Space-Time Trade-Offs for the Shortest Unique Substring Problem

Arnab Ganguly*[1], Wing-Kai Hon†[2], Rahul Shah[3], and Sharma V. Thankachan[4]

1 School of EECS, Louisiana State University, Baton Rouge, USA
agangu4@lsu.edu
2 Department of CS, National Tsing Hua University, Taiwan
wkhon@cs.nthu.edu.tw
3 School of EECS, Louisiana State University, Baton Rouge, USA; and
National Science Foundation, USA
rahul@csc.lsu.edu, rahul@nsf.gov
4 Department of CS, University of Central Florida, Orlando, USA
sharma.thankachan@gmail.com

Abstract

Given a string $\mathsf{X}[1,n]$ and a position $k \in [1,n]$, the *Shortest Unique Substring* of X covering k, denoted by S_k, is a substring $\mathsf{X}[i,j]$ of X which satisfies the following conditions: (i) $i \leq k \leq j$, (ii) i is the only position where there is an occurrence of $\mathsf{X}[i,j]$, and (iii) $j-i$ is minimized. The best-known algorithm [Hon et al., ISAAC 2015] can find S_k for all $k \in [1,n]$ in time $\mathcal{O}(n)$ using the string X and additional $2n$ words of working space. Let τ be a given parameter. We present the following new results. For any given $k \in [1,n]$, we can compute S_k via a deterministic algorithm in $\mathcal{O}(n\tau^2 \log \frac{n}{\tau})$ time using X and additional $\mathcal{O}(n/\tau)$ words of working space. For every $k \in [1,n]$, we can compute S_k via a deterministic algorithm in $\mathcal{O}(n\tau^2 \log n)$ time using X and additional $\mathcal{O}(n/\tau)$ words and $4n + o(n)$ bits of working space. For both problems above, we present an $\mathcal{O}(n\tau \log^{c+1} n)$-time randomized algorithm that uses $n/\log^c n$ words in addition to that mentioned above, where $c \geq 0$ is an arbitrary constant. In this case, the reported string is unique and covers k, but with probability at most $n^{-\mathcal{O}(1)}$, may not be the shortest. As a consequence of our techniques, we also obtain similar space-and-time tradeoffs for a related problem of finding Maximal Unique Matches of two strings [Delcher et al., Nucleic Acids Research 1999].

1998 ACM Subject Classification F.2.2 Pattern Matching

Keywords and phrases Suffix Tree, Sparsification, Rabin-Karp Fingerprint, Probabilistic z-Fast Trie, Succinct Data-Structures

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2016.34

1 Introduction

We consider a string $\mathsf{X}[1,n]$ with characters from an ordered alphabet Σ of cardinality σ. The ith character, $i \in [1,n]$, is denoted by $\mathsf{X}[i]$, and $\mathsf{X}[i,j]$, $1 \leq i \leq j \leq n$, is the substring $\mathsf{X}[i]\ \mathsf{X}[i+1] \ldots \mathsf{X}[j]$. We denote by $|\mathsf{X}[i,j]|$ the length $(j-i+1)$ of the substring $\mathsf{X}[i,j]$. A *suffix starting at* i is the string $\mathsf{X}[i,n]$ and a *prefix ending at* i is the string $\mathsf{X}[1,i]$. A *right*

* The work of Arnab Ganguly was supported by National Science Foundation Grants CCF–1218904 and CCF–1527435.
† The work of Wing-Kai Hon was supported by MOST Grant 105-2918-I-007-006 and MOST Grant 102-2221-E-007-068-MY3.

extension of $\mathsf{X}[i,j]$ is a string $\mathsf{X}[i,j']$, where $j' > j$. A substring $\mathsf{X}[i,j]$ *covers a position* k iff $i \le k \le j$. A substring $\mathsf{X}[i,j]$ is *unique* iff $\mathsf{X}[i,n]$ is the only suffix having $\mathsf{X}[i,j]$ as a prefix. A substring $\mathsf{X}[i,j]$ is *repeating* iff there exists $i' \neq i$ such that $\mathsf{X}[i,j]$ is a prefix of $\mathsf{X}[i',n]$.

▶ **Definition 1** (Shortest Unique Substring Covering k)**.** A substring $\mathsf{X}[i,j]$ is a shortest unique substring covering a position k iff (i) $\mathsf{X}[i,j]$ covers k, and (ii) there does not exist a substring $\mathsf{X}[i',j']$ that covers k and satisfies $j' - i' < j - i$.

We now present the following problems that will be discussed in the rest of the paper. Throughout this paper, we will use S_k to denote any shortest unique substring of X covering k. Note that there may be multiple choices of S_k.

▶ **Problem 2** (Single k)**.** *Given* $\mathsf{X}[1,n]$ *and a position* $k \in [1,n]$*, find any* S_k*.*

▶ **Problem 3** (All k)**.** *Given* $\mathsf{X}[1,n]$*, find any* S_k *for every* $k \in [1,n]$*.*

Previous Works and Our Contribution. To the best of our knowledge, the formal definitions presented in Problems 2 and 3 were introduced by Pei et al. [19]. They also listed several potential applications, for e.g., document searching on the internet. Arguably, the most important applications lie in the field of Computational Biology. A few of them are (see [19] and references therein): finding unique DNA signatures between closely related organisms, aiding polymerase chain reaction (PCR) primer design, genome mapability, and next-generation short reads sequencing.

For Problem 2, Pei et al. [19] presented an $\mathcal{O}(n)$ time and $\Theta(n)$ space (in words) solution. For the second problem, their method incurred a time of $\mathcal{O}(n \cdot h)$, where h is a variable which for most practical purposes can be taken to be a constant. In the worst-case, however, h is $\mathcal{O}(n)$; therefore, their solution takes $\mathcal{O}(n^2)$ time, the space remains at $\Theta(n)$ words. This is the first drawback of their approach. More importantly though, their solution is intrinsically based on the *Suffix Tree* of the string X. A suffix tree [11] ST of a string $S[1,m]$ is a compacted trie on the set of all non-empty suffixes of the string S. The suffix tree has m leaves (one per each suffix), and at most $(m-1)$ internal nodes. The leaves in the suffix tree are arranged from left-to-right in the lexicographic order of the corresponding suffix they represent. The space occupied is $\Theta(m)$ words, or equivalently $\Theta(m \log m)$ bits. (We assume the standard Word-RAM model of computation, where the size of a machine word is $\Theta(\log m)$ bits. Also, all logarithms are in base 2.)

Unfortunately, for most practical purposes, the suffix tree of a string S occupies space much larger (15-50 times) compared to the $|S| \log |\Sigma_S|$ bits of space needed by S. Here, Σ_S is the alphabet from which the characters in S are drawn. (Typically, $\Sigma_S = \{1, 2, \ldots, |\Sigma_S|\}$.) The space occupancy issue becomes more profound in the case when strings are much larger in comparison to the size of the alphabet. An example is the DNA, in which the alphabet has size four, but the lengths of the strings (such as in Human Genome) are typically in the billions. Even with a space-efficient implementation, such as in [16], a suffix tree occupies 40 Gigabytes, whereas the input Human Genome occupies only 700 Megabytes. Since a primary application of the Shortest Unique Substring (SUS) problem involves DNA, this presents a serious bottleneck, as has been corroborated by the experimental results of Ileri et al. [14], who were unable to run the algorithm of Pei et al. [19] for massive data sizes.

