Range Non-overlapping Indexing and Successive List Indexing

Orgad Keller, Tsvi Kopelowitz, and Moshe Lewenstein*

Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel
{kellero, kopelot, moshe}@cs.biu.ac.il

Abstract. We present two natural variants of the indexing problem:
In the *range non-overlapping indexing* problem, we preprocess a given text to answer queries in which we are given a pattern, and wish to find a maximal-length sequence of occurrences of the pattern in the text, such that the occurrences *do not overlap* with one another. While efficiently solving this problem, our algorithm even enables us to efficiently perform so in *substrings* of the text, denoted by given start and end locations. The methods we supply thus generalize the *string statistics* problem [4, 5], in which we are asked to report merely the *number* of non-overlapping occurrences in the *entire* text, by reporting the occurrences themselves, even only for substrings of the text.
In the related *successive list indexing* problem, during query-time we are given a pattern and a list of locations in the preprocessed text. We then wish to find a list of occurrences of the pattern, such that the ith occurrence is the leftmost occurrence of the pattern which starts to the right of the ith location given by the input list.
Both problems are solved by using tools from computational geometry, specifically a variation of the *range searching for minimum* problem of Lenhof and Smid [12], here considered over a grid, in what appears to be the first utilization of range searching for minimum in an indexing-related context.

1 Introduction

Given a text string $T = t_1 \dots t_n$ and a pattern string $P = p_1 \dots p_m$, in the *pattern matching* problem [11] we wish to report all the occurrences of P in T. Its online counterpart, the *indexing* problem, is one of the most important paradigms in searching: the idea is to preprocess a text and construct a mechanism that will later provide answers to queries of the form "does a pattern P occur in the text" in time proportional to the length of the *pattern* rather than the text. In addition, if we want to return the occurrences themselves, the time will be proportional to the length of the pattern and the number of actual occurrences.

The *suffix tree* [15, 14, 7, 13] has proven to be an invaluable data structure for indexing, using $O(n)$ space, where n is the text length. Algorithms for the

* This work was supported by a German-Israel Foundation (G.I.F.) young scientists program research grant.

construction of a suffix tree enable $O(n)$ preprocess time when $|\Sigma|$ is constant (where Σ is the alphabet set), and $O(n \log \min(n, |\Sigma|))$ time when $|\Sigma|$ is not. In fact, the suffix tree can be constructed in linear time even for alphabets drawn from a polynomially-sized range, see [7].

The size of the alphabet also affects the query time of the suffix tree: given a pattern P of length m, we can find the set of all occurrences of P in T in $O(m \log \min(n, |\Sigma|) + tocc)$ time for unbounded alphabets, where $tocc$ is the actual number of occurrences of P in T, or accordingly, $O(m + tocc)$ time for constant-sized alphabets.

While the search for P yields an unsorted set of occurrences in T, some may overlap others: a specific location i in the text might participate in several different occurrences of P in T. However, sometimes only *non-overlapping* occurrences are of importance. Such requirement is of interest in fields such as pattern recognition, computational linguistics, speech processing, bio-molecular sequence analysis, code optimization and data compression [4]. For instance, we might want to compress a text by replacing each non-overlapping occurrence of a substring of it with a pointer to a single copy of the substring.

In the *string statistics* problem [4, 5], we are interested in finding the maximal *number* of non-overlapping occurrences of P in the entire text T. The solutions proposed in [4] and [5] use properties of periodicity in the text and pattern. However, the methods described there do not report the *actual occurrences* of the pattern. In this paper, we present a solution that returns the *maximal* (sorted by location) sequence of non-overlapping occurrences of P in T[1]. Furthermore, we generalize it such that it can return the maximal sequence of non-overlapping occurrences of P in some substring of T, denoted by start and end locations given alongside the pattern at query time.

