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 be viewed as a relaxed 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 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.

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

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 [3]. 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 [5] provided a linear-time algorithm for computing the *maximal cover array* of a word, and showed that, analogous to the border array [6], 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 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. [7] 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 [8]. 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. The significant advantage of partial covers is that they enjoy a more combinatorial structure, and consequently the algorithms solving the most natural problems are much more efficient than those concerning approximate covers, where the time complexity rarely drops below quadratic and some problems are even NP-hard.

Let $\mathit{Covered}(v, w)$ denote the number of positions in w covered by occurrences of the word v in w; we call this value the *cover index* of v within w. For example, $\mathit{Covered}(\mathtt{aba}, \mathtt{aababab}) = 5$. We primarily concentrate on the following problem, but the tools we develop can be used to answer various questions concerning partial covers.

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 = `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* $n = |w|$.

We extensively use suffix trees, for an exposition see [6, 9]. 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*. 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. Such a representation of the unique node in the trie corresponding to a factor is called the *locus* of that factor. Our algorithm finds the loci of the shortest partial covers.

Informal Structure of the 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 [10, 11]. 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.

We define the *Cover Suffix Tree* of w, denoted by $CST(w)$, as 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 [12, 13]).

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 $\hat{v}$. 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.

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. Moreover we annotate all explicit nodes (including extra nodes) with the values cv, Δ; see, for example, Fig. 2. The number of extra nodes is linear [14], so $CST(w)$ takes $O(n)$ space.

b c c c a c c c a c c a c c b

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

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 $\Delta(v)$. □

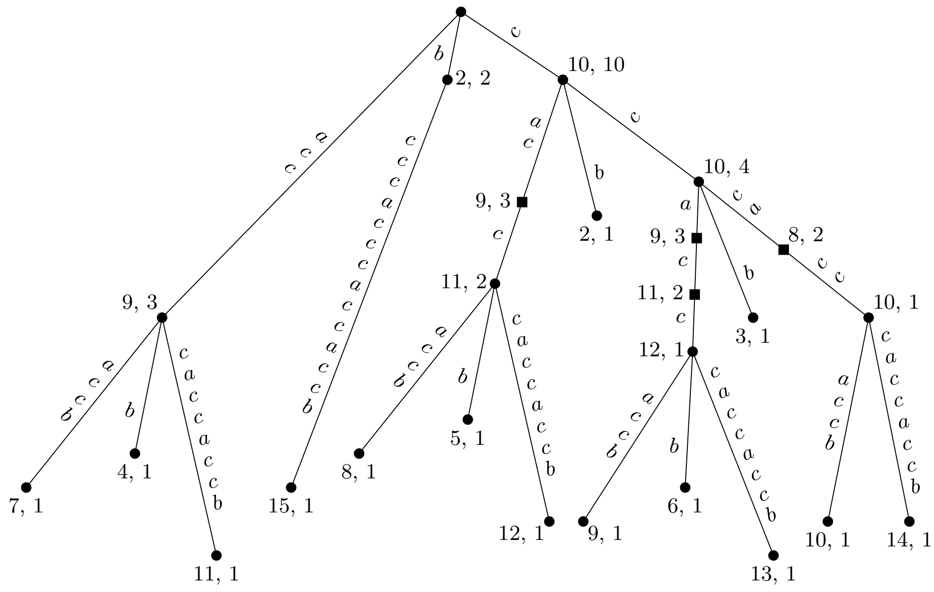

Fig. 2. $CST(w)$ for w = bcccacccaccaccb. It contains four extra nodes that are denoted by squares in the figure. Each node is annotated with $cv(v), \Delta(v)$. Leaves are omitted for clarity.

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 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 $cv(v''') = 8 - 2 = 6$.

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 a factor u in the suffix tree $CST(w)$, the cover index $Covered(u, w)$ 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* (see also Fig. 3) by

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

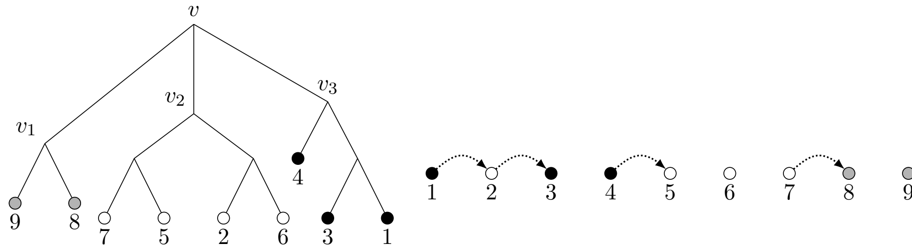

Fig. 3. Let $\mathcal{P}$ be the partition of $\{1, \ldots, 9\}$ whose classes consist of leaves in the subtrees rooted at children of v, $\mathcal{P} = \{\{1, 3, 4\}, \{2, 5, 6, 7\}, \{8, 9\}\}$, and let $\mathcal{P}' = \{\{1, \ldots, 9\}\}$. Then $ChangeList(\mathcal{P}, \mathcal{P}') = \{(1, 2), (2,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

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 (injective) mapping. A label $\ell \in L$ is called *valid* if $id(P) = \ell$ for some $P \in \mathcal{P}$ and *free* otherwise.

