Persistency in Suffix Trees with Applications to String Interval Problems

Tsvi Kopelowitz[1], Moshe Lewenstein[2,⋆], and Ely Porat[2,⋆⋆]

[1] Weizmann Institute of Science Rehovot, Israel
[2] Bar-Ilan University, Ramat-Gan, Israel

Abstract. The suffix tree has proven to be an invaluable indexing data structure, which is widely used as a building block in many applications. We study the problem of making a suffix tree persistent. Specifically, consider a streamed text T where characters are prepended to the beginning of the text. The suffix tree is updated for each character prepended. We wish to allow access to any previous version of the suffix tree. While it is possible to support basic persistence for suffix trees using classical persistence techniques, some applications which can make use of this persistency cannot be solved efficiently using these techniques alone.

A collection of such problems is that of queries on string intervals of the text indexed by the suffix tree. In other words, if the text $T = t_1...t_n$ is indexed, one may want to answer different queries on string intervals, $t_i...t_j$, of the text. These types of problems are known as position-restricted and contain querying, reporting, rank, selection etc. Persistency can be utilized to obtain solutions for these problems on prefixes of the text, by solving these problems on previous versions of the suffix tree. However, for substrings it is not sufficient to use the standard persistency.

We propose more sophisticated persistent techniques which yield solutions for position-restricted querying, reporting, rank, and selection problems.

1 Introduction

Text indexing is one of the most important paradigms in searching. The idea is to preprocess a text $T = t_1 \cdots t_n$ over alphabet Σ and construct a mechanism that will later provide answers to queries of the form "report all of the occurrences of a pattern P in the text" in time proportional to the size of the *pattern* and output, rather than the size of the text. The suffix tree [10,14,16,17] has proven to be an invaluable data structure for indexing. It is also considered a building block for various other indexing and non-indexing problems.

Some of the suffix tree constructions work in the *online* model [16,17], in which one maintains a suffix tree for a text that arrives character by character, and at any given time we might receive a pattern query. For simplicity, we assume that

⋆ This research was partially supported by the ISF (grant no. 1848/04).
⋆⋆ This research was partially supported by the BSF (grant no. 2006334) and the ISF grant (grant no. 1484/08).

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 67–80, 2011.

the text arrives from the end towards the beginning. Otherwise a single character added at the end of the text can impose a linear number of changes to the suffix tree. Of course, if the text arrives from beginning to end we can view the text in reversed form and then a queried pattern is reversed as well in order to obtain the correct results. The best currently known results for the online suffix tree are an $O(\log |\Sigma|)$ amortized time per character by Weiner in [17], and $O(\log n)$ worst case per character by Amir et.al in [2]. We also note that for constant-size alphabets there is a different indexing structure by Amir and Nor [3].

Data structures which have the ability to allow access to previous versions of themselves over the updates are known as *persistent data structures* [5,8,9]. For a good survey see [12]. We focus on two types of persistent data structures. The first type is *fully persistent data structures*, in which an update can be made to any version of the data structure. In this type, one can imagine a tree of versions of the data structure as update operations are performed on various versions. The second type is known as *partially persistent data structures*, in which an update can be made only to the last version created. In this type, one can imagine a list of versions of the data structure as update operations are performed only on the tail of the list. In Section 2 we will provide a quick review on some of the known results in this field which we will later use.

To the best of our knowledge persistent suffix trees have not been considered before. Nevertheless, since suffix trees have constant indegree it follows that one can make suffix trees persistent using the result of [8]. However, this persistency is useful solely for navigation purposes, which is sufficient for various standard applications, e.g. queries of the sort "report all of the times in which a specific stock has a series of consecutive values in the stock market, before last March". More sophisticated queries cannot be answered with navigational data on the current text alone.

One subset of problems that we focus on is string interval problems, a.k.a. position restricted problems. Here one has a suffix tree for the full text $T = t_1 \cdots t_n$ but is interested in queries that are narrowed down onto an interval $t_i \cdots t_j$. One problem is known as position restricted indexing, see [13], position restricted reporting, where one desires to report all matches within an interval of the text, position restricted rank, where one desires to know the rank of a given pattern within the interval of the text, and position restricted select, where one desires to find the i'th appearance of a given pattern. The intuition for using a persistent suffix tree for these type of problems is that by accessing the version of the suffix tree just after t_i was added, one may reduce the problem to searching within the prefix of $t_i \cdots t_n$ of length $j - i + 1$.

Unfortunately, the persistent suffix tree on its own does not suffice for efficient solutions for these problems. This happens because versions of the data structure provide bounds for one side of the desired interval query, but not both. Hence, we need to provide a persistent mechanism which supplies the capability for answering the different queries for position restricted problems. We do this by providing a general framework solution, and then show how each of the above applications can be solved using this general framework.