In addition, we provide a solution to another problem that incorporates indexing with added location constraints: in the *successive list indexing* problem, we are given a list $L = \langle i_1, \ldots, i_\ell \rangle$ of locations in T together with P, and we wish to find the sequence of occurrences of P in T where the jth occurrence returned is the leftmost occurrence of P in T that occurs after the i_jth location (if such exists). Other kinds of proximity-related indexing variants (for instance, finding the single occurrence of the pattern that overlaps the i_jth location) can be solved by using exactly the same method. We also note that the definition of the matching can be generalized to pattern matching with errors and such [3], but we leave the discussion for the full version of this paper, and assume for the rest of the paper the common matching definition.

Solutions to both problems rely heavily on tools taken from the *computational geometry* area. In the *range searching* problem (see survey in [1]), which is common to this field, we are given a set S of n geometric objects (e.g. points) in a d-dimensional space, which we store in some data structure. When a query object Q (e.g. a hyper-rectangle $[a_1, b_1] \times \cdots \times [a_d, b_d]$) is given, we wish to

[1] Note that we discuss the indexing variant of this problem. If one would like to solve non-overlapping pattern matching, then one could use the simple greedy method discussed in Sect. 5.

return the result of some sort of query on a subset of the points, usually the subset $S \cap Q$. A popular variant of range searching is *range reporting*, in which we are asked to report all the points which are included in the query range $Q = [a_1, b_1] \times \cdots \times [a_d, b_d]$, i.e. the set $S \cap Q$ itself (see [2]).

While range reporting has been used before in several indexing-related papers (e.g. [9, 3, 8]), to the best of our knowledge, this is the first indexing-related work using a variant of *range searching for minimum* of Lenhof and Smid [12], itself a generalization of a problem presented by Gabow et al. [10]. In Lenhof and Smid's variant, we are given a set of n d-dimensional points, and query them with ranges of type $[a_1, b_1] \times \cdots \times [a_{d-1}, b_{d-1}] \times [a_d, \infty]$, wishing to find a single point in range with minimal dth coordinate. When $d = 2$, they obtain the following bounds: $O(n \log n \log\log n)$ expected preprocessing time, $O(n \log n)$ space, and $O(\log n)$ query time.

We modify the solution from [12] to work on a 2-*dimensional grid*, which suits our purposes. We find it more appropriate to call this variant the *range successor query on a grid* problem. We obtain the following bounds: $O(n \log n \log\log n)$ expected preprocessing time, $O(n \log n)$ space, and $O(\log\log n)$ worst-case query time.

The rest of this paper is organized as follows: in Sect. 2 we provide some formal definitions of our problems. In Sect. 3 we supply an outline of the method we use for the range successor query on a grid problem. In Sect. 4 we solve the successive list indexing problem. In Sect. 5 we finally solve the range non-overlapping indexing problem, and in Sect. 6 we present some concluding remarks.

1.1 Notations

For two integers $i \leq j$, denote by $[i, j]$ the set $\{i, \ldots, j\}$. For an integer u, denote by $[u]$ the set $[0, u-1]$.

Given a string S, denote by $|S|$ the length of S. An integer i is a *location* in S if $i = 1, \ldots, |S|$. Given a string $T = t_1 \ldots t_n$ (i.e. $|T| = n$, hereafter the *text*), a *suffix* of T is a string of the form $t_i \ldots t_n$, for some location i. Given another string $P = p_1 \ldots p_m$ (hereafter the *pattern*), a location i in T is an *occurrence* of P in T if $t_i \ldots t_{i+m-1} = p_1 \ldots p_m = P$. Two occurrences i, j of P in T are said to be *non-overlapping* if $|j - i| \geq m$. The *suffix tree* of T is essentially a compressed trie of the suffixes of T, used as a data structure to efficiently find the occurrences of P in T.

2 Problem Definitions

The *successive list indexing* problem is defined as follows:
Input: a text $T = t_1 \ldots t_n$ over alphabet Σ.
The text will be preprocessed to answer the following:
Query: a pattern $P = p_1 \ldots p_m$ over Σ, and a list $L = \langle i_1, \ldots, i_\ell \rangle$ of locations in T.
Output: the ℓ-length list of occurrences of P in T where the jth occurrence

is the leftmost (i.e. minimal) occurrence of P in T that appears after the i_jth location (if such exists).

