Fast Algorithm for Partial Covers in Words

Tomasz Kociumaka[1], Solon P. Pissis[4,5⋆], Jakub Radoszewski[1], Wojciech Rytter[1,2⋆⋆], and Tomasz Waleń[3,1]

[1] Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Warsaw, Poland
[kociumaka,jrad,rytter,walen]@mimuw.edu.pl
[2] Faculty of Mathematics and Computer Science, Copernicus University, Toruń, Poland
[3] Laboratory of Bioinformatics and Protein Engineering, International Institute of Molecular and Cell Biology in Warsaw, Poland
[4] Laboratory of Molecular Systematics and Evolutionary Genetics, Florida Museum of Natural History, University of Florida, USA
[5] Scientific Computing Group (Exelixis Lab & HPC Infrastructure), Heidelberg Institute for Theoretical Studies (HITS gGmbH), Germany
solon.pissis@h-its.org

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 *partial cover*, which can also be viewed as an *approximate* variant of cover, that is, a factor covering at least a given number of positions in w. Our main result is an $O(n \log n)$-time algorithm for computing the shortest partial covers of a word of length n.

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. 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 single periods 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 characterisation of periods.

⋆ Supported by the NSF–funded iPlant Collaborative (NSF grant #DBI-0735191).
⋆⋆ Supported by grant no. N206 566740 of the National Science Centre.

The most well-known formalization of quasiperiodicity is the cover of word. A factor u of length m of a word w of length n is said to be a *cover* of w if $m < n$, 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. [2], and a linear-time algorithm for computing all the covers of a word was proposed by Moore & Smyth [16]. 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 & Smyth [15] provided a linear-time algorithm for computing the *maximal cover array* of a word, and showed that, analogous to the border array [8], it actually determines the structure of *all* the covers of every prefix of the word.

Still it remains unlikely that an arbitrary word, even over the binary alphabet, has a cover; 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 and widely applicable 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. [12] suggested a related notion of *enhanced covers* which are additionally required to be borders of the word. Let $\mathit{Covered}(v, w)$ denote the number of positions in w covered by occurrences of the word v in w; for example, $\mathit{Covered}(\mathtt{aba}, \mathtt{aababab}) = 5$. We consider the following problem.

PartialCovers **problem**
Input: a word w and a positive integer $\alpha \leq |w|$.
Output: all shortest factors v such that $\mathit{Covered}(v, w) \geq \alpha$.

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

Our contribution. The following summarizes our main result.

Theorem 1. *The* PartialCovers *problem can be solved in* $O(n \log n)$ *time and* $O(n)$ *space, where where* $n = |w|$.

We extensively use suffix trees, for an exposition see [8, 10]. A suffix tree is a compact trie of suffixes, the nodes of the trie which become nodes of the suffix tree are called *explicit* nodes, while the other nodes are called *implicit*. Then each edge of the suffix tree can be viewed as an *upward*

maximal path of implicit nodes starting with an explicit node. A factor corresponds either to an explicit or to an implicit node, the *locus* of the factor. Our algorithm finds the loci of the shortest partial covers.

Informal structure of our algorithm. The algorithm first augments the suffix tree of w, and a linear number of implicit extra nodes become explicit. Then, for each node of the augmented tree, two integer values are computed. 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.

2 Augmented and annotated suffix trees

Let w be a word of length n over a totally ordered alphabet Σ. Then the suffix tree T of w can be constructed in $O(n \log |\Sigma|)$ time [11, 17]. 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 $|v| = |\hat{v}|$. The leaves of T play an auxiliary role and do not correspond to factors, instead they are labeled with the starting positions of the suffixes.

For a given word w, we define the *Cover Suffix Tree* of w, denoted by $CST(w)$. It is an *augmented* – new nodes are added – suffix tree in which the nodes are *annotated* with information relevant to covers. $CST(w)$ is similar to the data structure named *MAST* (see [3, 5]).

In $CST(w)$, we introduce additional explicit nodes called *extra nodes*, which correspond to halves of square factors in w, i.e. we make v explicit if $\hat{v}\hat{v}$ is a factor of w.