To alleviate the running time of $\mathcal{O}(n^2)$ for Problem 3, Ileri et al. [14] introduced an $\mathcal{O}(n)$-time and $\Theta(n)$-word algorithm. More importantly, their algorithm is more space-efficient than the algorithm of Pei et al. [19]. They showed that their algorithm not only saves space by a factor of 20, but also attains a speedup by a factor of 4. The space efficiency is achieved by replacing the suffix tree with a combination of the *Suffix Array*, its inverse, and the *LCP array* of the string X. The suffix array [11] of $S[1,m]$ is an array $\mathsf{SA}_S[1,m]$ such that $\mathsf{SA}_S[i]=j$ iff $S[j,m]$ is the ith lexicographically smallest suffix. The inverse suffix array $\mathsf{SA}_S^{-1}[1,m]$ is an array such that $\mathsf{SA}_S^{-1}[j]=i$ iff $\mathsf{SA}_S[i]=j$. The Longest Common Prefix (LCP) array [11] of S is an array $\mathsf{LCP}[1,m]$ such that $\mathsf{LCP}[m]=-1$ and for $i<m$, $\mathsf{LCP}[i]$ equals the length of the LCP of the suffixes starting at $\mathsf{SA}_S[i]$ and $\mathsf{SA}_S[i+1]$. Hon et al. [13] achieved further space improvements by introducing an in-place framework. Specifically, their algorithm needs space $2n$ words in addition to that needed for storing the string X. Remarkably, the time needed to compute S_k for every k still remains $\mathcal{O}(n)$. Furthermore, they argued that $2n$ words is the minimum space needed to store S_k values explicitly, as we need to store the start position and length of each S_k.

Despite all the efforts that have been invested into the SUS problems, the current best solution of Hon et al. [13] still uses $2n$ words of space in addition to the space needed by the input string X. Therefore, an important question is whether we can solve the problems using $o(n)$ words of additional space. We consider the following sub-linear space setting. In addition to the input string X of length n, a parameter τ is provided. The task is to find S_k using space $\mathcal{O}(n/\tau)$ words in addition to the space needed for storing X. In this setting, we present the following solutions for Problems 2 and 3.

- For any given $k\in[1,n]$, we can compute S_k via a deterministic algorithm in $\mathcal{O}(n\tau^2\log\frac{n}{\tau})$ time using $\mathcal{O}(n/\tau)$-words of additional working space.
- For every $k\in[1,n]$, we can compute S_k via a deterministic algorithm in $\mathcal{O}(n\tau^2\log n)$ time using $\mathcal{O}(n/\tau)$-words and $4n+o(n)$-bits of additional working space.

We assume $\tau=\omega(1)$. Otherwise, we can simply use the algorithm of Hon et al. [13]. Thus, we present the first algorithm which needs $o(n)$ words of additional space for computing SUS. We also present a randomized algorithm which reduces the above running time to $\mathcal{O}(n\tau\log^{c+1}n)$ by using an additional $n/\log^c n$ words, where $c\geq 0$ is any arbitrary constant. Each computed S_k is unique and covers k, but with probability at most $n^{-\mathcal{O}(1)}$, may not be the shortest. Note that in this case, even by choosing $c=0$, our space requirements are strictly better (in the asymptotic sense) than that of Hon et al. [13]. By choosing $\tau=\log n$, our algorithm achieves a space-factor improvement of $\mathcal{O}(\log n)$, while matching the best known running time of $\mathcal{O}(n)$ within poly-logarithmic factors.

We remark that our techniques imply (almost) the same results (compact space and succinct index) attained by Belazzougui and Cunial [2] for a related problem of finding the shortest unique prefix of every suffix of X. Our techniques also imply the first sub-linear space algorithm for the related problem of finding Maximal Unique Matches (MUM) of two strings [6].

Roadmap. We first present the two deterministic algorithms in Sections 2 and 3 respectively. Section 4 introduces the randomized algorithms. A brief discussion on the MUM problem [6] is presented Section 5.

2 Deterministic Algorithm for Single k

We begin with the following key observation.

▶ **Observation 4.** S_k *is either the shortest unique prefix of a suffix that starts at a position* $i\leq k$*, or is the smallest right extension till* k *of such a prefix.*

With this key intuition, we define LS_i as the shortest unique prefix of the suffix $\mathsf{X}[i,n]$.

▶ **Observation 5.** LS_1 *is defined, whereas* LS_i *for* $i > 1$ *may not be defined. If* LS_i *is not defined, then for any* $i' > i$, $\mathsf{LS}_{i'}$ *is also not defined.*

For any $i \leq k$, we define LS_i^k as LS_i if LS_i covers k; otherwise, LS_i^k is the right extension of LS_i up to the position k, i.e., $\mathsf{LS}_i^k = \mathsf{LS}_i \circ \mathsf{X}[i + |\mathsf{LS}_i|, k]$, where $\circ$ denotes concatenation. By this definition, S_k is a minimum length LS_i^k, where $i \leq k$ and LS_i is defined. Moving forward, we will represent S_k by two integers: the starting position of S_k and the length $|\mathsf{S}_k|$.

We first present the general idea behind the previous works. Once we know LS_i for every $i \leq k$, where defined, we first compute LS_i^k. Following this, S_k is computed simply by selecting a LS_i^k of minimum length. Specifically, start at $i = 1$, and compute the longest repeating prefix of $\mathsf{X}[i, n]$. Using the inverse suffix array and the LCP array, this can be easily computed. If the length of this prefix is $(n - i + 1)$, then clearly LS_i is not defined. Otherwise, compute LS_i^k from LS_i, and repeat the process with $(i + 1)$. Finally, compute the minimum length LS_i^k, once we reach an i such that either LS_i is not defined, or $i > k$.

In our case, we cannot construct the entire suffix array and LCP array, as it will violate our space constraints. Also, storing all the LS_i or LS_i^k values is not an option, as in the worst-case the space will become $\Theta(n)$ words. Therefore, we will compute LS_i for a carefully chosen set of $\mathcal{O}(n/\tau)$ suffixes. Based on this, we present the following crucial lemma.

▶ **Lemma 6.** *Let* $\mathcal{I}_i = \{i, i + \tau, i + 2\tau, \dots\}$, $i \in [1, \tau]$, *be a set of at most* $\lceil n/\tau \rceil$ *suffixes. For every* $i' \in \mathcal{I}_i$, *we can compute* $\mathsf{LS}_{i'}$ *in* $\mathcal{O}(n\tau \log \frac{n}{\tau})$ *time using* X *and additional* $\mathcal{O}(n/\tau)$ *words of working space.*

Using the above lemma, we prove the following theorem, which presents our first result.