Lemma 3. *Let $n \le k$ be positive integers such that k is of magnitude $\Theta(n)$. 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 the label of $P \in \mathcal{P}$ containing x.*
- *$Union(I, \ell)$ for a set I of valid labels 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.*

Any valid sequence of Union operations is performed in $O(n \log n)$ time and $O(n)$ space in total. A single Find operation takes $O(1)$ time.

Proof. Note that these are actually standard disjoint-set data structure operations except for the fact that we require *Union* to return the change list.

We use an approach similar to Brodal and Pedersen [15] (who use the results of [16]) originally devised for computation of maximal quasiperiodicities.

Theorem 3 of [15] 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:

$X.MultiInsert(Y)$**:** insert all elements of Y to X,
$X.MultiPred(Y)$**:** return all (y, x) for $y \in Y$ and $x = \max\{z \in X, z < y\}$,
$X.MultiSucc(Y)$**:** 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.

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 $x \in \{1,\ldots,n\}$ we maintain a value $next[x] = next_{\mathcal{P}}(x)$ and a pointer $tree[x]$ 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 $id[P]$ and for each $\ell \in L$ we store $id^{-1}[\ell]$, 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 perfomed according to the pseudocode given below (for brevity we write P_i instead of $id^{-1}[i]$).

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

Proof. 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. On the other hand, 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. □

```
Function Union(I, ℓ)
    i_0 := argmax{|P_i| : i ∈ I}; S := P_{i_0};
    foreach i ∈ I \ {i_0} do
        foreach x ∈ P_i do tree[x] := S;
        ;
        S.MultiInsert(P_i);
    C := ∅;
    foreach i ∈ I \ {i_0} do
        foreach (b, a) ∈ S.MultiPred(P_i) do
            if next[a] ≠ b then C := C ∪ {(a, b)};
            ;
        foreach (a, b) ∈ S.MultiSucc(P_i) do
            if next[a] ≠ b then C := C ∪ {(a, b)};
            ;
        id^{-1}[i] := null;
    id[S] := ℓ; id^{-1}[ℓ] := S;
    foreach (x, y) ∈ C do next[x] := y;
    ;
    return C;
```

Claim. Any sequence of *Union* operations takes $O(n \log n)$ 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

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

so this suffices to obtain a desired $O(n \log n)$ bound.

Let us consider a *Union* operation that merges partition classes of sizes $p_1 \geq p_2 \geq \ldots \geq p_k$ 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. $\log \frac{p}{p_i} \geq 1$). This lets us bound the time complexity of the *Union* operations as follows:

$$\sum_{i=2}^{k} \max\left(p_i, p_i \log \tfrac{p}{p_i}\right) = \sum_{i=2}^{k} p_i \log \tfrac{p}{p_i} \leq \sum_{i=1}^{k} p_i \log \tfrac{p}{p_i} =$$

$$\sum_{i=1}^{k} p_i (\log p - \log p_i) = p \log p - \sum_{i=1}^{k} p_i \log p_i,$$

which is equal to the potential growth. □

This conclues the proof of Lemma 3. □

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

The suffix tree of w augmented with extra nodes is called the *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 [17, 18], 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:

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

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 non-increasing height $|v|$. We maintain the partition $\mathcal{P}$ of $\{1, \dots, n\}$ 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 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 $O(n)$. 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 $i \in \{1, \dots, n\}$ we store $Dist[i] = \delta(i,\ Find(i))$.
(C) For each $d < h$ we store $List[d] = \{i \ : \ Dist[i] = d\}$.

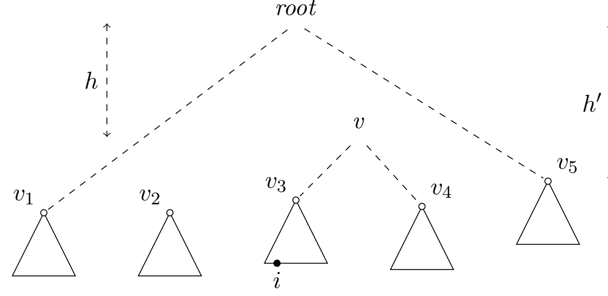

Fig. 4. One stage of the algorithm, where the peak nodes are $v_1, \ldots, v_5$ while the currently processed node is v. If $i \in List[d]$ and $v_3 = Find(i)$, then $d = \delta(i, v_3) = Dist[i]$. The current partition is $\mathcal{P} = \{Leaves(v_1), Leaves(v_2), Leaves(v_3), Leaves(v_4), Leaves(v_5)\}$.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

Algorithm ComputeCST*(w)*
  $T := sCST(w)$;
  $\mathcal{P}$:= partition of $\{1, \ldots, n\}$ into singletons;
  foreach v : *a leaf of* T **do** $cv'[v] := 0$; $\Delta'[v] := 1$;
  ;
  $h := n + 1$;
  foreach v : *an inner node of* T*, in non-increasing 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 $h = n + 1$.

