Order-Preserving Incomplete Suffix Trees and Order-Preserving Indexes

Maxime Crochemore[4,6], Costas S. Iliopoulos[4,5], Tomasz Kociumaka[1], Marcin Kubica[1], Alessio Langiu[4], Solon P. Pissis[6,7⋆], Jakub Radoszewski[1], Wojciech Rytter[1,3⋆⋆], and Tomasz Waleń[2,1]

[1] Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Warsaw, Poland
[kociumaka,jrad,rytter,walen]@mimuw.edu.pl
[2] Laboratory of Bioinformatics and Protein Engineering, International Institute of Molecular and Cell Biology in Warsaw, Poland
[3] Faculty of Mathematics and Computer Science, Copernicus University, Toruń, Poland
[4] Dept. of Informatics, King's College London, London WC2R 2LS, UK
[maxime.crochemore,csi]@dcs.kcl.ac.uk
[5] Faculty of Engineering, Computing and Mathematics, University of Western Australia, Perth WA 6009, Australia
[6] Université Paris-Est, France
[7] Laboratory of Molecular Systematics and Evolutionary Genetics, Florida Museum of Natural History, University of Florida, USA
[8] Scientific Computing Group (Exelixis Lab & HPC Infrastructure), Heidelberg Institute for Theoretical Studies (HITS gGmbH), Germany
solon.pissis@h-its.org

Abstract. Recently Kubica et al. (*Inf. Process. Let.*, 2013) and Kim et al. (*submitted to Theor. Comp. Sci.*) introduced order-preserving pattern matching: for a given text the goal is to find its factors having the same "shape" as a given pattern. The known results include a linear-time algorithm for this problem (in case of polynomially-bounded alphabet) and a generalization to multiple patterns. We extend these results and give an $O(n \log\log n)$ time construction of an index that enables order-preserving pattern matching queries in time linear with respect to the length of the pattern. The main novel component is a data structure being an incomplete suffix tree in the order-preserving setting. The tree can miss single letters related to branchings at internal nodes. Such incompleteness results from the weakness of our so called *weak character oracle*. However, due to its weakness, such oracle can be computed in $O(\log\log n)$ time on-line using a sliding-window approach. For most of the applications such incomplete suffix-trees provide the same functional power as the complete ones. We also give an $O(\frac{n \log n}{\log\log n})$ time algorithm constructing complete order-preserving suffix trees.

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

1 Introduction

We introduce order-preserving suffix trees that can be applied for pattern matching and repetition discovery problems in the order-preserving setting. In particular, this setting can be used to model finding trends in time series which appear naturally when considering e.g. the stock market or melody matching of two musical scores, see [11].

Two strings x and y of the same length over an integer alphabet are called *order-isomorphic* (or simply isomorphic), written $x \approx y$, if

$$\forall_{1 \le i,j \le |x|}\ x_i \le x_j \Leftrightarrow y_i \le y_j.$$

Example 1. $(5, 2, 7, 5, 1, 4, 9, 4, 5) \approx (6, 4, 7, 6, 3, 5, 8, 5, 6)$, see Fig. 1.

The notion of order-isomorphism was introduced in [11] and [14]. Both papers independently study the *order-preserving pattern matching problem* that consists in identifying all consecutive factors of a string x that are order-isomorphic to a given string y. If $|x| = n$ and $|y| = m$, an $O(n + m \log m)$ time algorithm for this problem is presented in both papers. Under a natural assumption that the characters of y can be sorted in linear time, the algorithm can be implemented in $O(n+m)$ time. Moreover, in [11] the authors present extensions of this problem to multiple-pattern matching based on the algorithm of Aho–Corasick.

The problem of order-preserving pattern matching has evolved from the combinatorial study of patterns in permutations. This field of study is concentrated on pattern avoidance, that is, counting the number of permutations not containing a subsequence which is order-isomorphic to a given pattern. Note that in this problem the subsequences need not to be consecutive. The first results on this topic were given by Knuth [12] (avoidance of 312), Lovász [16] (avoidance of 213) and Rotem [17] (avoidance of both 231 and 312). On the algorithmic side, pattern matching in permutations (as a subsequence) was shown to be NP-complete [3] and a number of polynomial-time algorithms for special cases of patterns were developed [1, 9, 10].