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

$$next_X(x) = \min\{y \in X, y > x\},$$

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

$$\delta(i, v) = next_{Occ(v)}(i) - i.$$

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

$$cv(v) = Covered(\hat{v}, w), \quad \Delta(v) = \big|\{i \in Occ(v) : \delta(i, v) \geq |v|\}\big|.$$

See, for example, Fig. 1. We call $cv(v)$ the *cover index* of v.

The $CST(w)$ is the suffix tree of w augmented with extra nodes and annotated with the values cv, Δ for all explicit nodes (including extra nodes); see, for example, Fig. 2.

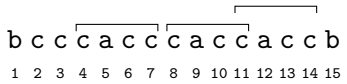

Fig. 1. Let $w =$ `bcccacccaccaccb` and let v be the node corresponding to `cacc`. We have $Occ(v) = \{4, 8, 11\}$, $cv(v) = 11$, $\Delta(v) = 2$.

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*

$$(\, cv(v_1),\, cv(v_2),\, cv(v_3),\, \ldots,\, cv(v_k)\,) \;=$$
$$(cv(v) - \Delta(v),\, cv(v) - 2\Delta(v),\, cv(v) - 3\Delta(v),\, \ldots, c(v) - k \cdot \Delta(v)).$$

Proof. Consider any v_i, $1 \le i \le k$. Note that $Occ(v_i) = Occ(v)$, 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 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 $\Delta(v)$. □

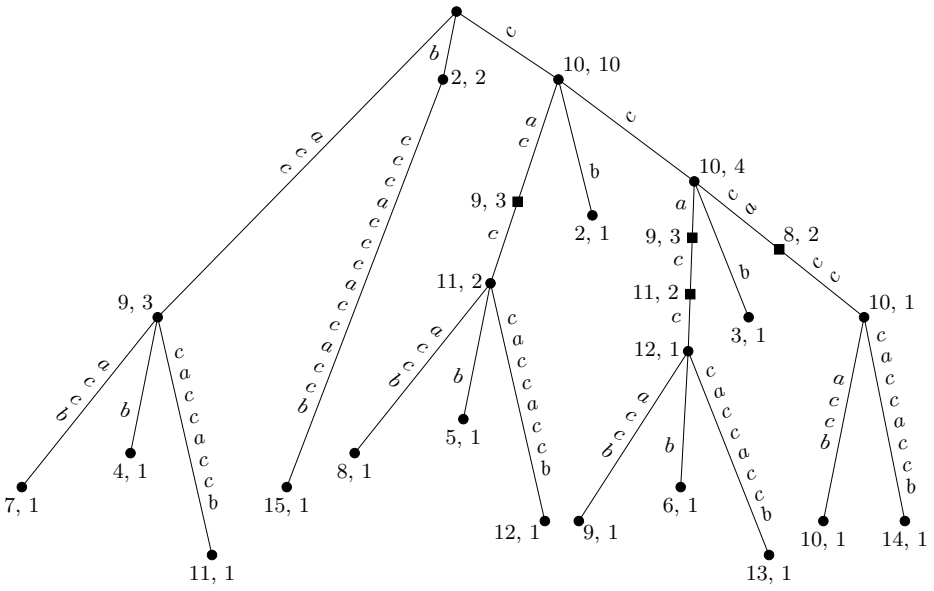

Fig. 2. $CST(w)$ for $w =$ `bcccacccaccaccb`. It contains four extra nodes that are denoted by squares in the figure. Each node is annotated with $cv(v), \Delta(v)$. Leaves are omitted for clarity.

Example 2. Consider the word w from Fig. 2. 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 and $cv(\texttt{cccac}) = 10-1 = 9$. 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 and $cv(\texttt{ccc}) = 8 - 2 = 6$.

As a consequence of Lemma 1 we obtain the following result.

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

(1) for any α, the loci of the shortest partial covers in linear time;
(2) given the locus of v in the suffix tree, $cv(v)$ in $O(1)$ 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 (Theorem 1) relies on the complexity of $CST(w)$ construction.