The most natural problem that our general framework solves is the position restricted indexing (PRI) problem. In this problem we wish to preprocess the text $T = t_1 \cdots t_n$, to support subsequent queries of the form "Given a pattern $P = p_1 \cdots p_m$ and two indexes i, j report all of the occurrences of P in $t_i \cdots t_j$". This PRI problem is a very natural one, and was introduced by Mäkinen and Navarro in [13]. PRI has also been addressed by Chien et al [7] where some connections between PRI and range searching in 2D are shown. Recently, Bille and Gørtz in [4] provided a solution for the PRI problem using $O(n \log n)$ preprocessing time, $O(n \log^{\epsilon} n)$ space (for any constant $\epsilon > 0$), and achieve optimal query time of $O(m + occ_{i,j})$ (where $occ_{i,j}$ is the size of the output).

We show in this paper, using the notion of persistency and our general framework, how one can solve PRI using $O(n \log n)$ preprocessing time, $O(n \log^{\epsilon} n)$ space (for any constant $\epsilon > 0$), and achieve query time of $O(m + \log\log|\Sigma| + occ_{i,j})$. For constant size alphabets we achieve the same complexities as those of Bille and Gørtz in [4], however, for general alphabets their solution is more efficient. Nevertheless, we choose to demonstrate our solution to this application as it provides an easier understanding of the use of our general framework, which later will allow us to solve other applications efficiently.

We discuss several other problems for which our general framework can help to achieve better query time from what is currently known. For the counting version of PRI (where we need to count the number of occurrences of P in $t_i \cdots t_j$) we obtain a data structure using $O(n \log n / \log\log n)$ space data structure which answers queries in $O(m + \log\log|\Sigma|)$ time. This is comparable with the results of [13] which by using $n + o(n)$ space can answer queries in $O(m + \log n)$ time. We note that while the authors in [13] claim that they can construct a data structure achieving $O(m + \log\log n)$ query time, using $O(n \log^{\epsilon} n)$ space, is based on an (unproven) claim that the data structure of [1] can answer 2D orthogonal range counting queries in $O(\log\log n)$ time on a grid. However, this cannot be true as that would contradict the lower bounds of Patrascu [15]. In addition, our solution for the counting version improves on the substring rank (SR) problem. In the *substring rank (SR) problem* we need to preprocess T for a substring rank query: given a pattern P and an integer k return the number of occurrences of P in $t_1 \cdots t_k$. The SR problem is a special case of the counting version of PRI since we can answer it using by setting $i = 1$ and $j = k - m + 1$. Thus the bounds of the counting version of PRI apply to SR as well.

Finally, we address the *substring selection problem* where we wish to preprocess T for a substring selection query: given a pattern P and an integer k locate the k^{th} occurrence of P in T. This problem was presented by Mäkinen and Navarro [13], where the authors construct a data structure that requires $O(n\ell \log_n |\Sigma|)$ space where ℓ is an upper bound on the size of the queried patterns, and answer queries in $O(m \log_{\log n} |\Sigma|)$ time. Our data structure requires $O(n \log n / \log\log n)$ space and can answer queries in optimal $O(m)$ time. Furthermore, we do not require any bound on the pattern length to be known in advance.

Our paper is organized as follows. In Section 2 we provide some definitions and preliminaries, including a quick review on some of the known persistent data structures which we will later use. In Section 3 we describe the persistent suffix tree. Then, in Section 4 we discuss the general framework used to solve the applications which we follow up on in Section 5.

2 Definitions and Preliminaries

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*) where for every location i $t_i \in \Sigma$ (hereafter the alphabet), 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 $t_i \ldots t_{i+m-1} = p_1 \ldots p_m = P$. The suffix tree of a string T is denoted by $ST(T)$.

We would like to assume that $|\Sigma| \leq n$. If this is not the case we can use a hash function in order to reduce Σ to a new alphabet Σ_T such that $|\Sigma_T| \leq n$ as there are at most n different characters in T. However, this can be done efficiently only if we assume that the subset of Σ which we use in T is known in advance. While this is true in the off-line (static) model of the text, it might not be true in the online model.

