Suffix-Prefix Queries on a Dictionary

Grigorios Loukides
Department of Informatics, King's College London, London, UK

Solon P. Pissis
CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands

Sharma V. Thankachan
North Carolina State University, Raleigh, USA

Wiktor Zuba
CWI, Amsterdam, The Netherlands

Abstract

In the *all-pairs suffix-prefix* (APSP) problem, we are given a dictionary R of k strings, $S_1, \ldots, S_k$, of total length n, and we are asked to find the length $\mathsf{SPL}_{i,j}$ of the *longest* string that is both a suffix of S_i and a prefix of S_j, for all $i, j \in [1, k]$. APSP is a classic problem in string algorithms with many applications in bioinformatics. When all strings of the dictionary are over an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$, APSP can be solved in the optimal $\mathcal{O}(n + k^2)$ time with the use of the generalized suffix tree of the dictionary [Gusfield et al., *Inf. Process. Lett.* 1992].

In many bioinformatics applications, such as in sequence assembly, the size k of dictionary R is very large. In particular, k^2 usually dominates n, and thus the k^2 factor is the bottleneck both in the time and in the space complexity of such applications. We thus initiate a holistic study on several data structure variants of APSP. In particular, we consider the following types of queries:

- One-to-One(i, j): output $\mathsf{SPL}_{i,j}$.
- One-to-All(i): output $\mathsf{SPL}_{i,j}$ for every $j \in [1, k]$.
- Report(i, ℓ): output all distinct $j \in [1, k]$ such that $\mathsf{SPL}_{i,j} \geq \ell$, where $\ell \geq 0$ is an integer.
- Count(i, ℓ): output the number of distinct $j \in [1, k]$ such that $\mathsf{SPL}_{i,j} \geq \ell$, where $\ell \geq 0$ is an integer.
- Top(i, K): output K distinct $j \in [1, k]$ with the highest values of $\mathsf{SPL}_{i,j}$ breaking ties arbitrarily.

We assume the standard word RAM model of computation with word size $w = \Omega(\log n)$ and an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$. We show the following upper bounds:

Query	Space (words)	Query time	Note
One-to-One(i, j)	$\mathcal{O}(n)$	$\mathcal{O}(\log\log k)$	Theorem 11
One-to-All(i)	$\mathcal{O}(n)$	$\mathcal{O}(k)$	Theorem 14
Report(i, ℓ)	$\mathcal{O}(n)$	$\mathcal{O}(\log n/\log\log n + \text{output})$	Theorem 19(i)
Count(i, ℓ)	$\mathcal{O}(n)$	$\mathcal{O}(\log n/\log\log n)$	Theorem 19(ii)
Top(i, K)	$\mathcal{O}(n)$	$\mathcal{O}(\log^2 n/\log\log n + K)$	Theorem 22

We also present efficient algorithms for constructing these data structures.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases all-pairs suffix-prefix, suffix-prefix queries, internal pattern matching

Funding This work is supported in part by the Royal Society grant IES\R3\193209.
Solon P. Pissis: Supported in part by the PANGAIA and ALPACA projects that have received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreements No 872539 and 956229, respectively.
Sharma V. Thankachan: Supported by the U.S. National Science Foundation (NSF) grant CCF-2146003.
Wiktor Zuba: Supported by the Netherlands Organisation for Scientific Research (NWO) through Gravitation-grant NETWORKS-024.002.003.

1 Introduction

The *all-pairs suffix-prefix* problem (APSP, in short) is a classic problem in string algorithms. APSP finds numerous applications in bioinformatics because it is the first step in sequence assembly [26, 37, 46, 8, 11]. Given a dictionary R of k strings, $S_1, \ldots, S_k$, of total length n, the APSP problem asks us to find, for each string S_i, $i \in [1, k]$, its longest suffix that is a prefix of string S_j, for all $j \neq i$, $j \in [1, k]$. Gusfield et al. [27] presented an algorithm running in the optimal $\mathcal{O}(n + k^2)$ time for solving APSP, assuming all strings in R are over an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$. The algorithm is based on the generalized suffix tree [53] of R. Ohlebusch and Gog [39] gave another optimal algorithm which is based on the generalized suffix array [36] of R. Tustumi et al. [49] gave yet another optimal algorithm based on the generalized suffix array of R. Thus the common denominator of all existing optimal algorithms for APSP is that they rely on sorting the suffixes of all strings in R, and therefore they require space $\Omega(n)$ in any case and for any alphabet. In a very recent work, Loukides and Pissis [34] presented a different optimal algorithm, which is based on the Aho-Corasick automaton of R [1], and it thus requires space linear in the size of the automaton.

Due to the practical relevance of APSP, there also exists a large body of works devoted to implementing algorithms for APSP that are suboptimal but practically fast on real-world datasets; see [25, 42, 33] and references therein for some of the state-of-the-art implementations. For a parallel implementation of the algorithm by Tustumi et al. see [35]. For approximate variants of APSP, under the Hamming or edit distance, see [44, 52, 32, 5, 47].

In many bioinformatics applications, such as in sequence assembly, the size k of dictionary R is very large. In particular, k^2 usually dominates n, and thus the k^2 factor is the bottleneck both in the time and the space complexity of such applications. For instance, in typical benchmark datasets[1] for genome assembly using short DNA reads (fragments), k is in the order of 10^6 to 10^8 and n is in the order of 10^8 to 10^{10}. Hence k^2 dominates n significantly.

We thus initiate a holistic study on several data structure variants of APSP. Let $\mathsf{SPL}_{i,j}$ (short for suffix-prefix length), for any $i, j \in [1, k]$, denote the length of the *longest* string, that is both a suffix of S_i and a prefix of S_j. We consider the following types of queries:

- One-to-One(i, j): output $\mathsf{SPL}_{i,j}$.
- One-to-All(i): output $\mathsf{SPL}_{i,j}$ for every $j \in [1, k]$.
- Report(i, ℓ): output all distinct $j \in [1, k]$ such that $\mathsf{SPL}_{i,j} \geq \ell$, where $\ell \geq 0$ is an integer.
- Count(i, ℓ): output the number of distinct $j \in [1, k]$ such that $\mathsf{SPL}_{i,j} \geq \ell$, where $\ell \geq 0$ is an integer.
- Top(i, K): output K distinct $j \in [1, k]$ with the highest values of $\mathsf{SPL}_{i,j}$ breaking ties arbitrarily.

By being able to answer different types of such queries efficiently, one may be able to design alternative algorithms, depending on the application in scope, which avoid the k^2 factor in their time or space complexity. Indeed, we stress that most works studying APSP from a practical perspective (e.g., [25, 42, 33]), in fact considered the ℓ-APSP problem in their experimental part; namely, the problem in which we are asked to output only the $\mathsf{SPL}_{i,j}$ values with $\mathsf{SPL}_{i,j} \geq \ell$, for some integer $\ell \geq 0$, which, however, is given *a priori* and is *fixed for all pairs* S_i, S_j. This inflexibility would be surpassed should one have *space-efficient* (e.g., linear-space) data structures for answering these different types of queries *fast*.

---

[1] For example, see `http://gage.cbcb.umd.edu/data/index.html`.

Our Results We assume the standard word RAM model of computation with word size $w = \Omega(\log n)$ and an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$. We show the following upper bounds:

Query	Space (words)	Query time	Note
One-to-One(i,j)	$\mathcal{O}(n)$	$\mathcal{O}(\log\log k)$	Theorem 11
One-to-All(i)	$\mathcal{O}(n)$	$\mathcal{O}(k)$	Theorem 14
Report(i,ℓ)	$\mathcal{O}(n)$	$\mathcal{O}(\log n/\log\log n + \text{output})$	Theorem 19(i)
Count(i,ℓ)	$\mathcal{O}(n)$	$\mathcal{O}(\log n/\log\log n)$	Theorem 19(ii)
Top(i,K)	$\mathcal{O}(n)$	$\mathcal{O}(\log^2 n/\log\log n + K)$	Theorem 22

We also provide efficient construction algorithms for Theorems 11 and 14: Theorem 11 can be implemented in $\mathcal{O}(n\log\log k)$ time and Theorem 14 can be implemented in $\mathcal{O}(n)$ time. For Theorems 19 and 22, no guaranteed construction time is provided: the query times for Report, Count, and Top rely on the construction of a 2D rectangle stabbing data structure for reporting [45] and counting [28], but unfortunately the construction times for these data structures are not mentioned in [45] or [28]. However, by constructing the classic data structure for 2D rectangle stabbing [15], we obtain $\mathcal{O}(n\log n)$ construction time, $\mathcal{O}(n)$ words of space, $\mathcal{O}(\log n + \text{output})$ query time for Report, $\mathcal{O}(\log n)$ query time for Count, and $\mathcal{O}(\log^2 n + K)$ query time for Top. We also make the following straightforward observation.

▶ **Observation 1.** *The symmetric versions of One-to-All, Report, Count and Top, where we are given string S_j as the query and we are asked to output information about $\mathsf{SPL}_{i,j}$, for all $i \in [1,k]$, can be addressed by constructing the corresponding data structures for the dictionary R^r of k strings $S_1^r,\ldots,S_k^r$, where $S^r = S[|S|]\cdots S[2]S[1]$ denotes the reverse of string $S = S[1]S[2]\cdots S[|S|]$. Hence, the same space/query-time trade-offs can be achieved.*

Related Work In addition to the data structure variants of APSP that are studied here, two other versions of APSP have been studied in the literature. The first version consists in enumerating all pairwise suffix-prefix matches (not necessarily the longest ones) in decreasing order of their lengths. This version of the problem was solved by Ukkonen [50], who used this solution as the crux of his classic linear-time implementation of the greedy algorithm for constructing approximate shortest common superstrings. The second APSP version studied consists in enumerating the *set* of longest suffix-prefix matches (not however their association with the corresponding pairs of strings) [12]. Since any suffix-prefix match in this set is a prefix of some input string, the size of this set is $\mathcal{O}(n)$. This version of the problem was solved in the optimal $\mathcal{O}(n)$ time, independently, by Park et al. [40] and by Khan [29].

