

A word u is called *primitive* if $u = y^k$ for a word y and an integer k implies that $y = u$, and non-primitive otherwise. A square u^2 is called *primitively rooted* if u is primitive.

Observation 1 *Let v be a node in the suffix trie of w. Then $\hat{v}\hat{v}$ is a primitively rooted square in w if and only if there exists $i \in Occ(v, w)$ such that $\delta(i, v) = |v|$.*

Proof Recall that, by the synchronization property of primitive words (see [9]), $\hat{v}$ is primitive if and only if it occurs exactly twice in $\hat{v}\hat{v}$.

($\Rightarrow$) If $\hat{v}\hat{v}$ occurs in w at position i then $\delta(i, v) = |v|$.
($\Leftarrow$) If $\delta(i, v) = |v|$ then obviously $\hat{v}\hat{v}$ occurs in w at position i. Additionally, if $\hat{v}$ was not primitive then $\delta(i, v) < |v|$ would hold.

□

In $CST(w)$, we introduce additional explicit nodes called *extra nodes*, which correspond to halves of primitively rooted square factors of w. Moreover we annotate all explicit nodes (including extra nodes) with the values cv, Δ; see, for example, Fig. 3. The number of extra nodes is bounded by the number of distinct squares, which is linear [14], so $CST(w)$ takes $\mathcal{O}(n)$ space.

Lemma 1 *Let $v_1, v_2, \ldots, v_k$ be the consecutive implicit nodes on the edge from an explicit node v of $CST(w)$ to its explicit parent. Then for $1 \le i \le k$ we have*

$$cv(v_i) = cv(v) - i\Delta(v),$$

in particular $(cv(v_i))_{i=1}^{k}$ forms an arithmetic progression.

Proof Note that $Occ(v_i, w) = Occ(v, w)$, since otherwise v_i would be an explicit node of $CST(w)$. Also note that if any two occurrences of $\hat{v}$ in w overlap, then the corresponding occurrences of $\hat{v_i}$ overlap. Otherwise, by Observation 1, the path from v to v_i (excluding v) would contain an extra node. Hence, when we go up from v (before reaching its parent) the size of the covered area decreases at each step by $\Delta(v)$. □

Example 2 Consider the word w from Fig. 3. The word cccacc corresponds to an explicit node of $CST(w)$; we denote it by v. We have $cv(v) = 10$ and $\Delta(v) = 1$ since the two occurrences of the factor cccacc in w overlap. The word cccac corresponds to an implicit node v' and $cv(v') = 10 - 1 = 9$. Now the word ccca corresponds to an extra node v'' of $CST(w)$. Its occurrences are adjacent in w and $cv(v'') = 8$, $\Delta(v'') = 2$. The word ccc corresponds to an implicit node v''' and $cv(v''') = 8 - 2 = 6$.

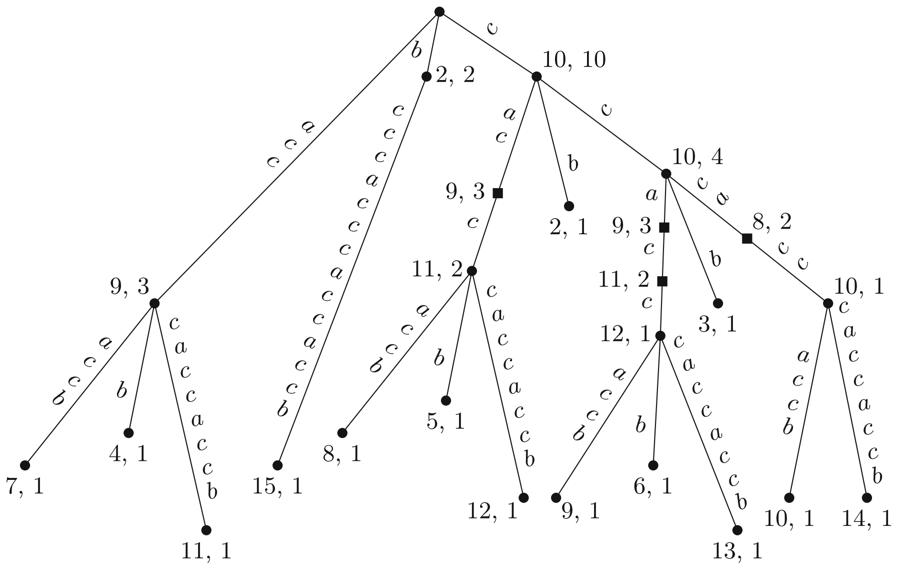

Fig. 3 *CST*(w) for $w = \texttt{bcccacccaccaccb}$. It contains four extra nodes that are denoted by *squares* in the figure. *Each node* is annotated with $cv(v)$, $\Delta(v)$. Leaves are omitted for clarity

As a consequence of Lemma 1 we obtain the following result. Recall that the locus of a factor v of w, given by its start and end position in w, can be found in $\mathcal{O}(\log\log|v|)$ time [20].