The suffix tree data structure is comprised of three different types of structures:

1. Tree Structure - The nodes and edges of the tree.
2. Text Structure - The label for each edge in the tree needs to be maintained. This can be costly, therefore, the suffix tree maintains for each edge a pointer into a substring of the text corresponding to the label, instead of explicitly maintaining it.
3. Navigation Structure - Each node in the suffix tree can have up to $|\Sigma|$ children, each one corresponding to one of the characters in Σ. Given a character $\sigma \in \Sigma$ the navigation structure allows us to quickly decide which of the edges to the node's children, if any, correspond to σ. There are two main approaches to solve this problem. The first is using a *pointer array* for each node, where location σ in the array corresponds to the appropriate outgoing edge. While this could induce a cost of $O(|\Sigma|)$ space per node, one can use hash functions in order to reduce the space to be linear in the number of children of each node. However, in an online setting this would require a dynamic hash-function which needs to be persistent, and we are not aware of any good solutions for this problem. The second option is using a balanced search tree over the children of every node, where the order is determined by the appropriate character. This will induce an extra $O(\log |\Sigma|)$ time per node encountered when traversing down the suffix tree.

Note that in the static setting one can maintain for each node two pointers to the beginning and end of the sub-list of leaves which are in the node's subtree. This

task is too difficult in the online setting. The main use of these pointers is to allow reporting the occurrences of a queried pattern. This happens once its appropriate node is located in time proportional to the size of the output (by scanning the list). However, we can overcome this in the online setting by performing a scan of the subtree of the node and outputting the leaves we encounter. This suffices as the size of the subtree is linear in the number of leaves of the subtree (since all of the inner nodes have more than one child).

2.1 Some Persistent Data Structures

Fully Persistent Arrays. Consider an array A, with the following operations:

1. $Store(v, i, x)$ - Store the information x in the i^{th} location of the version named v of A. This also returns v' which is the name of the new version of A.
2. $Access(v, i)$ - Return the information in the i^{th} location of the version named v of A.

Dietz in [8] presented a data structure (DDS for short) that performs *Store* operations in $O(\log \log m)$ expected amortized time, where m is the number of *Store* operations performed so far, and *Access* operations in $O(\log \log m)$ worst-case time. Unfortunately, it is unknown if improved bounds exist for the partially persistent version.

Fully and Partially Persistent Data Structures of Bounded In-degree. The task of making any data structure in which each node has a constant in-degree (meaning the number of pointers to any node is $O(1)$) fully persistent was solved by Dietz et. al. [9] with an overhead of amortized $O(1)$ space and time per operation. The partially persistent version was solved by Brodal [5] with a worst-case overhead of $O(1)$ per operation. Note that in order to be able to use this in a rooted tree with a non constant fan-out, one cannot maintain pointers to parents.

Fully and Partially Persistent Balanced Search Trees. Dietz et. al. in [9] show how one can support a fully persistent red-black tree over n nodes where each update (insertion or deletion) costs $O(\log n)$ worst-case time, and $O(1)$ worst-case space overhead.

2.2 Problems

Problem 1. **PRI-Report** Preprocess text $T = t_1 \cdots t_n$, such that given subsequent queries of the form $PRI - Report(P = p_1 \cdots p_m, i, j)$, report all of the occurrences of P in $t_i \cdots t_j$.

Problem 2. **PRI-Count** Preprocess text $T = t_1 \cdots t_n$, such that given subsequent queries of the form $PRI - Count(P = p_1 \cdots p_m, i, j)$, return the number of occurrences of P in $t_i \cdots t_j$.

Problem 3. **SSR** Preprocess text $T = t_1 \cdots t_n$, such that given subsequent queries of the form $SSR(P = p_1 \cdots p_m, k)$, return the number of occurrences of P in $t_1 \cdots t_k$.

Problem 4. **SSS** Preprocess text $T = t_1 \cdots t_n$, such that given subsequent queries of the form $SSS(P, k)$, return the k^{th} occurrence of P in T.

3 The Persistent Suffix Tree

In this section we briefly explain how to convert a suffix tree to be persistent in the online text model, where characters are prepended to the beginning of the text. Specifically, in a persistent suffix tree for $T = t_1 \cdots t_n$ we wish to be able to access the suffix tree of $t_i \cdots t_n$ for every $1 \leq i \leq n$.

Consider $ST(T)$ and $ST(\sigma T)$ for some $\sigma \in \Sigma$. It is a well know fact that the amount of tree structure changed or added in the transition from $ST(T)$ to $ST(\sigma T)$ is constant [2] [17]. Specifically, the new leaf corresponding to the new suffix is added, and in addition, a new node might be inserted into an existing edge in order to insert the new leaf's parent (if it does not already exist). The process of locating these changes can cost either amortized $O(\log |\Sigma|)$ [17], or worst case $O(\log n)$ [2]. We use the result of Brodal [5] in order to obtain partial persistency. Dealing with the text structure is standard. Thus the remaining task is that of maintaining the navigation structure, which depends upon implementation.