Although our work is inspired by real-world applications, the underlying data structure problems are also appealing from a theoretical perspective: (i) they are analogous to *distance oracles* for networks [48, 41, 17, 16, 13]; and (ii) they are special types of *internal pattern matching* (IPM) data structures [31, 30, 3, 14, 4]. For instance, an existing, more general, IPM data structure [30, 31] can be employed to answer One-to-One queries in $\mathcal{O}(\log n)$ time using $\mathcal{O}(n)$ words of space; see Section 2.3 for more details. By designing a specialized data structure for One-to-One, we obtain $\mathcal{O}(\log\log k)$ query time using $\mathcal{O}(n)$ words of space.

Paper Organization In Section 2, we provide basic definitions and notation on strings. We also describe basic data structures for representing a dictionary, some more advanced data structures that are necessary to obtain our upper bounds, and a few previous solutions to APSP (variants). In Section 3, we provide the solution to One-to-One queries. In Section 4, we provide the solution to One-to-All queries. In Section 5, we provide the solutions to Report and Count queries. Finally, in Section 6, we provide the solution to Top queries.

2 Preliminaries

An *alphabet* Σ is a finite nonempty set of $\sigma = |\Sigma|$ elements called *letters*. By Σ^* we denote the set of all strings over Σ including the *empty string* ε of length 0. A *string* S over Σ is a sequence of letters of Σ. For a string $S = S[1] \cdots S[n]$ over Σ, by $n = |S|$ we denote its length. The fragment $S[i\,..\,j]$ of S is an *occurrence* of the underlying *substring* $P = S[i] \cdots S[j]$. We also say that P *occurs* at *(starting) position* i in S. A *prefix* of S is a fragment of S of the form $S[1\,..\,j]$ and a *suffix* of S is a fragment of S of the form $S[i\,..\,n]$.

Let M be a finite nonempty set of strings over Σ of total length m. We call M a *dictionary*. We define the *trie* of M, denoted by $\mathsf{TR}(M)$, as a deterministic finite automaton that recognizes M. Its set of states (nodes) is the set of prefixes of the elements of M; the initial state (root node) is ε; the set of terminal states is M; and transitions (edges) are of the form $\delta(u, \alpha) = u\alpha$, where u and $u\alpha$ are nodes and $\alpha \in \Sigma$. The size of $\mathsf{TR}(M)$ is thus $\mathcal{O}(m)$. The *compacted trie* of M, denoted by $\mathsf{CT}(M)$, contains the root, the branching nodes, and the terminal nodes of $\mathsf{TR}(M)$. The term compacted refers to the fact that $\mathsf{CT}(M)$ reduces the number of nodes by replacing each maximal branchless path segment with a single edge, and that it uses a fragment of a string from M to represent the label of this edge in $\mathcal{O}(1)$ words of space. The nodes of $\mathsf{TR}(M)$ that are included in $\mathsf{CT}(M)$ are called *explicit*; all other nodes are called *implicit*. The size of $\mathsf{CT}(M)$ is thus $\mathcal{O}(|M|)$. The most well-known form of compacted trie is the suffix tree described next.

2.1 Suffix Tree and Aho-Corasick Automaton

We are given a dictionary R of k strings, $S_1, S_2, \ldots, S_k$, whose total length is $n = |S_1| + |S_2| + \cdots + |S_k|$. Every string in R is over an integer alphabet Σ whose size σ is polynomial in n, i.e., $\Sigma = \{1, 2, \ldots, n^{\mathcal{O}(1)}\}$ and thus $\sigma \leq n^{\mathcal{O}(1)}$. For constructing specialized data structures and answering internal pattern matching queries, non-trivial representations of R (different than a simple set of strings) are usually more efficient.

Let us set $T_R := S_1\$_1 S_2\$_2 \cdots S_k\$_k$, where $\$_1 < \$_2 < \cdots < \$_k$ are letters that are strictly lexicographically smaller than any letter from Σ (and as such they do not belong to Σ).

Let $\mathsf{ST}(S)$ denote the *suffix tree* of string S, that is the compacted trie of all the suffixes of S. For any node v of $\mathsf{ST}(S)$, by $\mathsf{str}(v)$ we denote the concatenation of the edge labels on the path from the root to v, and by $d(v) = |\mathsf{str}(v)|$ we denote the *string depth* of v. The *suffix array* $\mathsf{SA}(S)$ of S is the lexicographically sorted array of the set of suffixes of S, represented by their starting positions; see Figure 1 for an example.

▶ **Lemma 2** ([53, 22]). *For any string S of length m over an integer alphabet of size $\sigma \leq m^{\mathcal{O}(1)}$, the suffix tree and the suffix array of S can be constructed in $\mathcal{O}(m)$ time.*

We also denote $\mathsf{ST}_i = \mathsf{ST}(S_i\$_i)$ and $\mathsf{ST}_R = \mathsf{ST}(S_1\$_1, \ldots, S_k\$_k)$; that is ST_R is the generalized suffix tree [51] of the k strings from R. The generalized suffix tree can be built in linear time; here, however, this more complicated construction is not needed since this compacted trie is equivalent to $\mathsf{ST}(T_R)$ as the letters $\$_i$ occur uniquely in this string (and hence a compacted edge containing any label $\$_i$ must end at a leaf node).

Another useful representation of R is given by its Aho-Corasick (AC) automaton [1]; the set of states of the AC automaton of R, denoted by $\mathsf{AC}(R)$, corresponds to the set of the prefixes of the strings in R. Let $\mathsf{node}(S)$ denote the node corresponding to string S. After reading an input string the automaton must be in a state corresponding to a suffix of this string (the longest one that is also a prefix of some string in R and has a corresponding state); such a state always exists as ε is always represented (recall ε is

i	$\mathsf{SA}(S)[i]$	Suffix
1	7	`$`
2	6	`A$`
3	4	`AGA$`
4	2	`AGAGA$`
5	1	`CAGAGA$`
6	5	`GA$`
7	3	`GAGA$`

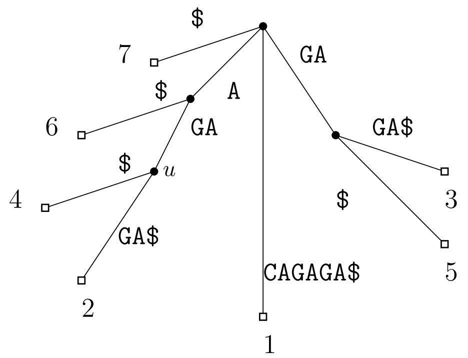


Figure 1 Suffix array $\mathsf{SA}(S)$ and suffix tree $\mathsf{ST}(S)$ of string $S = \texttt{CAGAGA\$}$, where `$` is a terminal letter, which is the lexicographically smallest letter occurring in S. For node u in $\mathsf{ST}(S)$, $\mathsf{str}(u) = \texttt{AGA}$ and $d(u) = 3$.

the string of length 0). As such, the automaton $\mathsf{AC}(R)$ is often represented by the trie $\mathsf{TR}(R)$ with transitions $\delta(\mathsf{node}(S), \alpha) = \{\mathsf{node}(S\alpha)\}$ if $S\alpha$ is a prefix of a string in R, and $\delta(\mathsf{node}(S), \varepsilon) = \{\mathsf{node}(S')\}$, where S' is the longest suffix of S which is also a prefix of a string in R. The ε-transitions are called *failure transitions*. The existence of ε-transitions makes the automaton nondeterministic, and even though this nondeterminism can be avoided, we are going to actually employ those ε-transitions to construct the data structure for One-to-All queries.

▶ **Lemma 3** ([1, 20])**.** *For any dictionary R of k strings of total length n over an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$, $\mathsf{AC}(R)$ can be constructed in $\mathcal{O}(n)$ time.*

By $\mathsf{FT}(R)$ we denote the so-called *Failure Transition tree* (FTtree) of R, introduced by Loukides and Pissis in [34] for solving the APSP problem: the FTtree nodes correspond to the states of the AC automaton (that is, to prefixes of strings in R), and the edges correspond to its ε-transitions with *reversed* direction. Notice that, since every state of $\mathsf{AC}(R)$ has *exactly one* outgoing failure transition, $\mathsf{FT}(R)$ is indeed a tree rooted at $\mathsf{node}(\varepsilon)$. We additionally decorate every node u of $\mathsf{FT}(R)$ by a labeled interval $I_u = [i, j]_d$: $S_i, S_{i+1}, \ldots, S_j$ have as a common prefix the string of length d represented by node u; see [34]. We will generally assume that R is given lexicographically sorted at construction time; otherwise, the sorted version of R can be produced in linear time using, for example, Lemma 3 or Lemma 2.

