Locally Consistent Parsing for Text Indexing in Small Space

Or Birenzwige* Shay Golan* Ely Porat*

Abstract

We consider two closely related problems of text indexing in a sub-linear working space. The first problem is the Sparse Suffix Tree (SST) construction, where a text S is given in read-only memory, along with a set of suffixes B, and the goal is to construct the compressed trie of all these suffixes ordered lexicographically, using only $\mathcal{O}(|B|)$ words of space. The second problem is the Longest Common Extension (LCE) problem, where again a text S of length n is given in read-only memory with some parameter $1 \leq \tau \leq n$, and the goal is to construct a data structure that uses $\mathcal{O}(\frac{n}{\tau})$ words of space and can compute for any pair of suffixes their longest common prefix length. We show how to use ideas based on the Locally Consistent Parsing technique, that were introduced by Sahinalp and Vishkin [44], in some non-trivial ways in order to improve the known results for the above problems. We introduce new Las-Vegas and deterministic algorithms for both problems.

For the randomized algorithms, we introduce the first Las-Vegas SST construction algorithm that takes $\mathcal{O}(n)$ time. This is an improvement over the last result of Gawrychowski and Kociumaka [22] who obtained $\mathcal{O}(n)$ time for Monte Carlo algorithm, and $\mathcal{O}(n\sqrt{\log |B|})$ time with hight probability for Las-Vegas algorithm. In addition, we introduce a randomized Las-Vegas construction for a data structure that uses $\mathcal{O}(\frac{n}{\tau})$ words of space, can be constructed in linear time with high probability and answers LCE queries in $\mathcal{O}(\tau)$ time.

For the deterministic algorithms, we introduce an SST construction algorithm that takes $\mathcal{O}(n \log \frac{n}{|B|})$ time (for $|B| = \Omega(\log n)$). This is the first almost linear time, $\mathcal{O}(n \cdot \operatorname{polylog} n)$, deterministic SST construction algorithm, where all previous algorithms take at least $\Omega\left(\min\{n|B|, \frac{n^2}{|B|}\}\right)$ time. For the LCE problem, we introduce a data structure that uses $\mathcal{O}(\frac{n}{\tau})$ words of space and answers LCE queries in $\mathcal{O}(\tau\sqrt{\log^* n})$ time, with $\mathcal{O}(n \log \tau)$ construction time (for $\tau = \mathcal{O}(\frac{n}{\log n})$). This data structure improves both query time and construction time upon the results of Tanimura et al. [47].

1 Introduction

Text indexing is one of the most fundamental problems in the area of string algorithms and information retrieval (e.g., see [50, 29, 49, 38, 16, 4, 17, 18, 7, 19, 2, 31, 28, 10, 42]).

In this paper we consider two closely related problems of text indexing in sub-linear working space. In both problems, we are given a string S of length n to process, in read-only memory.

The first problem is the Sparse Suffix Tree (SST) construction which is a generalization of the classical Suffix Tree data structure. The Suffix Tree index of S is a compact trie of all S's suffixes. Since the '70s, several algorithms were introduced for the suffix tree construction which take $\mathcal{O}(n)$ time [50, 38, 49]. In the Sparse Suffix Tree construction problem only a subset of the suffixes, $B \subseteq \{1, 2, \ldots, n\}$, need to be indexed. This generalization was introduced in 1996 by Kärkkäinen and Ukkonen [29] who introduced a construction algorithm for the special case where all the suffixes are in the positions which are integer multiples of some $k \in \mathbb{N}$. The algorithm of [29] takes $\mathcal{O}(n)$ time and uses linear working space in the number of suffixes. For the general case of an arbitrary set of suffixes, one can construct an SST by pruning of the full Suffix Tree, which requires $\Theta(n)$ words of working space. However, when the space usage is limited to be linear in $|B|$, another approach is required. Recently, Gawrychowski and Kociumaka [22] introduced an optimal randomized Monte Carlo algorithm which builds an SST in linear time, using only $\mathcal{O}(|B|)$ words of space.

The second problem we consider in this paper is the Longest Common Extension (LCE) problem. In this problem we are given a text S of length n in read-only memory, and the goal is to construct a data structure that can compute for any pair of suffixes their longest common prefix length, or formally $\mathsf{LCE}(i, j) = \min\{k \geq 0 \mid S[i + k] \neq S[j + k]\}$. It is well known that one can solve this problem using $\mathcal{O}(n)$ words of space with $\mathcal{O}(n)$ construction time and $\mathcal{O}(1)$

*Bar Ilan University.

This research is supported by ISF grant no. 1278/16, by a grant from the United States - Israel Binational Science Foundation (BSF) and by an ERC grant MPM under the European Union's Horizon 2020 research and innovation programme (grant no. 683064).

query time [21, 36]. We consider the trade-off version, where there is a parameter $1 \leq \tau \leq n$ and the goal is to construct a data structure using only $\mathcal{O}(\frac{n}{\tau})$ words of working space while achieving fast construction and query time. The trade-off LCE problem is closely tied to the SST problem, and in several papers the LCE was used as the main tool for constructing the SST efficiently [26, 4, 18, 22].

For the LCE problem, Kosolobov [35] proved a lower bound in the cell-probe model, that any data structure for the LCE problem that uses $S(n) = \Omega(n)$ *bits* of working space and $T(n)$ query time, must have $S(n) \cdot T(n) = \Omega(n \log n)$ for large enough alphabet. Kosolobov also conjectured that for data structure which used $S(n) < o(n)$ bits of space a similar lower bound of $T(n) \cdot S(n) = \Omega(n \log S(n))$ holds. A trade-off of $\mathcal{O}(\tau)$ query time and $\mathcal{O}(\frac{n}{\tau})$ words of space is already known, both in randomized and deterministic construction algorithms [5], and therefore the research efforts are invested in reducing the construction time. Tanimura et al. [47] introduced a deterministic data structure with $\mathcal{O}(n\tau)$ construction time algorithm, and with a slightly increasing of the query time to $\mathcal{O}(\tau \min\{\log \tau, \log \frac{n}{\tau}\})$. Gawrychowski and Kociumaka [22] have shown a linear time Monte Carlo construction of an LCE data structure with $\mathcal{O}(\tau)$ query time. In a close model of *encoding* data structures, Tanimura et al. [48] introduce a data structure that encodes the string, and then can delete the original string. The complexity of the data structure is dependent on the Lempel–Ziv 77 factorization of the string, and in some cases it can beat the lower bound of Kosolobov [35].

1.1 Our Contribution

LCE data structures. We provide new LCE data structures that answer queries deterministically and use only $\mathcal{O}(\frac{n}{\tau})$ words of space. The construction algorithms are either deterministic or randomized Las-Vegas algorithms. Our results are stated in the following theorems, see a full comparison in Table 1.

THEOREM 1.1. (LAS-VEGAS ALGORITHM) *For any $1 \leq \tau \leq \mathcal{O}(\frac{n}{\log^2 n})$ there exists a data structure of size $\mathcal{O}(\frac{n}{\tau})$ words that can be constructed in $\mathcal{O}(n)$ time with high probability*[1] *and answers LCE queries in $\mathcal{O}(\tau)$ time. For $\tau = \omega(\frac{n}{\log^2 n})$ the same data structure can be constructed in* expected *$\mathcal{O}(n)$ time.*

For the deterministic trade-off, we begin with a data structure with the following complexities.

[1]With probability $1 - n^{-\Theta(1)}$ at least.

LEMMA 1.1. (DETERMINISTIC ALGORITHM) *For any $1 \leq \tau \leq \mathcal{O}(\frac{n}{\log n})$ there exists a data structure of size $\mathcal{O}(\frac{n}{\tau})$ words which can be constructed deterministically in $\mathcal{O}(n \log \tau)$ time and answers LCE queries in $\mathcal{O}(\tau \log^* n)$ time.*

Then, we improve the results as stated in the following theorem.

THEOREM 1.2. (DETERMINISTIC ALGORITHM) *For any $1 \leq \tau \leq \mathcal{O}(\frac{n}{\log n})$ there exists a data structure of size $\mathcal{O}(\frac{n}{\tau})$ words which can be constructed deterministically in $\mathcal{O}(n \log \tau)$ time and answers LCE queries in $\mathcal{O}(\tau \sqrt{\log^* n})$ time.*

Sparse Suffix Tree construction. For the SST construction problem we introduce an optimal randomized Las-Vegas algorithm, which improves upon the previous algorithm of Gawrychowski and Kociumaka [22] who developed an optimal Monte Carlo algorithm and a Las-Vegas algorithm with additional $\sqrt{\log |B|}$ factor. In addition, we introduce the first almost-linear deterministic SST construction algorithm. Our results are stated in the following theorems.

THEOREM 1.3. *Given a set $B \subseteq [n]$ of size $\Omega(\log^2 n) \leq |B| \leq n$, there exists a randomized Las-Vegas algorithm which constructs the Sparse Suffix Tree of B in $\mathcal{O}(n)$ time with high probability using only $\mathcal{O}(|B|)$ words of space. For $|B| < o(\log^2 n)$ there exists a randomized Las-Vegas algorithm with the same complexities except that the running time is just* expected *to be $\mathcal{O}(n)$.*

THEOREM 1.4. *Given a set $B \subseteq [n]$ of size $\Omega(\log n) \leq |B| \leq n$, there exists a deterministic algorithm which constructs the Sparse Suffix Tree of B in $\mathcal{O}(n \log \frac{n}{|B|})$ time using only $\mathcal{O}(|B|)$ words of space.*

1.2 Algorithmic Overview We first give an overview of the LCE data structures and then describe how to use them (with additional techniques) to build an SST using only a small amount of space. All the techniques mentioned in the overview are described later in detail. For both problems, we use a parameter $1 \leq \tau \leq n$, such that the working space of the data structures and the construction algorithms are limited to $\mathcal{O}(\frac{n}{\tau})$ words of space.