Description of the $Lift(h_{old}, h_{new})$ **Operation.** The procedure *Lift* is of auxiliary nature but plays an important preparatory role in processing the current node. According to part (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 $LocalCorrect(p, q, v)$ **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. $(p, q) \in ChangeList(v)$. The *LocalCorrect* procedure updates $Dist[p]$ 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  Δ'[v] := Δ'[v] − 1;
    ;
    if d < |v| then  cv'[v] := cv'[v] + d ;
    else  Δ'[v] := Δ'[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 total size of lists $List[h]$ processed in this operation. As for each h the list $List[h]$ is processed 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 data structure, called the *Cover Suffix Tree*, in $O(n \log n)$ time and $O(n)$ space. In the algorithm, to simplify its presentation, we used all halves of square factors as extra nodes. However, it suffices to consider 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 [17, 18].

The Cover Suffix Tree has been developed in order to solve the PartialCovers problem, but it gives a well-structured description of the cover indices of all factors. Consequently, various queries related to partial covers can be answered efficiently. For example, with the Cover Suffix Tree one can solve in linear time a problem symmetric to PartialCovers: given constraints on factors of w (e.g. on their length), find a factor that maximizes the number of positions covered.

An interesting open problem is to reduce the construction time 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 [19]); and the only known linear-time algorithms for these problems are rather complex.

References

1. Apostolico, A., Ehrenfeucht, A.: Efficient detection of quasiperiodicities in strings. Theor. Comput. Sci. **119**(2) (1993) 247–265
2. Apostolico, A., Farach, M., Iliopoulos, C.S.: Optimal superprimitivity testing for strings. Inf. Process. Lett. **39**(1) (1991) 17–20
3. Moore, D., Smyth, W.F.: An optimal algorithm to compute all the covers of a string. Inf. Process. Lett. **50**(5) (1994) 239–246
4. Breslauer, D.: An on-line string superprimitivity test. Inf. Process. Lett. **44**(6) (1992) 345–347
5. Li, Y., Smyth, W.F.: Computing the cover array in linear time. Algorithmica **32**(1) (2002) 95–106
6. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press (2007)
7. Flouri, T., Iliopoulos, C.S., Kociumaka, T., Pissis, S.P., Puglisi, S.J., Smyth, W.F., Tyczyński, W.: New and efficient approaches to the quasiperiodic characterisation of a string. In Holub, J., Žďárek, J., eds.: PSC, Czech Technical University in Prague, Czech Republic (2012) 75–88
8. Sim, J.S., Park, K., Kim, S., Lee, J.: Finding approximate covers of strings. Journal of Korea Information Science Society **29**(1) (2002) 16–21
9. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific (2003)
10. Farach, M.: Optimal suffix tree construction with large alphabets. In: FOCS. (1997) 137–143
11. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14**(3) (1995) 249–260
12. Apostolico, A., Preparata, F.P.: Data structures and algorithms for the string statistics problem. Algorithmica **15**(5) (1996) 481–494

13. 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.: ICALP. Volume 2380 of Lecture Notes in Computer Science., Springer (2002) 728–739
14. Fraenkel, A.S., Simpson, J.: How many squares can a string contain? J. Comb. Theory, Ser. A **82**(1) (1998) 112–120
15. Brodal, G.S., Pedersen, C.N.S.: Finding maximal quasiperiodicities in strings. In Giancarlo, R., Sankoff, D., eds.: CPM. Volume 1848 of Lecture Notes in Computer Science., Springer (2000) 397–411
16. Brown, M.R., Tarjan, R.E.: A fast merging algorithm. J. ACM **26**(2) (1979) 211–226
17. Gusfield, D., Stoye, J.: Linear time algorithms for finding and representing all the tandem repeats in a string. J. Comput. Syst. Sci. **69**(4) (2004) 525–546
18. Crochemore, M., Iliopoulos, C.S., Kubica, M., Radoszewski, J., Rytter, W., Waleń, T.: Extracting powers and periods in a string from its runs structure. In Chávez, E., Lonardi, S., eds.: SPIRE. Volume 6393 of Lecture Notes in Computer Science., Springer (2010) 258–269
19. Kociumaka, T., Kubica, M., Radoszewski, J., Rytter, W., Waleń, T.: A linear time algorithm for seeds computation. In Rabani, Y., ed.: SODA, SIAM (2012) 1095–1112