▶ **Example 4.** Let $R = \{S_1, S_2, S_3, S_4\} = \{\texttt{ACAA}, \texttt{ACAG}, \texttt{ACGC}, \texttt{CACA}\}$ be a dictionary of $k = 4$ strings. The AC automaton and the FTtree of R is shown in Figure 2. Consider the path from the root to leaf node S_4 (shown in red) in the FTtree of R, where the non-root nodes have the following labeled intervals $[i, j]_d$: $[1, 3]_1$, $[4, 4]_2$, $[1, 2]_3$, $[4, 4]_4$. By recording the largest string depth d of an interval containing j, for every $j \in [1, k]$, along this path, we compute all $\mathsf{SPL}_{4,j}$: $\mathsf{SPL}_{4,1} = 3$, $\mathsf{SPL}_{4,2} = 3$, $\mathsf{SPL}_{4,3} = 1$, and $\mathsf{SPL}_{4,4} = 4$. Loukides and Pissis [34] showed how to compute this information, for all i, in $\mathcal{O}(n + k^2)$ total time, thus solving the APSP problem optimally using only the FTtree of R.

2.2 Advanced Data Structures

Let T be a rooted tree. A *lowest common ancestor* (LCA) query on T for two given nodes u and v, denoted by $w = \mathsf{LCA}_T(u, v)$, returns the last (i.e., the lowest) common node w on their paths from the root.

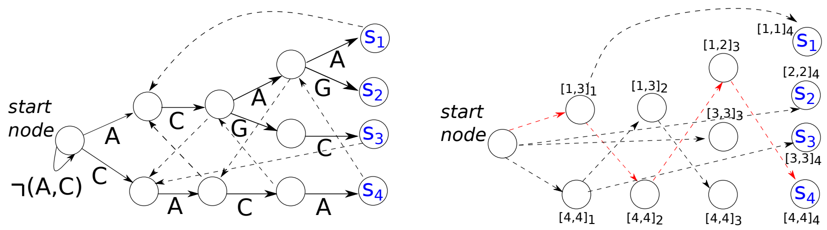


Figure 2 The AC automaton $\mathsf{AC}(R)$ (on the left) and FTtree $\mathsf{FT}(R)$ (on the right) of the dictionary of strings $R = \{S_1, S_2, S_3, S_4\} = \{\texttt{ACAA}, \texttt{ACAG}, \texttt{ACGC}, \texttt{CACA}\}$. In $\mathsf{AC}(R)$, solid arrows correspond to transitions and dashed arrows to failure transitions. To avoid cluttering the figure, failure transitions to the start node in $\mathsf{AC}(R)$ have been omitted.

▶ **Lemma 5** ([9]). *For any rooted tree T with m nodes, after $\mathcal{O}(m)$-time preprocessing, we can answer LCA_T queries in $\mathcal{O}(1)$ time per query.*

A *rank and select* data structure (also known as *succinct indexable dictionary* [43]) is a classic data structure, constructed over an array A of length m over alphabet $[1, \sigma]$, which supports two types of queries:

- $\mathsf{rank}_A(i, x) = |\{\ell \in [1, x] : A[\ell] = i\}|$, for $i \in [1, \sigma]$ and $x \in [1, m]$;
- $\mathsf{select}_A(i, x) = \min\{\ell \in [1, m] : \mathsf{rank}_A(i, \ell) = x\}$, for $i \in [1, \sigma]$ and $x \in [1, m]$.

In other words, $\mathsf{rank}_A(i, x)$ returns the number of elements with value equal to i occurring at positions in $[1, x]$ of S, while $\mathsf{select}_A(i, x)$ returns the position of the xth element of A with value equal to i.

▶ **Lemma 6** ([7, 38, 18]). *For any array $A = A[1 \mathinner{.\,.} m]$ over $[1, \sigma]$, $\sigma \le m$, after $\mathcal{O}(m \log\log \sigma)$-time preprocessing, we can construct a data structure of $\mathcal{O}(m)$ words of space that supports $\mathcal{O}(\log\log \sigma)$-time rank and select queries on A.*

Let T be a rooted tree of m nodes with integer weights on nodes. Further assume that the weight of every node of T satisfies the *min-heap property*: the weight of each node is greater than or equal to the value of its parent (the smallest weight is hence at the root). A *weighted ancestor* (WA) query for a given node u of T and an integer d, denoted by $w = \mathsf{WA}_T(u, d)$, returns its deepest ancestor w whose weight is at most d [23]. This problem is the generalization of the classic *predecessor search* problem on rooted trees. In the special case when T is a suffix tree and the nodes are weighted by *string depth*, the problem admits an optimal solution due to the recent result of Belazzougui et al. [6] (see also [24]).

▶ **Lemma 7** ([6]). *For any suffix tree T with m nodes weighted by string depth, after $\mathcal{O}(m)$-time preprocessing, we can answer WA_T queries in $\mathcal{O}(1)$ time per query.*

In this special case, the ancestor at string depth exactly d may be an implicit node of T, in which case the query outputs its closest explicit ancestor.

2.3 Previous Solutions

$\mathcal{O}(n + k^2)$-time Algorithm for APSP We describe the optimal solution to APSP given by Gusfield et al. in [27]. We set $T_R := S_1\$_1 S_2\$_2 \cdots S_k\$_k$, where $\$_1 < \$_2 < \cdots < \$_k$ are letters that are strictly lexicographically smaller than any letter from Σ. We start by constructing the suffix tree $\mathsf{ST}_R = \mathsf{ST}(T_R)$. Using a DFS traversal on ST_R, we construct lists $L(v)$ for all nodes v of ST_R: $L(v)$ stores all i such that the suffix of length $d(v)$ of string S_i is $\mathsf{str}(v)$.

Consider a string S_j from R and focus on the path P_j from the root of ST_R to the leaf node representing the longest suffix of S_j, i.e., the entire string S_j. Let v be a node on P_j. A suffix of string S_i of length $d(v)$ is a prefix of string S_j of the same length if and only if i is in $L(v)$. However, for each index i, we want to record the *deepest* node v on P_j such that i is in $L(v)$. It then follows that $d(v) = \mathsf{SPL}_{i,j}$. In order to achieve a linear-time complexity, we perform another DFS maintaining k stacks (one for each S_i). Upon visiting v, we push it on stack i for every $i \in L(v)$. When the leaf node representing the entire string S_j is reached, we scan the k stacks and record, for each index i, the current top of the ith stack. When v is reached in a backward edge traversal, we pop the top of any stack whose index is in $L(v)$. We obtain the following result.

▶ **Lemma 8** ([27]). *For any dictionary of k strings of total length n over an integer alphabet of size $\sigma \le n^{\mathcal{O}(1)}$, APSP can be solved in the optimal $\mathcal{O}(n + k^2)$ time.*

In what follows, we assume that $k \ge \sqrt{n}$; otherwise, when $k < \sqrt{n}$, Lemma 8 implies an optimal solution to our data structure problems (linear preprocessing time, linear size and time-optimal queries), which precomputes and stores all answers.

Internal Prefix-Suffix Queries for One-to-One Kociumaka considered the following data structure problem in [30]: Given two fragments x and y of a string T and a positive integer d, report all suffixes of y of length between d and $2d - 1$ that also occur as prefixes of x (represented as an arithmetic progression of their lengths). This is the *Internal Prefix-Suffix Queries* problem. Kociumaka showed the following result (see also [31]).

▶ **Lemma 9** (Theorem 1.1.3 in [30]). *For any string T of length m over an integer alphabet of size $\sigma \le m^{\mathcal{O}(1)}$, after $\mathcal{O}(m)$-time preprocessing, we can answer Internal Prefix-Suffix Queries in $\mathcal{O}(1)$ time per query.*

By employing Lemma 9 on T_R, after an $\mathcal{O}(n)$-time preprocessing, we can answer One-to-One queries in $\mathcal{O}(\log(\min(|S_i|, |S_j|))) = \mathcal{O}(\log n)$ time. In particular, we query for $x = S_j$, $y = S_i$, and $d = 2^\ell$, for all integers $0 \le \ell \le \log \min(|S_i|, |S_j|)$, to compute a representation of all the suffixes of S_i that are also prefixes of S_j and then return the length of the longest one as $\mathsf{SPL}_{i,j}$. We obtain the following result, which we improve in Section 3.

▶ **Corollary 10.** *For any dictionary of k strings of total length n over an integer alphabet of size $\sigma \le n^{\mathcal{O}(1)}$, we can construct a data structure of $\mathcal{O}(n)$ words of space answering One-to-One queries in $\mathcal{O}(\log n)$ time.*

3 Answering One-to-One Queries

Main Idea Say we want to find the longest suffix of S_i that is a prefix of S_j. We first find the maximal longest common prefix between S_j and any suffix of S_i. Say this suffix is $S_i[q \mathinner{.\,.} |S_i|]$ and we have that $S_i[q \mathinner{.\,.} q + r - 1] = S_j[1 \mathinner{.\,.} r]$ is this longest common prefix. If this prefix is the whole $S_i[q \mathinner{.\,.} |S_i|]$, i.e., $|S_i| = q + r - 1$, then r is clearly the answer. If this longest common prefix is not a suffix of S_i, i.e., $|S_i| > q + r - 1$, then the answer is the longest prefix of $S_i[q \mathinner{.\,.} q + r - 1]$, that is also a suffix of S_i.

Recall that $\mathsf{ST}_i = \mathsf{ST}(S_i\$_i)$ and $\mathsf{ST}_R = \mathsf{ST}(T_R)$. Consider the path in ST_R obtained by reading $S_j\$_j$ from its root (this path ends in a leaf node). When spelling any suffix of S_i that is also a prefix of S_j in ST_R we use exactly the same path and end by going out of it when reading $\$_i$. This means, that $\mathsf{SPL}_{i,j}$ is represented by the lowest node on this path that has an outgoing edge with label $\$_i$.

In the following we focus on enhancing ST_R and ST_i, for all $i \in [1, k]$, to obtain a data structure that allows finding the string depth of such a node (equal to $\mathsf{SPL}_{i,j}$) efficiently. We will prove the following result.