Lemma 2 *Assume we are given CST(w). Then we can compute:*

(1) *for any α, the loci of the shortest α-partial covers in linear time;*
(2) *given the locus of a factor u in the suffix tree CST(w), the cover index Covered(u, w) in $\mathcal{O}(1)$ time.*

Proof Part (2) is a direct consequence of Lemma 1. As for part (1), for each edge of *CST*(w), leading from v to its parent v', we need to find minimum $|v| \geq j > |v'|$ for which $cv(v) - \Delta(v) \cdot (|v| - j) \geq \alpha$. Such a linear inequality can be solved in constant time. □

Due to this fact the efficiency of the *PartialCovers* problem relies on the complexity of *CST*(w) construction. In turn, the following lemma, also a consequence of Lemma 1, can be used to solve *AllPartialCovers* problem provided that *CST*(w) is given. As a tool a solution to the geometric problem of upper envelope [16] is applied.

Lemma 3 *Assume we are given CST(w). Then we can compute the locus of a shortest α-partial cover for each $\alpha = 1, 2, \ldots, n$ in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space.*

Proof Consider an edge of *CST*(w) from v to its parent v' containing k implicit nodes. For each such edge, we form a line segment on the plane connecting points $(|v|, cv(v))$ and $(|v| - k, cv(v) - k \cdot \Delta(v))$ (if there are no implicit nodes on the edge, the line segment is a single point). Denote all such line segments obtained from *CST*(w) as

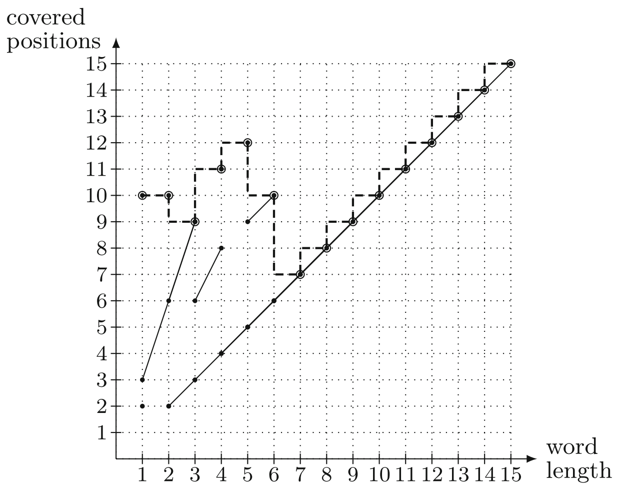

Fig. 4 Line segments constructed as in Lemma 3 for the $CST(w)$ from Fig. 3. E.g., the line segment corresponding to the edge from cccacc to ccca is (5, 9)–(6, 10). The *marked points* joined with a *dashed polyline* show the values of the integer upper envelope function $\mathcal{E}'$. The prefix maxima sequence for $\mathcal{E}'$ is as follows: $\mu_1 = \mu_2 = \mu_3 = 10$, $\mu_4 = 11$, $\mu_5 = \ldots = \mu_{12} = 12$, $\mu_{13} = 13$, $\mu_{14} = 14$, $\mu_{15} = 15$. We infer that the lengths of the shortest α-partial covers of w are: 1 for $\alpha \le 10$, 4 for $\alpha = 11$, 5 for $\alpha = 12$, and α for $\alpha \ge 13$

$s_1, \ldots, s_m$; we have $m = \mathcal{O}(n)$. We consider the upper envelope $\mathcal{E}$ of the set of these segments. Formally, if each s_i connecting points (x_i, y_i) and (x'_i, y'_i), $x_i \le x'_i$, is interpreted as a linear function on a domain $[x_i, x'_i]$, $\mathcal{E}$ is defined as a function $\mathcal{E} : [1, n] \to [1, n]$ such that:

$$\mathcal{E}(x) = \max\{s_i(x) \; : \; i \in \{1, \ldots, m\}, \quad x \in [x_i, x'_i]\}.$$

Here we are actually interested in an *integer envelope* $\mathcal{E}'$, that is, $\mathcal{E}$ limited to integer arguments, see Fig. 4. By Lemma 1, for any $j \in \{1, \ldots, n\}$, $\mathcal{E}'(j)$ equals the maximum of $Covered(u, w)$ over all factors u of w such that $|u| = j$. A piecewise linear representation of $\mathcal{E}$ can be computed in $\mathcal{O}(m \log m)$ time and $\mathcal{O}(m)$ space [16], therefore the function $\mathcal{E}'$ for all its arguments can be computed in the same time complexity.

Let us introduce a prefix maxima sequence for $\mathcal{E}'$: $\mu_i = \max\{\mathcal{E}'(j) \; : \; j \in \{1, \ldots, i\}\}$, with $\mu_0 = 0$. Note that μ_i is non-decreasing. If $\mu_i > \mu_{i-1}$ then the shortest α-partial cover for all $\alpha \in (\mu_{i-1}, \mu_i]$ has length i. An example of such a partial cover can be recovered if we explicitly store the initial line segments used in the pieces of the representation of $\mathcal{E}$. Thus the solution of the *AllPartialCovers* problem can be obtained from the sequence μ_i in $\mathcal{O}(m) = \mathcal{O}(n)$ time. $\square$

In the following two sections we provide an $\mathcal{O}(n \log n)$ time construction of $CST(w)$. Together with Lemmas 2 and 3, it yields Theorem 1.