▶ **Theorem 7.** *For any given* $k \in [1, n]$, *we can find* S_k *in* $\mathcal{O}(n\tau^2 \log \frac{n}{\tau})$ *time using* X *and additional* $\mathcal{O}(n/\tau)$ *words of working space.*

Proof. Initialize $S = n$ and $sp = 1$. Using Lemma 6, we first compute LS_j for every $j \in \mathcal{I}_i$ by choosing $i = 1$. Use LS_j to compute LS_j^k for every $j \in \mathcal{I}_1$. Assign $S = \min_{j \in \mathcal{I}_i}\{S, |\mathsf{LS}_j^k|\}$. If S is updated, then assign sp to the corresponding j. Repeat the process with $i = 2, 3, \dots, \tau$. Now, it remains to find LS_j for all suffixes with $j \in [n - \tau + 2, n]$. To find LS_j and LS_j^k for these suffixes simply use a brute-force approach. Since there are $\tau - 1$ suffixes, each of length at most $\tau - 1$, the time needed is $\mathcal{O}(n\tau^2)$. At each step, update S and sp accordingly. Finally, S_k is given by S and sp. Clearly, the claimed time and space bounds are met. ◀

2.1 Proof of Lemma 6

The central idea behind the proof is the use of a sparsification technique introduced by Hon et al. [12]. In particular, we create a sampled suffix tree ST_i by using a set of roughly n/τ regularly spaced suffixes, where the first suffix starts at position i. Now, we match the string X in ST_i, starting with the position $j = 1$ if $i > 1$, and with $j = 2$ otherwise. Using ST_i, we can find the longest repeating prefix of every sampled suffix w.r.t the positions $j, j + \tau, j + 2\tau, \dots$. Then the process is repeated with every value of $j \in [1, \tau]$, where $j \neq i$. Finally, we use the longest repeating prefix of each sampled suffix, and extend it by one character to find $\mathsf{LS}_{i'}$ for each $i' \in \mathcal{I}_i$. We now present the details.

Pre-process: Consider every substring of X of length τ that starts at a position which lies in the set $\mathcal{I}_i = \{i, i + \tau, i + 2\tau, \dots\}$. We first create a compacted trie $\mathcal{T}$ of these substrings, and ignore the last substring, say X_i', of X if it has length less than τ. While creating $\mathcal{T}$, for every node u, store in a balanced binary search tree (BST) the first characters that label the edges starting from u. This BST will allow us to efficiently select a correct edge (or create a new one) when a new string is inserted. Since the number of strings considered is at most $\lceil n/\tau \rceil$, the number of nodes in $\mathcal{T}$ is at most $2\lceil n/\tau \rceil$. Likewise, at any moment, the number of nodes in all the BSTs combined is at most $2\lceil n/\tau \rceil$, implying a search or insert operation requires $\mathcal{O}(\log \frac{n}{\tau})$ time. Clearly, the space needed to create $\mathcal{T}$ is $\mathcal{O}(n/\tau)$, and the time required is $\mathcal{O}(n + \frac{n}{\tau} \log \frac{n}{\tau})$. Note that each τ-length substring corresponds to a (not necessarily unique) leaf in $\mathcal{T}$, where the leaves are numbered according to the lexicographic order of the substring they represent. We create a new (compressed) string X_i^τ by mapping each τ-length substring of X starting at a position in $\mathcal{I}_i$ to the corresponding leaf number. (We ignore the string X_i' and the characters before i while creating X_i^τ.) Let Σ_i denote the alphabet of X_i^τ. Note that $\Sigma_i^\tau = \{1, 2, \ldots, |\Sigma_i^\tau|\}$, where $|\Sigma_i^\tau| \leq \lceil \frac{n}{\tau} \rceil$ is the number of leaves in $\mathcal{T}$. Also note that for any two integers $p, q \in \Sigma_i^\tau$, $p < q$ iff the string corresponding to leaf p in $\mathcal{T}$ is lexicographically smaller than the one corresponding to q.

Construct a suffix tree ST_i of $\mathsf{X}_i^\tau\$$, where \$ is a unique special character. Since $|\mathsf{X}_i^\tau| \leq \lceil n/\tau \rceil$, the number of nodes in ST_i is at most $2\lceil n/\tau \rceil$. Append X_i' to the label (ignoring \$) on each edge from a leaf to its parent. We remark that the edge labels are not explicitly written, but are obtained using two pointers to the start and end positions of the label in X. Each non-root node u in ST_i has a *suffix link* pointing to a node $\Psi(u)$, such that the string (over Σ) obtained by concatenating the edge labels from root to $\Psi(u)$ is same as the string from root to u with the first τ characters truncated. By using the algorithm of Farach-Colton [8], constructing ST_i along with the suffix links requires $\mathcal{O}(n/\tau)$ time and space. Now, consider the set E_u of outgoing edges of a node u. We will order them from left-to-right according to the lexicographic order of the τ-length substring of X represented by the first character. Since the lexicographic rank of the τ-length strings can be compared directly in $\mathcal{O}(1)$ time based on its leaf index in $\mathcal{T}$ (i.e., based on its representative in Σ_i^τ), we can order the edges in all such sets E_u in $\sum_u \mathcal{O}(|E_u| \log |E_u|) = \mathcal{O}((n/\tau) \log \frac{n}{\tau})$ time using $\mathcal{O}(n/\tau)$ space. Each leaf in ST_i corresponds to a suffix of X with starting position in $\mathcal{I}_i$, where leaves are numbered from left-to-right in lexicographic order of the suffix they represent. For the pth leftmost leaf, denoted by ℓ_p, let $\mathsf{SA}_i[p]$ be the suffix array value, i.e., the starting position in $\mathcal{I}_i$ of the corresponding suffix. Summarizing, the time needed to construct ST_i is $\mathcal{O}(n + \frac{n}{\tau} \log \frac{n}{\tau})$, and the space usage is $\mathcal{O}(n/\tau)$ words. We create a compacted trie $\mathsf{ST}_i(u)$ with the edges in E_u by mapping the edge labels over Σ_i^τ to the corresponding τ-length string over Σ. Call this the *navigation trie of node u*. Note that each leaf in $\mathsf{ST}_i(u)$ corresponds to a unique child of u in ST_i. As before, the edge labels are obtained using two pointers to X. The outgoing edges of a node in the navigation trie are ordered based on the lexicographic order of the first character from Σ, such that given a character $x \in \Sigma$, we can find the outgoing edge (if any) beginning with x in $\mathcal{O}(\log \frac{n}{\tau})$ time. The number of nodes in all navigation tries combined is at most $2\lceil n/\tau \rceil$. Since the first τ-length strings labeling the outgoing edges of a node are distinct, all navigation tries can be created in $\mathcal{O}(n + \frac{n}{\tau} \log \frac{n}{\tau})$ time.

Equip ST_i with the data structures in [3, 4], such that in $\mathcal{O}(1)$ time, we can (i) find $\mathsf{lca}(u, v)$ i.e., the Lowest Common Ancestor (LCA) of two nodes u and v, and (ii) find $\mathsf{levelAncestor}(u, W)$ i.e., the ancestor of u which has node-depth W. Likewise, equip all navigation tries with these data structures. Using these, given two leaves ℓ_k and $\ell_{k'}$ in ST_i, we can easily find their LCA in a particular navigation trie in $\mathcal{O}(1)$ time. For any node u in ST_i, let $\mathsf{path}(u)$ denote the string formed by concatenating the edge labels over Σ from root to u. Likewise, for any node u^* in a navigation trie $\mathsf{ST}_i(u)$, let $\mathsf{path}(u^*)$ be the string $\mathsf{path}(u)$ appended with the edge labels from u to u^*. Store $|\mathsf{path}(u)|$ (resp. $|\mathsf{path}(u^*)|$) at each node u in ST_i (resp. u^* in $\mathsf{ST}_i(u)$). The space and time required for these pre-processing steps can both be bounded by $\mathcal{O}(n/\tau)$. With the aid of these pre-processing steps, in $\mathcal{O}(1)$ time, we can find $\mathsf{lcp}(\ell_k, \ell_{k'})$ i.e., the Longest Common Prefix (LCP) of $\mathsf{path}(\ell_k)$ and $\mathsf{path}(\ell_{k'})$.