▶ **Theorem 11.** *For any dictionary of k strings of total length n over an integer alphabet of size $\sigma \le n^{\mathcal{O}(1)}$, we can construct a data structure of $\mathcal{O}(n)$ words of space answering* One-to-One *queries in $\mathcal{O}(\log \log k)$ time. The data structure can be constructed in $\mathcal{O}(n \log \log k)$ time.*

Let us start with a straightforward auxiliary lemma.

▶ **Lemma 12.** *For any dictionary of k strings $S_1, \ldots, S_k$ of total length n over an integer alphabet of size $\sigma \le n^{\mathcal{O}(1)}$, in $\mathcal{O}(n)$ time we can construct a data structure of $\mathcal{O}(k)$ words of space that answers queries of the type "Is S_j a suffix of S_i?" in $\mathcal{O}(1)$ time.*

Proof. Let X^r denote the *reverse* of string X, i.e., $X^r = X[|X|] \cdots X[1]$. We first sort $S_1^r, \ldots, S_k^r$ lexicographically, and store for each $j \in [1, k]$ a value $\mathsf{rlex}[j] \in [1, k]$ equal to the rank of S_j^r in this sorted list. S_j is a suffix of S_i if and only if S_j^r is a prefix of S_i^r. The crucial property of this ordering is that all the strings such that S_j^r is their prefix form an interval from the position $\mathsf{rlex}[j]$ to a position $\mathsf{rlex}[j] + l[j] - 1$, where $l[j]$ is the total number of strings $S_1^r, \ldots, S_k^r$ starting with S_j^r; that is, $\mathsf{rlex}[j] + l[j]$ is the position of the first string having a longest common prefix with S_j^r shorter than $|S_j^r|$. The values $\mathsf{rlex}[j]$ and $l[j]$, for all $j \in [1, k]$, can be computed in $\mathcal{O}(n)$ time [19].

As for the querying, for any i, j, we have that S_j is a suffix of S_i if and only if $\mathsf{rlex}[j] \le \mathsf{rlex}[i] < \mathsf{rlex}[j] + l[j]$, which is checked in $\mathcal{O}(1)$ time. The total size of arrays l and rlex is $\Theta(k)$. ◀

Construction We start the construction of the data structure by constructing the data structure underlying Lemma 12. We also construct ST_R and ST_i, for all $i \in [1, k]$, using Lemma 2. We enhance ST_R with the data structure for LCA queries underlying Lemma 5, and link the leaf nodes originating from suffixes of $S_i\$_i$ with the corresponding leaf nodes of ST_i, for all $i \in [1, k]$. We construct an array $A = A[1 \mathinner{.\,.} |T_R|]$ over $[1, k]$ such that $A[\ell] = i$ if the ℓth leaf node (from the left) of ST_R originates from a suffix of $S_i\$_i$; since the leaf nodes are ordered according to the lexicographic order of the suffixes they originate from, array A can be easily extracted from $\mathsf{SA}(T_R)$ constructed by means of Lemma 2. We enhance array A with the rank and select data structure underlying Lemma 6. We link the leaf nodes of ST_R with the corresponding elements of A. For each ST_i, we construct the data structure for WA queries underlying Lemma 7. For every node w of ST_i, we store the string depth of its closest ancestor (including w itself) that has an outgoing edge with label $\$_i$ and hence corresponds to a suffix of S_i; since the root always has such an edge, this assignment is always well-defined. In order to efficiently compute and store all those values, we simply process the information through the tree in a top-down manner. This completes the construction.

The part of the data structure that relies on Lemmas 2, 5, 7, and 12 is implemented in $\mathcal{O}(n)$ time and it occupies $\mathcal{O}(n)$ words of space. By Lemma 6, array A occupies $\mathcal{O}(n)$ words of space, and it can be implemented in $\mathcal{O}(n \log \log k)$ time as it stores k distinct values.

Querying Consider a $\mathsf{One\text{-}to\text{-}One}(i, j)$ query; that is, we want to compute $\mathsf{SPL}_{i,j}$, the length of the longest suffix of S_i that is a prefix of S_j. Let x be the position in array A that corresponds to the leaf node l_j of ST_R reached after conceptually reading $S_j\$_j$. We first check if the entire S_j is a suffix of S_i by means of Lemma 12. If this is the case then we return $\mathsf{SPL}_{i,j} = |S_j|$. If this is not the case (inspect Figure 3), we perform the following sequence

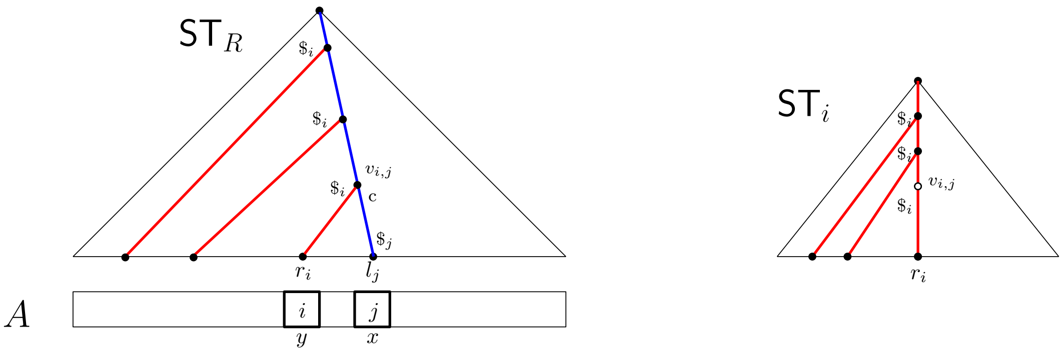


Figure 3 An illustration of the One-to-One(i, j) query algorithm. The node $v_{i,j}$, which is explicit in ST_R but implicit in ST_i, has an outgoing edge labeled with $\$_i$ and hence the string depth $d(v_{i,j})$ of node $v_{i,j}$ is the answer to the query.

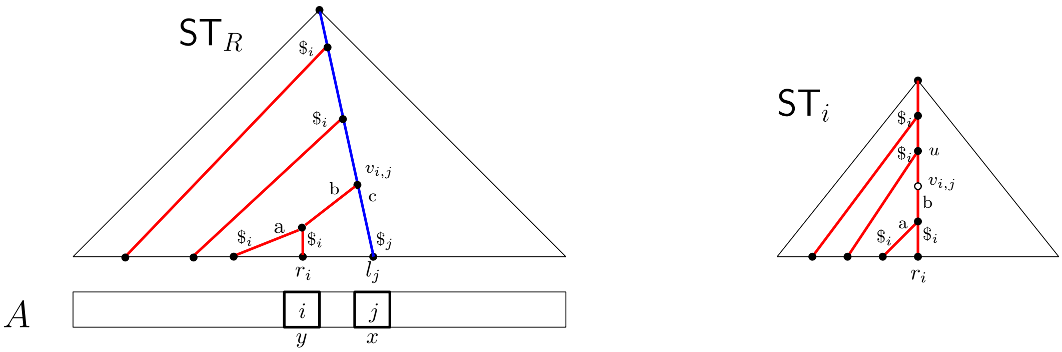


Figure 4 An illustration of the One-to-One(i, j) query algorithm. The closest ancestor of node $v_{i,j}$, which is explicit in ST_R but implicit in ST_i, with an outgoing edge labeled with $\$_i$ is node u and hence the string depth $d(u)$ of node u is the answer to the query.

of queries, $\mathsf{select}_A(i, \mathsf{rank}_A(i, x))$, which finds the position y in array A that corresponds to the leaf node r_i; this corresponds to the suffix of $S_i\$_i$ that is closest to the left of l_j. We then compute the lowest common ancestor of r_i and l_j: $v_{i,j} = \mathsf{LCA}_{\mathsf{ST}_R}(r_i, l_j)$. If node $v_{i,j}$ has an outgoing edge labeled with $\$_i$, which ends at r_i, then we return $\mathsf{SPL}_{i,j} = d(v_{i,j})$ (this is the case in Figure 3). We check this by checking whether $d(r_i) = d(v_{i,j}) + 1$. If $v_{i,j}$ does not have such an outgoing edge (this is the case in Figure 4), we locate the explicit node corresponding to $v_{i,j}$ in ST_i (or its closest explicit ancestor if it is implicit) by asking a WA query: $w = \mathsf{WA}_{\mathsf{ST}_i}(r_i, d(v_{i,j}))$. Finally, we return the string depth of the closest ancestor of w with an outgoing edge labeled $\$_i$ as $\mathsf{SPL}_{i,j}$; recall that every node of ST_i stores this information.

The time complexity of the query is $\mathcal{O}(\log\log k)$; the bottleneck is the complexity of the rank and select queries on A – all other operations take constant time. Let us now explain why the faster $\mathcal{O}(1)$-time select and $\mathcal{O}(1 + \log \frac{\log k}{\log w})$-time rank queries presented in [7], where w is the machine word, cannot improve our query time further. The size of the problem is $\Theta(n)$, hence the size of the machine word in the word-RAM model is $\Theta(\log n)$, thus the query time equals $\mathcal{O}(1 + \log \frac{\log k}{\log\log n})$. However, we have assumed that $k \geq \sqrt{n}$ (otherwise the structure of Lemma 8 implies an optimal solution – linear size and constant time queries – for the One-to-One queries), hence this is equal to $\mathcal{O}(1 + \log\log k) = \mathcal{O}(\log\log k)$ as stated.