A simpler version of this problem is the *successive indexing* problem that is defined as follows:
Input: a text $T = t_1 \dots t_n$ over alphabet Σ.
The text will be preprocessed to answer the following:
Query: a pattern $P = p_1 \dots p_m$ over Σ, and a location i in T.
Output: an occurrence $i' \geq i$ of P (i.e. $t_{i'} \dots t_{i'+m-1} = P$) in T for which i' is minimal (if such exists).

The *range non-overlapping indexing* problem is defined as follows:
Input: a text $T = t_1 \dots t_n$ over alphabet Σ.
The text will be preprocessed to answer the following:
Query: a pattern $P = p_1 \dots p_m$ over Σ, and two locations $i \leq j$ in T.
Output: an ascending sequence $L = \langle i_1, \dots, i_k \rangle$ of non-overlapping occurrences of P in T for which $i \leq i_1$ and $i_k \leq j$ (alternatively we can say we require L to be a subsequence of the sorted set $[i, j]$) and k is maximal. Formally, we require that for each $j = 1, \dots, k$, $t_{i_j} \dots t_{i_j+m-1} = P$ and that for each $j = 1, \dots, k-1$, $i_{j+1} - i_j \geq m$.

The (two-dimensional) *range successor query on a grid* problem is defined as follows:
Input: a set $S = \{(x_1, y_1), \dots, (x_n, y_n)\}$ of n points on an $[n] \times [n]$ grid.
Given this input, we will efficiently preprocess it to answer the following queries:
Query: a triplet (x', x'', y).
Output: a specific point $(x_i, y_i) \in S \cap ([x', x''] \times [y, n-1])$ whose y-coordinate (i.e. y_i) is minimal. In other words, (x_i, y_i) is the point with minimal value y_i corresponding to the following conditions:

1. $y_i \geq y$.
2. $x' \leq x_i \leq x''$.

3 Range Successor Query on a Grid

Both solutions for the successive list indexing and range non-overlapping indexing rely heavily on an efficient solution to the range successor query on a grid problem. As mentioned before, this problem, in its version where the points' coordinates are not on a grid (meaning, they are not necessarily integers and are not drawn from a restricted universe $[u]$), and for which the points can also be of dimension greater than 2, was solved by Lenhof and Smid [12]. In their definition, given n points in a d-dimensional space, the query object is a d-dimensional range $[a_1, b_1] \times \dots \times [a_{d-1}, b_{d-1}] \times [a_d, \infty]$ in which we wish to find the point having the minimal dth coordinate. They issued the problem with the name "range searching for minimum", which was used prior by Gabow et al. [10] to indicate the more particular problem in which the query object is of the form $[a_1, b_1] \times \dots \times [a_{d-1}, b_{d-1}] \times [-\infty, \infty]$. Again, the goal there is to find the point in the range having the minimal dth coordinate. As Lenhof and Smid's problem is actually the problem of finding the successor of a value, with added range

restrictions, we find it more appropriate to name it (in our context) the *range successor query on a grid* problem.

In the solution presented in [12], they used a rank space reduction in order to reduce the given point set in $\mathbb{R}^d$ to a point set on an $[n]^d$ grid. As a result, the query time for the two-dimensional case suffered from an additive $O(\log n)$ time. However, when we solve the problem on an $[n] \times [n]$ grid, we do not need the rank space reduction. Unfortunately, in [12] there is no complete analysis of the query time in absence of the rank space reduction. It can be shown that the query time in such a case is worst-case $O(\log \log n)$. We leave the full details for the full version, as it requires a complete description of the solution presented in [12].