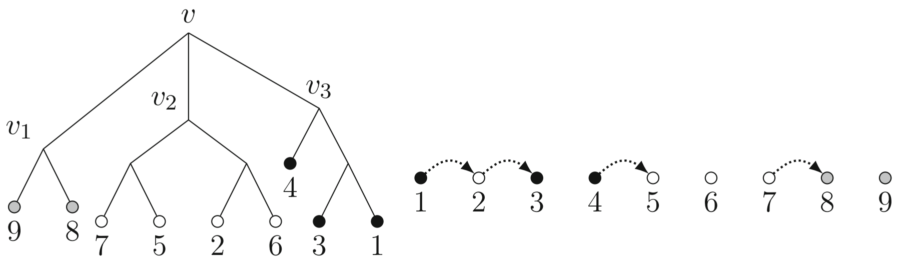

Fig. 5 Let $\mathcal{P}$ be the partition of $\{1, \ldots, 9\}$ whose classes consist of leaves in the subtrees rooted at children of v, $\mathcal{P} = \{\{1, 3, 4\}, \{2, 5, 6, 7\}, \{8, 9\}\}$, and let $\mathcal{P}' = \{\{1, \ldots, 9\}\}$. Then $\mathit{ChangeList}(\mathcal{P}, \mathcal{P}') = \{(1, 2), (2, 3), (4, 5), (7, 8)\}$ (depicted by *dotted arrows*)

3 Extension of Disjoint-Set Data Structure

In this section we extend the classic disjoint-set data structure to compute the *change lists* of the sets being merged, as defined below. First, let us extend the *next* notation. For a partition $\mathcal{P} = \{P_1, \ldots, P_k\}$ of $U = \{1, \ldots, n\}$, we define

$$\mathit{next}_{\mathcal{P}}(x) = \mathit{next}_{P_i}(x) \quad \text{where} \quad x \in P_i.$$

Now for two partitions $\mathcal{P}, \mathcal{P}'$ let us define the *change list* (see also Fig. 5) by

$$\mathit{ChangeList}(\mathcal{P}, \mathcal{P}') = \{(x, \mathit{next}_{\mathcal{P}'}(x)) : \mathit{next}_{\mathcal{P}}(x) \neq \mathit{next}_{\mathcal{P}'}(x)\}.$$

We say that $(\mathcal{P}, \mathit{id})$ is a partition of U *labeled* by L if $\mathcal{P}$ is a partition of U and $\mathit{id} : \mathcal{P} \to L$ is a one-to-one (injective) mapping. A label $\ell \in L$ is called *active* if $\mathit{id}(P) = \ell$ for some $P \in \mathcal{P}$ and *free* otherwise.

Lemma 4 *Let $n \leq k$ be positive integers such that k is of magnitude $\Theta(n)$. There exists a data structure of size $\mathcal{O}(n)$, which maintains a partition $(\mathcal{P}, \mathit{id})$ of $\{1, \ldots, n\}$ labeled by $L = \{1, \ldots, k\}$ and supports the following operations:*

- *$\mathit{Find}(x)$ for $x \in \{1, \ldots, n\}$ gives the label of $P \in \mathcal{P}$ containing x.*
- *$\mathit{Union}(I, \ell)$ for a set I of active labels ($|I| \geq 2$) and a free label ℓ replaces all $P \in \mathcal{P}$ with labels in I by their set-theoretic union with the label ℓ. The change list of the corresponding modification of $\mathcal{P}$ is returned.*

Initially $\mathcal{P}$ is a partition into singletons with $\mathit{id}(\{x\}) = x$. Any valid sequence of Union operations is performed in $\mathcal{O}(n \log n)$ time. A single Find operation takes $\mathcal{O}(1)$ time.

Note that these are actually standard disjoint-set data structure operations except for the fact that we require *Union* to return the change list. The technical proof of Lemma 4 is postponed until Sect. 5.

4 $\mathcal{O}(n \log n)$-Time Construction of $\mathit{CST}(w)$

The suffix tree of w augmented with extra nodes is called the *skeleton* of $\mathit{CST}(w)$, which we denote by $\mathit{sCST}(w)$. It could be constructed using the fact that all square factors of a word can be computed in linear time [11,15]. However, we do not need such a complicated machinery here. We will compute $sCST(w)$ on the fly, simultaneously annotating the nodes with cv, $\varDelta$.

We introduce auxiliary notions related to covered area of nodes:

$$cv_h(v) = \sum_{\substack{i \in Occ(v,w) \\ \delta(i,v) < h}} \delta(i, v), \quad \Delta_h(v) = |\{i \in Occ(v, w) \,:\, \delta(i, v) \geq h\}|.$$

Observation 2 $cv(v) = cv_{|v|}(v) + \varDelta_{|v|}(v) \cdot |v|$, $\Delta(v) = \Delta_{|v|}(v)$.

Example 3 Let $w =$ bcccacccaccaccb and let v be the node corresponding to cacc, as in Fig. 2; $Occ(v, w) = \{4, 8, 11\}$. We have: $cv_4(v) = 3$, $\varDelta_4(v) = \varDelta(v) = 2$, $cv(v) = 3 + 2 \cdot 4 = 11$.

In the course of the algorithm some nodes will have their values c, $\varDelta$ already computed; we call them *processed nodes*. Whenever v will be processed, so will its descendants.

The algorithm processes inner nodes v of $sCST(w)$ in the order of non-increasing height $h = |v|$. The height is not defined for leaves, so we start with $h = n + 1$. Extra nodes are created on the fly using Observation 1 (this takes place in the auxiliary *Lift* routine).