Correctness Recall that the answer to $\mathsf{One\text{-}to\text{-}One}(i, j)$ equals to the string depth of the closest ancestor of l_j in ST_R that has an outgoing edge labeled with $\$_i$. By construction, this ancestor ends on the right of l_j only if the entire S_j is a suffix of S_i, which we check separately. Otherwise, this ancestor is also an ancestor of r_i (which is on the left of l_i) as $\$_i$ goes out of the path from the root to l_j to the left (by construction, it is lexicographically smaller than the next letter on this path), and hence this edge labeled with $\$_i$ must end either in r_i or further to the left (by the definition of r_i). As an ancestor of l_j and r_i, it is also the closest ancestor of $v_{i,j}$ with such an outgoing edge; the latter actually exists (possibly as an implicit node) in ST_i (unlike l_j). The final steps of the query algorithm find the string depth of the node corresponding to the searched ancestor in ST_i (string depth is a shared property of the corresponding nodes).

We have arrived at Theorem 11. Note that the construction time for our data structure is $\mathcal{O}(n \log \log k)$. The bottleneck for the construction time is the construction time for the rank and select data structure (Lemma 6).

4 Answering One-to-All Queries

The spine of the data structure described in this section is $\mathsf{FT}(R)$, the FTtree of R (see Section 2). Recall that for each node in $\mathsf{FT}(R)$ (representing each prefix of a string S_i), we store information about which strings from R it is a prefix of (see Figure 2).

Main Idea The Aho-Corasick lemma [1] states that for any two nodes, $\mathsf{node}(U)$ and $\mathsf{node}(V)$, in $\mathsf{AC}(R)$, we have a failure transition from $\mathsf{node}(U)$ to $\mathsf{node}(V)$ if and only if V is the longest suffix of U that is also a prefix of some string in R. As a consequence, in $\mathsf{FT}(R)$, $\mathsf{node}(S)$ is an ancestor of $\mathsf{node}(S')$ if and only if S is a suffix of S' (and both are prefixes of some strings from R as nodes of $\mathsf{FT}(R)$). Thus the path from $\mathsf{node}(\varepsilon)$ (the root) to $\mathsf{node}(S_i)$ in $\mathsf{FT}(R)$ contains exactly the nodes $\mathsf{node}(S)$ such that S is a suffix of S_i and a prefix of some string in R. Those nodes are ordered according to the string length, hence the nodes closer to $\mathsf{node}(S_i)$ on this path will correspond to *longer* suffix-prefix matches.

A $\mathsf{One\text{-}to\text{-}All}(i)$ query can thus be answered by simply reading the path from the root to $\mathsf{node}(S_i)$ recording, for each $j \in [1, k]$, the last node on the path corresponding to a prefix of S_j. The space occupied by $\mathsf{FT}(R)$ is in $\mathcal{O}(n)$; and such a query algorithm can take $\Theta(|S_i|)$, that is even $\Theta(n)$ time. Hence, by such an algorithm, we would not really gain anything from constructing $\mathsf{FT}(R)$ in the preprocessing. On the other extreme, by running this algorithm not for a single path, but for the whole $\mathsf{FT}(R)$ using a DFS traversal, we can precompute the answers for all the values of $i \in [1, k]$ in $\mathcal{O}(n + k^2)$ total time (and space), and then answer a query in $\mathcal{O}(k)$ time by simply outputting the k stored values; this would not be faster than using the algorithm by Gusfield et al. [27] or the one by Loukides and Pissis [34]. We will augment $\mathsf{FT}(R)$ to obtain a more efficient solution combining the space efficiency of the first approach with the low query time of the second one.

A *τ-micro-macro decomposition*, introduced for rooted binary trees in [2], and then generalized for rooted general trees in [10] (after an appropriate mapping), is a partition of a rooted tree T of N nodes into $\mathcal{O}(N/\tau)$ connected subtrees, called *micro trees*. In the case of binary trees each micro tree is of size at most τ and at most two of its nodes are adjacent to nodes in other micro trees. These nodes are referred to as *top* and *bottom boundary* nodes of the micro tree. The top boundary node is chosen as the root of the micro tree. The *macro tree* is a rooted tree of size $\mathcal{O}(N/\tau)$ whose nodes correspond to micro trees as follows (inspect Figure 5): The top boundary node $t(C)$ of a micro tree C is connected to a boundary

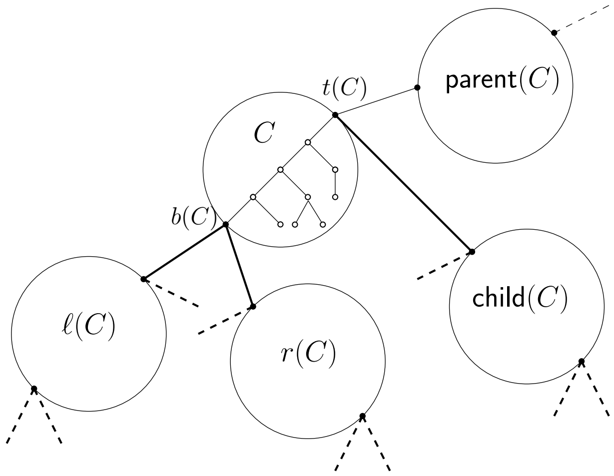


Figure 5 The structure of a micro-macro decomposition of a rooted binary tree.

node $\mathsf{parent}(C)$ in the parent micro tree (apart from the root). The boundary node $t(C)$ might also be connected to a top boundary node of a child micro tree, which we denote by $\mathsf{child}(C)$. Such a τ-micro-macro decomposition can be computed in $\mathcal{O}(N)$ time for binary [2] and general [10] rooted trees. We summarize the above discussion in the lemma below.

▶ **Lemma 13** ([2, 10])**.** *For any rooted tree T with N nodes and for any integer $\tau \in [1, N]$, the τ-micro-macro decomposition of T can be computed in $\mathcal{O}(N)$ time.*

We will prove the following result.

▶ **Theorem 14.** *For any dictionary of k strings of total length n over an integer alphabet of size $\sigma \le n^{\mathcal{O}(1)}$, we can construct a data structure of $\mathcal{O}(n)$ words of space answering One-to-All(i) queries in $\mathcal{O}(k)$ time. The data structure can be constructed in $\mathcal{O}(n)$ time.*

Construction We start the construction of the data structure by constructing $\mathsf{FT}(R)$ from $\mathsf{AC}(R)$ using Lemma 3. We compute the τ-micro-macro decomposition of $\mathsf{FT}(R)$, for a parameter τ defined later, using Lemma 13. For each node u of the $\mathsf{FT}(R)$, corresponding to a prefix S of some string S_i in R, we store the labeled interval I_u. For each boundary node in the τ-micro-macro decomposition of $\mathsf{FT}(R)$, we store an array of k integers, which for each $i \in [1, k]$, stores the string depth of its lowest ancestor $\mathsf{node}(S)$ such that S is a prefix of S_i. The additional size for storing this information in all the boundary nodes is $\mathcal{O}(k \cdot n/\tau)$. We compute these arrays by performing a DFS over $\mathsf{FT}(R)$ with a set of k stacks, one for every string in R, storing the string depths of ancestors of the visited node of each type (which S_i they originate from). As there are only $2n$ updates of the stacks (each prefix of a string S_i is stored and removed once from the ith stack) and the information is stored by simply reading the top values of the k stacks, the total computation time is bounded by $\mathcal{O}(n + k \cdot n/\tau)$.

Querying Let us start with the following observation from [34] (inspect also Figure 2).

▶ **Observation 15** ([34])**.** *Let u and v be two non-root nodes of FT(R) with labeled intervals $I_u = [i_u, j_u]_{d(u)}$ and $I_v = [i_v, j_v]_{d(v)}$, respectively, and such that u is an ancestor of v. Then $d(u) < d(v)$ and either $[i_u, j_u]$ contains $[i_v, j_v]$ or $[i_u, j_u]$ and $[i_v, j_v]$ do not intersect.*

Consider a One-to-All(i) query; that is, we want to compute an array of length k, which stores $\mathsf{SPL}_{i,j}$, for all $j \in [1,k]$. We start by finding the closest boundary node on the path from the root to $\mathsf{node}(S_i)$; that is, the top boundary node of the micro tree containing $\mathsf{node}(S_i)$. On the path between this top boundary node and $\mathsf{node}(S_i)$, there are at most τ nodes. We compute the information coming from just those nodes in $\mathcal{O}(k+\tau)$ time with a sweep line approach: there are $\mathcal{O}(\tau)$ (labeled) intervals from $[1,k]$, the intervals are labeled by different values (string depth), but, by Observation 15, two intervals are either disjoint or the one with the larger string depth is contained in the one with the smaller one. Thus, it is enough to hold the active intervals on a stack to keep track of the longest possible suffix-prefix match: the interval on the top of the stack has the highest value and will end the soonest. The solution is then obtained as the position-wise maximum of the computed array and the array stored in the top boundary node, which we compute in $\mathcal{O}(k)$ time.

Correctness The correctness of the algorithm follows by the Aho-Corasick lemma (see also the discussion of the "main idea" paragraph above).

The data structure occupies $\mathcal{O}(n+k\cdot n/\tau)$ words of space and supports One-to-All queries in $\mathcal{O}(k+\tau)$ time. By setting τ to k (or to ck, for some positive constant c that balances the operation costs more efficiently) we obtain the complexities claimed in Theorem 14. Note that the data structure is constructed in $\mathcal{O}(n+k\cdot n/\tau)$ time, which is $\mathcal{O}(n)$ for $\tau=\Theta(k)$. Thus the presented data structure for One-to-All queries is optimal.

5 Answering Report and Count Queries

In this section we are going to use ST_R again. This time, however, instead of augmenting ST_R with an LCA data structure and linking its nodes with the rank and select array, we are going to link the nodes with rectangles and employ classic results from computational geometry for reporting (see Lemma 16) and counting (see Lemma 17).