3.1 Using Pointer Arrays

If the pointer array solution is implemented, one can use the result by Dietz [8] in order to obtain an expected amortized overhead of $O(\log \log n)$ per update, and a worst case $O(\log \log n)$ overhead per node encountered when traversing down the suffix tree. While one might think that the number of changes to any pointer array is bounded by $|\Sigma|$ due to each pointer changing at most once, this is not the case as the pointer from, say, a node u corresponding to character σ can change many times when internal nodes are added. Thus the query time would be $O((m + occ) \log \log n)$. There are two ways to further reduce the query time. The first is by noting that when traversing the subtree corresponding to the queried pattern one can use the tree structure (and not the navigation structure) in order to scan the tree using, say, a post-order search. The second is using the techniques we discuss later in Section 4.2 which will allow us to reduce the overhead to $O(\log \log |\Sigma|)$. However, this is under the assumption that $\Sigma = \{1, 2, 3, \cdots, |\Sigma|\}$ to avoid the need of using a persistent dynamic hash function. Thus, the query time can be reduced to $O(m \log \log |\Sigma| + occ)$, and the following theorem has been established.

Theorem 5. *A persistent suffix tree can be maintained such that the cost of prepending a new character suffers an additive overhead of* $O(\log \log |\Sigma|)$ *expected amortized time and* $O(1)$ *worst-case space, and indexing queries can be answered in time* $O(m \log \log |\Sigma| + occ)$.

3.2 Using Balanced Search Trees

If the choice of implementation is to use a balanced search tree for the navigation data, one can use the results of Dietz et.al. [9] so that each update will cost an additional $O(\log|\Sigma|)$ time, and the traversal will suffer from an $O(\log|\Sigma|)$ time worst-case overhead per node. Thus, the query time would be $O(m\log|\Sigma|+occ)$ time.

Theorem 6. *A persistent suffix tree can be maintained such that the cost of prepending a new character suffers an additive overhead of* $O(\log|\Sigma|)$ *expected amortized time and* $O(1)$ *worst-case space, and indexing queries can be answered in time* $O(m\log|\Sigma|+occ)$.

4 The General Framework

While in some applications that text is static, it is useful to treat each text location as a timestamp. Each timestamp will have its own version of the suffix tree using persistency techniques. More precisely, for text location τ we have a data structure similar to the suffix tree containing only the suffixes of $t_\tau\cdots t_n$. Also, time is defined in reverse - so first create the version at time n and move backwards towards time 1. However, the focus here is on applications that a given query is confined to the substring $t_i\cdots t_j$ and there is a need for additional tools so that the version of the suffix tree at time stamp i can answer the queries for every possible j. Unfortunately, using the persistent suffix tree from the previous section does not suffice for answering such queries. The problem is that we would like the running time to depend only on the output in the restricted range given in the query, while the persistent suffix tree is only able to efficiently filter the locations which appear before i. Using the persistent suffix tree we still need to filter all of the locations after j.

We show a general framework of a data structure that for some height h constructs a data structure which implements a persistent version of the first h levels of the suffix tree. The choice of h is application dependent, but generally speaking we do not wish for h to be too large as the space used will depend on it. Nevertheless, the filtering process for all locations after j will be fast for all traversals in the suffix tree which end in a node of depth at most h using this persistent data structure.

4.1 Snapshots

We define the notion of a *snapshot* for a node u in the suffix tree of T at time τ. This snapshot is denoted by u^τ, and contains the following information:

- A pointer array A_u^τ of $|\Sigma|$ pointers to the at most $|\Sigma|$ children of u^τ which are the children of u in the version of the suffix tree at time τ. The pointer at location σ in A_u^τ will direct to the correct snapshot of child v of u^τ such that the label of the edge (u,v) begins with σ. Of course, if no such edge exists (whether it does not exist in the suffix tree of T, or it does not exist yet in the current timestamp) the pointer will be a null pointer.

- A pointer to the previous snapshot of u. This chain of pointers is called the snapshot list of u, or $SL(u)$.
- The timestamp of the snapshot τ.

For sake of simplicity, a conceptual timestamp $n+1$ is added. This timestamp has a snapshot for every node of distance at least h from the root in the suffix tree of T, in which the pointer array is all null pointers. We use this timestamp in order to have a so called first snapshot of every node. However, in order to save space we do not maintain an array for these snapshots (which would all be null pointers as at timestamp $n+1$ the suffix tree is empty), and instead we use a bit to indicate that these conceptual pointer are all null.