We introduce an index for order-preserving pattern matching. The preprocessing time is $O(n \log \log n)$ and queries are answered in $O(m)$ time for a pattern of length m over polynomially bounded integer alphabet Σ. The index is based on incomplete order-preserving suffix trees (incomplete op-suffix-trees, in short). We also introduce (complete) order-preserving suffix trees (op-suffix-trees) and show how they can be constructed using their incomplete counterpart in $O(n \log n / \log \log n)$ time.

In the literature there are a number of results in the related field of indexing for parameterized pattern matching. This problem is solved using parameterized suffix trees, a notion first introduced by Baker [2] who proposed an $O(n \log n)$ time construction algorithm. The result was then improved by Cole and Hariharan [5] to $O(n)$ construction time. Recently, Lee et al. [15] presented an online algorithm with the same time complexity. What Cole and Hariharan [5] proposed was actually a general scheme for construction of suffix trees for so-called quasi-suffix families with a constant time character oracle. This result can also

be applied in the order-preserving setting, however the resulting index has higher complexity than ours. We further comment on this in the Final Remarks section.

Structure of the paper. In Sections 2 (preliminary notation) and 3 we give a formal definition of a complete and an incomplete op-suffix-tree and describe their basic properties. Then in Sections 4 and 5 we show an $O(n \log\log n)$ construction of an incomplete op-suffix-tree. The former section contains an algorithmic toolbox that is also used in further parts of the paper. Applications of our data structure for order-preserving pattern matching and longest common factor problems are presented in Section 6. Finally in Section 7 we obtain a construction of complete op-suffix-trees and give some final remarks in Section 8.

2 Order-Preserving Code

Let $w = w_1 \dots w_n$ be a string of length n over an integer alphabet Σ. We assume that Σ is polynomially bounded in terms of n, i.e. $\Sigma = \{1, \dots, n^c\}$ for an integer constant c. We denote the length of a string w by $|w| = n$. By $w[i\,..\,j]$ we denote the factor $w_i \,..\, w_j$. Denote by suf_i the i-th suffix of w, that is, $w[i\,..\,n]$.
For any $i \in \{1, \dots, n\}$ define:

$$\alpha_w(i) = i - j \quad \text{if} \quad w_j = \max\{w_k \ : \ k < i,\ w_k \le w_i\},$$

if there is no such j then $\alpha_w(i) = i$, similarly define:

$$\beta_w(i) = i - j \quad \text{if} \quad w_j = \min\{w_k \ : \ k < i,\ w_k \ge w_i\},$$

and $\beta_w(i) = i$ if no such j exists. If several equally good values of j exist, we select the greatest good value of j.

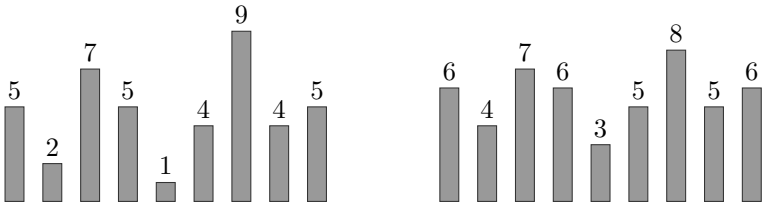

Fig. 1. Example of two order-isomorphic strings. Their codes are equal to $(1,1)$ $(2,1)$ $(2,3)$ $(3,3)$ $(5,3)$ $(4,2)$ $(4,7)$ $(2,2)$ $(5,5)$.

We introduce codes of strings in a similar way as in [14]:

$$Code(w) = ((\alpha_w(1), \beta_w(1)),\ (\alpha_w(2), \beta_w(2)),\ \dots,\ (\alpha_w(|w|), \beta_w(|w|))).$$

We also denote $LastCode(w) = (\alpha_w(|w|), \beta_w(|w|))$. The following property is a consequence of Lemma 2 in [14].

Lemma 1. *Let x and y be two strings of length t and $x' = x[1\,..\,t-1]$, $y' = y[1\,..\,t-1]$. Then:*

(a) $x \approx y \;\Leftrightarrow\; x' \approx y' \;\wedge\; (y_i \le y_t \le y_j)$,
(b) $x \approx y \;\Leftrightarrow\; x' \approx y' \;\wedge\; \mathit{LastCode}(x) = \mathit{LastCode}(y)$,

where $i = t - \alpha_x(t)$, $j = t - \beta_x(t)$.