We maintain the partition $\mathcal{P}$ of $\{1, \ldots, n\}$ given by sets of leaves of subtrees rooted at *peak nodes*. Initially the peak nodes are the leaves of $sCST(w)$. Each time we process v, all its children are peak nodes. Consequently, after processing v they are no longer peak nodes and v becomes a new peak node. The sets in the partition are labeled with identifiers of the corresponding peak nodes. Recall that leaves are labeled with the starting positions of the corresponding suffixes. We allow any labeling of the remaining nodes as long as each node of $sCST(w)$ has a distinct label of magnitude $\mathcal{O}(n)$. For this set of labels we store the data structure of Lemma 4 to compute the change list of the changing partition.

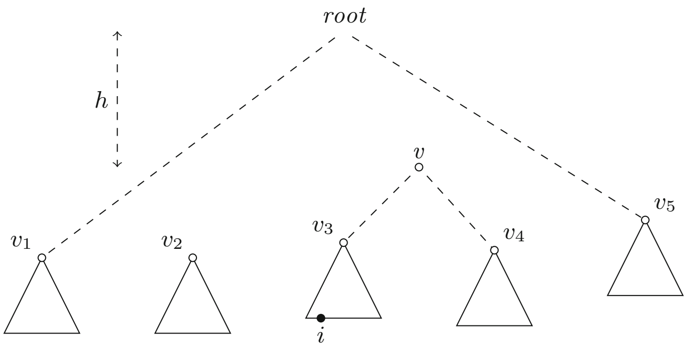

Fig. 6 One stage of the algorithm, where the peak nodes are $v_1, \ldots, v_5$ while the currently processed node is v. If $i \in List[d]$ and $v_3 = Find(i)$, then $d = \delta(i, v_3) = Dist[i]$. The current partition is $\mathcal{P} = \{Leaves(v_1), Leaves(v_2), Leaves(v_3), Leaves(v_4), Leaves(v_5)\}$. After v is processed, the partition changes to $\mathcal{P} = \{Leaves(v_1), Leaves(v_2), Leaves(v), Leaves(v_5)\}$. The *Union* operation merges $Leaves(v_4)$, $Leaves(v_3)$ and returns the corresponding change list

We maintain the following technical invariant (see Fig. 6).
Invariant(h):

(A) For each peak node z we store: $cv'[z] = cv_h(z),\ \Delta'[z] = \Delta_h(z)$.
(B) For each $i \in \{1, \ldots, n\}$ we store $Dist[i] = \delta(i,\ Find(i))$.
(C) For each $d < h$ we store $List[d] = \{i : Dist[i] = d\}$.

We use two auxiliary routines. The *Lift* operation updates cv' and Δ' values when h decrements. It also creates all extra nodes of depth h. The *LocalCorrect* operation is used for updating cv' and Δ' values for children of the node v. The *Dist* and *List* arrays are stored to enable efficient implementation of these two routines.

```
Algorithm ComputeCST(w)
    $T$ := suffix tree of $w$;
    $\mathcal{P}$ := partition of $\{1, \ldots, n\}$ into singletons;
    foreach $v$ : a leaf of $T$ do $cv'[v] := 0$, $\Delta'[v] := 1$;
    for $h := n + 1$ downto $0$ do
        $Lift(h)$;
        {Now part (A) of Invariant($h$) is satisfied}
        foreach $v$ : an inner node of $T$, $|v| = h$ do
            $cv'[v] := \sum_{u \in children(v)} cv'[u]$;
            $\Delta'[v] := \sum_{u \in children(v)} \Delta'[u]$;
            $ChangeList(v) := Union(children(v), v)$
            foreach $(p, q) \in ChangeList(v)$ do $LocalCorrect(p, q, v)$;
            $cv[v] := cv'[v] + \Delta'[v] \cdot |v|$;
            $\Delta[v] := \Delta'[v]$;
    return $T$ together with values of $cv$, $\Delta$;
```

4.1 Description of the *Lift*(h) Operation

The procedure *Lift* plays an important preparatory role in processing the current node. According to part (A) of our invariant, for all peak nodes z we know the values: $cv'[z] = cv_{h+1}(z),\ \Delta'[z] = \Delta_{h+1}(z)$. Now we have to change $h+1$ to h and guarantee validity of the invariant: $cv'[z] = cv_h(z),\ \Delta'[z] = \Delta_h(z)$. This is exactly how the following operation updates cv' and Δ'.

It also creates all extra nodes of depth h that were not explicit nodes of the suffix tree. By Observation 1, if $i \in List[h]$ then at position i in w there is an occurrence of a primitively rooted square of half length h. Consequently, an extra node corresponding to this occurrence is created in the *Lift* operation.

```
Function Lift(h)
    foreach $i$ in $List[h]$ do
        $v := Find(i)$;
        $\Delta'[v] := \Delta'[v] + 1$; $cv'[v] := cv'[v] - h$;
        if $|parent(v)| < h$ then
            Create a node of depth $h$ on the edge from $parent(v)$ to $v$;
```