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

$$next_{\mathcal{P}}(x) = next_{P_i}(x) \text{ where } x \in P_i.$$

Now for two partitions $\mathcal{P}, \mathcal{P}'$ let us define the *change list* by

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

Example 3. For the partitions $\mathcal{P} = \{\{1,3,4\},\{2,5,6,7\},\{8,9\}\}$ and $\mathcal{P}' = \{\{1,\ldots,9\}\}$ we have: $ChangeList(\mathcal{P}, \mathcal{P}') = \{(1,2), (2,3), (4,5), (7,8)\}$, see also Fig. 3.

We say that $(\mathcal{P}, id)$ is a partition of U *labeled* by L if $\mathcal{P}$ is a partition of U and $id : \mathcal{P} \to L$ is a one-to-one (that is, injective) mapping. We say that $\ell \in L$ is a *valid* label if $id(P) = \ell$ for some $P \in \mathcal{P}$, the remaining labels are called invalid.

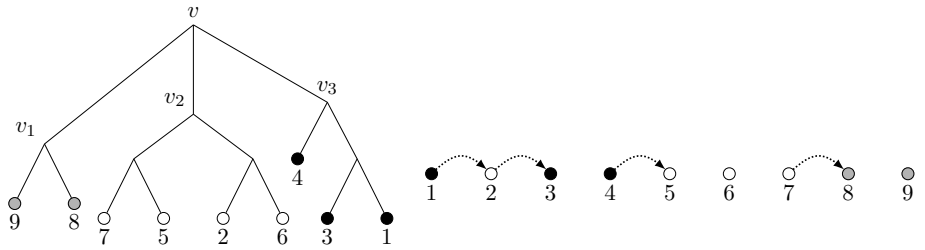

Fig. 3. Let $\mathcal{P}$ be the partition of $\{1,\ldots,9\}$ whose elements consist of leaves in the subtrees rooted at children of v and let $\mathcal{P}' = \{\{1,\ldots,9\}\}$. Then $\mathit{ChangeList}(\mathcal{P},\mathcal{P}') = \{(1,2),(2,3),(4,5),(7,8)\}$ (depicted by dotted arrows).

Lemma 3. *Let $n \leq k$ be positive integers such that $k = O(n)$. There exists a data structure of size $O(n)$, which maintains a partition $(\mathcal{P}, id)$ of $\{1,\ldots,n\}$ labeled by $L = \{1,\ldots,k\}$. Initially $\mathcal{P}$ is a partition into singletons with $id(\{x\}) = x$. The data structure supports the following operations:*

- *Find(x) for $x \in \{1,\ldots,n\}$ gives a label of $P \in \mathcal{P}$ containing x.*
- *Union(I,ℓ) for a set I of valid labels and a new 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.*

Any sequence of m operations is performed in $O(n \log n + m)$ total time and $O(n)$ space.

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 proof of Lemma 3 can be found in the Appendix; it is based on an approach presented in [6].

4 $O(n \log n)$-time construction of $CST(w)$

The suffix tree of w augmented with extra nodes is called *skeleton* of $CST(w)$, which we denote by $sCST(w)$. The following lemma follows from the fact that all square factors can be computed in linear time [13, 9], and the nodes corresponding to them (a linear number) can be inserted into the suffix tree easily in $O(n \log n)$ time.

Lemma 4. *$sCST(w)$ can be constructed in $O(n \log n)$ time.*

We introduce auxiliary notions related to covered area of nodes:

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

Observation 1 $cv(v) = cv_{|v|}(v) + \Delta_{|v|}(v) \cdot |v|$, $\Delta(v) = \Delta_{|v|}(v)$.

In the course of the algorithm some nodes will have their values c, Δ 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 nonincreasing height $|v|$. 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; see, for example, Fig. 4. The sets in the partition are labeled with the labels of the corresponding peak nodes. Recall that leaves are labeled with the starting positions of the corresponding suffixes. The remaining nodes may be labeled arbitrarily as long as the labels of all nodes are distinct positive integers of magnitude $O(n)$. We maintain the following technical invariant.