Proof. Part (a) is an equivalent formulation of Lemma 2 in [14]. Part (b) is a technical consequence of part (a). □

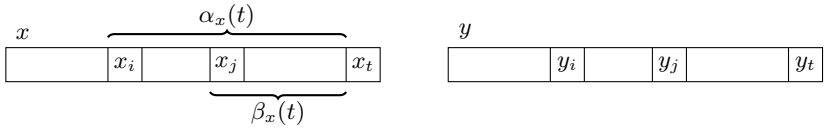

Fig. 2. An illustration of Lemma 1, part (a): $x[1\,..\,t] \approx y[1\,..\,t]$ iff $x[1\,..\,t-1] \approx y[1\,..\,t-1]$ and $y_i \le y_t \le y_j$.

As a corollary of part (b) of Lemma 1 we obtain that the codes provide an equivalent characterization of order-isomorphism:

Lemma 2. $x \approx y \;\Leftrightarrow\; \mathit{Code}(x) = \mathit{Code}(y)$.

The codes of strings can be computed efficiently. Applying Lemma 1 from [14] to strings over polynomially-bounded alphabet we obtain:

Lemma 3. *For a string w of length n, $\mathit{Code}(w)$ can be computed in $O(n)$ time.*

3 Order-Preserving Suffix Trees

Let us define the following family of sequences:

$$\mathit{SufCodes}(w) = \{\mathit{Code}(\mathit{suf}_1)\#,\ \mathit{Code}(\mathit{suf}_2)\#,\ \ldots,\ \mathit{Code}(\mathit{suf}_n)\#\},$$

see Fig. 3. The *order-preserving suffix tree* of w (*op-suffix-tree* in short), denoted $\mathit{opSufTree}(w)$, is a compacted trie of all the sequences in $\mathit{SufCodes}(w)$.

Example 2. Let $w = (1, 2, 4, 4, 2, 5, 5, 1)$. All $\mathit{SufCodes}(w)$ are given in Fig. 3.

The nodes of $\mathit{opSufTree}(w)$ with at least two children are called branching nodes, together with the leaves they form explicit nodes of the tree. All the remaining nodes (that 'disappear' due to compactification) are called implicit nodes. For a node v, its explicit descendant (denoted as $\mathit{FirstDown}(v)$) is the top-most explicit node in the subtree of v (possibly $\mathit{FirstDown}(v) =$

suffixes of w:								$SufCodes(w)$:								
1	2	4	4	2	5	5	1	(1,1)	(1,2)	(1,3)	(1,1)	(3,3)	(2,6)	(1,1)	(7,7)	#
	2	4	4	2	5	5	1		(1,1)	(1,2)	(1,1)	(3,3)	(2,5)	(1,1)	(7,3)	#
		4	4	2	5	5	1			(1,1)	(1,1)	(3,1)	(2,4)	(1,1)	(6,3)	#
			4	2	5	5	1				(1,1)	(2,1)	(2,3)	(1,1)	(5,3)	#
				2	5	5	1					(1,1)	(1,2)	(1,1)	(4,3)	#
					5	5	1						(1,1)	(1,1)	(3,1)	#
						5	1							(1,1)	(2,1)	#
							1								(1,1)	#

Fig. 3. $SufCodes(w)$ for $w = (1, 2, 4, 4, 2, 5, 5, 1)$.

v). By $Locus_{Code(x)}$ we denote the (explicit or implicit) locus of $Code(x)$ in $opSufTree(w)$. Only the explicit nodes of $opSufTree(w)$ are stored. The tree contains $O(n)$ leaves, hence its size is $O(n)$.

The leaf corresponding to $Code(suf_i)\#$ is labeled with the number i. Each branching node stores its depth and one of the leaves in its subtree. Each edge stores the code only of its first character. The codes of all the remaining characters of any edge can be obtained using a *character oracle* that can efficiently provide the code $LastCode(suf_i[1\,..\,j])$ for any i, j.

Each explicit node v stores a suffix link, $SufLink(v)$, that may lead to an implicit or an explicit node (see an example in Fig. 5). The suffix link is defined as:

$$SufLink(Locus_{Code(x)}) = Locus_{Code(DelFirst(x))},$$

where $DelFirst(x)$ results in removing the first character of x, see Fig. 4.

Observation 1 $Code(x) = Code(y) \Rightarrow Code(DelFirst(x)) = Code(DelFirst(y))$.

We also introduce an *incomplete* order-preserving suffix tree of w, denoted $T(w)$, in which the character oracle is not available and each explicit node v has at most one outgoing edge that does not store its first character (*incomplete edge*). This edge is located on the longest path leading from v to a leaf.

Example 3.
Let $w = (1, 2, 4, 4, 2, 5, 5, 1)$. The op-suffix-tree of w is presented in Fig. 5.