We thus obtain the following:

Theorem 1. *The range successor query on an* $[n]\times[n]$ *grid problem can be solved with* $O(n \log n)$ *space and* $O(\log \log n)$ *query time, using* $O(n \log n \log \log n)$ *expected preprocess time.*

4 Successive List Indexing

We now present a solution for the successive indexing problem which applies a reduction to the range successor query problem. Later, we will explain how to generalize the solution for solving the successive list indexing problem.

Let $T = t_1 \dots t_n$ be a text over alphabet Σ. When given a pattern $P = p_1 \dots p_m$ over Σ and a location i in T, we wish to find the leftmost occurrence of P in T that still starts to the right of i. Formally, we wish to find the minimal $i' \geq i$ such that $t_{i'} \dots t_{i'+m-1} = P$, if such exists.

We first construct the suffix tree of T, denoted $\mathrm{ST}(T)$. In order to prevent the effect of unbounded alphabets on the suffix tree, we can present hashing, as depicted in the following:

Theorem 2. *There exists a* randomized *suffix tree, which can be constructed in* expected $O(n)$ *time (where* n *is the length of the text), and in which queries can be made in worst-case* $O(m + tocc)$ *time (where* m *is the length of the pattern, and* $tocc$ *is the actual number of occurrences of the pattern in text), for* general *alphabets.*

Proof. Note the construction and query times for constant size alphabets are $O(n)$ and $O(m+tocc)$ respectively. In addition, note that the number of children of any node in the suffix tree is bounded by both $n+1$ and $|\Sigma|$. Hence, we obtain a $\min(n+1, |\Sigma|)$ bound on the number of children of a given node. Thus, if for every node in the suffix tree we maintain pointers to its children in a balanced search tree, the multiplicative $O(\log \min(n, |\Sigma|))$ factor comes from the need to search or to insert elements to balanced search trees. Substituting this balanced search tree with a dynamic hash table (e.g. of Dietzfelbinger et al. [6], supporting worst-case $O(1)$ query time and amortized expected $O(1)$ insertion time), using as before the symbols of the alphabets associated with the edges as keys, would

eliminate that factor, thus giving us an *expected* $O(n)$ construction time, and worst-case $O(m + tocc)$ query time (worst-case, since in query-time we do not modify the tree and therefore do not insert elements to those hash tables). □

Note that besides the obvious disadvantage of introducing randomness, another disadvantage of the randomized suffix tree is the fact that now, given a node, the order of its children cannot be efficiently derived from the structure used to hold the pointers to them. As this order eventually determines the order of the leaves of the suffix tree, which is crucial to us during preprocess, suffix trees built throughout this paper will hold the pointers to the children of a given node in both a balanced search tree and a hash table. During query time, since the aforementioned order is of no importance to us, we will use the hash tables option to efficiently navigate through the tree.

4.1 Algorithm Outline

In the suffix tree, each leaf l is associated with a suffix of T and is therefore marked with an integer $y(l)$ which is the start location of that suffix. Assume we go over the leaves of $\mathrm{ST}(T)$ in a left-to-right manner linking them to create a linked list (by using a depth first search). Note that now if we traverse the list, we actually traverse the leaves according to the lexicographical order of the suffixes they are associated with. For a leaf l, let $x(l)$ be the position of l in the linked list. It immediately follows that $x(l)$ is the lexicographical rank of the suffix associated with l. Equivalently: If we lexicographically sort all suffixes of T in an ascending order, then the $x(l)$th suffix is the one associated with l. Setting $x(l)$ for each l can be done by going over the list, marking each leaf l with its position in the list.

When given a pattern $P = p_1 \dots p_m$, we can find all the occurrences of P in T, by traversing $\mathrm{ST}(T)$ from the root downwards according to the symbols in P, until we either conclude that P does not occur in T (in the case we got 'stuck' in the tree, figuratively speaking: this is the case where the next symbol of the pattern cannot be found in our current location in the tree), or that we conclude the traversal at a node v in $\mathrm{ST}(T)$. In the latter, all the leaves in the subtree rooted at v correspond to occurrences of P in T. Denote the subtree rooted at v as T_v. Hence the set $L' = \{y(l) \mid l \text{ is a leaf in } T_v\}$ is the set of all occurrences of P in T.