4.2 Description of the *LocalCorrect*(p, q, v) Operation

Here we assume that $\hat{v}$ occurs at positions $p < q$ and that these are consecutive occurrences. Moreover, we assume that these occurrences are followed by distinct characters, i.e. $(p, q) \in \mathit{ChangeList}(v)$. The *LocalCorrect* procedure updates $\mathit{Dist}[p]$ to make part (B) of the invariant hold for p again. The data structure *List* is updated accordingly so that (C) remains satisfied.

```
Function LocalCorrect(p, q, v)
    $d := q - p$; $d' := \mathit{Dist}[p]$;
    if $d' < |v|$ then $cv'[v] := cv'[v] - d'$;
    else $\Delta'[v] := \Delta'[v] - 1$;
    if $d < |v|$ then $cv'[v] := cv'[v] + d$;
    else $\Delta'[v] := \Delta'[v] + 1$;
    $\mathit{Dist}[p] := d$;
    $\mathit{remove}(p, \mathit{List}[d'])$; $\mathit{insert}(p, \mathit{List}[d])$;
```

4.3 Complexity of the Algorithm

In the course of the algorithm we compute $\mathit{ChangeList}(v)$ for each $v \in T$. We have:

$$\sum_{v \in T} |\mathit{ChangeList}(v)| = \mathcal{O}(n \log n),$$

since each *ChangeList* is a result of *Union* operation and the total cost of such operations, by Lemma 4, is $\mathcal{O}(n \log n)$. Consequently we perform $\mathcal{O}(n \log n)$ operations *LocalCorrect*. In each of them at most one element is added to a list $\mathit{List}[d]$ for some d. Hence the total number of insertions to these lists is also $\mathcal{O}(n \log n)$.

The cost of each operation *Lift* is proportional to the total size of the list $\mathit{List}[h]$ processed in this operation. For each h, the list $\mathit{List}[h]$ is processed once and the total number of insertions into lists is $\mathcal{O}(n \log n)$, therefore the total cost of all operations *Lift* is also $\mathcal{O}(n \log n)$. This proves the following fact which, together with Lemmas 2 and 3, implies our main result (Theorem 1).

Lemma 5 *Algorithm* COMPUTECST *constructs* $\mathit{CST}(w)$ *in* $\mathcal{O}(n \log n)$ *time and* $\mathcal{O}(n)$ *space, where* $n = |w|$.

5 Implementation Details

In this section we give a proof of Lemma 4. We use an approach similar to Brodal and Pedersen [5] (who use the results of [7]) originally devised for computation of maximal quasiperiodicities. Theorem 3 of [5] states that a subset X of a linearly ordered universe can be stored in a height-balanced tree of linear size supporting the following operations:

$X.\mathit{MultiInsert}(Y)$: insert all elements of Y to X,

$X.\mathit{MultiPred}(Y)$: return all (y, x) for $y \in Y$ and $x = \max\{z \in X, z < y\}$,
$X.\mathit{MultiSucc}(Y)$: return all (y, x) for $y \in Y$ and $x = \min\{z \in X, z > y\}$,

in $O\left(|Y| \max\left(1, \log \frac{|X|}{|Y|}\right)\right)$ time.

Recall that our goal is to implement a sequence od *Find* and *Union* operations on a dynamic partition $(\mathcal{P}, \mathit{id})$ of $\{1, \dots, n\}$ labeled by identifiers from a set L. Each *Union* operation is given a list of labels of sets in the partition and is to return a change list of these sets after merge. The label of $P \in \mathcal{P}$ is denoted as $\mathit{id}(P)$.

In the data structure we store each $P \in \mathcal{P}$ as a height-balanced tree. Additionally, we store several auxiliary arrays, whose semantics follows. For each $x \in \{1, \dots, n\}$ we maintain a value $\mathit{next}[x] = \mathit{next}_{\mathcal{P}}(x)$ and a pointer $\mathit{tree}[x]$ to the tree representing P such that $x \in P$. For each $P \in \mathcal{P}$ (technically for each tree representing $P \in \mathcal{P}$) we store $\mathit{id}[P]$ and for each $\ell \in L$ we store $\mathit{id}^{-1}[\ell]$, a pointer to the corresponding tree (null for free labels).

Answering *Find* is trivial as it suffices to follow the *tree* pointer and return the *id* value. The *Union* operation is performed according to the pseudocode given below (for brevity we write P_i instead of $\mathit{id}^{-1}[i]$).

```
Function Union(I, ℓ)
    $i_0 := \mathrm{argmax}\{|P_i| : i \in I\}$;
    $S := P_{i_0}$;
    foreach $i \in I \setminus \{i_0\}$ do
        foreach $x \in P_i$ do $\mathit{tree}[x] := S$;
        $S.\mathit{MultiInsert}(P_i)$;
    $C := \emptyset$;
    foreach $i \in I \setminus \{i_0\}$ do
        foreach $(b, a) \in S.\mathit{MultiPred}(P_i)$ do
            if $\mathit{next}[a] \neq b$ then $C := C \cup \{(a, b)\}$;
        foreach $(a, b) \in S.\mathit{MultiSucc}(P_i)$ do
            if $\mathit{next}[a] \neq b$ then $C := C \cup \{(a, b)\}$;
        $\mathit{id}^{-1}[i] :=$ null;
    $\mathit{id}^{-1}[i_0] :=$ null;
    $\mathit{id}[S] := \ell$; $\mathit{id}^{-1}[\ell] := S$;
    foreach $(x, y) \in C$ do $\mathit{next}[x] := y$;
    return $C$;
```

Claim *The Union operation correctly computes the change list and updates the data structure.*