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 $List[d] = \{i : Dist[i] = d\}$.

Algorithm ComputeCST*(w)*

$T := sCST(w)$;
$\mathcal{P} :=$ partition of $\{1, \ldots, n\}$ into singletons;
$id(\{i\}) = i$, which we identify with leaf of T containing i;
foreach v : *a leaf of* T **do** $cv'[v] := 0$; $\Delta'[v] := 1$;
$h := n + 1$;
foreach v : *an inner node of* T*, in nonincreasing order of* $|v|$ **do**
  $Lift(h, |v|)$; $h := |v|$;
  {Now part (A) of Invariant(h) is satisfied}
  $cv'[v] := \sum_{u \in children(v)} cv'[u]$;
  $\Delta'[v] := \sum_{u \in children(v)} \Delta'[u]$;
  $ChangeList(v) := Union(children(v), v)$
  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, Δ*;*

In the algorithm, h is the smallest height (the smallest value of $|z|$) among the current set of peak nodes z; the height is not defined for leaves, so we start with $h = n + 1$.

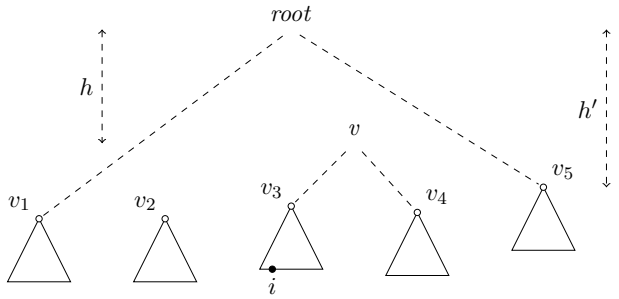

Fig. 4. One stage of the algorithm, where the peak nodes are $v_1, \ldots, v_5$ while the currently processed node is v. If $i \in \mathit{List}[d]$ and $v_3 = \mathit{Find}(i)$, then $d = \delta(i, v_3) = \mathit{Dist}[i]$. The current partition is $\mathcal{P} = \{\mathit{Leaves}(v_1), \mathit{Leaves}(v_2), \mathit{Leaves}(v_3), \mathit{Leaves}(v_4), \mathit{Leaves}(v_5)\}$. After v is processed, the partition changes to $\mathcal{P} = \{\mathit{Leaves}(v_1), \mathit{Leaves}(v_2), \mathit{Leaves}(v), \mathit{Leaves}(v_5)\}$. The *Union* operation merges $\mathit{Leaves}(v_4), \mathit{Leaves}(v_3)$ and returns the corresponding change list.

Description of the $\mathit{Lift}(h_{old}, h_{new})$ operation. The procedure *Lift* is of auxiliary nature but plays an important preparatory role in processing the current node. According to point **(A)** of our invariant, for all peak nodes z we know the values: $cv'[z] = cv_{h_{old}}(z)$, $\Delta'[z] = \Delta_{h_{old}}(z)$. Now we have to change h_{old} to h_{new} and guarantee validity of the invariant: $cv'[z] = cv_{h_{new}}(z)$, $\Delta'[z] = \Delta_{h_{new}}(z)$. This is exactly what the following operation does.

```
Function Lift(h_old, h_new)
    for h := h_old − 1 downto h_new do
        foreach i in List[h] do
            v := Find(i);
            Δ′[v] := Δ′[v] + 1; cv′[v] := cv′[v] − h;
```

Description of the $\mathit{LocalCorrect}(p, q, v)$ operation. Here we assume that 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)
$d := q - p$; $d' := Dist[p]$;
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** $\Delta'[v] := \Delta'[v] + 1$;
$Dist[p] := d$;
$remove(i, List[d'])$; $insert(i, List[d])$;

Complexity of the algorithm. In the course of the algorithm we compute $ChangeList(v)$ for each $v \in T$. Due to Lemma 3 we have:

$$\sum_{v \in T} |ChangeList(v)| \;=\; O(n \log n).$$

Consequently we perform $O(n \log n)$ operations *LocalCorrect*. In each of them at most one element is added to a list $List[d]$ for some d. Hence the total number of insertions to these lists is also $O(n \log n)$.

