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[7,8,⋆⋆], 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, UK
{maxime.crochemore,c.iliopoulos,alessio.langiu}@kcl.ac.uk
[5] Faculty of Engineering, Computing and Mathematics, University of Western Australia, Perth, 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. Known results include a linear-time algorithm for this problem (in case of polynomially-bounded alphabet) and a generalization to multiple patterns. We give an $O(n \log\log n)$ time construction of an index that enables order-preserving pattern matching queries in time proportional to pattern length. The main component is a data structure being an incomplete suffix tree in the order-preserving setting. The tree can miss single letters related to branching at internal nodes. Such incompleteness results from the weakness of our so called *weak character oracle*. However, due to its weakness, such oracle can answer queries on-line in $O(\log\log n)$ time 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.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-3-319-02432-5_33

⋆ Supported by Polish budget funds for science in 2013-2017 as a research project under the 'Diamond Grant' program.
⋆⋆ Supported by the NSF–funded iPlant Collaborative (NSF grant #DBI-0735191).
⋆⋆⋆ The author receives financial support of Foundation for Polish Science.
† Supported by grant no. N206 566740 of the National Science Centre.

O. Kurland, M. Lewenstein, and E. Porat (Eds.): SPIRE 2013, LNCS 8214, pp. 84–95, 2013.

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 and 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. We provide randomized (Las Vegas) algorithms for the word-RAM model with $\Omega(\log n)$ word size.

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 larger construction time, $O(n \log n)$ or $O(n \log n / \log\log n)$ depending on the codes used.

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.

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 $w_i \,..\, w_j$, and by suf_i – the i-th suffix of w, that is, $w[i\,..\,n]$.
For any $i \in \{1, \dots, n\}$ define:

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

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

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

and $\beta_w(i) = i$ if no such j exists. If several equally good values of j exist, we select the greatest possible value of j that is smaller than i.
We introduce codes of strings in a similar way as in [14]:

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

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

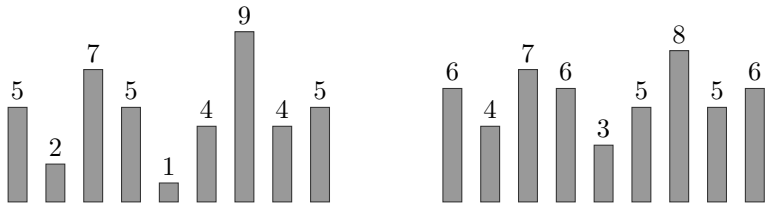

Fig. 1. Example of two order-isomorphic strings. Their codes are equal to $(1,1)\ (2,1)\ (2,3)\ (3,3)\ (5,3)\ (4,2)\ (4,7)\ (2,2)\ (5,5)$.

Lemma 2. *Let x and y be two strings of length t and $x' = x[1\,..\,t-1]$, $y' = y[1\,..\,t-1]$. Then:*

(a) $x \approx y \;\Leftrightarrow\; x' \approx y' \;\wedge\; (y_i \le y_t \le y_j)$, *where* $i = t - \alpha_x(t)$, $j = t - \beta_x(t)$;
(b) $x \approx y \;\Leftrightarrow\; x' \approx y' \;\wedge\; LastCode(x) = LastCode(y)$.

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

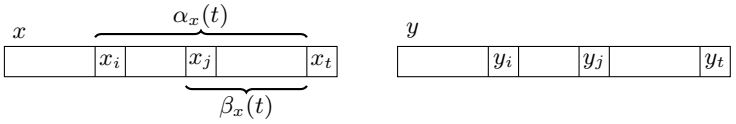

Fig. 2. An illustration of Lemma 2, part (a): $x[1\,..\,t] \approx y[1\,..\,t]$ is equivalent to $x[1\,..\,t-1] \approx y[1\,..\,t-1]$ and $y_i \le y_t \le y_j$

Part (b) of Lemma 2 implies that the codes provide an equivalent characterization of order-isomorphism:

Lemma 3. $x \approx y \;\Leftrightarrow\; Code(x) = Code(y)$.