LCE data structure. The naïve method to compute $\mathsf{LCE}(i, j)$ is to compare pairs of characters until a mismatch is found. Since this naïve methos takes $\mathcal{O}(1+\mathsf{LCE}(i, j))$ time, the case where $\mathsf{LCE}(i, j) = \mathcal{O}(\tau)$ is solvable in $\mathcal{O}(\tau)$ time without any preprocessing. Therefore, we invest our efforts on speeding up the computation of queries where $\mathsf{LCE}(i, j) = \ell > \omega(\tau)$. In this

	Data Structure				Trade-off range	Reference
	Space	Query time	Preprocessing time	Correct		
Monte Carlo	$\frac{n}{\tau}$	τ	n	*w.h.p.*	$1 \le \tau \le n$	[22, Theorem 3.3]
Las Vegas	$\frac{n}{\tau}$	$\tau \log \frac{\ell}{\tau}$	$n(\tau + \log n)$ *w.h.p.*	always	$1 \le \tau \le n$	[6]
	$\frac{n}{\tau}$	τ	$n^{3/2}$ *w.h.p.*	always	$1 \le \tau \le n$	[5, Section 3.5]
	$\frac{n}{\tau}$	τ	$n\sqrt{\log \frac{n}{\tau}}$ *w.h.p.*	always	$1 \le \tau \le n$	[22, 5][a]
	$\frac{n}{\tau}$	τ	n *expected*	always	$1 \le \tau \le n$	**new**
	$\frac{n}{\tau}$	τ	n *w.h.p.*	always	$1 \le \tau \le \frac{n}{\log^2 n}$	**new**
Deterministic	1	n	1	always	-	naïve algorithm
	n	1	ℓ	always	-	Suffix tree+LCA
	$\frac{n}{\tau}$	τ	$n^{2+\varepsilon}$	always	$1 \le \tau \le n$	[5, Section 4]
	$\frac{n}{\tau}$	$\tau \min\{\log \tau, \log \frac{n}{\tau}\}$	$n\tau$	always	$1 \le \tau \le n$	[47, Corollary 12]
	$\frac{n}{\tau}$	$\tau\sqrt{\log^* n}$	$n \log \tau$	always	$1 \le \tau \le \frac{n}{\log n}$	**new**

[a] The LCE data structure is not introduced explicitly, but it can be constructed with the techniques of [5] and [22].

Table 1: LCE data structures — The table introduces the previous most efficient data structures for the LCE problem together with our new results. The integer ℓ denoted the LCE query result.

case, we use a special partitioning of S into blocks. The partitioning has the property that if a string appears as a substring of S multiple times, then all these substrings are composed of exactly the same blocks, except for possibly small margins of the substrings. More detailed, if $S[i..i+\ell-1] = S[j..j+\ell-1]$ are long enough substrings, then both $S[i..i+\ell-1]$ and $S[j..j+\ell-1]$ can be considered as composed of three parts, a prefix, a suffix and the middle part between them with the following properties. Both the prefix and the suffix are short strings (e.g., of length $\mathcal{O}(\tau)$ for the randomized algorithm), or they have some other special properties, and the middle part is composed of full blocks, which are exactly the same in both substrings, regardless the length of this part. Thus, using this partitioning, the computation of $\mathsf{LCE}(i,j)$ is done in three phases: first, the algorithm verifies that $\mathsf{LCE}(i,j)$ is long enough and if so — the algorithm finds the beginning of the common middle part. Then, the algorithm computes the length of the middle part (which might be very long) by a designated data structure that performs this computation in constant time. At the end, the algorithm just computes the length of the short suffix, right after the end of the last block of the middle part.

The key concepts of this technique are similar to the Locally Consistent Parsing of Sahinalp and Vishikin [45, 12] and also to the techniques of Kociumaka et al. [34]. However, we design novel methods to implement these ideas in a small space. For the randomized algorithms, we combine Karp–Rabin fingerprints with approximately min-wise hash functions, and for the deterministic algorithms we show how to compute the parsing in an online fashion in order to reduce the space usage. In previous results for the LCE problem [4, 5, 22, 47], a set of positions of size $\mathcal{O}(\frac{n}{\tau})$ was also considered by the data structures. In contrast to these algorithms, where the selected positions were dependent only on the *length* of the text, we introduce a novel method that exploits the *actual text* using local properties in order to decide which positions to select.

Partitioning set. In order to achieve the desired special partitioning of the string into blocks, our algorithms pick positions in S, which will be the endpoints of the blocks. Our goal is to pick $\mathcal{O}(\frac{n}{\tau})$ positions, such that the distance between any two consecutive positions is $\mathcal{O}(\tau)$ and, for the locality, the decision whether a position $p \in \{1,2,\ldots,n\} = [n]$ is picked, depends only on the *neighbourhood* of p up to $\mathcal{O}(\tau)$ characters away, which is the substring $S[p-\delta..p+\delta]$ for some $\delta \in \mathcal{O}(\tau)$.

The above discussion captures the key concepts of the algorithms, however, in certain cases it is impossible to pick such a set of positions. For example, if the string is $S = aaa\ldots a$, it is obvious that any local selection must pick all the positions (except possibly for the margins) or none of them. Thus, it is impossible to fulfill both the requirement of $\mathcal{O}(\frac{n}{\tau})$ positions and that the distance between any two consecutive positions will be $\mathcal{O}(\tau)$. Generally, this problem occurs when the string S contain long substrings whose *period* (see Section 2) length is $\mathcal{O}(\tau)$. Thus, our algorithms use period-based techniques to treat positions in such substrings. So, the partitioning of S will contain two types of blocks: periodic blocks, which are substrings with period length at most τ, and regular blocks, that

contains no repetition. For the sake of clarity, all the details about periodic blocks are omitted in this overview.

It remains to describe how the algorithm picks the special positions, and how the algorithm uses these positions to answer LCE queries on these positions fast. We introduce two methods, and therefore two algorithms, for the position selection. The former is a randomized Las-Vegas method and the later is a deterministic method, with a slightly worse construction and query time.

We note that after discover the idea of partitioning set, we found that similar ideas were invented in the previous decade in the area of Bioinformatics, without theoretical analysis, by Schleimer et al. [46] called *winnowing* and by Roberts et al. [43] called *minimizers*. In addition, Kempa and Kociumaka [32, 33] *synchronizing set* was influenced by our definition of partitioning set.

The randomized method. For the randomized data structure, the positions are picked in the following way. The algorithm uses a novel hashing technique, that maps each string of length τ into an integer value in the range $[1..n^c]$ for some constant c. The mapping function is *almost min-wise* function, which means that given k different strings the probability of each one of them to be mapped into the minimum value among all the strings is at most $\mathcal{O}(\frac{1}{k})$. For any position $p \in [n]$ the algorithm computes the fingerprint of $S[p..p+\tau-1]$ and considered it as the ID of position p. Then, the algorithm scans the positions with a sliding window of length τ, and for any consecutive τ positions, the algorithm picks the position with the minimum ID. Notice that it is possible, and even advisable, that a single position will be picked due to multiple overlapping shifts of the sliding window. It is straightforward that in any sequence of τ positions this method will pick at least one position and that the selection of each position depends only on a substring of S of length 3τ. The only possible problem with fulfilling the desired properties is the possibility of choosing too many positions ($\omega(\frac{n}{\tau})$). With a simple probabilistic analysis of this method we prove that under random selection of the hash function, the *expected* number of picked positions is $\mathcal{O}(\frac{n}{\tau})$, assuming that in each offset of the sliding window, each substring of length τ appears only once, i.e., the non-periodic case. The details of this method, including the treatment of periodic substring, appears in Section 4. Later, in Section 8 we show how to improve this method to $\mathcal{O}(\frac{n}{\tau})$ picked positions, with linear construction time *with high probability*. The main idea of this improvement is to evaluate whether the selected min-wise hash function provides a small enough set of positions by sampling a few substrings of S, and if the set of picked positions in these substrings is too large, the algorithm chooses another function.

The deterministic method. In the deterministic method, we mimic the randomized selection, but with a technique of deterministic coin-tossing, going back to Cole and Vishkin [11] and appeared in several other papers [12, 23, 39, 45, 1, 25, 18]. In the high level, this technique transforms a string over an alphabet Σ into a same length string over a constant alphabet in $\mathcal{O}(n \log^* n)$ time. The transformed string has the special property that for any pair of consecutive positions, if the original characters before the transformations were distinct, then the characters on these positions after the transformation are distinct as well. In addition, the transformation of each position depends only on the *neighbourhood* of the position of $\mathcal{O}(\log^* n)$ positions, assuming that S does not contain any kind of periodic substring. Using this transformation, the algorithm splits S into blocks of size at least 2 and at most $\mathcal{O}(1)$, by selecting all the positions which their transformed value is a local minimum. By a repeatedly applying this transformation, while after the first transformation we consider each block as a character over a larger alphabet, the algorithm creates $\mathcal{O}(\frac{n}{\tau})$ blocks of size $\mathcal{O}(\tau)$ characters. Each of these blocks depends only on a substring of S of length $\mathcal{O}(\tau \log^* n)$, which adds only a factor of $\log^* n$ over the complexities of algorithms that use the randomized selection. The total computation time is $\mathcal{O}(n \log \tau)$ and with a novel scheduling technique, which scans the text in one pass from right to left, the computation is implemented using only $\mathcal{O}(\frac{n}{\tau} + \log \tau)$ words of space. During the transformations of the string into blocks, the algorithm recognizes all the long substrings of S with a period length of at most τ characters for the special treatment.