The only nodes that change between version $\tau+1$ of the suffix tree and version τ of the suffix tree are nodes on the path from the root of the suffix tree to the leaf corresponding to the suffix at location τ. Since only the first h nodes on this path are of interest, there is no need to create snapshots for the entire path. Furthermore, for any node u not on the aforementioned path we have that $u^\tau = u^{\tau+1}$ and hence, a new snapshot is not created for such nodes. Therefore, each timestamp induces at most h snapshots. Say that node u is *stamped* at time τ if u^τ is created.

Lemma 7. *For every node u which is stamped at time τ there exists at most one child v that is also stamped at time τ. Therefore, the difference between A_u^τ and $A_u^{\tau+1}$ is only at location σ which corresponds to the edge (u,v).*

The time required to create each snapshot other than the $n+1$ snapshots is $O(|\Sigma|)$ by copying the pointer array. Each of the n timestamps creates at most h snapshots, and so the total construction time for those timestamps is $O(n|\Sigma|h)$.

Navigation. Navigating down the suffix tree at timestamp i is done as usual through the pointer arrays, and thus the cost of locating the node corresponding to P is $O(m)$.

4.2 Using Persistent Arrays

Consider all of the different snapshots of a node u: $u^{\tau_1}, u^{\tau_2}, ..., u^{\tau_t}, u^{n+1}$, and their corresponding pointer arrays $A_u^{\tau_1}, A_u^{\tau_2}, ..., A_u^{\tau_t}, A_u^{n+1}$. Assume that there exists a positive integer k such that $t = k \cdot |\Sigma|$. It will be shown later how to deal with the case where t is not a multiple of $|\Sigma|$. Thus one can divide the snapshots of u to k lists, each of $|\Sigma|$ consecutive snapshots, and in addition the very last snapshot u^{n+1}. Again, for simplicity, assume that $k=1$, and thus $t = |\Sigma|$. For larger k the process is a repetition of this simpler case k times.

A pointer array is only created for u^{τ_1} and u^{n+1} (using 1 bit). For the other versions of u one can use the data structure of Dietz [8] (denoted by DDS) which was briefly mentioned in Section 2. A bit is used within each snapshot to indicate if it is a *full* snapshot with a complete pointer array, or an *incomplete* snapshot using the DDS. Note that while the pointer array of u^{n+1} is implemented via

a bit to indicate that it is all null pointers, the data structure of Dietz can be adjusted to still work in this situation (the details are standard, but require an exposition of the data structure, and so we choose to omit it here).

The DDS is used on the pointer array of u^{n+1} (which as we mentioned does not actually exist). Next, iterate over u^{τ_x} which is a version of u in $u^{\tau_2}, ..., u^{\tau_t}$, starting with $x = t$ and ending with $x = 2$. Next, let σ be the character leading u^{τ_x} to its appropriate child v^{τ_x}. We use DDS to perform $Store(x, \sigma, (u^{\tau_x}, v^{\tau_x}))$ inserting the appropriate edge/pointer into the array. u^{τ_x} is given a pointer to the DDS, its current timestamp τ_x, the value $|\Sigma| - x$ which we call the relative timestamp of u^{τ_x}, and finally u^{τ_x} is added to $SL(U)$.

If $t < |\Sigma|$, the DDS is still used, but a pointer array is never fully constructed. If there is no integer k such that $t = k \cdot |\Sigma|$, and $t > |\Sigma|$, then the tail of snapshots use the DDS on the last fully constructed pointer array.

As before, each suffix tree version induces h snapshots. Each incomplete snapshot, the DDS uses $O(\log\log|\Sigma|)$ time. Each full snapshot, except for snapshots at time $n+1$, requires $O(|\Sigma|)$ time which amortizes to $O(1)$ time over the previous $O(\Sigma)$ incomplete snapshots. Thus, the total time spent is $O(nh \log\log|\Sigma|)$, and the space used is $O(nh)$.

Navigation. Navigation is basically done as before; the only difference is the method of traversing down the current version of the suffix tree without complete pointer arrays (for the full snapshots simply use the array). This is done using the DDS. If the navigation reached u^{τ}, whose relative timestamp is x, and needs to continue traversing with σ, then use $Access(x, \sigma)$ on the DDS of u^{τ} in order to obtain the correct pointer. This implies that each traversal through a node in the current version of the suffix tree will take $O(\log\log|\Sigma|)$ time. So the cost of locating the node corresponding to P is $O(m \log\log|\Sigma|)$.