Query: Initially, each node u^* in every navigation trie is *unmarked*; also, assign $\Delta(u^*) = 0$. Starting with $j = 1$ (if $i > 1$, and with $j = 2$ otherwise), we match successive symbols of the string $\mathsf{X}[j, n]$ in ST_i as follows. Suppose, we are at a node u in ST_i. Find the correct edge (if any) in $\mathsf{ST}_i(u)$ to traverse using the character $\mathsf{X}[j + |\mathsf{path}(u)|]$. Now, use the characters starting from $\mathsf{X}[j + |\mathsf{path}(u)| + 1]$ to traverse $\mathsf{ST}_i(u)$ until either we reach a leaf ℓ^* (which corresponds to a child u' of u in ST_i), or we find a mismatch. In the first case, mark the leaf ℓ^*, set $\Delta(\ell^*) = |\mathsf{path}(\ell^*)|$, and repeat the process from u'. Otherwise, suppose we find a failure on an edge to a node v^* in $\mathsf{ST}_i(u)$ after successfully matching D characters starting from u. *Mark* the node v^*, and store $\Delta(v^*) = \max\{\Delta(v^*), |\mathsf{path}(u)| + D\}$. Follow the suffix link of u to the node $\Psi(u)$. We have the following two cases to consider.

- If $D < \tau$, then use the string $\mathsf{X}[j + |\mathsf{path}(u)|, j + |\mathsf{path}(u)| + D - 1]$ and traverse $\mathsf{ST}_i(\Psi(u))$. Then, we resume matching from the reached position using $\mathsf{X}[j + |\mathsf{path}(u)| + D, n]$.
- If $D \geq \tau$, then v^* is a leaf in $\mathsf{ST}_i(u)$ and represents a child v of u in ST_i. Use the string $\mathsf{X}[j + |\mathsf{path}(u)|, j + |\mathsf{path}(u)| + \tau - 1]$ and traverse $\mathsf{ST}_i(\Psi(u))$. At this point, we are on an edge from a node w^* to a leaf node in $\mathsf{ST}_i(\Psi(u))$. The desired position to resume matching on this edge is given by $(D - |\mathsf{path}(w^*)| + 1)$.

In either case, we compare at most τ characters. Observe that on following a suffix link we truncate τ characters starting from j, and are now trying to match $\mathsf{X}[j + \tau, n]$. Therefore, the total time needed to mark a node for $j, j + \tau, j + 2\tau, \ldots$ is $\mathcal{O}(n \log \frac{n}{\tau})$. Repeat this process for every value of $j \in [1, \tau]$, $j \neq i$. The total time needed is $\mathcal{O}(n\tau \log \frac{n}{\tau})$.

We initialize an array LS of length $|\mathcal{I}_i|$ as follows. Assign $\mathsf{LS}[1] = \mathsf{lcp}(\ell_1, \ell_2)$, and $\mathsf{LS}[p] = \max\{\mathsf{lcp}(\ell_{p-1}, \ell_p), \mathsf{lcp}(\ell_p, \ell_{p+1})\}$, where $p \in [2, |\mathcal{I}_i| - 1]$. Finally, $\mathsf{LS}[|\mathcal{I}_i|] = \mathsf{lcp}(\ell_{|\mathcal{I}_i|-1}, \ell_{|\mathcal{I}_i|})$. Now, for each leaf ℓ_p in ST_i, we find its nearest marked ancestor ℓ_p^*. This is easily achieved in $\mathcal{O}(n/\tau)$ time and space by traversing ST_i and the navigation tries using lca and $\mathsf{levelAncestor}$ queries. Simply assign $\mathsf{LS}[p] = 1 + \max\{\mathsf{LS}[p], \Delta(\ell_p^*)\}$. If $\mathsf{LS}[p] = |\mathsf{path}(\ell_p)|$, then assign $\mathsf{LS}[p] = n + 1$. (This implies that LS value for the position $\mathsf{SA}_i[p]$ is not defined.) Clearly, $\mathsf{LS}[p]$ and $\mathsf{SA}_i[p]$ together give us $\mathsf{LS}_{\mathsf{SA}_i[p]}$. The time needed is $\mathcal{O}(n/\tau)$.

3 Deterministic Algorithm for All k

▶ **Observation 8** ([13, 14]). $|\mathsf{LS}_k| \leq |\mathsf{LS}_{k+1}| + 1$*,* $k \in [1, n-1]$*, where* LS_k *and* LS_{k+1} *are defined.* S_1 *is the same as* LS_1*. For any* $k \in [2, n]$*, if* S_k *is the right extension of some* $\mathsf{LS}_{k'}$*,* $k' < k$*, then* (i) S_{k-1} *ends at the position* $(k-1)$*, and* (ii) $\mathsf{S}_k = \mathsf{S}_{k-1} \circ \mathsf{X}[k]$*.*

Assume that S_{k-1}, $k > 1$, is computed, where $\mathsf{S}_1 = \mathsf{LS}_1$ is known. We want to compute S_k. The following are immediate from the above observation. If S_{k-1} does not end at $(k-1)$, then S_k is simply the shortest $\mathsf{LS}_{k'}$ that covers k. (Note that such an $\mathsf{LS}_{k'}$ must exist.) Otherwise, S_{k-1} ends at $(k-1)$, and S_k is simply the shorter of: (i) the shortest $\mathsf{LS}_{k'}$, $k' \leq k$, that covers k, if such a string exists, and (ii) $\mathsf{S}_{k-1} \circ \mathsf{X}[k]$. Thus the focus is to compute the shortest $\mathsf{LS}_{k'}$ that covers k, if such a string exists. We prove the following theorem.

▶ **Theorem 9.** *We can compute* S_k *for every* $k \in [1, n]$ *in* $\mathcal{O}(n\tau^2 \log n)$ *time using* X *and additional* $\mathcal{O}(n/\tau)$ *words and* $4n + o(n)$ *bits of working space.*

Following are a couple of well-known results that will be needed.