4 Algorithmic Toolbox

We use y-fast trees, see [19], to compute the last symbols of the code of a sequence changing in a queue-like manner.

Lemma 4. [Weak Character Oracle] *An initially empty sequence x over $\{1, \ldots, n\}$ can be maintained in a data structure $\mathcal{D}(x)$ of $O(|x|)$ size so that the following queries are supported in $O(\log \log n)$ expected time:*

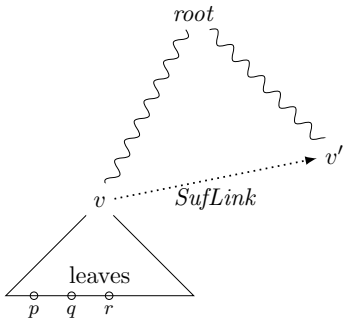

Fig. 4. Let $\gamma = \mathit{PathLabel}(\mathit{root}, v)$, $|\gamma| = k$, and $\gamma' = \mathit{PathLabel}(\mathit{root}, v')$, where $v' = \mathit{SufLink}(v)$. Not necessarily γ' is a suffix of γ, but $\gamma' = \mathit{Code}(\mathit{DelFirst}(x))$, where $x = w[p\,..\,p+k-1]$ or $x = w[q\,..\,q+k-1]$ or $x = w[r\,..\,r+k-1]$.

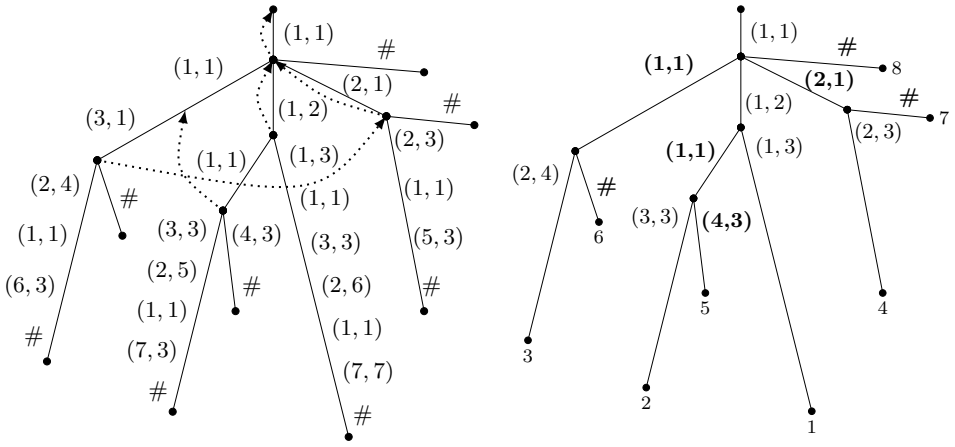

Fig. 5. The uncompacted trie of $\mathit{SufCodes}(w)$ for $w = (1, 2, 4, 4, 2, 5, 5, 1)$ (to the left) and its compacted version, the complete op-suffix-tree of w (to the right). The dotted arrows (left figure) show suffix links for branching nodes, note that one of them leads to an implicit node. Labels in the right figure that are in bold are present also in the incomplete op-suffix-tree.

compute $\mathit{LastCode}(x)$; append a single letter to x; and $\mathit{DelFirst}(x)$. The first and the third operations are invalid if x is empty.

Proof. The main tool is here the y-fast tree, a data structure for dynamic predecessor queries. The following fact has been shown in [19].

Claim. Let N be an integer such that $N = O(\log w)$, where w is the machine word-size. There exists a data structure that uses $O(|X|)$ space to maintain a

set X of key-value pairs with keys from $\{1, \ldots, N\}$ and supports the following operations in $O(\log \log N)$ expected time:

find(k): find the value associated with k, if any,
predecessor(k): return the pair $(k', v) \in X$ with the largest $k' \leq k$,
successor(x): return the pair $(k', v) \in X$ with the smallest $k' \geq k$,
remove(k): remove the pair with key k,
insert(k, v): insert (k, v) to X removing the pair with key k, if any.