Let $[x_1,x_2]\times[y_1,y_2]$ denote a rectangle in a 2D space with edges parallel to the axes, where the intervals $[x_1,x_2]$ and $[y_1,y_2]$ are the projections of this rectangle to the x-axis and y-axis, respectively. In the reporting version of the *2D rectangle stabbing* problem [15], we are given a set S of n rectangles to preprocess, so that when we are given a query point $q=(x,y)$, we report the subset $Q\subseteq S$ of rectangles $[x_1,x_2]\times[y_1,y_2]$ that contain q: $x_1\le x\le x_2$ and $y_1\le y\le y_2$. In the counting version of 2D rectangle stabbing, we are asked to return $|Q|$.

▶ **Lemma 16** ([45]). *For any set S of n rectangles, we can construct a data structure of $\mathcal{O}(n)$ words of space answering 2D rectangle stabbing reporting queries in $\mathcal{O}(\log n/\log\log n+f)$ time, where f is the output size $|Q|$.*

2D rectangle stabbing counting is known to be reducible to 2D orthogonal range counting [21], and such a data structure for 2D orthogonal range counting can be found in [28].

▶ **Lemma 17** ([21, 28]). *For any set S of n rectangles, we can construct a data structure of $\mathcal{O}(n)$ words of space answering 2D rectangle stabbing counting queries in $\mathcal{O}(\log n/\log\log n)$ time.*

Main Idea For every suffix S of a string in R that is represented by a node in ST_R, we define a rectangle in 2D space: the x dimension corresponds to the lexicographically sorted list of all suffixes of strings in R whose prefix is S; and the y dimension corresponds to interval $[0,|S|]$. A Report (resp. a Count) query is defined by two parameters, which form a point in the 2D space: i corresponds to string S_i in the same sorted list (x dimension) and ℓ

corresponds to the smallest length of interest (y dimension). By reporting (resp. counting) all rectangles *enclosing this point* (Lemmas 16 and 17), we locate all suffix-prefix matches. Extra care, however, needs to be taken in order to avoid double reporting (resp. counting).

Construction We start the construction of the data structure by constructing ST_R using Lemma 2. Let u be an explicit or implicit node of ST_R that is the parent of a leaf node reached with $\$_i$: the labels of the path from root to u form a suffix of S_i. For every such node u and every i, we create a tuple $(L(u), R(u), d(u), i)$, where $L(u)$ and $R(u)$ are the (pre-order rank of) the leftmost and the rightmost leaf node under u, respectively.[2] Note that such a node may correspond to multiple tuples for different i values – this occurs when distinct elements of R share the same suffix. There are exactly n such tuples (one for every suffix) coming from ST_R and we can compute them in $\mathcal{O}(n)$ total time using a DFS traversal.

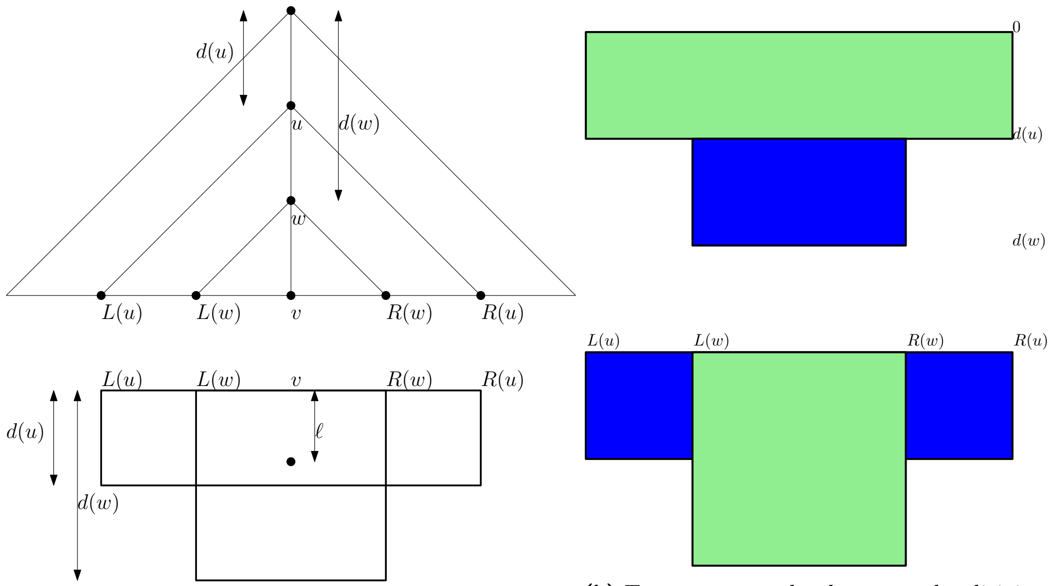
Recall that if we spell $S_j\$_j$ in ST_R and the obtained leaf node v has an ancestor of string depth ℓ which has an outgoing edge with label $\$_i$, then $\mathsf{SPL}_{i,j} \geq \ell$. The same property ($\mathsf{SPL}_{i,j} \geq \ell$) can be expressed by $L(v) \in [L(u), R(u)]$ (namely, u is an ancestor of v), and $\ell \in [0, d(u)]$ (namely, the string depth of u is at least ℓ) for a tuple $(L(u), R(u), d(u), i)$. Now note that $(L(u), R(u), d(u), i)$ forms a rectangle, whose identifier is i. In particular, $(L(u), R(u), d(u), i)$ can be viewed as rectangle $[L(u), R(u)] \times [0, d(u)]$ with satellite data i.

Now consider constructing the 2D rectangle stabbing data structure for reporting (resp. counting) for these n rectangles, and then ask the query for a point $(L(v), \ell)$, where v is the leaf node reached from the root by conceptually reading $S_j\$_j$. The data structure will report (resp. count) all of the suffixes of S_i, for $i \in [1, k]$, of length at least ℓ that are also prefixes of S_j. Unfortunately, such a solution differs from the expected results of $\mathsf{Report}(i, \ell)$ and $\mathsf{Count}(i, \ell)$ in the following two ways:

1. Instead of finding all $j \in [1, k]$ such that $\mathsf{SPL}_{i,j} \geq \ell$ for a given i, we find all such $i \in [1, k]$ for a given j. This issue is addressed by Observation 1, which states that $\mathsf{Report}(i, \ell)$ and $\mathsf{Count}(i, \ell)$ reduce trivially to the problems considered here, denoted by $\mathsf{Report}^r(i, \ell)$ and $\mathsf{Count}^r(i, \ell)$, respectively (recall that the r superscript refers to reversing the input strings);
2. If there are multiple prefixes of S_j of length at least ℓ that are also suffixes of S_i, then we will report (resp. count) each of them leading to double reporting (resp. counting). Although one may actually be interested in reporting or counting those multiple suffix-prefixes, in this paper, we are only interested in the *longest* ones. We address this issue by modifying the rectangles before the construction.

As mentioned earlier the first issue is resolved by Observation 1. To solve the second issue, we have to make the set of rectangles, for a single $i \in [1, k]$, pairwise disjoint while leaving their union unchanged. Notice that two such non-disjoint rectangles must come from a pair of nodes u and w in an ancestor-descendant relationship. An easy solution is to take, for every node w which has an outgoing edge with label $\$_i$, its closest ancestor u which also has an outgoing edge with label $\$_i$, and change the $[L(w), R(w)] \times [0, d(w)]$ rectangle into $[L(w), R(w)] \times [d(u) + 1, d(w)]$; inspect Figure 6. Since the part $[L(w), R(w)] \times [0, d(u)]$ is already contained in $[L(u), R(u)] \times [0, d(u)]$ the union remains unchanged, and since u is the closest such ancestor, the other rectangles (for this i) cannot have a nonempty intersection with the newly obtained one (the intersection with the ones coming from the descendants of w is empty after the modification of those rectangles). We can perform these

[2] $[L(u), R(u)]$ is also known as the suffix array interval of node u.



(a) Two intersecting rectangles implied by ST_R.

(b) Two ways to make the rectangles disjoint.

Figure 6 On the bottom left part, the rectangles obtained from two nodes u and w of ST_R (top left), both having an outgoing edge with label $\$_i$, forming a suffix-prefix match of S_i and S_j for node v reached by reading $S_j\$_j$ from the root. The rectangles have a nonempty intersection. To avoid double reporting (or double counting), we make the rectangles disjoint while leaving their union unchanged. We can do this (by taking the intersection *once*) in two ways (on the right): a simple one (top) or a more complicated one (bottom), which allows us to efficiently output $\mathsf{SPL}_{i,j}$.

modifications with a single DFS traversal with k stacks of nodes on the path from the root to the currently processed node, which has an outgoing edge with label $\$_i$, $i \in [1, k]$. A more complicated solution is obtained by replacing the two rectangles $[L(u), R(u)] \times [0, d(u)]$ and $[L(w), R(w)] \times [0, d(w)]$ with three rectangles: $[L(u), L(w) - 1] \times [0, d(u)]$, $[L(w), R(w)] \times [0, d(w)]$ and $[R(w)+1, R(u)] \times [0, d(u)]$; inspect Figure 6. Unlike the previous construction, a single rectangle can be spliced into smaller ones many times (a node can be a direct ancestor of many other nodes); at the same time a single rectangle can splice only its direct ancestor, hence the number of rectangles obtained this way is bounded from above by $2n$. This set of modified intervals can be obtained similarly: in a DFS traversal, when a node which has an outgoing edge with label $\$_i$ is reached, we access its closest ancestor, which also has an outgoing edge with label $\$_i$, and splice its rectangle. As such descendants of a node are visited from left to right, we always know which part of the rectangle will be spliced next, hence each such splice takes $\mathcal{O}(1)$ time leading to computing $\mathcal{O}(n)$ such modified rectangles in $\mathcal{O}(n)$ total time.