The technique is similar to previous algorithms as [12], however, this is the first implementation where the recursive construction does not use any randomness and also does not require $\Omega(n)$ working space. Another previous algorithm that uses similar techniques is by Fischer et al. [18] which also consider the LCE and SST problems using the Locally Consistent Parsing. However, the model of computation considered in [18] is weaker, where in addition to the $\mathcal{O}(\frac{n}{\tau})$ working space, the algorithm may also edit the space of the input, as long as the algorithm recovers the original input at the end of the construction. While in [18] the usage of the Locally Consistent Parsing technique is involved, and a lot of effort is invested in merging parsing of different parts of the text, in our algorithm we introduce a space- and time-efficient implementation of the parsing for the whole text.

Compute LCE query using the selected positions. Given the partitioning of S induced by

the selected positions, we define for any two indices i', j' which are beginning of some blocks, the longest common block-prefix as the largest ℓ' such that $S[i'..i'+\ell'-1] = S[j'..j'+\ell'-1]$ and all the blocks that compose the two substrings are exactly the same. In order to compute this length, we create a new string S' by converting each block into a unique symbol. To detect repeated occurrences of the same block, and give them the same symbol, we sort the blocks in linear time using a technique from [22]. The result string is of length $\mathcal{O}(\frac{n}{\tau})$ and the alphabet size is also $\mathcal{O}(\frac{n}{\tau})$, so the algorithm builds the suffix tree of this string in $\mathcal{O}(\frac{n}{\tau})$ time and space. Using this suffix tree augmented with an LCA data structure, one can compute the value of ℓ' in constant time.

As described above, given the partitioning of S induced by the selected positions, for any two indices $i, j \in [n]$ with $\mathsf{LCE}(i,j) = \ell > \Omega(\tau)$ the substrings $S[i..i+\ell-1] = S[j..j+\ell-1]$ are composed of a short prefix, a short suffix, and a middle part which is composed of exactly the same blocks in both substrings. Thus, the computation of $\mathsf{LCE}(i,j)$ follows this decomposition and computes the LCE in three phases. In the first phase, the algorithm compares $\mathcal{O}(\tau)$ pairs of characters (or $\mathcal{O}(\tau \log^* n)$ for the deterministic selection). If there is an exact match, the algorithm finds the beginning of the middle part and computes its end using the suffix tree of the blocks. Then, the algorithm just compares additional $\mathcal{O}(\tau)$ pairs of characters (or $\mathcal{O}(\tau \log^* n)$ for the deterministic selection) after the end of the middle part, and it is guaranteed that the mismatch will be found.

Sparse Suffix Tree construction. It is well-known that the SST of a set $B \subseteq [n]$ of suffixes can be built in $\mathcal{O}(|B|)$ time and space given the Sparse Suffix Array (SSA) and the LCP array [31] of B, which are the lexicographic order of the suffixes, and the longest common prefix of each adjacent suffixes due to this order, respectively. Hence, our focus is on sorting the suffixes and on computing their LCP. Our algorithm works in two steps. At the beginning, the algorithm selects a partitioning set P for $\tau = \frac{n}{|B|}$ and builds the SSA of all suffixes starting at positions of P. Then, the algorithm uses the SSA of P to construct the SSA of B.

Our novel technique for computing the SSA of P is based on a special property that is guaranteed by our algorithms that selects partitioning sets. This property ensures that if two suffixes of positions from P share a long prefix, then both prefixes are composed of the same blocks in the partitioning, except for possibly a small portion at the end of the common prefix. Intuitively, the algorithm sorts the blocks and creates a new string S'_P of length $|P|$, such that the ith character in S'_P is the rank of the ith block. Then, we prove that the lexicographic order of any two suffixes of S starting at positions from P is exactly the same as the lexicographic order of the corresponding suffixes of S'_P. The actual construction is a little bit more complicated since it is required to extend the blocks we sort in order to achieve the aforementioned result.

On the second stage for creating the SSA of B, the algorithm uses methods which are similar to the methods that were used to compute the SSA of P. The algorithm creates for any suffix $i \in B$ a short representative string, which basically, ignoring some details, is a prefix of the ith suffix of a proper length. Then the algorithm sorts all the representative strings of the suffixes. Ranking of the representative strings is sufficient to determine the order of two suffixes with different corresponding representative strings. To determine the order of two suffixes which have the same representative string we use the SSA of P from the first stage. The length of the representative string ensures that if two suffixes from B share the same representative string then their order can be determined solely by the order of two corresponding suffixes from P. Thus, by combining the representative strings ranking with the SSA of P, the algorithm can sort all suffixes of B in $\mathcal{O}(n)$ time using only $\mathcal{O}(\frac{n}{\tau})$ words of space.

Even better Deterministic LCE data structure. Using the SST construction algorithm, we show how to combine the deterministic selection with the difference covers [37, 10] technique, to get almost optimal trade-off for the LCE data structure of $\mathcal{O}(\frac{n}{\tau})$ space and $\mathcal{O}(\tau\sqrt{\log^* n})$ query time with $\mathcal{O}(n \log \tau)$ deterministic construction time. We mention that although the technique of difference covers has already been used for the LCE problem (see [42, 5, 22]), the usage in our algorithm is different. While in all previous work the considered positions were all the positions in a given range, in our algorithm the positions are a subset of the locally selected positions. Thus, some properties such as the ability to compute in constant time the common difference for any two indices are impossible to exploit. Even though we need to spend time on finding the synchronized selected positions, our selection gives an $\mathcal{O}(\tau\sqrt{\log^* n})$ bound on the number of comparison steps needed to take until the synchronized positions are found.

Organization. In section 2 we give the basic definitions and tools we use in the paper. Then in Section 3 we give the formal definition for the set of selected positions and show how to use it to create an LCE data structure. We complete the data structures with the randomized Las-Vegas position selection in Section 4 and with the deterministic selection in Section 5. In

Section 6 we show how to use these selections in order to achieve SST construction algorithms. Sections 7 and 8 show how to improve the basic results, where Section 7 discuss on the trade-off improvement for deterministic LCE data structure, and Section 8 discuss on how to ensure the Las-Vegas selection will run in linear time with high probability. In these sections we omit the discussion about the case of periodic block, which we postpone to the appendices, so in Section 9 we overview the main technique that we use for this case.

Due to space constraints, a lot of the details are omitted from this version of the paper, the complete details appear in the full version of the paper [8].

Acknowledgments. We thank an anonymous reviewer for his usefull comments and suggestions.

2 Preliminaries

For $1 \leq i < j \leq n$ denote the *interval* $[i..j] = \{i, i+1, \ldots, j\}$, and $[n] = \{1, 2, \ldots, n\}$. We consider in this paper strings over an integer alphabet $\Sigma = \{1, 2, \ldots, n^{\mathcal{O}(1)}\}$. A string W of length $|W| = \ell$ is a sequence of characters $W[1]W[2]\ldots W[\ell]$ over Σ. A *substring* of W is denoted by $W[x..y] = W[x]W[x+1]\ldots W[y]$ for $1 \leq x \leq y \leq \ell$. If $x = 1$ the substring is called a *prefix* of W, and if $y = \ell$, the substring is called a *suffix* of W.

A prefix of W of length $y \geq 1$ is called a *period* of W if and only if $W[i] = W[i+y]$ for all $1 \leq i \leq \ell - y$. The shortest period of W is called *the principal period* of W, and its length is denoted by ρ_W. If $\rho_W \leq \frac{|W|}{2}$ we say that W is *periodic*.

Sparse suffix trees. Given a set of strings, the compact trie [40] of these strings is the tree induced by shrinking each path of nodes of degree one in the trie [13, 20] of the strings. Each edge in the compact trie has a label which is a substring of part of the given strings. Each node in the tree is associated with a string, which is the concatenation of all the labels of the edges of the simple path from the root to the node. The suffix tree of a string S is the compact trie of all the suffixes of S. The sparse suffix tree of a string S of length n, with a set $B \subseteq [n]$ is the compact trie of all the suffixes $\{S[i..n] \mid i \in B\}$.

Fingerprints. Given n, for the following let $u, v \in \bigcup_{i=0}^{n} \Sigma^i$ be two strings of size at most n. Porat and Porat [41] and Breslauer and Galil [9] proved that for every constant $c > 1$ there exists a *fingerprint function* $\phi : \bigcup_{i=0}^{n} \Sigma^i \to [n^c]$, which is closely related to the hash function of Karp and Rabin [30] and Dietzfelbinger et al. [14], such that:

1. If $|u| = |v|$ and $u \neq v$ then $\phi(u) \neq \phi(v)$ with high probability (at least $1 - \frac{1}{n^{c-1}}$).
2. *The sliding property:* Let $w{=}uv$ be the concatenation of u and v. If $|w| \leq n$ then given the length and the fingerprints of any two strings from u,v and w, one can compute the fingerprint of the third string in constant time.

Approximately Min-Wise Hash Function. We say that $\mathcal{F}$ is an (ϵ, k)-min-wise independent hash family if for any $X \subset [n], |X| < k$ and $x \in [n] \setminus X$ when picking a function $h \in \mathcal{F}$ uniformly we have: $\frac{1-\varepsilon}{|X|+1} \leq \Pr_h[h(x) < \min\{h(y) \mid y \in X\}] \leq \frac{1+\varepsilon}{|X|+1}$

Indyk [27] have proved that any $\mathcal{O}(\log \frac{1}{\epsilon})$-independent hash family is also a $(\epsilon, \frac{\epsilon n}{c})$-min-wise independent hash family, for some constant $c > 1$. Furthermore, since the family of polynomials of degree $\ell - 1$ over $GF(n)$, assuming n is a prime, is an ℓ-independent hash family, then for $\ell = \mathcal{O}(\log \frac{1}{\epsilon})$ the polynomials' family is $(\epsilon, \frac{\epsilon n}{c})$-min-wise independent hash family. For any polynomial from the family, evaluating its value for a given input can be done in $\mathcal{O}(\log \frac{1}{\epsilon})$ time.

Periodicity We prove the following lemmas in the full version of the paper [8].