▶ **Fact 10** (Munro [18]). *Consider a binary string $B[1,m]$. By using a data structure occupying $o(m)$ bits, in $\mathcal{O}(1)$ time, we can find* (i) $\mathsf{rank}(i,c) = |\{j \le i \mid B[j] = c\}|$ *and* (ii) $\mathsf{select}(j,c) = \min_i\{i \mid \mathsf{rank}(i,c) = j\}$, *where $c \in \{0,1\}$. The data structure can be constructed in $\mathcal{O}(m)$ time using $o(m)$ bits of working space in addition to the string B.*

▶ **Fact 11** (Fischer and Heun [9]). *Consider an array A of m integers. By using a data structure occupying $2m + o(m)$ bits, in $\mathcal{O}(1)$ time, we can find $\mathsf{rmq}_A(i,j)$ i.e., a position $t \in [i,j]$ such that $A[t] = \min\{A[t'] \mid t' \in [i,j]\}$. The data structure can be constructed in $\mathcal{O}(m)$ time using $2m + o(m)$ bits of working space in addition to the array A.*

Proof of Theorem 9. The key idea is to compute LS_j for all values of j, where defined, and then store it in a compact way. Specifically, use Lemma 6 to compute LS_j for every $j \in \mathcal{I}_i = \{i, i+\tau, i+2\tau, \dots\}$, first by choosing $i = 1$. Store these values explicitly, and initialize $|\mathcal{I}_1|$ empty binary strings $B_1, B_2, \dots, B_{|\mathcal{I}_1|}$. Compute LS_j for every $j \in \mathcal{I}_2$. For each $j \in \mathcal{I}_2$, append $(|\mathsf{LS}_j| + 1 - |\mathsf{LS}_{j-1}|)$ many 1s followed by a 0 to the binary string $B_{\lceil j/\tau \rceil}$. (Note that $(j-1) \in \mathcal{I}_1$, and LS_{j-1} has already been computed.) Now, compute LS_j for every $j \in \mathcal{I}_3$. For each $j \in \mathcal{I}_3$, append $(|\mathsf{LS}_j| + 1 - |\mathsf{LS}_{j-1}|)$ many 1s followed by a 0 to the binary string $B_{\lceil j/\tau \rceil}$. Delete the LS_j values computed for $j \in \mathcal{I}_2$. Repeat the process with $i = 4, 5, \dots, \tau$. Suppose r is the last position such that LS_r is defined. We will ignore $\mathsf{LS}_{r+1}, \mathsf{LS}_{r+2}, \dots, \mathsf{LS}_n$ while creating the binary strings. Now, we create a binary string $B = B_1 B'_1 B_2 B'_2 \dots B'_{|\mathcal{I}_1|-1} B_{|\mathcal{I}_1|}$, where B'_p, $p \in [1, |\mathcal{I}_1| - 1]$, is the string containing $(\mathsf{LS}_{p\tau+1} + 1 - \mathsf{LS}_{p\tau})$ many 1s followed by a 0. Delete the binary strings B_1 through $B_{|\mathcal{I}_1|}$. If $r > n - \tau + 1$, compute LS_j for $j \in [n-\tau+1, r]$ in $O(n\tau^2)$ time using a brute-force approach. For each $j \in [n-\tau+2, r]$, append $(|\mathsf{LS}_j| + 1 - |\mathsf{LS}_{j-1}|)$ many 1s followed by a 0 to the binary string B. Finally, construct the rank-select structure of Fact 10 over B.

Since we will make τ calls to Lemma 6, the time required is $\mathcal{O}(n\tau^2 \log \frac{n}{\tau})$. Note that $r + |\mathsf{LS}_r| \le n + 1$. By Observation 8, for any $r' < r$, we have $r' + |\mathsf{LS}_{r'}| \le r' + 1 + |\mathsf{LS}_{r'+1}|$. By Observation 5, $\mathsf{LS}_{r''}$ is not defined for any $r'' > r$. It immediately follows that $\sum_{p=1}^{r-1}(|\mathsf{LS}_{p+1}| + 1 - |\mathsf{LS}_p|) \le n$. Observe that B is a binary string which is a concatenation of $(|\mathsf{LS}_{p+1}| + 1 - |\mathsf{LS}_p|)$ many 1s followed by 0 for all values of p from 1 to $(r-1)$. Therefore, the total length of B is at most $2n$. Then, $|\mathsf{LS}_k| = |\mathsf{LS}_1| + \mathsf{rank}(\mathsf{select}(k-1,0),1) - k + 1$, where $k \in [1,r]$. By storing $|\mathsf{LS}_1|$ and the position r explicitly in $\lceil 2\log n \rceil = o(n)$ bits, LS_k can be retrieved in $\mathcal{O}(1)$ time. Since at any point we are storing LS_j values for at most $3\lceil n/\tau \rceil$ choices of j, the working space needed is $\mathcal{O}(n/\tau)$ words and $2n + o(n)$ bits.

Now, we build an RMQ data structure over a conceptual array A. The length of A is the number of zeroes in B i.e., $|A| = r \le n$, and $A[p] = |\mathsf{LS}_p| = |\mathsf{LS}_1| + \mathsf{rank}(\mathsf{select}(p-1,0),1) - p + 1$. Using Fact 11, we can construct this data structure using $2n + o(n)$ bits of additional space.

Summarizing, the working space needed at any point is $\mathcal{O}(n/\tau)$ words and $4n + o(n)$ bits.

We return to the task of computing the shortest $\mathsf{LS}_{k'}$, $k' \le k$, that covers k. First locate the smallest position $k'' \le k$, such that $\mathsf{LS}_{k''}$ covers k. This is achieved in $\mathcal{O}(\log n)$ time via a binary search using LS_1, B and its associated rank-select structure. If k'' does not exist, then we are done. Otherwise, $k' = \mathsf{rmq}_A(k'', k)$. The total time needed for all such computations is $\mathcal{O}(n \log n)$, and the claimed space and time bounds are met. ◀

▶ **Corollary 12.** *Suppose, we can compute $\mathsf{LS}_1, \mathsf{LS}_2, \dots, \mathsf{LS}_n$ in that order. We can store $|\mathsf{LS}_k|$, $k \in [1,n]$, in total $2n + o(n)$ bits, such that a particular $|\mathsf{LS}_k|$ value can be accessed in $\mathcal{O}(1)$ time.* (This is the same result as obtained by Belazzougui and Cunial [2].) *Also, by maintaining an additional $2n + o(n)$-bit structure, for any k, in $\mathcal{O}(\log n)$ time, we can compute the shortest $\mathsf{LS}_{k'}$, $k' \le k$, that covers k, or verify that no such k' exists. The total time (in addition to that for computing every LS_k value) to construct this $4n + o(n)$ data* *structure is $\mathcal{O}(n)$. The working space required is $4n + o(n)$ bits. Using this, we can compute S_k for every $k \in [1, n]$ in an additional $\mathcal{O}(n \log n)$ time and $\mathcal{O}(1)$ space.*

4 Randomized Algorithm

We prove the following theorem in this section.

▶ **Theorem 13.** *For a string X of length n, any given $k \in [1, n]$, and any arbitrary constant $c \geq 0$, we can find S_k in $\mathcal{O}(n\tau \log^{c+1} n)$ time using X and additional $n/\log^c n + \mathcal{O}(n/\tau)$-words of working space. By using additional $4n + o(n)$ bits, we can compute S_k for all values of k in $\mathcal{O}(n\tau \log^{c+1} n)$ time. Each S_k computed is correct with probability at least $1 - n^{-\mathcal{O}(1)}$.*

4.1 Proof of Theorem 13

The key idea to reduce the time from $\mathcal{O}(n\tau^2 \log \frac{n}{\tau})$ is to modify Lemma 6 so that we can carry out the same task in time $\mathcal{O}(n \log^{c+1} n)$ time, with $n/\log^c n$ words of additional space. In this context, we present the following lemma.