The cost of each operation *Lift* is proportional to the number of elements removed from $List[d]$ for some d. As each list is processed at most once and the total number of insertions into lists is $O(n \log n)$, the total cost of all operations *Lift* is also $O(n \log n)$. This proves the following fact which, together with Lemma 3, implies our main result (Theorem 1).

Lemma 5. *Algorithm* ComputeCST *computes* $CST(w)$ *in* $O(n \log n)$ *time and* $O(n)$ *space, where* $n = |w|$.

5 Final remarks

We have presented an algorithm which constructs a useful data structure, called the *Cover Suffix Tree*, in $O(n \log n)$ time and $O(n)$ space. Then several queries related to partial covers can be answered efficiently. Our construction is based on extended approaches used in [3, 5] to solve a similar problem related to statistics of occurrences of factors.

Having our data structure one can solve in linear time also a symmetric problem: given constraints on factors of w (e.g. on their length), find a factor that maximizes the number of positions covered.

In our algorithm, to simplify its presentation, we used all halves of square factors as extra nodes. However, it suffices to insert primitive square halves only and all such nodes can be shown to be necessary for Lemma 1 to hold. As such, they can be introduced on the fly (in the *Lift* operation) without using the algorithms of [13, 9].

An interesting open problem is to reduce the complexity of our construction to $O(n)$. This could be difficult, though, since this would yield alternative linear-time algorithms finding primitively rooted squares and computing seeds (for a definition see [14]); and the only known linear-time algorithms for these problems are rather complex.

References

1. A. Apostolico and A. Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theor. Comput. Sci.*, 119(2):247–265, 1993.
2. A. Apostolico, M. Farach, and C. S. Iliopoulos. Optimal superprimitivity testing for strings. *Inf. Process. Lett.*, 39(1):17–20, 1991.
3. A. Apostolico and F. P. Preparata. Data structures and algorithms for the string statistics problem. *Algorithmica*, 15(5):481–494, 1996.
4. D. Breslauer. An on-line string superprimitivity test. *Inf. Process. Lett.*, 44(6):345–347, 1992.
5. G. S. Brodal, R. B. Lyngsø, A. Östlin, and C. N. S. Pedersen. Solving the string statistics problem in time $O(n \log n)$. In P. Widmayer, F. T. Ruiz, R. M. Bueno, M. Hennessy, S. Eidenbenz, and R. Conejo, editors, *ICALP*, volume 2380 of *Lecture Notes in Computer Science*, pages 728–739. Springer, 2002.
6. G. S. Brodal and C. N. S. Pedersen. Finding maximal quasiperiodicities in strings. In R. Giancarlo and D. Sankoff, editors, *CPM*, volume 1848 of *Lecture Notes in Computer Science*, pages 397–411. Springer, 2000.
7. M. R. Brown and R. E. Tarjan. A fast merging algorithm. *J. ACM*, 26(2):211–226, 1979.
8. M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.
9. M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, and T. Walen. Extracting powers and periods in a string from its runs structure. In E. Chávez and S. Lonardi, editors, *SPIRE*, volume 6393 of *Lecture Notes in Computer Science*, pages 258–269. Springer, 2010.
10. M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific, 2003.
11. M. Farach. Optimal suffix tree construction with large alphabets. In *FOCS*, pages 137–143, 1997.
12. T. Flouri, C. S. Iliopoulos, T. Kociumaka, S. P. Pissis, S. J. Puglisi, W. F. Smyth, and W. Tyczyński. New and efficient approaches to the quasiperiodic characterisation of a string. In J. Holub and J. Žďárek, editors, *PSC*, pages 75–88, Czech Technical University in Prague, Czech Republic, 2012.
13. D. Gusfield and J. Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *J. Comput. Syst. Sci.*, 69(4):525–546, 2004.
14. T. Kociumaka, M. Kubica, J. Radoszewski, W. Rytter, and T. Walen. A linear time algorithm for seeds computation. In Y. Rabani, editor, *SODA*, pages 1095–1112. SIAM, 2012.
15. Y. Li and W. F. Smyth. Computing the cover array in linear time. *Algorithmica*, 32(1):95–106, 2002.
16. D. Moore and W. F. Smyth. An optimal algorithm to compute all the covers of a string. *Inf. Process. Lett.*, 50(5):239–246, 1994.
17. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

Appendix: Proof of Lemma 3

We use an approach similar to Brodal and Pedersen [6] (who use the results of [7]) originally devised for computation of maximal quasiperiodicities.