4.3 Renaming

First a solution is provided for navigating with patterns of length which is a multiple of $\log\log|\Sigma|$. The more general case is considered in the following subsection. Given T over alphabet Σ, construct a new alphabet $\Sigma' \subseteq \Sigma^{\log\log|\Sigma|}$ by taking every substring of T of size $\log\log|\Sigma|$ and renaming it. The renaming scheme needs to maintain the lexicographical order between the substrings. For this, construct a trie for all of the substrings of T of size $\log\log|\Sigma|$, and then label each leaf. The labels should maintain the order defined by the leaves of the trie. Thus, the labels preserve the lexicographical ordering of the substrings corresponding to them. The construction of this trie takes $O(n \log\log|\Sigma|)$ time.

Next, construct $\log\log|\Sigma|$ new text strings, named $T_1, T_2, ..., T_{\log\log|\Sigma|}$ as follows. For each $1 \leq i \leq \log\log|\Sigma|$, the new text T_i is a text over alphabet Σ' where the j^{th} character in T_i is the renamed label corresponding to the substring $t_{i+(j-1)\log\log|\Sigma|} \cdots t_{i+j\log\log|\Sigma|-1}$. In other words, T_i is the renamed version of $t_i t_{i+1} \cdots .t_n$, removing any tail of characters that might remain, as $n - i + 1$ might not be a multiple of $\log\log|\Sigma|$. Note that only the first $\log\log|\Sigma|$ suffixes

of T are being renamed, as the renamed versions of any other suffix (not one of the first $\log\log|\Sigma|$) is a proper suffix of one of the renamed suffixes. Next, construct the text $T' = T_1\$_1T_2\$_2...T_{\log\log|\Sigma|}\$$, and use the previous solution with persistent pointer arrays on this text, but only for height $h' = \frac{h}{\log\log|\Sigma|}$ as each renamed character corresponds to $\log\log|\Sigma|$ original characters (and we wish to only maintain this data structure for nodes in the original tree of height at most h).

It is important to note that while the order of the timestamps defined by the original text become scrambled, it is still possible to maintain the correct order of timestamps by first constructing the suffix tree for T', and then creating snapshots for paths by order of the original timestamps rather than the order defined by T'.

The construction time is now $O(n\log\log|\Sigma|)$ time for the renaming, and $O(nh') = O(nh/\log\log|\Sigma|)$ for the solution using the persistent pointer arrays for a total of $O(n(\log\log|\Sigma| + \frac{h}{\log\log|\Sigma|}))$.

Navigation. Recall the assumption that m is a multiple of $\log\log|\Sigma|$. So, one can use the renaming trie in order to obtain a new pattern P' over Σ' of size $\frac{m}{\log\log|\Sigma|}$, and then the node corresponding to P' in $O(m'\log\log|\Sigma'|) = O(\frac{m\log\log|\Sigma'|}{\log\log|\Sigma|})$ time, which is $O(m)$ as:

$$\log\log|\Sigma'| \leq \log\log(|\Sigma|^{\log\log|\Sigma|}) = \log\log|\Sigma| + \log\log\log|\Sigma| = O(\log\log|\Sigma|).$$

4.4 Renaming for Any Size Pattern

For the case in which m is not a multiple of $\log\log|\Sigma|$, assume that $m \equiv k$ modulo $\log\log|\Sigma|$ where $0 < k < \log\log|\Sigma|$. The previous renaming scheme can still be used on the first $m-k$ characters of P in order to obtain a pattern P' over Σ'. Then, one can locate the node corresponding to P' in $O(m)$ time. What is left to be done is to provide another renaming scheme in order to deal with the last k characters. To this end, the renaming process is done for every length from 1 to $\log\log|\Sigma| - 1$, by using a renaming trie for each possible length.

For every node u in the suffix tree of T', $\log\log|\Sigma| - 1$ pointer arrays are maintained, each corresponding to one of the renaming schemes for each possible length, and each of length corresponding to the length of that specific renaming scheme. So for the renaming scheme for every length $x < \log\log|\Sigma|$, node u uses a pointer array of size $|\Sigma|^x$. Each location in the pointer array corresponds to an edge corresponding to the appropriate renamed character in the renaming scheme of length x. It is crucial to note that this is only done for nodes u in T', and not recursively. Each of the $\log\log|\Sigma| - 1$ pointer arrays then goes through the process of becoming persistent using the same techniques as above.

The additional construction time is $O(nh'\log\log|\Sigma|)$, which combines together with the previous section to $O(n(\log\log|\Sigma| + h))$ time for construction. the space is $O(nh'\log\log|\Sigma|) = O(nh)$.