▶ **Lemma 14.** *Let $\mathcal{I}_i = \{i, i+\tau, i+2\tau, \dots\}$, $i \in [1, \tau]$, be a set of at most $\lceil n/\tau \rceil$ suffixes. For each $i' \in \mathcal{I}_i$, we can compute $\mathsf{LS}_{i'}$ correctly with high probability in $\mathcal{O}(n \log^{c+1} n)$ time using X and additional $n/\log^c n + \mathcal{O}(n/\tau)$-words of working space.*

Here and henceforth, by high probability, we mean that each computed $\mathsf{LS}_{i'}$ is unique, but with probability at most $n^{-\mathcal{O}(1)}$, may not be the shortest. Likewise, each computed S_k is unique and covers k, but with probability at most $n^{-\mathcal{O}(1)}$, may not be the shortest.

We observe that in Lemma 6 the $n\tau$-factor in the time complexity is due to matching X in the sampled suffix tree ST_i by passing the string τ times, each time with a different choice of $j \in [1, \tau]$, $j \neq i$. Each such pass costs us $\mathcal{O}(n \log \frac{n}{\tau})$ time. The idea is to reduce this by speeding up (i) the time to find the correct outgoing edge of a node, and (ii) the time to update the Δ value of a node in a navigation trie. We will show that (i) can be achieved in $\mathcal{O}(\log^c n)$ time, with a slight probability of a false positive using Rabin-Karp Fingerprint [15] and perfect hashing [10]. For achieving (ii), the rough idea is to use randomization to binary search on the navigation trie, along the path containing the longest repeating prefix. This will cost us $\mathcal{O}(\log^{c+1} n)$ time. We begin by revisiting a couple of important results.

▶ **Fact 15** (Rabin-Karp Fingerprint [15])**.** *Let S be a string, and $p > |S|$ be a prime number. Choose $q \in \mathbb{F}_p$ uniformly at random. The fingerprint of S is*

$$\Phi(S) = \sum_{k=0}^{|S|-1} S[k]q^k \mod p$$

The following are a few well-known properties of fingerprints [5]. The probability of $\Phi(S) = \Phi(S')$ for two distinct strings S and S' is at most $m^{-\lambda+1}$, where $m = |S| = |S'|$, $p \in \Theta(m^\lambda)$, and $\lambda \geq 4$ is a constant. The factor λ may be amplified by a constant number of computations. For two strings S and S', where $m = |S|$, we have (i) $\Phi(SS') = \Phi(S) + \Phi(S')q^m \mod p$, (ii) $\Phi(S) = \Phi(SS') - \Phi(S')q^m \mod p$, and (iii) $\Phi(S') = \big(\Phi(SS') - \Phi(S)\big)q^{-m} \mod p$. Therefore, for these three equations, given the value of $q^m \mod p$ and the FP values on right, we compute the FP value on the left in $\mathcal{O}(1)$ time.

▶ **Fact 16** (Probabilistic z-fast Trie, Theorem 4.1, Belazzougui et al. [1])**.** *Consider the compacted trie $\mathcal{T}$ of a set of t strings. Each string has length at most m. Given any string S,* *by using a probabilistic data structure, we can find the deepest node u (called the **exit node**) in $\mathcal{T}$ such that* $\mathsf{path}(u)$ *is a prefix of S. The chances of an error is at most $m^{-\lambda}$, where $\lambda > 0$ is an arbitrary constant. The space occupied by the probabilistic data structure is $\mathcal{O}(t)$ words, and the time required is $\mathcal{O}(\log(m+t))$.*

The main technique behind Fact 16 is to associate each node u in the trie with a signature function. Specifically, the signature function is based on $\mathsf{path}(u)$. If two strings are distinct, then their signatures match with very low probability. Now, given the signature of each prefix of S, the overall idea is to carry out a binary search on the signature of each node to locate the desired node. Furthermore, given the compacted trie, and the signature of every prefix of each of the t strings, the data structure of Fact 16 can be constructed in $\mathcal{O}(t)$ time.

▶ **Lemma 17.** *Consider the compacted trie $\mathcal{T}$ of a set of t suffixes of a string Y having length m. Given a string S and fingerprint of every $\log^c m$ prefix, by maintaining an $m/\log^c m + \mathcal{O}(t)$ word data structure, we can find the deepest point (possibly, on an edge) such that the string formed by concatenating edge labels from root to this point is a prefix of S. The time required is $\mathcal{O}(\log^{c+1} m)$, and the probability of an error is at most $m^{-\mathcal{O}(1)}$. The data structure can be constructed in $\mathcal{O}(m + t(\log t + \log^c m))$ time using $m/\log^c m + \mathcal{O}(t)$ words of space.*

Proof. We will use a different signature function as that of Belazzougui et al. [1]. (See [5] for a similar usage.) Specifically, each node w is labeled with the fingerprint (FP) of $\mathsf{path}(w)$. Each edge in $\mathcal{T}$ is labeled by a substring of Y. We maintain two pointers *sp* and *ep* to the start and end position in Y, and store the value of $q^{sp} \mod p$. Here, p and q are defined as in Fact 15. Also, at each node w, we store the value of $q^{|\mathsf{path}(w)|} \mod p$. To compute this simply sort the edges based on *sp* and the nodes w based on $\mathsf{path}(w)$ in $O(t \log t)$ time. The time needed is $\mathcal{O}(m + t \log t)$, and the space occupied at any point is $\mathcal{O}(t)$ words.

Now, compute the FP of each of the prefixes of Y ending at the positions $1, 1+\log^c m, 1+2\log^c m, \ldots$ in $\mathcal{O}(m)$ time using Fact 15. The space needed to compute and store this information is at most $(1 + m/\log^c m)$ words. Using these, we can compute the FP of a prefix of an edge label in $\mathcal{O}(\log^c m)$ time by simply finding the nearest prefix of Y whose FP has been stored and then walking at most $\log^c m$ characters in Y. Also, by pre-processing the trie with $\mathsf{levelAncestor}$ queries [4], we can find the FP of a prefix of any of the t suffixes in $\mathcal{O}(\log t + \log^c m)$ time as follows. Binary search using $\mathsf{levelAncestor}$ queries and $|\mathsf{path}(u)|$ stored at each node u on the path from root to the leaf corresponding to the suffix. This binary search enables us to find the edge position corresponding to the prefix whose FP value we want to find. Then, the desired FP value is obtained using the edge pointers. Therefore, we can construct the z-fast trie of Fact 16 in $\mathcal{O}(t(\log t + \log^c m))$ time given $\mathcal{T}$ and the FP of each prefix of Y. The space at any point is bounded by $m/\log^c m + \mathcal{O}(t)$ words.