LEMMA 2.1. *Let u be a string with period length p. If v is a substring of u of length at least p, and v has a period length of $q|p$, then u has a period length of q.*

LEMMA 2.2. *Let u and v be two strings of length $|u| = |v| \geq 2t$ for some $t \in \mathbb{N}$, such that both $\rho_u \leq t$ and $\rho_v \leq t$. If $u[1..2t] = v[1..2t]$ then $u = v$.*

Sorting Strings The following lemmas are direct results of two facts stated in [22].

LEMMA 2.3. ([22, FACT 3.1 AND FACT 3.2]) *For any $1 \leq \tau \leq n$, given random access to $\mathcal{O}(n/\tau)$ strings of length $\mathcal{O}(\tau \log^* n)$, the strings can be sorted in $\mathcal{O}(n \log^* n)$ time and $\mathcal{O}(n/\tau)$ words of space.*

LEMMA 2.4. ([22, FACT 3.1 AND FACT 3.2]) *For any $1 \leq \tau \leq n$, given random access to $\mathcal{O}(n/\tau)$ strings of length $\mathcal{O}(\tau)$, the strings can be sorted in $\mathcal{O}(n)$ time and $\mathcal{O}(n/\tau)$ words of space.*

Lowest Common Ancestor For any tree T, we denote by $LCA(u, v)$ the *lowest common ancestor* of nodes $u, v \in T$. The LCA of any two leaves in a trie is the longest common prefix (LCP) of the corresponding strings. Hence, an efficient data structure for LCA queries implies an efficient data structure for LCP queries.

LEMMA 2.5. ([24, 3]) *Any tree T can be preprocessed in $\mathcal{O}(|T|)$ time and space to support LCA queries in constant time.*

3 LCE with Selected Positions

Following the discussion above in Section 1.2, we introduce here the formal definition of the selected positions that defines the partitioning of S into blocks.

The partitioning of S is induced by a set P of indices from $[n]$, such that each block is a substring beginning in some index of $P \cup \{1\}$ and the concatenation of all the blocks is exactly S. The following definition introduces the essential properties of the set P, required for our data structures.

DEFINITION 1. *A set of positions $P \subseteq [n]$ is called a (τ, δ)-*partitioning set *of S, for some parameters $1 \leq \tau \leq \delta \leq n$, if and only if it has the following properties:*

1. **Local Consistency** — *For any two indices $i, j \in [1 + \delta..n - \delta]$ such that $S[i - \delta..i + \delta] = S[j - \delta..j + \delta]$ we have $i \in P \Leftrightarrow j \in P$.*
2. **Compactness** — *Let $p_i < p_{i+1}$ be two consecutive positions in $P \cup \{1, n + 1\}$ then we have one of the following.*
   (a) (Regular block) $p_{i+1} - p_i \leq \tau$.
   (b) (Periodic block) $p_{i+1} - p_i > \tau$ *such that $u = S[p_i..p_{i+1} - 1]$ is a periodic string with period length $\rho_u \leq \tau$.*

Here we describe how to use a (τ, δ)-partitioning set of size $\mathcal{O}(\frac{n}{\tau})$ in order to construct an LCE data structure in $\mathcal{O}(n)$ time using $\mathcal{O}(\frac{n}{\tau})$ words of space. The data structure uses $\mathcal{O}(\frac{n}{\tau})$ words of working space and answers queries in $\mathcal{O}(\delta)$ time. In Section 4, we introduce a randomized Las-Vegas algorithm that finds a $(2\tau, 2\tau)$-partitioning set of size $\mathcal{O}(\frac{n}{\tau})$ in $\mathcal{O}(n)$ expected time (which we later improve to be with high probability in Section 8), using only $\mathcal{O}(\frac{n}{\tau})$ words of space. In Section 5, we introduce a deterministic algorithm that finds an $(\mathcal{O}(\tau), \mathcal{O}(\tau \log^* n))$-partitioning set of size $\mathcal{O}(\frac{n}{\tau})$ in $\mathcal{O}(n \log \tau)$ time using only $\mathcal{O}(\frac{n}{\tau} + \log \tau)$ words of space.

Remark about border cases. When using a (τ, δ)-partitioning set we assume that all the queries to the algorithm are made with indices from the range $[1 + \delta..n - \delta]$. If this is not the case, the algorithm considers string $S' = \#^\delta S \#^\delta$ where $\#$ is a special character. Notice that $|S'| = n + 2\delta = \mathcal{O}(n)$ and if S is given in a read only memory, the algorithm is able to simulate S' with additional constant space.

The partitioning string. Given a (τ, δ)-partitioning set P of size $\mathcal{O}(\frac{n}{\tau})$ we consider the partitioning of S induced by P. Let $p_1 < \cdots < p_h$ be all the indices in $P \cup \{1, n + 1\}$, then the ith block of S (for $1 \leq i < h$) is $S_i = S[p_i..p_{i+1} - 1]$. Since there are at most $\mathcal{O}(\frac{n}{\tau})$ blocks, we associate with each block a unique symbol from $\Sigma' = \{1, \ldots, \mathcal{O}(\frac{n}{\tau})\}$, such that if two blocks are equal (correspond to the same string) they have the same symbol. The unique symbols are determined by sorting the blocks in $\mathcal{O}(n)$ time due to Lemma 2.4, where the only exception are periodic blocks whose length is larger than τ. For those blocks, we take only the first 2τ characters into the sorting and add at the end a new character which is the length of the block (we assume that the alphabet of S contains $\{1, 2, \ldots, n\}$, otherwise we extend the alphabet). The value of each block is its rank among all the (distinct) blocks. It is straightforward that the value of each block is a unique identifier which equals among all copies of the block. Using these values, we consider the *partitioning string* S_P of length $h - 1$ over Σ', defined such that $S_P[i]$ is the value of the ith block $S[p_i..p_{i+1} - 1]$. The string S_P is constructed in $\mathcal{O}(n)$ time from the partitioning set P and its length is $\mathcal{O}(\frac{n}{\tau})$.

The data structure. The algorithm builds a suffix tree T_P on the string S_P using the algorithm of Farach-Colton [15] in $\mathcal{O}(\frac{n}{\tau})$ time. Each node in T_P maintains, as its *actual depth*, the length of the substring of S (over Σ) corresponding to the substring of S_P written on the path from the root of T_P to the node. Moreover, T_P is preprocessed to support LCA queries (see Theorem 2.5), thus for any two leaves one can compute the length of the longest common prefix composed of complete blocks for the two corresponding positions in P in constant time.

All the positions in P are maintained in an ordered linked list, and each $p_i \in P$ is maintained with a pointer to the leaf in T_P corresponding to the block beginning at position p_i. In addition, let $\mathsf{succ}_P(\alpha) = \min\{p \in P \mid p \geq \alpha\}$ be the successor of position α, the data structure stores an array SUC of length $\lfloor n/\tau \rfloor$ such that $SUC[j]$ stores a pointer to the node of $\mathsf{succ}_P(j \cdot \tau)$ in the linked list. Notice that for any $i \in [n]$, the successor $\mathsf{succ}_P(i)$ can be found in $\mathcal{O}(\tau)$ time by sequential search on the list from the position pointed to by $SUC\left[\lfloor \frac{i}{\tau} \rfloor\right]$.

Query Processing. Due to property 2 of Definition 1 each block is either of length at most τ or it has period length of at most τ. For the sake of intuition we first consider the case where the length of each block is at most τ. In [8] we describe how to generalize the processing for the case where there exist large blocks with a short period as well.

We consider the computation of $\mathsf{LCE}(i, j) = \ell$ for some $i, j \in [n]$. The computation consist of (at most) three phases. At the first phase the algorithm compares $S[i + k]$ and $S[j + k]$ for k from 0 to $3\delta \geq 2\delta + \tau$. If

a mismatch is found, then the minimum k such that $S[i+k] \neq S[j+k]$ is $\mathsf{LCE}(i,j)$. Otherwise, since $\mathsf{LCE}(i,j) > 3\delta$ we use the partitioning into blocks. We use the following lemma that states that all the blocks of $S[i..i+\ell-1]$ and $S[j..j+\ell-1]$ are the same except for the margins of at most δ characters. The proof of the lemma appears in [8].

LEMMA 3.1. *Let $i,j \in [n]$ be two indices and let $\ell = \mathsf{LCE}(i,j)$. If $\ell > 2\delta$ then:*

$$\{p-i \,|\, p \in (P \cap [i+\delta..i+\ell-\delta-1])\} = \{p-j \,|\, p \in (P \cap [j+\delta..j+\ell-\delta-1])\}$$

In the second phase, the algorithm finds a correlated offset $\alpha = \min\{p-i \,|\, p \in (P \cap [i+\delta..i+\ell-\delta])\}$ which due to Lemma 3.1 has the property that $i+\alpha \in P$ and $j+\alpha \in P$. The computation of α can be done in $\mathcal{O}(\tau)$ time by retrieving $\mathsf{succ}_P(i+\delta)$. In addition, it must be that the substrings $S[i+\alpha..i+\ell-1]$ and $S[j+\alpha..j+\ell-1]$ are partitioned into blocks exactly the same, except for possibly the last $\tau+\delta$ characters. The algorithm finds the end of the last common block using an LCA query on T_P in constant time. Since we assume that all the blocks are of length τ at most, it must be that the offset of this block is at least $\ell-\delta-\tau-1$. Hence, in the third phase the algorithm compares the characters from the end of the last common block until a mismatch is encountered. It is guaranteed that such a mismatch will be found after at most $\tau+\delta$ comparisons.

To summarize, we show in this section the following lemma.