Note that for the node v mentioned above, the leaves of T_v appear consecutively in the linked list of leaves. Furthermore, since for each leaf l, $x(l)$ is its position in the list, the leaves of T_v form a range $[x(l'_v), x(l''_v)]$ (where l'_v and l''_v are the leftmost and rightmost leaves in T_v, respectively). It immediately follows that for a leaf l, l is a leaf in T_v iff $x(l) \in [x(l'_v), x(l''_v)]$. In other words: $x(l) \in [x(l'_v), x(l''_v)]$ iff P appears in T at location $y(l)$.

Consider the leaf f for which $x(f) \in [x(l'_v), x(l''_v)]$ and $y(f)$ is minimal such that $y(f) \geq i$. By the problem definition, $y(f)$ is exactly what we need to find and return. Now consider the set $\{(x(l), y(l)) \mid l \text{ is a leaf in } \mathrm{ST}(T)\}$. Clearly, this is a set of $n + 1$ points on an $[n + 1] \times [n + 1]$ grid. Since the point $(x(f), y(f))$

Algorithm 1: Successive indexing preprocess

Input: a text $T = t_1 \dots t_n$.

construct ST(T); `/* assume the field` $y(l)$ `is set for any leaf` l `by the suffix tree algorithm */`

traverse ST(T) and set the field $x(l)$ for each leaf l;

traverse ST(T) *using DFS* **:**

    foreach *node* u **do**

        store the values $x(l'_u)$ and $x(l''_u)$ in u; `/*` l'_u `and` l''_u `are the leftmost and rightmost leaves of` T_u`, respectively */`

preprocess the set $\{(x(l), y(l)) \mid l \text{ is a leaf in ST}(T)\}$ for range successor queries on an $[n+1] \times [n+1]$ grid;

Algorithm 2: Successive indexing query

Input: a pattern $P = p_1 \dots p_m$ and an integer $1 \leq i \leq n$.

traverse ST(T) *starting from the root, according to the symbols in* P **:**

    if *stuck* **then return** "no occurrences" **else** let v be the node we reached (if we stopped at a node) or the node immediately below the edge we are at (if we stopped on an edge);

if *the range successor query for* $(x(l'_v), x(l''_v), i)$ *yields a result* (x', y') **then**

    return y';

else return "no occurrence";

is exactly the y-axis successor of i in the range $[x(l'_v), x(l''_v)]$, we can find and return $y(f)$ by using a single range successor query.

The algorithm for the successive indexing problem thus immediately follows, and is presented as Algorithms 1 (preprocess) and 2 (query).

4.2 Analysis

We have obtained the following:

Theorem 3. *The successive indexing problem can by solved with* $O(n \log n)$ *storage and* $O(m + \log \log n)$ *query time, using* $O(n \log n \log \log n)$ *expected preprocess time.*

Proof. The correctness of the proposed algorithm follows immediately from the discussion above. Note that for the values $x(l), y(l)$ for each leaf l in ST(T), it holds that $x(l), y(l) \in [n+1]$. The space used is therefore:

1. $O(n)$ for the suffix tree itself.
2. $O(n \log n)$ for the data structure supporting range successor queries.

We conclude we use overall $O(n \log n)$ storage space.

The query time consists of:

1. $O(m)$ in order to find node v.

2. $O(\log \log n)$ time for the single range successor query.

Summing up, the query time is worst-case $O(m + \log \log n)$.

The preprocess time consists of:

1. $O(n \log \min(n, |\Sigma|))$ in order to construct the suffix tree with both a balanced search tree and a hash table in each node.
2. $O(n)$ for each traversal on the suffix tree.
3. $O(n \log n \log \log n)$ expected time for preprocessing in order to answer future range successor queries.