Proof In the *Union* operation for sets P_i, $i \in I$, we find the largest set P_{i_0} and *MultiInsert* all the elements of the remaining sets to P_{i_0}. If (a, b) is in the change list, then a and b come from different sets P_i, in particular at least one of them does not come from P_{i_0}. Depending on which one it is, the pair (a, b) is found by *MultiPred* or *MultiSucc* operation. While computing C, the table *next* is not updated yet (i.e. corresponds to the state before *Union* operation) while S is already updated. Consequently the pairs inserted to C indeed belong to the change list. Once C is proved to be the change list, it is clear that *next* is updated correctly. For the other components of the data structure, correctness of updates is evident. □

Claim *Any sequence of Union operations takes $\mathcal{O}(n \log n)$ time in total.*

Proof Let us introduce a potential function $\Phi(\mathcal{P}) = \sum_{P\in\mathcal{P}} |P| \log |P|$. We shall prove that the running time of a single *Union* operation is proportional to the increase in potential. Clearly

$$0 \le \Phi(\mathcal{P}) = \sum_{P\in\mathcal{P}} |P| \log |P| \le \sum_{P\in\mathcal{P}} |P| \log n = n \log n,$$

so this suffices to obtain the desired $\mathcal{O}(n \log n)$ bound.

Let us consider a *Union* operation that merges partition classes of sizes $p_1 \ge p_2 \ge \cdots \ge p_k$ to a single class of size $p = \sum_{i=1}^{k} p_i$. The most time-consuming steps of the algorithm are the operations on height-balanced trees, which, for single i, run in $O\left(\max\left(p_i, p_i \log \frac{p}{p_i}\right)\right)$ time. These operations are not performed for the largest set and for the remaining ones we have $p_i < \frac{1}{2}p$ (i.e. $\log \frac{p}{p_i} \ge 1$). This lets us bound the time complexity of the *Union* operation as follows:

$$\begin{aligned}\sum_{i=2}^{k} \max\left(p_i, p_i \log \tfrac{p}{p_i}\right) &= \sum_{i=2}^{k} p_i \log \tfrac{p}{p_i} \le \sum_{i=1}^{k} p_i \log \tfrac{p}{p_i}\\ &= \sum_{i=1}^{k} p_i (\log p - \log p_i) = p \log p - \sum_{i=1}^{k} p_i \log p_i,\end{aligned}$$

which is equal to the increase in potential. □

6 By-Products of Cover Suffix Tree

In this section we present two additional applications of the Cover Suffix Tree. We show that, given $CST(w)$ (or CST of a word that can be obtained from w in a simple manner), one can compute in linear time all distinct primitively rooted squares in w and a linear representation of all the seeds of w, in particular, the shortest seeds of w. This shows that constructing this data structure is *at least as hard* as computing all distinct primitively rooted squares and seeds. While there are linear-time algorithms for these problems [11,15,18,19], they are all complex and rely on the combinatorial properties specific to the repetitive structures they seek for.

Theorem 2 *Assume that the Cover Suffix Tree of a word of length n can be computed in $T(n)$ time. Then all distinct primitively rooted squares in a word w of length n can be computed in $T(2n)$ time.*

Proof Let $0 \notin \Sigma$ be a special symbol. Let $\varphi : \Sigma^* \to (\Sigma \cup \{0\})^*$ be a morphism such that $\varphi(c) = 0c$ for any $c \in \Sigma$. We consider the word $w' = \varphi(w)0$, that is, the

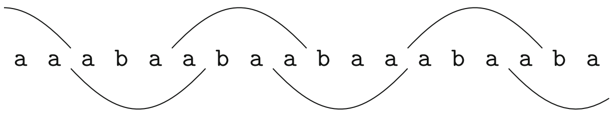

Fig. 7 Seed of string aaabaabaabaaabaaba

word w with 0-characters inserted at all its inter-positions, e.g. if $w =$ aabab then $w' =$ 0a0a0b0a0b0.

Let us consider the set of explicit non-branching nodes of $CST(w')$ and select among them the nodes corresponding to even-length factors of w' starting with the symbol 0. It suffices to note that there is a one-to-one correspondence between these nodes and the halves of distinct primitively rooted squares in w. □

Recall that a word u is a seed of w if u is a factor of w and w is a factor of some word y covered by u, see Fig. 7. The following lemma states that the set of all seeds of w has a representation of $\mathcal{O}(n)$ size, where $n = |w|$. This representation enables, e.g., simple computation of all shortest seeds of the word. By a range on a edge of a suffix tree we mean a number of consecutive nodes on this edge (obviously at most one of these nodes is explicit). Let w^R denote the reverse of the word w.

Lemma 6 ([17,18]) *The set of all seeds of w can be split into two disjoint classes. The seeds from one class form a single (possibly empty) range on each edge of the suffix tree of w, while the seeds from the other class form a range on each edge of the suffix tree of w^R.*

We will show that given $CST(w)$ and $CST(w^R)$ we can compute the representation of all seeds from Lemma 6 in $\mathcal{O}(n)$ time. Let us recall auxiliary notions of quasiseed and quasigap, see [18].