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

Lemma 4. *For a string w of length n, $Code(w)$ can be computed in $O(n)$ time.*

3 Order-Preserving Suffix Trees

Let us define the following family of sequences:

$$SufCodes(w) = \{Code(suf_1)\#,\ Code(suf_2)\#,\ \ldots,\ Code(suf_n)\#\},$$

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

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

The nodes of $opSufTree(w)$ 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 $FirstDown(v)$) is the top-most explicit node in the subtree of v (possibly $FirstDown(v) = v$). 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

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 $w = (1, 2, 4, 4, 2, 5, 5, 1)$

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. 4). 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. 5.

Observation 6. $Code(x) = Code(y) \Rightarrow Code(DelFirst(x)) = Code(DelFirst(y))$.

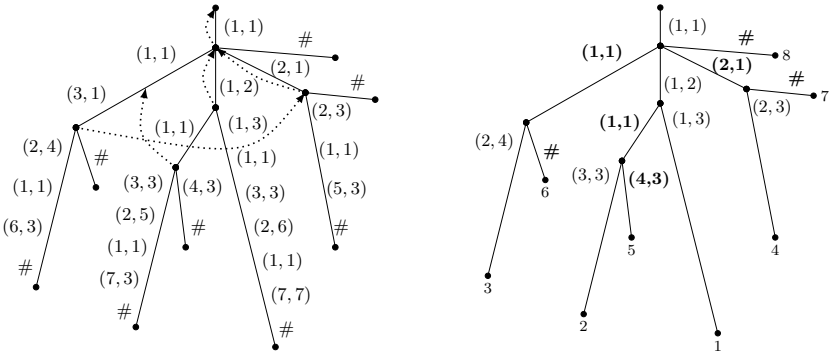

Fig. 4. The uncompacted trie of $SufCodes(w)$ for $w = (1, 2, 4, 4, 2, 5, 5, 1)$ (to the left) and its compacted version, the complete op-suffix-tree of w (to the right). The dotted arrows (left figure) show suffix links for branching nodes, note that one of them leads to an implicit node. Labels in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

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 can have 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.

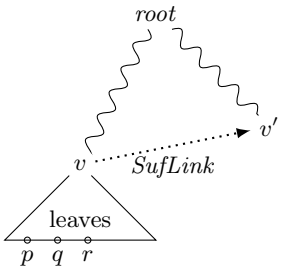

Fig. 5. Let γ be the text spelled out on a path from the root to v in the uncompacted op-suffix trie of w. Similarly, let γ' be the text on a path to $v' = \mathit{SufLink}(v)$. Observe that not necessarily γ' is a suffix of γ, but $\gamma' = \mathit{Code}(\mathit{DelFirst}(x))$, where $x = w[p\,..\,p+k-1]$ or $x = w[q\,..\,q+k-1]$ or $x = w[r\,..\,r+k-1]$, where p, q, r are the labels on the leaves in the subtree rooted in v.

Example 7.
Let $w = (1, 2, 4, 4, 2, 5, 5, 1)$. The op-suffix-tree of w is presented in Fig. 4.

4 Algorithmic Toolbox

We use a predecessor data structure to compute the last symbols of the code of a sequence changing in a queue-like manner.

Lemma 8. [Weak Character Oracle] *An initially empty sequence x over $\{1, \ldots, n\}$ can be maintained in a data structure $\mathcal{D}(x)$ of size $O(|x|)$ so that the following queries are supported in $O(\log \log n)$ expected time:*

compute $\mathit{LastCode}(x)$; append a single letter to x; and $\mathit{DelFirst}(x)$.

Only the second operation is valid if x is empty.

Proof. The main tool here is 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 $\omega = \Omega(\log n)$, where ω 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. The keys are the symbols present in x while the values associated with them are the locations of their last occurrences represented as a time-stamps (that is, the ordinal numbers of the push operations used to append them). Then the $\mathit{LastCode}()$ 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 integer weight assigned to each node, such that a monotonicity condition is satisfied: 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 g find $\mathit{WeightedAnc}(v, g)$ – the highest ancestor of v with weight at least g.

The following lemma is proved in [13].

Lemma 9. *Let N be an integer such that $\omega = \Omega(\log N)$, where ω is the machine word-size. There exists a data structure which maintains a weighted tree T with weights $\{1, \ldots, N\}$ in $O(|T|)$ space and supports the following operations in $O(\log \log N)$ expected time:*

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

The weights of inserted nodes must meet 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 weighted ancestor queries, see Fig. 6. The weights of explicit nodes represent their depths. In this case for a node u, by $\mathit{WeightedAnc}(u, d)$ we denote its (explicit or implicit) ancestor at depth d.