LEMMA 3.2. *Given a (τ,δ)-partitioning set of size $\mathcal{O}(\frac{n}{\tau})$ there exists a deterministic construction algorithm of an LCE data structure that takes $\mathcal{O}(n)$ time using $\mathcal{O}(\frac{n}{\tau})$ words of space. The space usage of the data structure is $\mathcal{O}(\frac{n}{\tau})$ and its query time is $\mathcal{O}(\delta)$.*

Thus, combining Lemma 3.2 with the randomized selection of Sections 4 and 8 (Lemma 8.4) we have proved Theorem 1.1. Similarly, combining Lemma 3.2 with the deterministic selection of Section 5 we have proved Lemma 1.1.

4 Randomized Selection

Using fingerprint hash functions and approximately min-wise independent hash functions, we prove that a $(2\tau,2\tau)$-partitioning set consisting of $\mathcal{O}(\frac{n}{\tau})$ positions can be constructed in linear time with high probability. In this section we show a method for constructing such a partitioning set of size $\mathcal{O}(\frac{n}{\tau})$ using linear time in expectation. Later, in Section 8 we improve the method to ensure linear construction time with high probability. All the algorithms use $\mathcal{O}(\frac{n}{\tau})$ words of working space.

We define a function $\mathrm{ID} : [1..n-\tau-1] \to \mathbb{R}$ such that $\mathrm{ID}(j)$ depends only on $S[j..j+\tau+1]$, hence if $S[j..j+\tau-1] = S[k..k+\tau-1]$ for some indices j and k then $\mathrm{ID}(j) = \mathrm{ID}(k)$. Using ID, we create P, a partitioning set of S. For each interval $[i..i+\tau-1]$ the positions within the interval with the smallest ID value are picked for P. In other words,

$$P = \left\{ j \in [1..n-\tau] \;\middle|\; \exists \ell \in [j-\tau+1..j] : \mathrm{ID}(j) = \min_{k \in [\ell..\ell+\tau-1]} \{\mathrm{ID}(k)\} \right\}$$

From the locality properties of the ID function, the following lemma follows immediately.

LEMMA 4.1. *Given an ID function as described above, the set P induced by ID is a $(2\tau,2\tau)$-partitioning set.*

Proof. We will prove that the two properties of Definition 1 hold.

Local Consistency. Let $i,j \in [1+2\tau..n-2\tau]$ such that $S[i-2\tau..i+2\tau] = S[j-2\tau..j+2\tau]$. For any $a \in [i-2\tau..i+\tau+1]$, since $\mathrm{ID}(a)$ depends only on $S[a..a+\tau-1]$, we have that $\mathrm{ID}(a) = \mathrm{ID}(a-i+j)$.

Assume $i \in P$ and let $\ell \in [i-\tau+1..i]$ be an index such that $\mathrm{ID}(i) = \min_{k \in [\ell..\ell+\tau-1]}\{\mathrm{ID}(k)\}$. We have that $[\ell..\ell+\tau-1] \subset [i-\tau+1..i+\tau-1] \subset [i-2\tau..i+\tau+1]$. Hence, for any $k \in [\ell..\ell+\tau-1]$ we have $\mathrm{ID}(k) = \mathrm{ID}(k-i+j)$. In particular,

$$\begin{aligned}\mathrm{ID}(j) = \mathrm{ID}(i) &= \min_{k \in [\ell..\ell+\tau-1]}\{\mathrm{ID}(k)\} \\ &= \min_{k \in [\ell-i+j..\ell-i+j+\tau-1]}\{\mathrm{ID}(k)\}\end{aligned}$$

Thus, due to the definition of P, $j \in P$. Hence, we prove that $i \in P \Rightarrow j \in P$. The opposite direction is symmetric.

Compactness. Let $1 = p_0 < p_1 < \cdots < p_k < p_{k+1} = n+1$ be all the positions in $P \cup \{1, n+1\}$. Notice that if the first selected position is not $p_0 = 1$, then p_1 is selected from the interval $[1..\tau]$, and therefore $p_1 - p_0 < \tau$. Moreover, the rightmost position in P, p_k, is the position picked on the interval $[n-2\tau+1..n-\tau]$. Thus, $p_{k+1} - p_k = n+1-p_k \leq 2\tau$ Finally, for the general case where both $p_i, p_{i+1} \in P$ assume by a contradiction that $p_{i+1} - p_i > \tau$. Therefore no position is selected in the interval $[p_i+1..p_i+\tau] \subseteq [p_i+1..p_{i+1}-1]$, and by definition of P this cannot happen. ∎

Remark about the periodic case. Notice that if S contains a long periodic substring with a small period,

then the same IDs will appear in every period (except perhaps in the last τ positions within the periodic substring). In particular, many positions will have the minimum ID value, hence P will be a very large partitioning set, no matter which ID function we use. This means that long periodic substring with small periods should be taken with special care. Formally we define:

DEFINITION 2. *A substring $S' = S[i..j]$ is a (d, ρ)-run of S if and only if $|S'| \geq d$ and the principal period length of S' is at most ρ.*

Overview of the solution. We show here the solution for the non-periodic case: We prove that when S contains no $(\tau, \frac{\tau}{6})$-runs, it is possible to construct a partitioning set of $\mathcal{O}(\frac{n}{\tau})$ positions in expected linear time. In [8], we show that all $(\tau, \frac{\tau}{6})$-runs can be found in linear time and constant space using fingerprints. We then combine the two methods to partition the string into blocks —find all $(\tau, \frac{\tau}{6})$-runs and mark them as blocks, and then use the non-periodic algorithm to partition the substring between every two runs in a total of expected linear time and $\mathcal{O}(\frac{n}{\tau})$ space.

The ID function. Our goal is to use an ID function that induces a partitioning set of size $\mathcal{O}(\frac{n}{\tau})$ that we will be able to compute in $\mathcal{O}(n)$ time in expectation. To achieve this, we fix a fingerprint function ϕ and an approximately min-wise hash function $h \in \mathcal{F}$ where $\mathcal{F}$ is a $\left(\frac{1}{2}, \tau\right)$-min-wise independent hash family (see Section 2). We denote $\phi_i = \phi\left(S[i..i+\tau-1]\right)$ and the algorithm uses the function $\mathrm{ID}_{\phi,h}(i) = h(\phi_i)$. The fingerprint function is used for the locality of $\mathrm{ID}_{\phi,h}$, and the hash function is used for diversifying the fingerprints that will have minimal value.

As mentioned before, we assume S contains no $(\tau, \frac{\tau}{6})$-runs. We show that when using $\mathrm{ID}_{\phi,h}$ where ϕ and h are picked uniformly, the size of the partitioning set P is $\mathcal{O}(\frac{n}{\tau})$ in expectation.

LEMMA 4.2. *If ϕ and h are picked uniformly and S contains no $(\tau, \frac{\tau}{6})$-runs then $\mathbb{E}_{\phi,h}\left[|P|\right] = \mathcal{O}\left(\frac{n}{\tau}\right)$.*

Proof. [Proof Sketch; full proof appears in [8]] We assume that there are no fingerprints collisions, since the contribution to the expectation of the other case is constant due to the small probability of fingerprints collision. We use the linearity of expectation, which means that $\mathbb{E}_{\phi,h}[|P|] = \sum_{i=1}^{n} \Pr_{\phi,h}(A_i)$ where A_i is the event that $i \in P$. We bound $\Pr_{\phi,h}(A_i)$ by noticing that for any i, we have $i \in P$ only if $h(\phi_i)$ is the minimum ID among all the IDs of $[i-\tau/2..i]$ or the minimum among all the IDs of $[i..i+\tau/2]$. Due to the almost min-wise function h, the probability of each one of these events is $\mathcal{O}(\frac{1}{\tau})$ and therefore $\Pr_{\phi,h}(A_i) \leq \mathcal{O}(\frac{1}{\tau})$. Hence, $\mathbb{E}_{\phi,h}[|P|] = \sum_{i=1}^{n} \Pr_{\phi,h}(A_i) = \mathcal{O}(\frac{n}{\tau})$. ∎

LEMMA 4.3. *The partitioning set P induced by the function $ID_{\phi,h}$ on a substring containing no $(\tau, \frac{\tau}{6})$-runs can be constructed in expected $\mathcal{O}(n)$ time using expected $\mathcal{O}(\frac{n}{\tau})$ words of space.*

Proof. For every interval of size τ, the algorithm will find the positions with minimum IDs and add them to P, using only $\mathcal{O}(1)$ additional space. When sliding the interval by one step, using the sliding property of the fingerprints, the next ID is calculated in $\mathcal{O}(1)$ time. Let $p_j \in P$ be the rightmost position picked for P for some interval $[k..k+\tau-1] \subset [1..n-\tau]$. Since p_j has the minimum ID value in that interval, then if $p_j \in [k+1..k+\tau]$ the algorithm can find out in $\mathcal{O}(1)$ if $k+\tau \in P$ by calculating $\mathrm{ID}(k+\tau)$ and comparing it with $\mathrm{ID}(p_j)$.

Otherwise, if $p_j = k$, then the algorithm must recalculate all the IDs on the interval $[k+1..k+\tau]$ to find the position with the minimum ID value. This recalculation takes $\mathcal{O}(\tau)$ time due to the sliding property of the fingerprint, and it happens only if the new interval does not contain the currently rightmost position in P (i.e., $p_j = k$). Therefore we can charge the $\mathcal{O}(\tau)$ time on p_j. Notice that p_j can be charged at most once since immediately after we charge p_j, a new rightmost position in P is picked.

Therefore, the algorithm runs in $\mathcal{O}(n+\tau \cdot |P|)$ using $\mathcal{O}(|P|)$ space, which by Lemma 4.2 is expected to be $\mathcal{O}(n)$ time using expected $\mathcal{O}(\frac{n}{\tau})$ words of space. ∎

Due to the Lemma 4.2 and Lemma 4.3, using Markov's inequality we conclude with the following corollary.