We conclude we use overall expected $O(n \log n \log \log n)$ time for preprocess. □

4.3 Solving the Successive List Indexing Problem

Note that after answering the query for some P and i, if we wish to answer this query for the same pattern P and a different location j, we can immediately perform the range successor query and return the result, since we have already found and thus know the x-axis range associated with P. Thus, we have obtained the following:

Theorem 4. *The successive list indexing problem can be solved with $O(n \log n)$ storage and $O(m+\ell \log \log n)$ query time, using $O(n \log n \log \log n)$ expected preprocess time.*

Proof. Given an ℓ-length list L of locations in T, we can use the solution for the successive indexing problem, but repeat the range successor query for every location given in the queried list. □

5 Range Non-overlapping Indexing

We now present a solution for the range non-overlapping indexing problem.

Let $T = t_1 \dots t_n$ be a text over alphabet Σ. When given a pattern $P = p_1 \dots p_m$ over Σ, and two locations $i \leq j$ in T, denote by L' the ascending sequence of locations in T where an occurrence of P in T starts. Denote by L'' the sequence of locations in T which (1) correspond to occurrences of P in T, and (2) are in the range $[i, j]$. Clearly, L'' is a subsequence of L'. We wish to find a subsequence $L = \langle i_1, \dots, i_k \rangle$ of L'' which corresponds to non-overlapping occurrences of P in T in the range $[i, j]$, with maximal k. Notice that L is a subsequence of L'' which is a subsequence of L'. Formally, we require that:

1. $i \leq i_1$.
2. $i_k \leq j$.
3. For each $d = 1, \dots, k$, $t_{i_d} \dots t_{i_d+m-1} = P$.
4. For each $d = 1, \dots, k-1$, $i_{d+1} - i_d \geq m$.

Consider the following greedy method for constructing L: we go over $t_i \dots t_{j+m-1}$ by using one of the linear pattern matching algorithms (for instance [11]) which scan the text and return occurrences of P in $t_i \dots t_{j+m-1}$ in an ascending order of positions. We choose the first occurrence the algorithm has outputted to be the first element in L. Note that the occurrence we have just chosen is the leftmost occurrence of P in $t_i \dots t_{j+m-1}$. We then proceed to choose every first occurrence outputted by the algorithm that does not overlap with the last occurrence we have chosen for L. It is easy to see that for the resulting sequence $L = \langle i_1, \dots, i_k \rangle$, it holds that for each $d = 2, \dots, k$, i_d is minimal such that $t_{i_d} \dots t_{i_d+m-1} = P$ and $i_d - i_{d-1} \geq m$.

Lemma 1. *The sequence L is a maximal-length sequence of non-overlapping occurrences of P in $t_i \dots t_{j+m-1}$.*

Proof. Recall that $|L| = k$ and assume by contradiction that L is not maximal, i.e. there is a sequence of non-overlapping occurrences of P in $t_i \dots t_{j+m-1}$, denoted H, such that $|H| \geq k+1$. For an integer d, denote by H_d the dth element of H, if such exists. Since the greedy method always chooses the leftmost non-overlapping occurrence to be included, we can say that for each $d = 1, \dots, k$, $i_d \leq H_d$. In particular, $i_k \leq H_k$. Since H is a sequence of non-overlapping occurrences, we notice that H_{k+1} does not overlap with H_k, and because $i_k \leq H_k$, it follows that H_{k+1} does not overlap with i_k as well. We conclude that the greedy method should have appended the occurrence H_{k+1} to L, which contradicts the fact that i_k is the last element of L. □

Denote $tocc = |L'|$, $k'' = |L''|$ and recall that $|L| = k$. The following lemma tells the relation between the three:

Lemma 2. *$tocc \geq k''$ and $k'' \leq m \cdot k$. Furthermore, there exists a text and a pattern for which $k'' = \Theta(m \cdot k)$.*

Proof. $tocc \geq k''$ since L'' is a subsequence of L'.