The y-fast trees are now used as follows. For each symbol present in x we store in a y-fast tree its last occurrence, represented as a time-stamp (the ordinal number of the push operation used to append it). Then the *LastCode*() query is answered using one predecessor and one successor query. □

Our second tool is the dynamic weighted ancestor data structure proposed by Kopelowitz and Lewenstein [13] and originally motivated by problems related to ordinary suffix trees. A *weighted tree* is a rooted tree with weights in each node that satisfies a monotonicity condition: the weight of a node is strictly greater than the weight of its parent. The *weighted ancestor query* is:

given a node v and a weight w find $\mathit{LevelAnc}(v, w)$ – the lowest ancestor of v with weight at least w.

The following lemma is proved in [13].

Lemma 5. *Let N be an integer such that $N = O(\log w)$, where w is the machine word-size. There exists a data structure which maintains a weighted tree T with weights $\{1, \ldots, N\}$ in space $O(|T|)$ and supports the following operations in $O(\log \log N)$ amortized time:*

- *answer $\mathit{LevelAnc}(v, w)$,*
- *insert a leaf with weight w and v as a parent,*
- *insert a node with weight w by subdividing the edge joining v with its parent.*

We require that the weights of inserted nodes satisfy the monotonicity condition.

5 Constructing Incomplete Order-Preserving Suffix Tree

We design a version of Ukkonen's algorithm [18] in which suffix links are computed using dynamic weighted ancestors, see Fig. 6. The weights of explicit nodes represent their depths. In this case for a node u, by $\mathit{LevelAnc}(u, d)$ we denote its (explicit or implicit) ancestor of depth d.

Our algorithm works online. After reading the string w it produces:

- the incomplete op-suffix-tree $T(w)$ for w;
- the longest suffix $\mathfrak{F}$ of w such that $\mathit{Code}(\mathfrak{F})$ corresponds to a non-leaf node of $T(w)$, together with the data structure $\mathcal{D}(\mathfrak{F})$; $\mathfrak{F}$ is called the *active suffix*;
- the node (explicit or implicit) $\mathit{Locus}_{\mathit{Code}(\mathfrak{F})}$, called the *active node*.

In the algorithm all implicit nodes are represented in a canonical form: the explicit descendant (*FirstDown*) and the length of the edge to this descendant. Each explicit node stores a hash table (see [5, 8]) of its explicit children, indexed by the labels of the respective edges. Note that the explicit child corresponding to an incomplete edge is stored outside of this hash table.

When w is extended by one character, say a, we traverse the *active path* in $T(w)$: we search for the longest suffix $\mathfrak{F}'$ of $\mathfrak{F}$ such that $Locus_{Code(\mathfrak{F}'a)}$ appears in the tree, and for each longer suffix $\mathfrak{F}''$ of $\mathfrak{F}$ we create a branch leading to a new leaf node $Locus_{Code(\mathfrak{F}''a)}$. The active path is found by jumping along suffix links, starting at the active node. The end point of the active path provides the new active node, and $\mathfrak{F}'a$ becomes the active suffix.

To compute the code of a in $Code(\mathfrak{F}a)$ we use the following observation.

Observation 2 *Due to Lemma 4 we can compute* $LastCode(\mathfrak{F}\cdot a)$ *in* $O(\log\log n)$ *time, where* $\mathfrak{F}$ *is the active suffix.*

We also use two auxiliary subroutines.

Function $Transition(v,(p,q))$**.** This function checks if v has an (explicit or implicit) child v' such that the edge from v to v' represents the code (p,q). It returns the node v' or **nil** if such a node does not exist. We check, using hashing, if any of the labeled edges outgoing from v starts with the code (p,q), for (at most one for v) incomplete edge we can check if its starting letter code equals (p,q) by checking two inequalities from part (a) of Lemma 1.

Function $Branch(v,(p,q))$**.** This function creates a new (open) transition from v with the code (p,q). If v was implicit then it is made explicit, at this moment the edge leading to its existing child remains incomplete.

Every single jump along a suffix link in the algorithm is performed in $O(\log\log n)$ time as presented in Fig. 6.

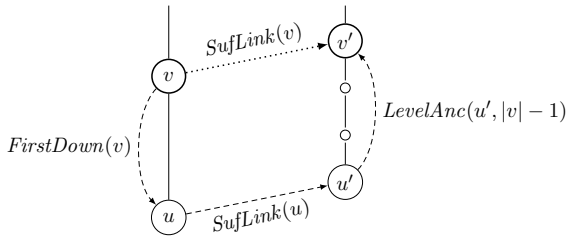

Fig. 6. The computation of $SufLink(v)$. Here u is explicit.