In order to finalize the construction of our data structure, we compute the set of modified rectangles of one of the two types described above, and construct for them the 2D rectangle stabbing data structures for reporting (Lemma 16) and counting (Lemma 17).

Querying To answer a $\mathsf{Report}^r(j, \ell)$ or a $\mathsf{Count}^r(j, \ell)$ query, we simply ask the corresponding 2D rectangle stabbing data structure for the point $(L(v), \ell) = (R(v), \ell)$, where v is the node reached in ST_R from the root by conceptually reading $S_j\$_j$. In case of a reporting query, the data structure returns a set of rectangles $[x, y] \times [\ell_1, \ell_2]$ labeled with distinct values $i \in [1, k]$. We can simply report the set of these i values. In case of a counting query, the result is simply an integer which we output. The two constructions of modified rectangles have additional nice properties however – each value i is associated with a value ℓ_2. In case of the first construction, this ℓ_2 is the length of the shortest suffix of S_i which is also a prefix of S_j of length at least ℓ; in case of the second construction, ℓ_2 is the length of the longest such suffix, that is $\ell_2 = \mathsf{SPL}_{i,j}$.

Correctness The correctness of the algorithm follows by the fact that point $(L(v), \ell) = (R(v), \ell)$ is enclosed by a rectangle $[L(u), R(u)] \times [0, d(u)]$ if and only if $S_j\$_j$ has a prefix of length at least ℓ that is also a suffix of S_i; and by the fact that the set of rectangles originating from a single i are made pairwise disjoint while their union remains unchanged.

We have thus arrived at the following lemma.

▶ **Lemma 18.** *For any dictionary of k strings of total length n over an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$, we can construct a data structure of $\mathcal{O}(n)$ words of space answering: (i) $\mathsf{Report}^r(j, \ell)$ queries in $\mathcal{O}(\log n / \log\log n + f)$ time, where f is the size of the output; and (ii) $\mathsf{Count}^r(j, \ell)$ queries in $\mathcal{O}(\log n / \log\log n)$ time.*

By combining Lemma 18 with Observation 1 we obtain the main result of this section.

▶ **Theorem 19.** *For any dictionary of k strings of total length n over an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$, we can construct a data structure of $\mathcal{O}(n)$ words of space answering: (i) $\mathsf{Report}(i, \ell)$ queries in $\mathcal{O}(\log n / \log\log n + f)$ time, where f is the output size; and (ii) $\mathsf{Count}(i, \ell)$ queries in $\mathcal{O}(\log n / \log\log n)$ time.*

Let us remark that the construction time for our data structures, excluding the implementation of the data structures underlying Lemmas 16 and 17, is $\mathcal{O}(n)$. Unfortunately, the construction time of the latter data structures (Lemmas 16 and 17) is not mentioned in [28, 45]. However, by using the construction from [15], we obtain $\mathcal{O}(n \log n)$ construction time, $\mathcal{O}(n)$ words of space, $\mathcal{O}(\log n + f)$ time for reporting, and $\mathcal{O}(\log n)$ time for counting.

▶ **Theorem 20.** *For any dictionary of k strings of total length n over an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$, we can construct a data structure of $\mathcal{O}(n)$ words of space answering: (i) $\mathsf{Report}(i, \ell)$ queries in $\mathcal{O}(\log n + f)$ time, where f is the output size; and (ii) $\mathsf{Count}(i, \ell)$ queries in $\mathcal{O}(\log n)$ time. The data structure construction time is $\mathcal{O}(n \log n)$.*

Let us also remark that $\mathsf{Report}(i, 0)$ (with the second construction of disjoint rectangles) actually answers any $\mathsf{One\text{-}to\text{-}All}(i)$ query within the same asymptotic time: $\mathcal{O}(\log n + f) = \mathcal{O}(\log n + k) = \mathcal{O}(k)$ as $k \geq \sqrt{n}$. While the data structure for answering Report queries occupies $\mathcal{O}(n)$ words of space, like the data structure for $\mathsf{One\text{-}to\text{-}All}$ queries, the construction time for the former is more expensive – and it is likely much slower in practice.

6 Answering Top Queries

Recall that a $\mathsf{Top}(i, K)$ query returns exactly K elements j for which $\mathsf{SPL}_{i,j}$ is the largest, breaking ties arbitrarily. In case we are given an additional bound $K' \leq k$ such that $K \leq K'$ (e.g., we are only interested in finding $\mathcal{O}(1)$ many such top elements), the obvious data structure would be to store, for each $i \in [1, k]$, the sorted list of size K' of the best answers. Such a data structure allows answering $\mathsf{Top}(i, K)$ queries, for $K \leq K'$, in the optimal $\mathcal{O}(K)$ time, but it requires $\mathcal{O}(kK')$ space, which for small K' may be $\mathcal{O}(n)$, but in general (i.e., when $K' = k$) leads back to the $\mathcal{O}(n + k^2)$-time APSP algorithm. We show how to use our results from Section 5 to answer $\mathsf{Top}(i, K)$ queries using $\mathcal{O}(n)$ space without this K' bound.

Clearly, we can assume that $K < k$. We start by making the following crucial observation.

▶ **Observation 21.** *For any $\mathsf{Top}(i, K)$ query, with $K < k$, there exists an integer $\ell \in [0, n-1]$ such that $\mathsf{Count}(i, \ell+1) \leq K < \mathsf{Count}(i, \ell)$.*

Using the results from Section 5, we can find such an ℓ in $\mathcal{O}(\log^2 n/\log\log n)$ time using binary search on $\ell \in [0, n-1]$ and the data structure for Count queries. Next we can simply compute $\mathsf{Report}(i, \ell+1)$ to be left with only choosing the remaining $(K - \mathsf{Count}(i, \ell+1))$ elements out of all $j \in [1, k]$ such that $\mathsf{SPL}_{i,j} = \ell$. Unfortunately, there can be many such elements (even k), and we do not want this to influence the query time. We have to report the remaining elements out of the ones such that $\mathsf{SPL}_{i,j} = \ell$ without computing or explicitly accessing all of them. Recall that, in ST_R, a list of elements i such that S_i has a suffix of length exactly ℓ which is also a prefix of S_j can be accessed in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$-time preprocessing by finding the ancestor of the node reached by conceptually reading $S_j\$_j$ at string depth ℓ (using a WA query) and reading the first letters of its outgoing edges from left to right; since $\$_1 < \cdots < \$_k$ are smaller than any element of Σ those values form a sorted list. Analogously, to access the list of elements j such that S_i has a suffix of length exactly ℓ which is also a prefix of S_j, we simply use the symmetric data structure by Observation 1.

Unfortunately, this list may contain elements j such that $\mathsf{SPL}_{i,j} > \ell$, and we do not want to report them again. This, however, can be fixed by maintaining a bitvector of size k as an integral part of our data structure; for each element $j \in \mathsf{Report}(i, \ell+1)$, we set the jth element of the bitvector to 1 in $\mathcal{O}(\mathsf{Count}(i, \ell+1)) = \mathcal{O}(K)$ time. When accessing the elements of the sorted list one-by-one, we simply check if the element was already outputted using the bitvector in $\mathcal{O}(1)$ time. In total, we can check up to K such elements, hence the total time of merging those two parts of the output is $\mathcal{O}(K)$ (including the bitvector reset). We summarize the solution in Theorem 22, which is the main result of this section.

▶ **Theorem 22.** *For any dictionary of k strings of total length n over an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$, we can construct a data structure of $\mathcal{O}(n)$ words of space answering $\mathsf{Top}(i, K)$ queries in $\mathcal{O}(\log^2 n/\log\log n + K)$ time.*

Proof. We start the construction of the data structure by constructing the data structures for $\mathsf{Report}(i, \ell)$ and $\mathsf{Count}(i, \ell)$ using Theorem 19. We also construct a data structure to find the list of elements j such that S_i has a suffix-prefix match of length ℓ with S_j in $\mathcal{O}(1)$ time using Lemmas 2 and 7 and Observation 1. Finally, we also maintain a bitvector of size $k = \mathcal{O}(n)$. The space required by our data structure is $\mathcal{O}(n)$ words.

Consider a $\mathsf{Top}(i, K)$ query. We ask $\mathcal{O}(\log n)$ Count queries and a single Report query in $\mathcal{O}(\log^2 n/\log\log n + K)$ total time, as the output is bounded by K. We index the Report result in the bitvector. We find the list (without reading its content) of elements j such that S_i has a suffix of length exactly ℓ which is also a prefix of S_j in $\mathcal{O}(1)$ time. Finally, we access and check at most K elements from the list in $\mathcal{O}(K)$ total time.

The correctness of the algorithm follows by Observation 21 and Theorem 19. ◀

Similar to Section 5, the construction time for our data structure, excluding the implementation of Theorem 19, is $\mathcal{O}(n)$. If instead of Theorem 19, we employ Theorem 20, we obtain $\mathcal{O}(n \log n)$ construction time, $\mathcal{O}(n)$ words of space, and $\mathcal{O}(\log^2 n + K)$ query time.

▶ **Theorem 23.** *For any dictionary of k strings of total length n over an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$, we can construct a data structure of $\mathcal{O}(n)$ words of space answering $\mathsf{Top}(i, K)$ queries in $\mathcal{O}(\log^2 n + K)$ time. The data structure construction time is $\mathcal{O}(n \log n)$.*

References

1 Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

2 Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Minimizing diameters of dynamic trees. In Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium, ICALP'97, Bologna, Italy, 7-11 July 1997, Proceedings*, volume 1256 of *Lecture Notes in Computer Science*, pages 270–280. Springer, 1997.