Our algorithm works online. While reading the string w it maintains:

- 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 distance to this descendant (depth difference). Each explicit node stores a dynamic hash table (see [5,8]) of its explicit children, indexed by the labels of the respective edges. Note that the explicit child corresponding to the incomplete edge is stored outside of the 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 $\mathit{Locus}_{\mathit{Code}(\mathfrak{F}'a)}$ appears in the tree, and for each longer suffix $\mathfrak{F}''$ of $\mathfrak{F}$ we create a branch leading to a new leaf node $\mathit{Locus}_{\mathit{Code}(\mathfrak{F}''a)}$. 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 last symbol of $\mathit{Code}(\mathfrak{F}a)$ we use the following observation.

Observation 10. *Due to Lemma 8 we can compute* $LastCode(\mathfrak{F} \cdot a)$ *in* $O(\log \log n)$ *expected 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 2.

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 already existing child remains incomplete.

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 }
    $v' := WeightedAnc(u', |v| - 1)$; { weighted ancestor query }
    $SufLink(v) := v'$; $v := v'$;
  $v := Transition(v, LastCode(\mathfrak{F}))$;
return T;

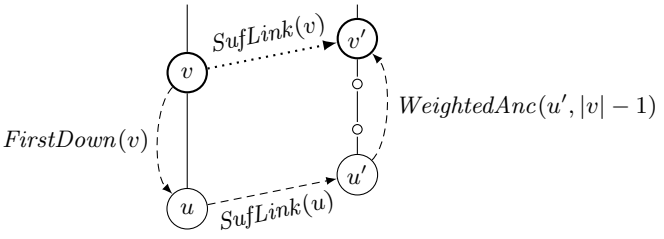

Fig. 6. Computation of $SufLink(v)$. Here u is explicit.

Remark 11. [Why incomplete?] At first glance it is not clear why incomplete edges appear. Consider the situation when we jump to an implicit node $v' = \mathit{SufLink}(v)$ 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) last code 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 2.

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 *WeightedAnc* are performed in $O(\log \log n)$ time and all the remaining operations require constant time only. We obtain the following result.

Theorem 12. *The incomplete op-suffix-tree $T(w)$ for a string w of length n can be computed in $O(n \log \log n)$ expected 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 13. *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 reported.*

Proof. First we compute the code of the pattern. This takes $O(m)$ time due to Lemma 4. 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 $\mathit{Locus}_{\mathit{Code}(x)}$. □

The motivating application of the standard suffix trees was 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 longest factor of x that is order-isomorphic to a factor of w. We show the usefulness of the suffix links in incomplete op-suffix-tree.

Theorem 14. *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 $O(m(\log \log m + \log \log n))$ expected 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 $pref(x)$ be the longest prefix of x such that $Code(pref(x))$ corresponds to a node in $T(w)$. Let suf_i^x be the i-th suffix of x. The algorithm computes $pref(suf_1^x)$, $pref(suf_2^x)$ etc. and finds the maximum depth among their loci.

At each point the data structure $\mathcal{D}(pref(suf_i^x))$ for the current suffix is stored. First, the locus of $pref(suf_1^x)$ is found by iterating $Transition(v,(p,q))$, as in the order-preserving pattern matching (Theorem 13). To proceed from $pref(suf_i^x)$ to $pref(suf_{i+1}^x)$, we remove the first letter ($DelFirst$), which also corresponds to a jump along a suffix link, and then keep traversing down the $T(w)$ using $Transition(v,(p,q))$.

By Lemmas 8 and 9, 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 15. *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 $LastCode(suf_i[1..j])$ for any i, j in $O(\log n)$ time. We use range trees for that, 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 16. 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 17. $x \approx y \;\Leftrightarrow\; Code'(x) = Code'(y)$.

The main advantage of the new order-preserving code is the existence of an $O(\log n/\log\log n)$ time character oracle with $o(n\log n/\log\log n)$ time construction. To design 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 18. *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 are to count the number of points in axis-aligned rectangles given as queries.