Algorithm *Construct incomplete opSufTree*(w)
Initialize T as incomplete *opSufTree* for w_1;
$v := root$; $\mathfrak{F} :=$ empty string;
for $i := 2$ **to** n **do**
  $a := w_i$; $\mathfrak{F} := \mathfrak{F} \cdot a$;
  while *Transition*$(v, LastCode(\mathfrak{F})) =$ **nil do**
    Branch$(v, LastCode(\mathfrak{F}))$;
    if $v = root$ **then break**;
    $\mathfrak{F} := DelFirst(\mathfrak{F})$;
    $u := FirstDown(v)$; { u is the first explicit node below v, including v }
    $u' := SufLink(u)$; { u' can be an implicit node }
    $v' := LevelAnc(u', |v| - 1)$; { weighted ancestor query }
    $SufLink(v) := v'$; $v := v'$;
  $v := Transition(v, LastCode(\mathfrak{F}))$;
return T;

Observation 3 [Why incomplete?] *At first glance it is not clear why incomplete edges appear. Consider the situation when we jump to an implicit node $v' = SufLink(v)$ and we later branch in this node. The node v' becomes explicit and the existing edge from this node to some node u' becomes an* incomplete edge. *Despite incompleteness of the edge (v', u') the equality test between the (known) lastcode letter of the active string and the first (unknown) code letter of the label of this edge can be done quickly due to part (a) of Lemma 1.*

In the pseudocode above we perform $O(n)$ operations in total. This follows from the fact that each step of the while-loop creates a new edge in the tree. The operations involving $\mathfrak{F}$ and the operation *LevelAnc* are performed in $O(\log \log n)$ time and all the remaining operations require only constant time. We obtain the following result.

Theorem 1. *The incomplete op-suffix-tree $T(w)$ for a string w of length n can be computed in $O(n \log \log n)$ time.*

6 Incomplete Suffix Tree as Order-Preserving Index

The most common application of suffix trees is pattern matching with time complexity independent of the length of the text.

Theorem 2. *Assume that we have $T(w)$ for a string w of length n. Given a pattern x of length m, one can check if w contains a factor order-isomorphic to x in $O(m)$ time and report all occurrences of such factors in $O(m + Occ)$ time, where Occ is the number of occurrences.*

Proof. First we compute the code of the pattern. This takes $O(m)$ time due to Lemma 3. To answer a query, we traverse down $T(w)$ using the successive symbols of the code. At each step we use the function $\mathit{Transition}(v, (p, q))$.

This enables to find the locus of $\mathit{Code}(x)$ in $O(m)$ time. Afterwards all the occurrences of factors that are order-isomorphic to x can be listed in the usual way by inspecting all leaves in the subtree of $\mathit{Locus}_{\mathit{Code}(x)}$. □

Another important application of standard suffix trees is related to finding the longest common factor of two strings. An analog of this problem in the order-preserving setting is especially important, since it provides a way to find common trends in time series. In this problem, given two strings w and x, we need to find the maximum length of their common factors that are order-isomorphic. We show the usefulness of the suffix links in incomplete op-suffix-tree.

Theorem 3. *Let w be a string of length n. Having $T(w)$, one can find the order-preserving longest common factor of w and x, the latter string of length m, in $O(m(\log\log m + \log\log n))$ time.*

Proof. The main principle of the algorithm is the same as in the standard setting (see Corollary 6.12 in [6]). However, it needs to be enhanced using our algorithmic tools.

Let $\mathit{pref}(x)$ be the longest prefix of x such that $\mathit{Code}(\mathit{pref}(x))$ corresponds to a node in $T(w)$. Let suf_i^x be the i-th suffix of x. The algorithm computes $\mathit{pref}(\mathit{suf}_1^x)$, $\mathit{pref}(\mathit{suf}_2^x)$ etc and finds the maximum depth among their loci.

At each point the data structure $\mathcal{D}(\mathit{pref}(\mathit{suf}_i^x))$ for the current suffix is stored. First, the locus of $\mathit{pref}(\mathit{suf}_1^x)$ is found by iterating $\mathit{Transition}(v, (p, q))$, as in the order-preserving pattern matching (Theorem 2). To proceed from $\mathit{pref}(\mathit{suf}_i^x)$ to $\mathit{pref}(\mathit{suf}_{i+1}^x)$, we remove the first letter ($\mathit{DelFirst}$), which also corresponds to a jump along a suffix link, and then keep traversing down the $T(w)$ using $\mathit{Transition}(v, (p, q))$.