Navigation. As for the navigation time, if $m \geq \log\log|\Sigma|$, the time will still be $O(m)$ as the extra $O(\log\log|\Sigma|)$ cost induced by having to move to a new renaming scheme and locating the appropriate snapshot can be amortized over previous work done in the suffix tree for T'. However, if $m < \log\log|\Sigma|$, the cost of searching the correct snapshot cannot be amortized, and so the cost will be $O(m-k+\log\log(|\Sigma|^k)) = O(m+\log\log|\Sigma|)$ time.

5 Applications

In this section we will see how to use the general framework from Section 4 in order to solve some problems that fit the model.

5.1 PRI-Report

As mentioned by Mäkinen and Navarro [13], one can obtain a data structure which supports PRI-Report by using the suffix tree and the data structure of Alstrup et.al. [1]. This provides a data structure that uses $O(n\log n)$ preprocessing time with $O(n\log^{\epsilon} n)$ space (for any constant $\epsilon > 0$), and achieves query time of $O(m+\log\log n+occ_{i,j})$. Thus, by setting $h = \log\log n$ one can answer PRI-Report as follows.

If $m \geq \log\log n$ then by using the data structure from [13], the query time is $O(m+occ_{i,j})$, which is optimal. If $m < \log\log n$ then navigate through the data structure from Section 4 with P on version i of the suffix tree till a snapshot u^{τ} is reached, where u is the node corresponding to P. Note that $i \leq \tau$. The next procedure is based on the following lemma.

Lemma 8. *Let u be a node in the suffix tree of T corresponding to P, and let $u^{\tau_1}, u^{\tau_2}, ..., u^{\tau_t}, u^{n+1}$ be the snapshots of u. Then P appears in T only at locations $\tau_1, \tau_2, ..., \tau_t$.*

Proof. Each of the snapshots is created due to P being at the location of that timestamp. □

Given that the snapshot list for u is ordered by the timestamps, one can scan the list starting at u^{τ} till a snapshot $u^{\tau'}$ is reached where $\tau' > j$. Due to Lemma 8 every snapshot encountered by the scan except for the last one corresponds to an occurrence of P. Thus each location in the output costs $O(1)$ time to output after u is located. This provides the following:

Theorem 9. *The PRI-Report problem can be solved using $O(n\log^{\epsilon} n)$ space (for any constant $\epsilon > 0$) and $O(n\log n)$ preprocessing time, with $O(m+\log\log|\Sigma|+occ_{i,j})$ query time.*

5.2 PRI-Count

In the counting version (PRI-Count), one wishes to report only the number of occurrences of P in $t_i \cdots t_j$, without listing the occurrences. While this can be

solved in $O(m + \log\log|\Sigma| + occ_{i,j})$ time using the solution to the reporting version, this can be wasteful if the output size is fairly large. Instead, the following data structure which is based on ideas similar to the reporting scenario is presented, which manages to avoid the cost of the output size all together.

If the pattern happens to be large enough ($m \geq \log / \log\log n$) the following solution can be used. Given a PRI-Count query, the suffix tree is first used in order to locate the node u corresponding to P. This node covers a consecutive range of suffixes, sorted by their lexicographical ordering (this can be viewed as a consecutive sub-array of the suffix array). Locating this range (l, r) can be done offline in linear time per every node in the suffix tree. Next, a 2D orthogonal range counting query is performed using the data structure of [11]. In [11], it is shown how to construct a data structure using $O(n)$ space and $O(n \log n)$ preprocessing time which allows to answer 2D orthogonal range counting queries in (optimal for this space [15]) $\emptyset(\log n / \log\log n)$. Thus the total query time is $O(m + \log n / \log\log n) = O(m)$ and optimal.

For the case where $m < \log n / \log\log n$ the general framework is used again. This time set $h = \log n / \log\log n$. When given a query, navigate through the data structure from Section 4 with P on both versions i and j of the suffix tree till a snapshot u^{τ} is reached from the i-version and a snapshot $u^{\tau'}$ is reached from the j-version. All of the snapshots in the snapshot list of u between u^{τ} and $u^{\tau'}$, excluding $u^{\tau'}$, correspond to occurrences we are interested in counting. By computing the distance of each snapshot from the end of its snapshot list in the preprocessing phase, one can subtract the distance of $u^{\tau'}$ from the distance of u^{τ} during the query phase in constant time. The correctness of this answer follows directly from Lemma 8.

This provides the following:

Theorem 10. *The PRI-Count problem can be solved using $O(n \log n / \log\log n)$ space and $O(n \log n)$ preprocessing time, with $O(m + \log\log|\Sigma|)$ query time.*

Substring Rank. The solution for $SSR(P, i)$ is a direct application PRI-Count, by computing $PRI - Count(P, 1, i)$ as this will count the number of times P appeared in $t_1 \cdots t_n$.