We return to our original task. Use the z-fast trie and the FP of each prefix of S to find the exit node u. Then, use the character $S[1 + |\mathsf{path}(u)|]$ to select an outgoing edge (u, v) of u. If no such edge exists, then the desired location is given by the node u. Otherwise, the desired location lies on the edge (u, v). Using the FP of a prefix of $S[|\mathsf{path}(u)| + 1, \mathsf{path}(v)]$, we binary search on the edge (u, v) to find the desired location. Each prefix computation needs $\mathcal{O}(\log^c m)$ time. Thus, the total time required is $\mathcal{O}(\log^{c+1} m)$. Since the number of FP comparisons is $\mathcal{O}(\log m)$, the probability of a false positive is $\mathcal{O}(\frac{\log m}{m^\lambda}) = m^{-\mathcal{O}(1)}$. ◀

Proof of Lemma 14. Construct the suffix tree ST_i for each suffix starting at a location lying in the set $\mathcal{I}_i = \{i, i+\tau, i+2\tau, \ldots\}$. Now, create the navigation trie $\mathsf{ST}_i(u)$ of every node u. The total time needed is $\mathcal{O}(n + \frac{n}{\tau}\log\frac{n}{\tau})$. Each edge in ST_i or a navigation trie has pointers *sp* and *ep* to X. Use Lemma 17 to compute and store (i) $q^{|\mathsf{path}(w)|} \mod p$ for each node w in

ST_i, and (ii) $q^{sp} \mod p$ for each edge. Likewise, we compute and store the values for each node and edge in every navigation trie. We maintain an array A_j which stores the values $q^j \mod p$, $q^{j+\tau} \mod p, q^{j+2\tau} \mod p, \ldots$. Initially, the array is maintained for $j = 2$ if $i = 1$, and for $j = 1$, otherwise. As described in Lemma 17, the total time needed is $\mathcal{O}(n)$. The construction space and that needed for storage are both bounded by $\mathcal{O}(n/\tau)$ words.

Using Fact 15, we compute and store $\Phi(\mathsf{X}[1, i])$ for every $i \in \{1, 1+\log^c n, 1+2\log^c n, \ldots\}$ in $\mathcal{O}(n)$ time. The space needed is $1 + n/\log^c n$ words. Compute the FP of the first τ-characters of the edge label in ST_i; this is achieved in $\mathcal{O}(\log^c n)$ time. Use a perfect hash function [10] at each node for selecting the correct outgoing edge based on the computed FP. The total time and space needed to incorporate this information is $\mathcal{O}((n/\tau)\log^c n)$. Finally, we maintain the probabilistic z-fast trie of Fact 16 for each navigation trie. This can be created as described in Lemma 17. Incorporating the probabilistic data structure for all navigation tries requires $\mathcal{O}(n + \frac{n}{\tau}(\log \frac{n}{\tau} + \log^c n))$ time.

Summarizing, the space needed to maintain the data structure comprising of ST_i, the navigation tries and their adjoining z-fast tries is $n/\log^c n + \mathcal{O}(n/\tau)$ words. Moreover, the data structure is constructed in $\mathcal{O}(n + \frac{n}{\tau}(\log \frac{n}{\tau} + \log^c n))$ time using $n/\log^c n + \mathcal{O}(n/\tau)$ words.

Now, we start matching X in ST_i starting with $j = 1$ if $i > 1$, and with $j = 2$, otherwise. To traverse ST_i, suppose we are at a node u, and have read up to position j' in X. Use $\Phi(\mathsf{X}[j' + 1, j' + \tau])$ to select a correct outgoing edge (if any). This is achieved in $\mathcal{O}(\log^c n)$ time first by computing the FP using the array A_j, and then using perfect hashing. Similarly, we can traverse every τ characters on an edge in ST_i in $\mathcal{O}(\log^c n)$ time. We do this until we find a failure, or reach a child v of u. In the latter case, we mark v's corresponding leaf ℓ^* in $\mathsf{ST}_i(u)$, update $\Delta(\ell^*)$, and continue matching from v. In the former case, use Lemma 17 to mark the correct node w^* in $\mathsf{ST}_i(u)$ and update $\Delta(w^*)$ in $\mathcal{O}(\log^{c+1} n)$ time. Now, follow the suffix link of u. The correct position to start matching in an outgoing edge of $\Psi(u)$ in ST_i can be found in $\mathcal{O}(\log^c n)$ time. Continue, until the entire string X has been processed. The total time needed is $\mathcal{O}(\frac{n}{\tau}\log^{c+1} n)$. Now, we repeat the process with $(j + 1)$ if $i \neq (j + 1)$, and with $(j + 2)$, otherwise. The array A_j can be updated in $\mathcal{O}(n/\tau)$ time to A_{j+1} or A_{j+2}, as the case is. The number of times this process is repeated is $(\tau - 1)$. Finally, for each $i' \in \mathcal{I}_i$, we can compute $\mathsf{LS}_{i'}$ as described in Section 2.1 in $\mathcal{O}(n/\tau)$ time. Hence, the total time needed is $\mathcal{O}(n\log^{c+1} n + \frac{n}{\tau}(\log \frac{n}{\tau} + \log^c n)) = \mathcal{O}(n\log^{c+1} n)$.

Note that if two strings are identical, then their FP values are necessarily the same. Hence, each $\mathsf{LS}_{i'}$ is definitely unique, but may not be the shortest. The number of queries to a z-fast trie, or a FP comparison are both bounded by $\mathcal{O}(n\log n)$. Therefore, the probability of an error is $n^{-\mathcal{O}(1)}$ (achieved by appropriately choosing λ in Facts 15 and 16). ◀

Wrapping Up. As in Theorem 7, we will invoke Lemma 14 by rotating the choices of $i \in [1, \tau]$. Finally, we compute the LS_j values for $j \in [n - \tau + 2, n]$ as follows. Maintain the FP of every $\log^c n$ prefix of $\mathsf{X}[n - \tau + 2, n]$. Now to find LS_j, binary search at each position (other than j) of X with the suffix starting at j to find the longest repeating prefix. The FP of a prefix of any of these suffixes (resp. of a suffix in X) is obtained in $\mathcal{O}(\log^c \tau)$ time (resp. $\mathcal{O}(\log^c n)$ time). The number of binary search operations is $\mathcal{O}(\log \tau)$. Thus, the overall time is bounded by $\mathcal{O}(n\tau(\log^c n)\log \tau) = \mathcal{O}(n\tau\log^{c+1} n)$.

The discussion in this section and the techniques used in proving Theorems 7 prove the first part of Theorem 13 for computing S_k for a single k. The latter part of the theorem is a consequence of Corollary 12, which follows from the proof of Theorem 9. However, one needs to be a little more careful while carrying out the steps in Theorem 9 because the relation $|\mathsf{LS}_i| \leq |\mathsf{LS}_{i+1}| + 1$ in Observation 8 maybe violated due to false positives in FP matches.

Since no false negatives occur in FP matches, each computed $\mathsf{LS}_{i'}$ for any i' is definitely unique. Therefore, we can simply start from the rightmost i' where $|\mathsf{LS}_{i'}| \le |\mathsf{LS}_{i'+1}| + 1$ is violated and set $|\mathsf{LS}_{i'}| = |\mathsf{LS}_{i'+1}| + 1$ for each successive i' from right to left. Observe that the total number of changes to the binary strings B_1 through $B_{|\mathcal{I}_1|}$ (see the proof of Theorem 9) is at most $2n$ for each invoking of Lemma 14. Therefore, the total time needed to affect these changes is $\mathcal{O}(n\tau)$. Finally, the binary string B is again computed in $\mathcal{O}(n)$ time. The rest of the steps remain the same as in the proof of Theorem 9. This completes the proof of Theorem 13.