COROLLARY 4.1. *A $(2\tau, 2\tau)$-partitioning set of S can be constructed using $\mathcal{O}(\frac{n}{\tau})$ words of space with expected $\mathcal{O}(n)$ construction time, assuming S contains no $(\tau, \frac{\tau}{6})$-runs.*

5 Deterministic Selection

In this section we introduce a deterministic algorithm that finds a $(\mathcal{O}(\tau), \mathcal{O}(\tau \log^* n))$-partitioning set of size $\mathcal{O}(\frac{n}{\tau})$. The key concepts we use are based on the techniques of Cormode and Muthukrishnan [12], which in turn are similar to other previous papers [23, 39, 11, 44, 18]. Here we focus on the definition of the positions selected by the algorithm. In [8] we describe how to select these positions using only $\mathcal{O}(\frac{n}{\tau} + \log \tau)$ words of space in $\mathcal{O}(n \log \tau)$ time.

The algorithm creates a hierarchical decomposition of S, such that the top-level has $\mathcal{O}(n/\tau)$ blocks. At the

bottom level (level 0) each character in S is considered as a block, so the set of picked positions is all $[n]$. We create levels in a bottom-up fashion, preserving the following properties. At each level μ there are at most $\mathcal{O}\left(\frac{n}{(3/2)^\mu}\right)$ blocks. Moreover, the length of each block is $\mathcal{O}((3/2)^\mu)$, except for blocks which are substrings of S that have period length of at most $(3/2)^\mu$, which can be longer. In addition, the locality of the partitioning is obtained by limiting the dependency of each block in level μ to at most $\mathcal{O}(\log^* n)$ blocks in level $\mu-1$. These properties are a generalization of Definition 1 (partitioning set) of size $\mathcal{O}(\frac{n}{\tau})$, so by repetitively applying the transformation from level $\mu = 0$ up to level $\mu = \log_{3/2}\tau$ we obtain a $(\mathcal{O}(\tau), \mathcal{O}(\tau \log^* n))$-partitioning set.

Now we explain how the decomposition of level μ is defined based on the decomposition of level $\mu - 1$. For clarity of representation, when describing the block of level μ, we call the blocks of level $\mu - 1$ *sub-blocks*. Given the $(\mu - 1)$th level of the hierarchy, we consider parsing of S into four types of maximal non-overlapping substrings.

1. Single sub-block of length at least $(3/2)^\mu$.
2. Maximal contiguous sequence of eqaul sub-blocks, each of which is not of type 1.
3. Sequences of at least $c \log^* n$ sub-blocks, that are not of type 1 or 2 (where c is a constant to be determined).
4. Sequences of less than $c \log^* n$ sub-blocks, that are not of type 1 or 2.

We identify each block with the index of its leftmost endpoint, which we called the *representative position* of the block. As a consequence, position 1 is a representative position at any level of the hierarchy. The algorithm treats each type of such maximal substrings separately. For the sake of simplicity, we describe it now as an algorithm which has no space constraints, which we overcome in [8].

The first type contains sub-blocks of length at least $(3/2)^\mu$ and the algorithm just maintains them as blocks in level μ. The substrings of the second type are sequences of (at least two) identical sub-blocks. In this case, the algorithm merges all the sub-blocks in each sequence into one large block. To identify such sub-blocks, the algorithm compares each sub-block with its successor, and if they are same, the algorithm merges them in level μ.

The alphabet reduction. For the third type of substrings, we use the following method. For the sake of intuition, we begin with the special case where $\mu = 1$, which is the transformation from level 0 (where each character of S has a unique block) to the next level. We follow the technique of [12] and reduce each character into a number from the set $\{0, 1, 2, 3, 4, 5\}$ using a *local* method. The alphabet reduction is done in $\mathcal{O}(\log^* n)$ iterations, which we call *inner-iterations* (to distinguish them from the *outer*-iterations of μ from 0 to $\log_{3/2}\tau$). We assume that the alphabet of S is $\Sigma = \{1, 2, \ldots, |\Sigma|\}$ of size $|\Sigma| = \mathcal{O}(n^{\mathcal{O}(1)})$. Hence, before the reduction begins each character can be considered as a bit string of length $\mathcal{O}(\log n)$. At each inner iteration, the algorithm reduces the length of the binary string logarithmically. We denote by $\mathsf{label}(i, 0) = S[i]$ the character of S in position i. For any two consecutive characters of the $(j-1)$th inner iteration, $\mathsf{label}(i, j-1)$ and $\mathsf{label}(i+1, j-1)$, we define $\mathsf{label}(i, j)$ as follows. Let ψ be the index of the least significant bit in which $\mathsf{label}(i, j-1)$ and $\mathsf{label}(i+1, j-1)$ are different[2], and let $bit(\mathsf{label}(i, j-1), \psi)$ be the value of the ψth bit of $\mathsf{label}(i, j-1)$. We define $\mathsf{label}(i, j) = 2\psi + bit(\mathsf{label}(i, j-1), \psi)$, i.e., the concatenation of the index ψ with the value of the bit in this index.

LEMMA 5.1. ([12, LEMMA 1]) *For any i, if $S[i] \neq S[i+1]$ then $\mathsf{label}(i, j) \neq \mathsf{label}(i+1, j)$ for any $j \leq 0$.*

Each inner iteration of this reduction, reduces an alphabet of size σ to an alphabet of size $2\lceil \log \sigma \rceil$. We run such inner-iterations until the size of the alphabet is constant. Hence, after $\mathcal{O}(\log^* n)$ inner-iterations the size of the alphabet becomes constant, we denote by c the appropriate constant for the $\mathcal{O}(\log^* n)$ inner-iterations. By the following lemma, the size of the final alphabet is exactly 6.

LEMMA 5.2. ([12, LEMMA 2]) *After the final inner-iteration of alphabet reduction, the alphabet size is* 6.

The main challenge with this technique is how to repeat this process with larger and larger blocks. In [12] there are two methods: a randomized algorithm that uses fingerprints to reduce the size of each block to a constant number of words ($\mathcal{O}(\log n)$ bits), or an algorithm with expensive $\mathcal{O}(n)$ additional working space that uses bucket sort. Since we want a deterministic algorithm, and since we have just $\mathcal{O}(\frac{n}{\tau})$ words of working space we use a different method, which costs us

[2]Notice that the label of position i depends on the label of position $i+1$, instead of $i-1$. This change from the original Locally Consistent Parsing, is very useful in the SST construction since it makes the partitioning set *forward synchronized* (see Definition 3).

$\log \tau$ factor in the running time and uses the following observation. We observe that each block at any outer-iteration is a string of size at most $\mathcal{O}(n)$, thus we consider each block as a number between one and $\mathcal{O}(|\Sigma|^n) = \mathcal{O}(n^{O(n)})$. Formally, we assume without loss of generality that $0 \notin \Sigma$, the alphabet of S, and we define $\Pi = \{0, 1, \ldots, 2^{\lceil \log_2(|\Sigma|+1) \rceil n}\}$ as the *blocks' alphabet*. Each block $u = S[x..y]$ at any iteration is a string of length at most n over Σ, and is represented by one symbol from Π by concatenating the binary representation of u's characters, with padding of zeros to the left, i.e., the symbol of u is $\sum_{k=0}^{|u|-1} u[|u|-k] \cdot 2^{\lceil \log_2(|\Sigma|+1) \rceil k} \in \Pi$. The alphabet reduction for any iteration $\mu > 1$ is done exactly as described for $\mu = 1$ except that instead of converting characters from Σ the algorithm considers the symbols of Π induced by the blocks of level $\mu - 1$.

Notice that $\Pi = \mathcal{O}(|\Sigma|^n) = \mathcal{O}(n^{\mathcal{O}(n)})$ and therefore, the alphabet reduction for any outer-iteration takes $\mathcal{O}(\log^*(n^{\mathcal{O}(n)})) = \mathcal{O}(\log^* n)$ inner-iterations. For any outer-iteration the algorithm executes $\mathcal{O}(\log^* n)$ inner iterations. Each first inner-iteration takes $\mathcal{O}(n)$ time since it has to take into account all the blocks which their length summed to $\Theta(n)$. However, after one inner-iteration each block reduced from $\mathcal{O}(\log n^{\mathcal{O}(n)}) = \mathcal{O}(n \log n)$ bits to $\mathcal{O}(\log(n \log n)) = \mathcal{O}(\log n)$ bits, and therefore can be maintained in a constant number of words. Thus, after the first inner-iteration each block comparison takes constant time. Since the number of blocks reduces exponentially and it is $\mathcal{O}\left(\frac{n}{(3/2)^\mu}\right)$, the total time for all the inner iterations, except the first inner iteration of each outer-iteration is $\mathcal{O}(n \log^* n)$. Together with the first inner-iterations the total time required for all the transformations is $\mathcal{O}(n(\log \tau + \log^* n))$. However, using lookup-table, we reduce this time to $O(n \log \tau)$.

Remark that at the end of the alphabet reduction, the blocks' symbols are all in $\{0, 1, 2, 3, 4, 5\}$. At this point, the algorithm scans all blocks and compares their symbol's value with those of their right and left neighbors. The algorithm picks all positions that have a local minimum symbol value as the selected positions in level μ. The only exceptions are regarding the margins of the sequence. For the rightmost $c \log^* n$ characters, the alphabet reduction cannot reach the last level, and therefore they do not have a corresponding number from $\{0, \ldots, 5\}$, thus, the algorithm treats them as sequences of type 4. In addition, the leftmost position in the sequence is always selected by the algorithm. Moreover, if the second block in the sequence (from left) receives a symbol which is a local minimum, the algorithm ignores it in order to avoid the case that the first sub-block will become a block in level μ. So, by the merging due to the minimal symbol, each block of level μ is composed of at least 2 sub-blocks of level $\mu - 1$ and at most 11 sub-blocks (10 is the maximal distance between minimal positions over the symbols $\{0, \ldots, 5\}$ and additional one block for the case of the leftmost block).