L is the maximal set of non-overlapping locations. For two consecutive elements i_d, i_{d+1} in L, if there exists an occurrence of P in T at location e such that $i_d < e < i_{d+1}$, then the occurrence at e certainly overlaps with the occurrence at i_d, otherwise the greedy method would have chosen e instead of i_{d+1}. Therefore, we charge every such occurrence e to i_d, in order to refrain from counting e twice (one time for i_d, and possibly another time for i_{d+1} if it also overlaps with it). Since for each i_d there are $m-1$ locations $i_d + \ell$ ($\ell = 1, \dots, m-1$) for which if P appears at, it would overlap with the occurrence at i_d, the lemma follows.

Finally, if the text is $T = a^n$ (the symbol a repeated n times), the pattern is $P = a^m$, and the query range is $[1, n]$, it is easy to see that $k'' = \Theta(m \cdot k)$ □

Assume we first index T by constructing the suffix tree of T, denoted $\mathrm{ST}(T)$. As described in Sect. 4, the suffix tree of T enables us to find all the occurrences of P in T. Therefore, a naive approach for solving the problem will be, when given P, to simply find all occurrences of P in T by using this method, sort them (thus obtaining the sequence L'), choose only those which are in the range

$[i, j]$, and then iterate through them, each time outputting the first location not overlapping with the last location outputted. However, it is clear the the time for such a method will be dependant on *tocc*, which can, as lemma 2 suggests, be $\Theta(m \cdot k)$, which is too large.

Another (slightly better) approach would be to transform the leaves of $\mathrm{ST}(T)$ to points on a grid as described in Sect. 4. Assume we have found the node v which is described there (by using the methods described there). Recall that $x(l) \in [x(l'_v), x(l''_v)]$ iff P appears in T at location $y(l)$. We can therefore recover the exact set of occurrences of P in T that are in the range $[i, j]$ by conducting a range reporting query (i.e. searching and reporting all the points in range) of the range $[x(l'_v), x(l''_v)] \times [i, j]$ (latest results of range reporting due to Alstrup et al. [2]). Again, we can sort them (thus obtaining L'') and then iterate through them, each time outputting the first location not overlapping with the last location outputted. However, it is clear that now we still have a dependency on k'', which could be $\Theta(m \cdot k)$. Note that the method of representing the lexicographic order of the suffixes of a text, and the locations in the text, as two axes of a grid, used in this paper, was used before by Ferragina [8]. The goal of [8] required performing range reporting queries on a grid, while we use range successor queries on a grid.

We resort therefore to using a similar method to that which was used to solve the successive indexing problem in Sect. 4: consider the leaf f for which $x(f) \in [x(l'_v), x(l''_v)]$, $y(f) \geq i$ and $y(f)$ is minimal. If $y(f) \leq j$, then $y(f)$ is the leftmost occurrence of P in $t_i \dots t_{j+m-1}$, so according to the greedy scheme, we can include it in L. It is clear that $y(f)$ is exactly the occurrence of P in T successive to i, subject to the requirement that $y(f) \leq j$. Suppose such f exists and therefore we included $y(f)$ in L. We now want to choose the leftmost occurrence of P in T in the range $[i, j]$ not overlapping with the occurrence we have just chosen. In other words: we wish to find a leaf l for which $x(l) \in [x(l'_v), x(l''_v)]$ and $y(l)$ is minimal such that $y(f) + m \leq y(l) \leq j$. Luckily, this is exactly the occurrence of P in T successive to $y(f) + m$, adding the constraint that the occurrence is less than or equal to j. Therefore, this can be solved also by querying for the y-axis successor of $y(f) + m$ in the x-axis range $[x(l'_v), x(l''_v)]$. We can repeat this process in order to obtain the sequence L as it was defined by the greedy method.

The algorithm for the range non-overlapping indexing problem immediately follows, and is described as Algorithms 3 (preprocess) and 4 (query).