By *first*(u) and *last*(u) let us denote min $Occ(u)$ and max $Occ(u)$, respectively. We say that u is a *complete cover* in w if u is a cover of the word $w[\mathit{first}(u), \mathit{last}(u)+|u|-1]$. The word u is called a *quasiseed* of w if u is a complete cover in w, $\mathit{first}(u) < |u|$ and $n+1-\mathit{last}(u) < 2|u|$. Alternatively, w can be decomposed into $w = xyz$, where $|x|, |z| < |u|$ and u is a cover of y.

All quasiseeds of w lying on the same edge of the suffix tree with lower explicit endpoint v form a range with the lower explicit end of the range located at v. The length of the upper end of the range is denoted as $\mathsf{quasigap}(v)$. If the range is empty, we set $\mathsf{quasigap}(v) = \infty$. Thus a representation of all quasiseeds of a given word can be provided using only the quasigaps of explicit nodes in the suffix tree. It is known that computation of quasiseeds is the hardest part of an algorithm computing seeds:

Lemma 7 ([17,18]) *Assume quasigaps of all explicit nodes of suffix trees of w and w^R are known. Then a representation of all seeds of w from Lemma 6 can be found in $\mathcal{O}(n)$ time.*

It turns out that the auxiliary data in $CST(w)$ and $CST(w^R)$ enable constant-time computation of quasigaps of explicit nodes. By Lemma 7 this yields an $\mathcal{O}(n)$ time algorithm for computing a representation of all the seeds of w. This is stated formally in the following theorem.

Theorem 3 *Assume that the Cover Suffix Tree of a word of length n can be computed in $T(n)$ time. Given a word w of length n, one can compute a representation of all seeds of w from Lemma 6 in $T(n)$ time. In particular, all the shortest seeds of w can be computed within the same time complexity.*

Proof We show how to compute quasigaps for all explicit nodes of $\mathit{CST}(w)$. The computation for $\mathit{CST}(w^R)$ is symmetric. Note that $\mathit{CST}(w)$ may contain more explicit nodes that the suffix tree of the word. In this case, the results from any maximal sequence of edges connected by non-branching explicit nodes in $\mathit{CST}(w)$ need to be merged into a single range on the corresponding edge of the suffix tree.

By the definition of $\mathit{cv}(v)$, an explicit node v of $\mathit{CST}(w)$ is a complete cover in w if the following condition holds:

$$\mathit{cv}(v) = \mathit{last}(v) - \mathit{first}(v) + |v|.$$

Thus for checking whether an explicit node v of $\mathit{CST}(w)$ is a quasiseed of w it suffices to check whether this condition and the following equalities hold:

$$\mathit{first}(v) < |v|, \quad n + 1 - \mathit{last}(v) < 2|v|.$$

If v is not a quasiseed of w, we have $\mathsf{quasigap}(v) = \infty$, otherwise we can assume that $\mathsf{quasigap}(v) \leq |v|$.

Example 4 Consider the word w from Fig. 3, $n = 15$. The word `cacc` corresponds to an explicit node of $\mathit{CST}(w)$; we denote it by v. We have $\mathit{cv}(v) = 11$, $\mathit{first}(v) = 4$, $\mathit{last}(v) = 11$, and $\mathit{last}(v) - \mathit{first}(v) + |v| = 11$. Therefore `cacc` is a quasiseed of w, see also Fig. 2.

By Lemma 1, the condition for any node on the edge ending at v to be a complete cover in w is very simple:

$$\Delta(v) = 1.$$

Assume this condition is satisfied and consider any implicit node v' on this edge, $|v'| = k$. Then v' is a quasiseed if both inequalities:

$$\mathit{first}(v) < k \quad \text{and} \quad n + 1 - \mathit{last}(v) < 2k$$

are satisfied. Thus in this case

$$\mathsf{quasigap}(v) = \max(\mathit{first}(v) + 1,\ \lceil (n - \mathit{last}(v) + 2)/2 \rceil,\ |\mathit{parent}(v)| + 1).$$

Example 5 Consider the word w from Fig. 3. The word `cccacc` corresponds to an explicit node of $\mathit{CST}(w)$; we denote it by v. We have $\mathit{cv}(v) = 10$, $\mathit{first}(v) = 2$, $\mathit{last}(v) = 6$, and $\mathit{last}(v) - \mathit{first}(v) + |v| = 10$. Therefore `cccacc` is a quasiseed of w. Since $\Delta(v) = 1$, $\mathsf{quasigap}(v)$ could be smaller than 6. However, $\lceil (n - \mathit{last}(v) + 2)/2 \rceil = 6$ and the above formula yields $\mathsf{quasigap}(v) = 6$.

This concludes a complete set of rules for computing $\mathsf{quasigap}(v)$ for explicit nodes of $\mathit{CST}(w)$. $\square$

7 Conclusions

We have presented an algorithm which constructs a data structure, called the *Cover Suffix Tree*, in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space. The Cover Suffix Tree has been developed in order to solve the *PartialCovers* and *AllPartialCovers* problem in $\mathcal{O}(n)$ and $\mathcal{O}(n \log n)$ time, respectively, but it also gives a well-structured description of the cover indices of all factors. Consequently, various questions related to partial covers can be answered efficiently. For example, with the Cover Suffix Tree one can solve in linear time a problem inverse to *PartialCovers*: find a factor of length between l and r that maximizes the number of positions covered. Also a similar problem to *AllPartialCovers* problem, to compute for all lengths $l = 1, \dots, n$ the maximum number of positions covered by a factor of length l, can be solved in $\mathcal{O}(n \log n)$ time. This solution was actually given implicitly in the proof of Lemma 3.