By Lemmas 4 and 5, we obtain the required time complexity. □

7 Constructing Complete Order-Preserving Suffix Tree

In Section 5 we presented an $O(n \log\log n)$ time construction of an incomplete op-suffix-tree. To obtain a complete op-suffix-tree, we need to put labels on incomplete edges and to provide a character oracle. Note that, using a character oracle working in $f(n)$ time, we can fill in the missing labels in $O(nf(n))$ time.

Observation 4 *The op-suffix-tree of a string of length n can be constructed in $O(n \log n)$ time.*

Proof. After $O(n \log n)$ preprocessing one can compute $\mathit{LastCode}(\mathit{suf}_i[1\,..\,j])$ for any i, j in $O(\log n)$ time. We use range trees, see [7]. Then we can fill in separately each missing label in the incomplete tree in $O(n \log n)$ time. □

Below we show a slightly faster construction. For this, however, we need a different encoding of strings that also preserves the order. A very similar code was already presented in [11]. For any $i \in \{1, \ldots, n\}$ define:

$$prev_w^{<}(i) = |\{k : k < i,\, w_k < w_i\}|, \quad prev_w^{=}(i) = |\{k : k < i,\, w_k = w_i\}|.$$

The *counting code* of a string w is defined as:

$$Code'(w) = ((prev_w^{<}(1), prev_w^{=}(1)), \ldots, (prev_w^{<}(|w|), prev_w^{=}(|w|))).$$

We also define $LastCode'(w) = (prev_w^{<}(|w|), prev_w^{=}(|w|))$.

Example 4. The counting code of each of the strings in Fig. 1 is $(0,0)\ (0,0)\ (2,0)\ (1,1)\ (0,0)\ (2,0)\ (6,0)\ (2,1)\ (4,2)$.

The following lemma states that $Code'$ is also an order-preserving code. In this version of the paper we omit the proof, since it is basically present in [11].

Lemma 6. $x \approx y \ \Leftrightarrow\ Code'(x) = Code'(y)$.

The main advantage of the new order-preserving code is the existence of an $O(\log n / \log\log n)$ time character oracle. To construct the oracle we use a geometric approach: the computation of $LastCode'$ for w corresponds to counting points in certain orthogonal rectangles in the plane.

Observation 5 *Let us treat the pairs (i, w_i) as points in the plane. Then we have $LastCode'(suf_i[1\,..\,j]) = (a, b)$, where a is the number of points that lie within the rectangle $A = [i, i+j-2] \times (-\infty, w_{i+j-1})$ and b is the number of points in the rectangle $B = [i, i+j-2] \times [w_{i+j-1}, w_{i+j-1}]$, see Fig. 7.*

The orthogonal range counting problem is defined as follows. We are given n points in the plane and we need to answer queries of the form:

"how many points are contained in a given axis-aligned rectangle?".

An efficient solution to this problem was given by Chan and Pătraşcu, see Theorem 2.3 in [4] which we state below as Lemma 7. We say that a point (p, q) dominates a point (p', q') if $p > p'$ and $q > q'$.

Lemma 7. *We can preprocess n points in the plane in $O(n\sqrt{\log n})$ time, using a data structure with $O(n)$ words of space, so that we can count the number of points dominated by a query point in $O(\log n / \log\log n)$ time.*

Theorem 4. *The op-suffix-tree of a string of length n using the counting code can be constructed in $O(n \log n / \log\log n)$ time.*

Proof. Due to Lemma 2 and the corresponding Lemma 6, the *skeleton* of the op-suffix-tree for each of the order-preserving codes is the same. Hence, to construct the op-suffix-tree for the counting code, we compute the skeleton of the suffix tree using the algorithm for incomplete op-suffix-tree. Afterwards we use the character oracle to insert the first characters on each edge of the skeleton.

Due to Observation 5 and Lemma 7 after $O(n\sqrt{\log n})$ time and $O(n)$ space preprocessing one can compute $LastCode'(suf_i[1\,..\,j])$ for any i, j in $O(\log n / \log\log n)$ time. □

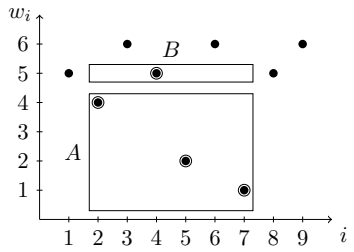

Fig. 7. Geometric illustration of the sequence $w = (5, 4, 6, 5, 2, 6, 1, 5, 6)$. The elements w_i are represented as points (i, w_i). The computation of $\mathit{LastCode}'(\mathit{suf}_2[1\,..\,7]) = (3, 1)$ corresponds to counting points in rectangles A, B.