Theorem 3 of [6] states that a subset X of a linearly ordered universe can be stored in a height-balanced tree supporting the following operations in $O\left(|Y| \max\left(1, \log \frac{|X|}{|Y|}\right)\right)$ time with linear space:

$X.\mathit{MultiInsert}(Y)$**:** insert all elements of Y to X,
$X.\mathit{MultiPred}(Y)$**:** for each $y \in Y$ compute $\max\{x \in X, x < y\}$,
$X.\mathit{MultiSucc}(Y)$**:** for each $y \in Y$ compute $\min\{x \in X, x > y\}$.

In the data structure we will store each $P \in \mathcal{P}$ as a height-balanced tree. For each $x \in \{1, \ldots, n\}$ we will store a value $\mathit{next}[x]$ (which eventually becomes $\mathit{next}_{\mathcal{P}}(x)$) and a pointer $\mathit{tree}[x]$ to the tree representing P such that $x \in P$. Moreover, for each $P \in \mathcal{P}$ we store $\mathit{id}[P]$ and for each $\ell \in L$ we store $\mathit{id}^{-1}[\ell]$, a pointer to the corresponding tree (or null if none).

Answering *Find* is trivial as it suffices to use *tree* pointer and the *id* value. The *Union* operation is performed as follows (we use P_i to denote the tree $\mathit{id}^{-1}[i]$):

```
Function Union(I, ℓ)
    let i_0 be the index of the largest set in {|P_i| : i ∈ I} and S = P_{i_0};
    foreach i ∈ I \ {i_0} do S.MultiInsert(P_i);
    Update the tree pointers;
    C := ∅;
    foreach i ∈ I \ {i_0} do
        foreach (y, x) ∈ S.MultiPred(P_i) do
            if next[x] ≠ y then C := C ∪ {(x, y)};
        foreach (y, x) ∈ S.MultiSucc(P_i) do
            if next[y] ≠ x then C := C ∪ {(y, x)};
    foreach (x, y) ∈ C do next[x] := y;
    id[S] := ℓ;
    id^{-1}[ℓ] := S;
    return C;
```

The correctness of this procedure follows from the fact that if (x, y) is in the change list, then x and y come from different sets, in particular at least one of them does not come from the largest set.

Let us analyze the complexity of the algorithm. We need to show that a sequence of valid *Union* operations is performed in $O(n \log n)$ time. First, observe that a single *Union* operation with $|I| > 2$ is not slower than $|I|-1$ union operations merging P_{i_0} with P_j for consecutive $j \in I \setminus \{i_0\}$. Thus, it suffices to prove that any sequence of (valid) *binary Union* operations can be performed in $O(n \log n)$ time.

Notice that such a sequence can be represented by a full binary tree with n leaves corresponding to elements of the universe and inner nodes corresponding to *Union* operations. The time complexity of a single *Union* operation can be bounded by $O\left(n_2 \max\left(1, \log \frac{n_1}{n_2}\right)\right)$, where $n_1 \geq n_2$ are sizes of the corresponding subtrees, this is due to the complexity of *Multi* operations.

Lemma 6 of [6] states that if we have a full binary tree of size n and if each inner node v supplies a term $n_2(v) \max\left(1, \log \frac{n_1(v)}{n_2(v)}\right)$, where $n_1(v) \geq n_2(v)$ are sizes of the subtrees rooted in children of v, then the sum over all terms is $O(n \log n)$. (This is a stronger version of the 'smaller-half trick'). Consequently the total time complexity of the algorithm is $O(n \log n)$.