An efficient solution to this problem was given by Chan and Pătraşcu, see Theorem 2.3 in [4] which we state below as Lemma 19. We say that a point (p, q) dominates a point (p', q') if $p > p'$ and $q > q'$.

Lemma 19. *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.*

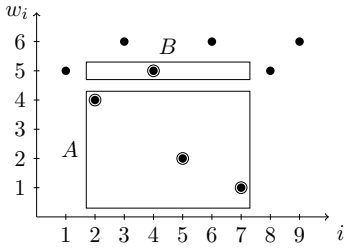

Fig. 7. Geometric illustration of the sequence $w = (5, 4, 6, 5, 2, 6, 1, 5, 6)$. The elements w_i are represented as points (i, w_i). The computation of $LastCode'(suf_2[1\,..\,7]) = (3, 1)$ corresponds to counting points in rectangles A, B.

Theorem 20. *The op-suffix-tree of a string of length n using the counting code can be constructed in $O(n\log n/\log\log n)$ expected time.*

Proof. Due to Lemma 3 and the corresponding Lemma 17, 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 18 and Lemma 19 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. □

References

1. Albert, M.H., Aldred, R.E.L., Atkinson, M.D., Holton, D.A.: Algorithms for pattern involvement in permutations. In: Eades, P., Takaoka, T. (eds.) ISAAC 2001. LNCS, vol. 2223, pp. 355–366. Springer, Heidelberg (2001)
2. Baker, B.S.: Parameterized pattern matching: Algorithms and applications. J. Comput. Syst. Sci. 52(1), 28–42 (1996)
3. Bose, P., Buss, J.F., Lubiw, A.: Pattern matching for permutations. Inf. Process. Lett. 65(5), 277–283 (1998)
4. Chan, T.M., Patrascu, M.: Counting inversions, offline orthogonal range counting, and related problems. In: Charikar, M. (ed.) SODA, pp. 161–173. SIAM (2010)
5. Cole, R., Hariharan, R.: Faster suffix tree construction with missing suffix links. SIAM J. Comput. 33(1), 26–42 (2003)
6. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press, USA (2007)
7. de Berg, M., Cheong, O., van Kreveld, M., Overmars, M.: Computational Geometry. Algorithms and Applications, 3rd edn. Springer, Heidelberg (2008)
8. Dietzfelbinger, M., Karlin, A.R., Mehlhorn, K., Meyer auf der Heide, F., Rohnert, H., Tarjan, R.E.: Dynamic perfect hashing: Upper and lower bounds. SIAM J. Comput. 23(4), 738–761 (1994)
9. Guillemot, S., Vialette, S.: Pattern matching for 321-avoiding permutations. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 1064–1073. Springer, Heidelberg (2009)
10. Ibarra, L.: Finding pattern matchings for permutations. Inf. Process. Lett. 61(6), 293–295 (1997)
11. Kim, J., Eades, P., Fleischer, R., Hong, S.-H., Iliopoulos, C.S., Park, K., Puglisi, S.J., Tokuyama, T.: Order preserving matching. CoRR, abs/1302.4064 (2013); Submitted to Theor. Comput. Sci.
12. Knuth, D.E.: The Art of Computer Programming, 2nd edn. Fundamental Algorithms, vol. I. Addison-Wesley (1973)
13. Kopelowitz, T., Lewenstein, M.: Dynamic weighted ancestors. In: Bansal, N., Pruhs, K., Stein, C. (eds.) SODA, pp. 565–574. SIAM (2007)
14. Kubica, M., Kulczynski, T., Radoszewski, J., Rytter, W., Walen, T.: A linear time algorithm for consecutive permutation pattern matching. Inf. Process. Lett. 113(12), 430–433 (2013)
15. Lee, T., Na, J.C., Park, K.: On-line construction of parameterized suffix trees for large alphabets. Inf. Process. Lett. 111(5), 201–207 (2011)
16. Lovász, L.: Combinatorial problems and exercices. North-Holland (1979)
17. Rotem, D.: Stack sortable permutations. Discrete Mathematics 33(2), 185–196 (1981)
18. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
19. Willard, D.E.: Log-logarithmic worst-case range queries are possible in space theta(n). Inf. Process. Lett. 17(2), 81–84 (1983)