5.3 Substring Select

In order to be able to answer substring select queries efficiently, a data structure which solves the *Range Selection* problem is required.

Problem 11. **Range Selection** Given an array of integers $A = (a_1, ..., a_n)$ where $\forall 1 \leq i \leq n, 1 \leq a_i \leq n$, preprocess A such that given a query (i, j, k) return the k^{th} smallest value in the set $\{a_x | i \leq x \leq j\}$.

Brodal and Jørgensen [6] show a solution for the Range Selection problem which uses linear space and $O(n \log n)$ construction time, and uses $O(\log n / \log\log n)$ query time. The construction time there is dominated by the need to sort the

input. By constructing their solution on the suffix array, which is the array of the locations of the suffixes in their lexicographical ordering, one can answer substring select queries efficiently for $m \geq \log n / \log \log n$ as follows. Furthermore, this construction will take linear time in this case as the range is bounded by n. Given a SSS query, the suffix tree is first used in order to locate the node u corresponding to P. This node covers a consecutive range of suffixes, sorted by their lexicographical ordering (this can be viewed as a consecutive sub-array of the suffix array). Locating this range (l, r) is done like in the PRI-Count query. Next, perform a range selection query (l, r, k) in order to obtain the answer for the query.

For the case where $m < \log n / \log \log n$ the general framework is used again. This time set $h = \log n / \log \log n$. To answer a query, navigate through the data structure from Section 4 with P on version 1 of the suffix tree till a snapshot u^{τ} is reached, where u is the node corresponding to P. Next, one can preprocess the snapshot lists to all be in arrays. Thus jumping to the k^{th} snapshot of u in $SL(u)$ to obtain $u^{\tau'}$ can be done in constant time. τ' is the answer to our query. Thus, the total query time is $O(m)$ which is optimal.

Theorem 12. *The SSS problem can be solved using $O(n \log n / \log \log n)$ space and preprocessing time, with $O(m)$ query time.*

References

1. Alstrup, S., Brodal, G.S., Rauhe, T.: New data structures for orthogonal range searching. In: IEEE Symposium on Foundations of Computer Science, pp. 198–207 (2000)
2. Amir, A., Kopelowitz, T., Lewenstein, M., Lewenstein, N.: Towards Real-Time Suffix Tree Construction. In: Consens, M.P., Navarro, G. (eds.) SPIRE 2005. LNCS, vol. 3772, pp. 67–78. Springer, Heidelberg (2005)
3. Amir, A., Nor, I.: Real-time indexing over fixed finite alphabets. In: Proc. of the Symposium on Discrete Algorithms (SODA), pp. 1086–1095 (2008)
4. Bille, P., Gørtz, L.: Substring Range Reporting. To Appear in Proc. 22nd Combinatorial Pattern Matching Conference (2011)
5. Brodal, G.S.: Partially Persistent Data Structures of Bounded Degree with Constant Update Time. Nord. J. Comput. 3(3), 238–255 (1996)
6. Brodal, G.S., Jørgensen, A.G.: Data structures for range median queries. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 822–831. Springer, Heidelberg (2009)
7. Chien, Y., Hon, W., Shah, R., Vitter, J.S.: Geometric Burrows-Wheeler Transform: Linking Range Searching and Text Indexing. In: Data Compression Conference (DCC), pp. 252–261 (2008)
8. Dietz, P.F.: Fully Persistent Arrays (Extended Array). In: Proc. of Symposium on Discrete Algorithms (SODA), pp. 235–244 (1999)
9. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making Data Structures Persistent. J. Comput. Syst. Sci. 38(1), 86–124 (1989)
10. Farach, M.: Optimal suffix tree construction with large alphabets. In: Proc. 38th IEEE Symposium on Foundations of Computer Science, pp. 137–143 (1997)

11. JáJá, J., Mortensen, C.W., Shi, Q.: Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In: Fleischer, R., Trippen, G. (eds.) ISAAC 2004. LNCS, vol. 3341, pp. 558–568. Springer, Heidelberg (2004)
12. Kaplan, H.: Persistent data structures. In: Handbook on Data Structures, pp. 241–246. CRC Press, Boca Raton (1995)
13. Mäkinen, V., Navarro, G.: Rank and select revisited and extended. Theor. Comput. Sci. 387(3), 332–347 (2007)
14. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. of the ACM 23, 262–272 (1976)
15. Patrascu, M.: Lower bounds for 2-dimensional range counting. In: Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC), pp. 40–46 (2007)
16. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14, 249–260 (1995)
17. Weiner, P.: Linear pattern matching algorithm. In: Proc. 14th IEEE Symposium on Switching and Automata Theory, pp. 1–11 (1973)