An interesting open problem is to reduce the construction time to $\mathcal{O}(n)$. This could be difficult, though, since by the results of Sect. 6 this would yield alternative linear-time algorithms finding all distinct primitively rooted squares and computing seeds. The only known linear-time algorithms for these problems (see [11,15,18]) are rather complex.

Acknowledgments Tomasz Kociumaka is supported by Polish budget funds for science in 2013–2017 as a research project under the 'Diamond Grant' program. Jakub Radoszewski receives financial support of Foundation for Polish Science. Tomasz Waleń is supported by Iuventus Plus Grant (IP2011 058671) of the Polish Ministry of Science and Higher Education.

References

1. Apostolico, A., Ehrenfeucht, A.: Efficient detection of quasiperiodicities in strings. Theor. Comput. Sci. **119**(2), 247–265 (1993)
2. Apostolico, A., Preparata, F.P.: Data structures and algorithms for the string statistics problem. Algorithmica **15**(5), 481–494 (1996)
3. Apostolico, A., Farach, M., Iliopoulos, C.S.: Optimal superprimitivity testing for strings. Inf. Process. Lett. **39**(1), 17–20 (1991)
4. Breslauer, D.: An on-line string superprimitivity test. Inf. Process. Lett. **44**(6), 345–347 (1992)
5. Brodal, G.S., Pedersen, C.N.S.: Finding maximal quasiperiodicities in strings. In: Giancarlo, R., Sankoff, D. (eds.) Combinatorial Pattern Matching, 11th Annual Symposium, CPM 2000. Lecture Notes in Computer Science, vol. 1848, pp. 397–411. Springer, Berlin (2000)
6. Brodal, G.S., Lyngsø, R.B., Östlin, A., Pedersen, C.N.S.: Solving the string statistics problem in time $O(n \log n)$. In: Widmayer, P., Ruiz, F.T., Bueno, R.M., Hennessy, M., Eidenbenz, S., Conejo, R. (eds.) Automata, Languages and Programming, 29th International Colloquium, ICALP 2002. Lecture Notes in Computer Science, vol. 2380, pp. 728–739. Springer, Berlin (2002)
7. Brown, M.R., Tarjan, R.E.: A fast merging algorithm. J. ACM **26**(2), 211–226 (1979)
8. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific, Singapore (2003)
9. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press, Cambridge (2007)
10. Crochemore, M., Ilie, L., Rytter, W.: Repetitions in strings: algorithms and combinatorics. Theor. Comput. Sci. **410**(50), 5227–5235 (2009)

11. Crochemore, M., Iliopoulos, C.S., Kubica, M., Radoszewski, J., Rytter, W., Waleń, T.: Extracting powers and periods in a word from its runs structure. Theor. Comput. Sci. **521**, 29–41 (2014)
12. Farach, M.: Optimal suffix tree construction with large alphabets. In: 38th Annual Symposium on Foundations of Computer Science, FOCS ’97, pp. 137–143 (1997)
13. Flouri, T., Iliopoulos, C.S., Kociumaka, T., Pissis, S.P., Puglisi, S.J., Smyth, W., Tyczyński, W.: Enhanced string covering. Theor. Comput. Sci. **506**(0), 102–114 (2013)
14. Fraenkel, A.S., Simpson, J.: How many squares can a string contain? J. Comb. Theory Ser. A **82**(1), 112–120 (1998)
15. Gusfield, D., Stoye, J.: Linear time algorithms for finding and representing all the tandem repeats in a string. J. Comput. Syst. Sci. **69**(4), 525–546 (2004)
16. Hershberger, J.: Finding the upper envelope of n line segments in $O(n \log n)$ time. Inf. Process. Lett. **33**(4), 169–174 (1989)
17. Iliopoulos, C.S., Moore, D., Park, K.: Covering a string. Algorithmica **16**(3), 288–297 (1996)
18. Kociumaka, T., Kubica, M., Radoszewski, J., Rytter, W., Waleń, T.: A linear time algorithm for seeds computation. In: Rabani, Y. (ed.) Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, pp. 1095–1112. SIAM (2012)
19. Kolpakov, R.M., Kucherov, G.: Finding maximal repetitions in a word in linear time. In: 40th Annual Symposium on Foundations of Computer Science, FOCS ’99, pp. 596–604. IEEE Computer Society (1999)
20. Kucherov, G., Nekrich, Y., Starikovskaya, T.A.: Cross-document pattern matching. In: Kärkkäinen, J., Stoye, J. (eds.) Combinatorial Pattern Matching—23rd Annual Symposium, CPM 2012. Lecture Notes in Computer Science, vol. 7534, pp. 196–207. Springer, Berlin (2012)
21. Li, Y., Smyth, W.F.: Computing the cover array in linear time. Algorithmica **32**(1), 95–106 (2002)
22. Moore, D., Smyth, W.F.: An optimal algorithm to compute all the covers of a string. Inf. Process. Lett. **50**(5), 239–246 (1994)
23. Sim, J.S., Park, K., Kim, S., Lee, J.: Finding approximate covers of strings. J. Korea Inf. Sci. Soc. **29**(1), 16–21 (2002)
24. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14**(3), 249–260 (1995)