5.1 Analysis

Theorem 5. *The range non-overlapping indexing problem can by solved with $O(n \log n)$ storage and $O(m + k \log \log n)$ query time (where k is the maximal number of non-overlapping occurrences of P in T, that are in the range $[i, j]$), using $O(n \log n \log \log n)$ expected preprocess time.*

Proof. The preprocess phase is identical to the one for the successive indexing problem and therefore the space and preprocess time analysis is omitted.

Algorithm 3: Range non-overlapping indexing preprocess

Input: a text $T = t_1 \dots t_n$

construct $\mathrm{ST}(T)$; `/* assume the field` $y(l)$ `is set for any leaf` l `by the suffix tree algorithm */`

traverse $\mathrm{ST}(T)$ and set the field $x(l)$ for each leaf l;

traverse $\mathrm{ST}(T)$ *using DFS* **:**

    foreach *node* u **do**

        store the values $x(l'_u)$ and $x(l''_u)$ in u; `/*` l'_u `and` l''_u `are the leftmost and rightmost leaves of` T_u`, respectively */`

preprocess the set $\{(x(l), y(l)) \mid l \text{ is a leaf in } \mathrm{ST}(T)\}$ for range successor queries on an $[n+1] \times [n+1]$ grid;

Algorithm 4: Range non-overlapping indexing query

Input: a pattern $P = p_1 \dots p_m$, and two integers $1 \leq i \leq j \leq n$

let L be the empty sequence;

traverse $\mathrm{ST}(T)$ *starting from the root, according to the symbols in* P **:**

    if *'stuck'* **then return** "no occurrences";

    else let v be the node we reached (if we stopped at a node) or the node immediately below the edge we are at (if we stopped on an edge);

$y \leftarrow i$;

while *the range successor query for* $(x(l'_v), x(l''_v), y)$ *yields a result* (x', y')*, for which* $y' \leq j$ **do**

    append y' to L;

    $y \leftarrow y' + m$;

return L;

The query time consists of:

1. $O(m)$ in order to find node v.
2. $O(\log \log n)$ time for a successor query to find each element of L, therefore overall $O(k \log \log n)$ for all k non-overlapping occurrences.

We conclude we use overall worst-case $O(m + k \log \log n)$ time. □

6 conclusions

We have presented solutions for the successive list indexing problem, and the range non-overlapping indexing problems, by using a tool from computational geometry — range successor queries on a grid, which, to our best knowledge, has not been used before in this context. It is conceivable that more indexing problems can be solved by using the tool of range successor queries on a grid.

References

1. P. Agarwal and J. Erickson. Geometric range searching and its relatives, 1999.

2. S. Alstrup, G. S. Brodal, and T. Rauhe. New data structures for orthogonal range searching. In *IEEE Symposium on Foundations of Computer Science*, pages 198–207, 2000.
3. A. Amir, D. Keselman, G. M. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh. Text indexing and dictionary matching with one error. *J. Algorithms*, 37(2):309–325, 2000.
4. A. Apostolico and F. P. Preparata. Data structures and algorithms for the string statistics problem. *Algorithmica*, 15(5):481–494, 1996.
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 *ICALP '02: Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, pages 728–739, London, UK, 2002. Springer-Verlag.
6. M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. M. auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. volume 23, pages 738–761, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.
7. M. Farach. Optimal suffix tree construction with large alphabets. In *FOCS '97: Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS '97)*, page 137, Washington, DC, USA, 1997. IEEE Computer Society.
8. P. Ferragina. Dynamic text indexing under string updates. *J. Algorithms*, 22(2):296–328, 1997.
9. P. Ferragina, S. Muthukrishnan, and M. de Berg. Multi-method dispatching: A geometric approach with applications to string matching problems. In *STOC*, pages 483–491, 1999.
10. H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 135–143, New York, NY, USA, 1984. ACM Press.
11. D. Knuth, J. H. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.
12. H.-P. Lenhof and M. Smid. Using persistent data structures for adding range restrictions to searching problems. *RAIRO Theoretical Informatics and Applications*, 28:25–49, 1994.
13. E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.
14. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
15. P. Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory*, pages 1–11. IEEE, 1973.