3 Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Dynamic and internal longest common substring. *Algorithmica*, 82(12):3707–3743, 2020.

4 Golnaz Badkobeh, Panagiotis Charalampopoulos, Dmitry Kosolobov, and Solon P. Pissis. Internal shortest absent word queries in constant time and linear space. *Theor. Comput. Sci.*, 922:271–282, 2022.

5 Carl Barton, Costas S. Iliopoulos, Solon P. Pissis, and William F. Smyth. Fast and simple computations using prefix tables under hamming and edit distance. In Jan Kratochvíl, Mirka Miller, and Dalibor Froncek, editors, *Combinatorial Algorithms - 25th International Workshop, IWOCA 2014, Duluth, MN, USA, October 15-17, 2014, Revised Selected Papers*, volume 8986 of *Lecture Notes in Computer Science*, pages 49–61. Springer, 2014.

6 Djamal Belazzougui, Dmitry Kosolobov, Simon J. Puglisi, and Rajeev Raman. Weighted ancestors in suffix trees revisited. In Pawel Gawrychowski and Tatiana Starikovskaya, editors, *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5-7, 2021, Wrocław, Poland*, volume 191 of *LIPIcs*, pages 8:1–8:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

7 Djamal Belazzougui and Gonzalo Navarro. Optimal lower and upper bounds for representing sequences. *ACM Trans. Algorithms*, 11(4):31:1–31:21, 2015.

8 Ilan Ben-Bassat and Benny Chor. String graph construction using incremental hashing. *Bioinform.*, 30(24):3515–3523, 2014.

9 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000.

10 Philip Bille and Inge Li Gørtz. The tree inclusion problem: In linear space and faster. *ACM Trans. Algorithms*, 7(3):38:1–38:47, 2011.

11 Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali, and Raffaella Rizzi. FSG: fast string graph construction for de novo assembly. *J. Comput. Biol.*, 24(10):953–968, 2017.

12 Bastien Cazaux and Eric Rivals. Hierarchical overlap graph. *Inf. Process. Lett.*, 155, 2020.

13 Panagiotis Charalampopoulos, Pawel Gawrychowski, Shay Mozes, and Oren Weimann. Almost optimal distance oracles for planar graphs. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 138–151. ACM, 2019.

14 Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal dictionary matching. *Algorithmica*, 83(7):2142–2169, 2021.

15 Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, 1988.

16 Shiri Chechik. Approximate distance oracles with constant query time. In David B. Shmoys, editor, *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 654–663. ACM, 2014.

17 Shiri Chechik. Approximate distance oracles with improved bounds. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *Proceedings of the Forty-Seventh Annual ACM on Symposium*

on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015, pages 1–10. ACM, 2015.

18 Nicola Cotumaccio, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Co-lexicographically ordering automata and regular languages. part I. *CoRR*, abs/2208.04931, 2022.

19 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.

20 Shiri Dori and Gad M. Landau. Construction of aho corasick automaton in linear time for integer alphabets. *Inf. Process. Lett.*, 98(2):66–72, 2006.

21 Herbert Edelsbrunner and Mark H. Overmars. On the equivalence of some rectangle problems. *Inf. Process. Lett.*, 14(3):124–127, 1982.

22 Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 137–143. IEEE Computer Society, 1997.

23 Martin Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In Daniel S. Hirschberg and Eugene W. Myers, editors, *Combinatorial Pattern Matching, 7th Annual Symposium, CPM 96, Laguna Beach, California, USA, June 10-12, 1996, Proceedings*, volume 1075 of *Lecture Notes in Computer Science*, pages 130–140. Springer, 1996.

24 Pawel Gawrychowski, Moshe Lewenstein, and Patrick K. Nicholson. Weighted ancestors in suffix trees. In Andreas S. Schulz and Dorothea Wagner, editors, *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, volume 8737 of *Lecture Notes in Computer Science*, pages 455–466. Springer, 2014.

25 Giorgio Gonnella and Stefan Kurtz. Readjoiner: a fast and memory efficient string graph-based sequence assembler. *BMC Bioinform.*, 13:82, 2012.

26 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

27 Dan Gusfield, Gad M. Landau, and Baruch Schieber. An efficient algorithm for the all pairs suffix-prefix problem. *Inf. Process. Lett.*, 41(4):181–185, 1992.

28 Joseph F. JáJá, Christian Worm Mortensen, and Qingmin Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In Rudolf Fleischer and Gerhard Trippen, editors, *Algorithms and Computation, 15th International Symposium, ISAAC 2004, Hong Kong, China, December 20-22, 2004, Proceedings*, volume 3341 of *Lecture Notes in Computer Science*, pages 558–568. Springer, 2004.

29 Shahbaz Khan. Optimal construction of hierarchical overlap graphs. In Pawel Gawrychowski and Tatiana Starikovskaya, editors, *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5-7, 2021, Wrocław, Poland*, volume 191 of *LIPIcs*, pages 17:1–17:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

30 Tomasz Kociumaka. Efficient data structures for internal queries in texts. *PhD thesis, University of Warsaw, October 2018.*, 2018.

31 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal pattern matching queries in a text and applications. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 532–551. SIAM, 2015.

32 Gregory Kucherov and Dekel Tsur. Improved filters for the approximate suffix-prefix overlap problem. In Edleno Silva de Moura and Maxime Crochemore, editors, *String Processing and Information Retrieval - 21st International Symposium, SPIRE 2014, Ouro Preto, Brazil, October 20-22, 2014. Proceedings*, volume 8799 of *Lecture Notes in Computer Science*, pages 139–148. Springer, 2014.

33 Jihyuk Lim and Kunsoo Park. A fast algorithm for the all-pairs suffix-prefix problem. *Theor. Comput. Sci.*, 698:14–24, 2017.

34 Grigorios Loukides and Solon P. Pissis. All-pairs suffix/prefix in optimal time using Aho-Corasick space. *Inf. Process. Lett.*, 178:106275, 2022.

35 Felipe A. Louza, Simon Gog, Leandro Zanotto, Guido Araujo, and Guilherme P. Telles. Parallel computation for the all-pairs suffix-prefix problem. In Shunsuke Inenaga, Kunihiko Sadakane, and Tetsuya Sakai, editors, *String Processing and Information Retrieval - 23rd International Symposium, SPIRE 2016, Beppu, Japan, October 18-20, 2016, Proceedings*, volume 9954 of *Lecture Notes in Computer Science*, pages 122–132, 2016.

36 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

37 Eugene W. Myers. The fragment assembly string graph. *Bioinformatics*, 21(suppl_2):ii79–ii85, 09 2005.

38 Gonzalo Navarro. *Compact Data Structures - A Practical Approach*. Cambridge University Press, 2016.

39 Enno Ohlebusch and Simon Gog. Efficient algorithms for the all-pairs suffix-prefix problem and the all-pairs substring-prefix problem. *Inf. Process. Lett.*, 110(3):123–128, 2010.

40 Sangsoo Park, Sung Gwan Park, Bastien Cazaux, Kunsoo Park, and Eric Rivals. A linear time algorithm for constructing hierarchical overlap graphs. In Pawel Gawrychowski and Tatiana Starikovskaya, editors, *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5-7, 2021, Wrocław, Poland*, volume 191 of *LIPIcs*, pages 22:1–22:9. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

41 Mihai Patrascu and Liam Roditty. Distance oracles beyond the thorup-zwick bound. *SIAM J. Comput.*, 43(1):300–311, 2014.

42 Maan Haj Rachid and Qutaibah Malluhi. A practical and scalable tool to find overlaps between sequences. *BioMed Res. Int.*, 2015(905261), 2015.

43 Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In David Eppstein, editor, *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA*, pages 233–242. ACM/SIAM, 2002.

44 Kim R. Rasmussen, Jens Stoye, and Eugene W. Myers. Efficient q-gram filters for finding all *epsilon*-matches over a given length. *J. Comput. Biol.*, 13(2):296–308, 2006.

45 Qingmin Shi and Joseph F. JáJá. Novel transformation techniques using q-heaps with applications to computational geometry. *SIAM J. Comput.*, 34(6):1474–1492, 2005.

46 Jared T. Simpson and Richard Durbin. Efficient construction of an assembly string graph using the fm-index. *Bioinform.*, 26(12):367–373, 2010.

47 Sharma V. Thankachan, Chaitanya Aluru, Sriram P. Chockalingam, and Srinivas Aluru. Algorithmic framework for approximate matching under bounded edits with applications to sequence analysis. In Benjamin J. Raphael, editor, *Research in Computational Molecular Biology - 22nd Annual International Conference, RECOMB 2018, Paris, France, April 21-24, 2018, Proceedings*, volume 10812 of *Lecture Notes in Computer Science*, pages 211–224. Springer, 2018.

48 Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005.

49 William H. A. Tustumi, Simon Gog, Guilherme P. Telles, and Felipe A. Louza. An improved algorithm for the all-pairs suffix-prefix problem. *J. Discrete Algorithms*, 37:34–43, 2016.

50 Esko Ukkonen. A linear-time algorithm for finding approximate shortest common superstrings. *Algorithmica*, 5(3):313–323, 1990.

51 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

52 Niko Välimäki, Susana Ladra, and Veli Mäkinen. Approximate all-pairs suffix/prefix overlaps. *Inf. Comput.*, 213:49–58, 2012.

53 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11. IEEE Computer Society, 1973.