5 Maximal Unique Matches Problem

Let X_1 and X_2 be two strings of length n_1 and n_2 respectively, where $n = n_1 + n_2$. Each character is drawn from a totally ordered alphabet Σ. We assume that X_1 and X_2 terminate in two special characters $\$_1$ and $\$_2$ that does not appear anywhere else.

▶ **Definition 18** (Maximal Unique Match)**.** A *Maximal Unique Match* (MUM) of two strings X_1 and X_2 is a string S that satisfies the following two properties: (i) S appears uniquely in each string X_1 and X_2, and (ii) a left or right extension of S in X_1 does not appear in X_2.

▶ **Problem 19.** *Given two strings X_1 and X_2, the task is to find the set $\mathcal{S}$ of all their maximal unique matches. Each match is represented by its starting position in X_1 and its length.*

To the best of our knowledge, Problem 19 was formulated by Delcher et al. [6]. The main motivation was its importance in aligning whole genome sequences consisting of millions of nucleotides. They presented a software known as MUMmer 1.0. Further improvements by Delcher et al. [7] and then by Kurtz et al. [17] lead to MUMmer 2.0 and MUMmer 3.0 respectively. The chief component of all these softwares (and underlying algorithm) is the (generalized) suffix trees (GST) – a compacted trie storing all the suffixes of X_1 and X_2, and occupying $\Theta(n)$ words. The following is the key observation.

▶ **Observation 20.** *Given two strings X_1 and X_2 and their GST, a string S is an MUM iff*

(a) *There exists a node v in the GST such that $S = \mathsf{path}(v)$. Moreover, v has exactly two children (leaves), each labeled by a suffix from X_1 and X_2.*

(b) *There does not exist a node u which simultaneously satisfies:* (i) *u has a suffix link to v, and* (ii) *u has exactly two children (leaves) that are labeled by suffixes from X_1 and X_2.*

The GST of X_1 and X_2 can be built in $\mathcal{O}(n)$ time using the algorithm of Farach-Colton [8], and leads to a simple $\mathcal{O}(n)$-space and $\mathcal{O}(n)$-time algorithm for Problem 19. The basic idea to reduce the space is to build a GST only on n_1/τ suffixes of X_1 and n_2/τ suffixes of X_2 at a time. This reduces the space to $\mathcal{O}(n/\tau)$ words. By rotating the choice of n_2/τ suffixes in X_2 roughly τ times, we will be able to determine the candidate set (i.e., a set containing the MUMs) among the n_1/τ suffixes of X_1. Using the next set of n_1/τ suffixes of X_1, we will be able to remove the incorrect choices from the candidate set. This idea, coupled with the techniques for the SUS problem, leads to the following theorem.

▶ **Theorem 21.** *Given X_1 and X_2, we can compute the set $\mathcal{S}$* (i) *in $\mathcal{O}(n\tau^2 \log \frac{n}{\tau})$ time using additional $\mathcal{O}(n/\tau)$ words of working space, and* (ii) *correctly with high probability in $\mathcal{O}(n\tau \log^{c+1} n)$ time using additional $n/\log^c n + \mathcal{O}(n/\tau)$ words of working space.*

References

1 Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: searching a sorted table with $O(1)$ accesses. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009*, pages 785–794, 2009.

2 Djamal Belazzougui and Fabio Cunial. Indexed matching statistics and shortest unique substrings. In *String Processing and Information Retrieval – 21st International Symposium, SPIRE 2014, Ouro Preto, Brazil, October 20-22, 2014. Proceedings*, pages 179–190, 2014. doi:10.1007/978-3-319-11918-2_18.

3 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, pages 88–94, 2000. doi:10.1007/10719839_9.

4 Michael A. Bender and Martin Farach-Colton. The level ancestor problem simplified. In *LATIN 2002: Theoretical Informatics, 5th Latin American Symposium, Cancun, Mexico, April 3-6, 2002, Proceedings*, pages 508–515, 2002. doi:10.1007/3-540-45995-2_44.

5 Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana A. Starikovskaya. Dictionary matching in a stream. In *Algorithms – ESA 2015 – 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, pages 361–372, 2015. doi:10.1007/978-3-662-48350-3_31.

6 Arthur L. Delcher, Simon Kasif, Robert D Fleischmann, Jeremy Peterson, Owen White, and Steven L. Salzberg. Alignment of whole genomes. *Nucleic acids research*, 27(11):2369–2376, 1999.

7 Arthur L. Delcher, Adam Phillippy, Jane Carlton, and Steven L. Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic acids research*, 30(11):2478–2483, 2002.

8 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, 1997. doi:10.1109/SFCS.1997.646102.

9 Johannes Fischer and Volker Heun. A new succinct representation of rmq-information and improvements in the enhanced suffix array. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, First International Symposium, ESCAPE 2007, Hangzhou, China, April 7-9, 2007, Revised Selected Papers*, pages 459–470, 2007. doi:10.1007/978-3-540-74450-4_41.

10 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984. doi:10.1145/828.1884.

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

12 Wing-Kai Hon, Tak Wah Lam, Rahul Shah, Siu-Lung Tam, and Jeffrey Scott Vitter. Compressed index for dictionary matching. In *2008 Data Compression Conference (DCC 2008), 25-27 March 2008, Snowbird, UT, USA*, pages 23–32, 2008. doi:10.1109/DCC.2008.62.

13 Wing-Kai Hon, Sharma V. Thankachan, and Bojian Xu. An in-place framework for exact and approximate shortest unique substring queries. In *Algorithms and Computation – 26th International Symposium, ISAAC 2015, Nagoya, Japan, December 9-11, 2015, Proceedings*, pages 755–767, 2015. doi:10.1007/978-3-662-48971-0_63.

14 Atalay Mert Ileri, M. Oguzhan Külekci, and Bojian Xu. Shortest unique substring query revisited. In *Combinatorial Pattern Matching – 25th Annual Symposium, CPM 2014, Moscow, Russia, June 16-18, 2014. Proceedings*, pages 172–181, 2014. doi:10.1007/978-3-319-07566-2_18.

15 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987. doi:10.1147/rd.312.0249.

16 Stefan Kurtz. Reducing the space requirement of suffix trees. *Softw., Pract. Exper.*, 29(13):1149–1171, 1999. doi:10.1002/(SICI)1097-024X(199911)29:13<1149::AID-SPE274>3.0.CO;2-O.

17 Stefan Kurtz, Adam Phillippy, Arthur L. Delcher, Michael Smoot, Martin Shumway, Corina Antonescu, and Steven L. Salzberg. Versatile and open software for comparing large genomes. *Genome biology*, 5(2):R12, 2004.

18 J. Ian Munro. Tables. In *Foundations of Software Technology and Theoretical Computer Science, 16th Conference, Hyderabad, India, December 18-20, 1996, Proceedings*, pages 37–42, 1996. doi:10.1007/3-540-62034-6_35.

19 Jian Pei, Wush Chi-Hsuan Wu, and Mi-Yen Yeh. On shortest unique substring queries. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 937–948, 2013. doi:10.1109/ICDE.2013.6544887.