Margin sub-blocks (type 4). Each sequence of type 4 contains at most $c \log^* n$ sub-blocks. If the sequence contains exactly one sub-block, this sub-block is kept as a block in level μ of the hierarchy. If the sequence contains two or three sub-blocks, the algorithm creates one block in level μ which is the concatenation of all these three sub-blocks. As for the last case, if the sequence contains at least four sub-blocks, the algorithm creates one block in the μth level which is the union of the two rightmost blocks, and repeats the process from right to left until there are no sub-blocks of level $i-1$ in the sequence which are not part of any block in level μ. Notice that each block created in such sequence contains at least two sub-blocks from the lower level and at most three of them, except for the case where the sequence contains exactly one sub-block.

Conclusion. The set of positions induced by level $\mu = \log_{3/2} \tau$ of the hierarchical decomposition, is a $(\mathcal{O}(\tau), \mathcal{O}(\tau \log^* n))$-partitioning set of size $\mathcal{O}(\frac{n}{\tau})$. Moreover, the algorithm finds this set using only $\mathcal{O}(\frac{n}{\tau} + \log \tau)$ words of space in $\mathcal{O}(n \log \tau))$ time. The complete details and proof of correctness appear in the full version of the paper [8].

6 Sparse Suffix Tree Construction

In this section we introduce a method to build the Sparse Suffix Tree of a set $B \subseteq [n]$ of suffixes of S. Since the output size is $\Theta(|B|)$ words of space, we are able to use $\mathcal{O}(|B|)$ words of working space without affecting the space usage of the algorithm. Thus, we will use the LCE data structure with $\tau = \frac{n}{|B|}$ that uses $\mathcal{O}(\frac{n}{\tau}) = \mathcal{O}(|B|)$ words of space. We focus on computing the Sparse Suffix Array (SSA) of the suffixes, which is an array that stores B's suffixes ordered by their lexicographic order. Then, one can compute the longest common prefix (LCP) of any pair of consecutive (by the lexicographic order) suffixes using the LCE data structure from Section 3. Using the SSA and LCP information one can build the desired SST in $\mathcal{O}(|B|) = \mathcal{O}(\frac{n}{\tau})$ time, using the algorithm of Kasai et al. [31]. Our construction algorithms will use the same (τ, δ)-partitioning sets that were introduced in Section 4 and Section 5. We will prove that these sets have a special property that enables us to compute efficiently the SSA of the set P that was selected by the algorithm. Then, we will show how to use this SSA to compute the desired SSA of the set B.

Recall that Lemma 3.1, which was induced by the definition of (τ, δ)-partitioning set, said that for any

two indices with large common extension both common substrings are composed of the same blocks except for possibly the right and left margins of length δ. For the SST construction, we require an additional property. We want that for every pair of selected positions, i.e., positions from P, with large LCE, the two suffixes also begin with composition into exactly the same blocks, and the only difference in the composition of the common extensions is at the right margin. Formally, we say that a partitioning set is *forward synchronized* if it has the following property.

Definition 3. (Forward synchronization) *A (τ, δ)-partitioning set P is called* forward synchronized *if and only if for any $p_i, p_j \in P$, which their consecutive positions in P are p_{i+1} and p_{j+1}, respectively, if $\mathsf{LCE}(p_i, p_j) > p_{i+1} - p_i + \delta$ then $p_{i+1} - p_i = p_{j+1} - p_j$.*

We will show how to build the sparse suffix tree of a forward synchronized (τ, δ)-partitioning set of size $\mathcal{O}(\frac{n}{\tau})$ in $\mathcal{O}(n \cdot \frac{\delta}{\tau})$ time, using $\mathcal{O}(\frac{n}{\tau})$ words of space. Fortunately, our two constructions of partitioning sets, from Section 4 and Section 5 are forward synchronized, as claimed in the following lemma. The proof is given in the full version of this papaer [8] separately for the randomized algorithm and for the deterministic algorithm.

Lemma 6.1. *Any (τ, δ)-partitioning set that is a result of the algorithms from Section 4 or Section 5 is forward synchronized.*

6.1 SSA of Forward Synchronized (τ, δ)-Partitioning Set Let P be a forward synchronized (τ, δ)-partitioning set, we show here how to compute the SSA of P efficiently. Let us consider the decomposition of S into blocks, induced by P. According to this decomposition, all the suffixes of S beginning in positions from P are composed only of complete blocks. Moreover, since P is forward synchronized, it is guaranteed that whenever comparing two suffixes with long LCE, the two suffixes must begin with identical blocks. As with the definition of S_P in Section 3, we aim to create a partitioning string S'_P of length $|P|$ such that each block of S will be transformed into one symbol in this string, and any two suffixes begins in positions from P will have the same order as the corresponding suffixes of the partitioning string. Recall that when we defined S_P we sorted the substrings of the blocks and transformed each block into its rank. This method seemed to work also for our aim, since when comparing two suffixes, we hope that the first non-identical blocks will determine the order of the suffixes properly, according to the sorting. However, this method has one problem, which is the order of two blocks when one of them is a prefix of the other. In such a case, the order of the suffixes is not determined by these blocks, but by pair of characters which at least one of them is not in these two blocks.

We use the following method to create a S'_P, which overcomes the difficulty described above. For each regular block (of type 2a in Definition 1) beginning at position $p_i \in P$ we consider the *representative string* of the block as $s_i = S[p_i..p_i + 3\delta - 1]$. Notice that this string may contain also characters outside the block of p_i. More detailed, since $\delta \geq \tau$, and due to property 2 of Definition 1, the only case where s_i do not contain characters outside the block is when the block $[p_i..p_{i+1} - 1]$ is a large block with a period length of at most τ characters. Hence, when comparing two suffixes which their first non-identical block is regular (of type 2a), the sorting of the representative strings guarantees that the order of these blocks is equal to the order of the suffixes. Thus, we just need to explain how to treat periodic blocks (of type 2b).

Periodic blocks. When comparing two periodic blocks which their prefixes of length $3\delta > 2\tau$ are equal, it must be the case that all the corresponding characters in the two blocks match (see Lemma 2.2), and therefore the shorter block is a prefix of the longer block. Of course, if the two blocks have exactly the same length then the corresponding substrings of S are identical, and comparison between the suffixes should be done according to the consecutive blocks.

For the sake of intuition let us consider first the case that all the periodic blocks are maximal. In this case for a periodic block $[p_i..p_{i+1} - 1]$ if $S[p_i..p_{i+1} - 1]$ has period of length ρ, then $S[p_{i+1}] \neq S[p_{i+1} - \rho]$. In such a case, when comparing two suffixes that have the same period, the lexicographic order of the suffixes is defined by the character right after the end of the short block and the corresponding character in the longer block. We want the representative string of each block to be composed of a long enough prefix of the block, say of length 3δ, with some additional characters to deal with the situation of one block that is a prefix of another block. Let us consider two suffixes $p_i, p_j \in P$ that begin with periodic blocks and one is the prefix of the other. Assume without loss of generality that the shorter block appears at position p_i, and that its length is α. In this case, the lexicographic order of the suffixes is determined by $S[p_i + \alpha]$ and the corresponding character in the second suffix, $S[p_j + \alpha]$. The suffix $S[p_i..n]$ should appear before $S[p_j..n]$ in the lexicographic order if and only if $S[p_i + \alpha]$ is larger than $S[p_j + \alpha]$. Both the block of p_i and the block of p_j have the same period, hence, let ρ be the length of

this period. Thus, for any position x in the block, we have $S[x] = S[x-\rho]$ except for possibly the first $\rho \leq \tau$ positions. In particular, $S[p_j+\alpha] = S[p_j+\alpha-\rho]$. On the other hand, since the shorter block is a prefix of the longer block we have $S[p_j+\alpha-\rho] = S[p_i+\alpha-\rho]$ and so $S[p_j+\alpha] = S[p_i+\alpha-\rho]$. Hence, instead of comparing $S[p_i+\alpha]$ with $S[p_j+\alpha]$ it is sufficient to compare $S[p_i+\alpha]$ with $S[p_i+\alpha-\rho]$. Thus, for the periodic block $[p_i, p_{i+1}-1]$ we focus on comparing the character $S[p_i+\alpha]$, which we call *the violation character* with the character $S[p_i+\alpha-\rho]$, which we call *the period character*. We distinguish between two cases: either the violation character is larger than the period character, or vice versa. If the violation character is larger than the period character, then this block should be after longer blocks with the same period while sorting the suffixes. Otherwise, if the violation character is smaller than the period character, then this block should appear before longer blocks while sorting the suffixes. When comparing two blocks that their violation characters are larger than the period characters, the longer blocks should be first, and when comparing two blocks which their violation characters are smaller than the period characters, the longer block should be last. Thus, for periodic blocks of length at least 3δ, instead of just defining the representative string $s_i = S[p_i..p_i+3\delta-1]$ we concatenate to this string two symbols. If the violation character is smaller than the period character, the first symbol added is -1 and the second is the block's length. Otherwise, the first symbol added is $+1$ and the second is the negative value of the block's length ($\cdot(-1)$).

In the previous paragraph, we assumed that the periodic blocks are maximal, and the characters right after the blocks violate the periodicity. However, this is not necessarily the case in a forward-synchronized partitioning set. It is possible that a periodic block will be terminated before the end of the periodic substring. Moreover, in the constructions that we introduce in this paper the periodic blocks are indeed terminated before the right end of the periodic substring. The exact position of the right endpoint could depend on the δ characters after the end of the periodic substring. Now, the character right after the end of the block could be part of the periodic substring. Thus, we define for every periodic block the *right-violation* of the block as the first position after the end of the periodic substring.

DEFINITION 4. *Let $[p_i..p_{i+1}-1]$ be a periodic block such that the principal period of $S[p_i..p_{i+1}-1]$ is ρ, then* $\mathsf{right\text{-}violation}(p_i) = \min\{j \mid j > p_i+\rho, S[j] \neq S[j-\rho]\}$.