8 Final Remarks

A family of strings $S_1, \ldots, S_n$ is called a quasi-suffix collection [5] if the following conditions hold: **(a)** $|S_1| = n$ and $|S_i| = |S_{i-1}| - 1$ for all $i > 1$; **(b)** no S_i is a prefix of another S_j; **(c)** if S_i and S_j have a common prefix of length $l > 0$ then S_{i+1} and S_{j+1} have a common prefix of length at least $l - 1$.

The suffix tree for a quasi-suffix collection is defined as a compacted trie of all the strings in the collection. Cole and Hariharan [5] presented a general framework for constructing suffix trees for quasi-suffix collections. Assuming they are given a character oracle that provides the j-th character of any S_i in $O(1)$ time, they construct the suffix tree for a quasi-suffix collection in $O(n)$ time and space. They assume that the alphabet of S_i has size polynomial in n.

One can observe that the strings in $\mathit{SufCodes}(w)$ form a quasi-suffix collection. Hence, we can apply the result of Cole and Hariharan [5] to obtain an op-suffix-tree by using our character oracle. The time complexity grows by a factor corresponding to the complexity of the oracle and matches the bound from our Theorem 4. However, our approach, tailored for the order-preserving setting, appears to be simpler.

The framework of Cole and Hariharan [5] is based on McCreight's algorithm for suffix tree construction. Recently Lee, Na and Park [15] presented a modified version of this algorithm that uses Ukkonen's suffix tree construction algorithm. If one applies their construction, a yet alternative construction of op-suffix-tree can be given, still within the same complexity bounds.

References

1. M. H. Albert, R. E. L. Aldred, M. D. Atkinson, and D. A. Holton. Algorithms for pattern involvement in permutations. In P. Eades and T. Takaoka, editors, *ISAAC*, volume 2223 of *Lecture Notes in Computer Science*, pages 355–366. Springer, 2001.
2. B. S. Baker. Parameterized pattern matching: Algorithms and applications. *J. Comput. Syst. Sci.*, 52(1):28–42, 1996.

3. P. Bose, J. F. Buss, and A. Lubiw. Pattern matching for permutations. *Inf. Process. Lett.*, 65(5):277–283, 1998.
4. T. M. Chan and M. Patrascu. Counting inversions, offline orthogonal range counting, and related problems. In M. Charikar, editor, *SODA*, pages 161–173. SIAM, 2010.
5. R. Cole and R. Hariharan. Faster suffix tree construction with missing suffix links. *SIAM J. Comput.*, 33(1):26–42, 2003.
6. M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, USA, 2007.
7. M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry. Algorithms and Applications. Third Edition*. Springer-Verlag Berlin Heidelberg, 2008.
8. M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.
9. S. Guillemot and S. Vialette. Pattern matching for 321-avoiding permutations. In Y. Dong, D.-Z. Du, and O. H. Ibarra, editors, *ISAAC*, volume 5878 of *Lecture Notes in Computer Science*, pages 1064–1073. Springer, 2009.
10. L. Ibarra. Finding pattern matchings for permutations. *Inf. Process. Lett.*, 61(6):293–295, 1997.
11. J. Kim, P. Eades, R. Fleischer, S.-H. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi, and T. Tokuyama. Order preserving matching. *CoRR*, abs/1302.4064, 2013. Submitted to Theor. Comput. Sci.
12. D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition*. Addison-Wesley, 1973.
13. T. Kopelowitz and M. Lewenstein. Dynamic weighted ancestors. In N. Bansal, K. Pruhs, and C. Stein, editors, *SODA*, pages 565–574. SIAM, 2007.
14. M. Kubica, T. Kulczynski, J. Radoszewski, W. Rytter, and T. Walen. A linear time algorithm for consecutive permutation pattern matching. *Inf. Process. Lett.*, 113(12):430–433, 2013.
15. T. Lee, J. C. Na, and K. Park. On-line construction of parameterized suffix trees for large alphabets. *Inf. Process. Lett.*, 111(5):201–207, 2011.
16. L. Lovász. *Combinatorial problems and exercices*. North-Holland, 1979.
17. D. Rotem. Stack sortable permutations. *Discrete Mathematics*, 33(2):185–196, 1981.
18. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
19. D. E. Willard. Log-logarithmic worst-case range queries are possible in space theta(n). *Inf. Process. Lett.*, 17(2):81–84, 1983.