We remark that both in the randomized construction of the partitioning set and in the deterministic construction, during the construction the algorithm computes the period length of any periodic block. Therefore in $\mathcal{O}(n)$ time the algorithm computes the $\mathsf{right\text{-}violation}(p_i)$ for any $p_i \in P$ which is an endpoint of a periodic block, without affecting the asymptotic run-time.

Following the discussion above, now the violation character of the periodic block p_i is $S[\mathsf{right\text{-}violation}(p_i)]$ instead of $S[p_{i+1}]$. Moreover, instead of inserting the length of the block p_i to s_i, we insert the distance $\mathsf{right\text{-}violation}(p_i) - p_i$. There is only one case that this construction does not cover. In the case when two blocks have the same period, and the same distance to the end of the period — they will have exactly the same representative string. Notice that if the two blocks have the same length, then it is fine that they have the same representative string since the comparison between the suffixes will be done according to the following blocks. However, it is possible that the blocks have different lengths. Since the right endpoint of the block depends on the δ characters after the periodic substring, the blocks could be of different lengths only if the δ characters after the end of the periodic substrings are different. Thus, to cover also this case, we add to the representative string of the periodic block p_i after the sign ($+1$ or -1) and number also the substring $S[\mathsf{right\text{-}violation}(p_i)..\mathsf{right\text{-}violation}(p_i)+2\delta-1]$.

To conclude for a periodic block $[p_i..p_{i+1}-1]$ we define the representative string s_i as to be the concatenation of the following. The substring (1) $S[p_i..p_i+3\delta-1]$, (2) if the violation character is smaller than the period character, the first character is -1 and the second character is the length of the periodic substring beginning at p_i (that is $\mathsf{right\text{-}violation}(p_i)-p_i$), otherwise, the first character is $+1$ and the second character is the negative value of the length of the periodic substring beginning at p_i (that is $(\mathsf{right\text{-}violation}(p_i)-p_i)\cdot(-1)$). At the end we add (3) $S[\mathsf{right\text{-}violation}(p_i)..\mathsf{right\text{-}violation}(p_i)+2\delta-1]$.

Conclusion. To conclude, for any position $p_i \in P$ we have a representative string of length at most $3\delta+2+2\delta = \mathcal{O}(\delta)$ over an alphabet of size $\mathcal{O}(n)$. The algorithm sorts all these substrings, in $\mathcal{O}(n \cdot \frac{\delta}{\tau})$ time due to Lemma 2.4 or Lemma 2.3 (recall that $\delta = \Theta(\tau)$ for the randomized selection and $\delta = \Theta(\tau \log^* n)$ for the deterministic selection). After the sorting, the algorithm passes over the sorted array, and computes for each s_i its rank, which is the number of distinct s_js smaller than s_i. We define the string S'_P as the string of length $|P|$ where the ith character is the rank of s_i. There is a natural bijective mapping between suffixes of S'_P and the suffixes of S beginning at positions of P. Moreover, in the following lemma we prove that for any

$p_i, p_j \in P$ the suffix $S[p_i..n] < S[p_j..n]$ if and only if $S'_P[i..|P|] < S'_P[j..|P|]$ (the $<$ relation denotes here the lexicographic less than relation).

LEMMA 6.2. *Let P be a forward synchronized (τ, δ)-partitioning set and let $p_i, p_j \in P$ be two distinct positions, then $S[p_i..n] < S[p_j..n]$ if and only if $S'_P[i..|P|] < S'_P[j..|P|]$.*

Proof. We will prove $S[p_i..n] < S[p_j..n] \Rightarrow S'_P[i..|P|] < S'_P[j..|P|]$, the proof is symmetric in the second case.

If the representative strings s_i and s_j are different, then by the discussion above, their relative rank determines the order of the corresponding suffixes, and we have $S'_P[i..|P|] < S'_P[j..|P|]$.

Otherwise, if $s_i = s_j$ we will prove that $S[p_i..p_{i+1} - 1] = S[p_j..p_{j+1} - 1]$. When proven, it follows that $S[p_i..n] < S[p_j..n]$ if and only if $S[p_{i+1}..n] < S[p_{j+1}..n]$. Using an induction argument on the length of the sequence of equal pairs s_{i+k}, s_{j+k}, we have that if $S[p_i..n] < S[p_j..n]$ then $S'_P[i..|P|] < S'_P[j..n]$.

To prove that if $s_i = s_j$ then $S[p_i..p_{i+1} - 1] = S[p_j..p_{j+1} - 1]$, let us distinguish between the two cases of s_i construction:

- If $[p_i..p_{i+1} - 1]$ is a regular block (of type 2a), then $p_{i+1} - p_i \leq \tau \leq \delta$ and $s_i = S[p_i..p_i + 3\delta - 1]$. Since $s_i = s_j$, we have $\mathsf{LCE}(p_i, p_j) \geq 3\delta \geq p_{i+1} - p_i + \delta$. Thus, due to the fact that P is forward synchronized, we have that $p_{i+1} - p_i = p_{j+1} - p_j$. It follows that $S[p_i..p_{i+1} - 1]$ and $S[p_j..p_{j+1} - 1]$ are of the same length, and they are both prefixes of $s_i = s_j$, therefore they must be equal.

- Otherwise, if $[p_i..p_{i+1} - 1]$ is a periocic block (of type 2b), then it must be that $[p_j..p_{j+1} - 1]$ is also a periodic block (of type 2b) since otherwise we can use the proof of the previous case to prove that $[p_i..p_{i+1} - 1]$ is not a periodic block. Since $S[p_i..p_i + 3\delta - 1]$ and $S[p_j..p_j + 3\delta - 1]$ are prefixes of $s_i = s_j$ respectively, then $S[p_i..p_i + 3\delta - 1] = S[p_j..p_j + 3\delta - 1]$. Due to the definition s_i and s_j we have that $s_i[3\delta + 2]$ is $\mathsf{right\text{-}violation}(p_i) - p_i$ in absolute value, and the same goes for p_j. Thus, it follows that $\mathsf{right\text{-}violation}(p_i) - p_i = \mathsf{right\text{-}violation}(p_j) - p_j$. Remark that $S[p_i..\mathsf{right\text{-}violation}(p_i) - 1]$ and $S[p_j..\mathsf{right\text{-}violation}(p_j) - 1]$ are two strings of the same length, and they both share periodic substrings with period lengths at most $\tau \leq \delta$. Hence, due to Lemma 2.2, we have that $S[p_i..\mathsf{right\text{-}violation}(p_i) - 1] = S[p_j..\mathsf{right\text{-}violation}(p_j) - 1]$. In addition, due to the definition of s_i and s_j we have that $S[\mathsf{right\text{-}violation}(p_i)..\mathsf{right\text{-}violation}(p_i) + 2\delta - 1] = S[\mathsf{right\text{-}violation}(p_j)..\mathsf{right\text{-}violation}(p_j) + 2\delta - 1]$. Therefore, it must be that $\mathsf{LCE}(p_i, p_j) \geq 2\delta + \mathsf{right\text{-}violation}(p_i) - p_i > \delta + p_{i+1} - p_i$. Since P is forward synchronized it follows that $p_{i+1} - p_i = p_{j+1} - p_j$ and thus $S[p_i..p_{i+1} - 1] = S[p_j..p_{j+1} - 1]$.

□

Hence, to compute the SSA of S with the suffixes of P, the algorithm computes the suffix array of S'_P using the algorithm of Kärkkäinen and Sanders [28] in $\mathcal{O}(|S'_P|) = \mathcal{O}(\frac{n}{\tau})$ time. To retrieve the SSA of P's suffixes on the string S the algorithm just need to replace every index i in the SA with the index p_i. So, in total, the construction of the SSA of P takes $\mathcal{O}(n \cdot \frac{\delta}{\tau})$ time.

6.2 Construct B's SST Using The SSA of P

Given the SSA of P's suffixes the algorithm builds the SST of B's suffixes, by first computing the SSA of B, then computing the LCP array of these suffixes and finally building the SST based on the SSA and LCP array. To compute the SSA of B, the algorithm uses the same technique we already used for the construction of P's SSA. For any $i \in B$, let $v_i = S[i..i + 3\delta - 1]$ be the string of length 3δ from position i. In addition, similar to the construction of the SSA of P, if the central part $S[i + \delta..i + 2\delta - 1]$ is part of a periodic block we add two characters due to the periodicity violation and the distance between the $\mathsf{right\text{-}violation}$ of the block and i, followed by 3δ characters from the right violation. The algorithm sorts all the v_is for $i \in B$ in $\mathcal{O}(n \cdot \frac{\delta}{\tau})$ time and computes the rank of each v_i, denoted as r_i. In addition, the algorithm computes for each $i \in B$ the index $\mathsf{succ}_P(i + \delta)$ which is the first selected position after index $i + \delta$. Let us denote by x_i the rank of the suffix $S[\mathsf{succ}_P(i + \delta)..n]$ among all the suffixes of P (which can be retrieved from the SSA of P). For each $i \in B$ we associate the pair (r_i, x_i). The SSA of B is obtained by sorting the list of pairs, due to the following lemma. The lemma's proof is omitted since it is almost the same as the proof of Lemma 6.2.

LEMMA 6.3. *Let $i, j \in B$ be two given suffixes, then $S[i..n] < S[j..n]$ if and only if $(r_i, x_i) < (r_j, x_j)$ (where $<$ is the lexicographic smaller than relation).*

The sorting of all the pairs is done in $\mathcal{O}(n)$ time using radix sort if $\tau \leq \sqrt{n}$ or by any standrad efficient sorting in $\mathcal{O}(\frac{n}{\tau} \log \frac{n}{\tau}) = \mathcal{O}(n)$ time if $\tau > \sqrt{n}$. After computing the SSA of B, to compute the LCP array,

the algorithm uses the LCE data structure from Section 3 with the same partitioning set P. Since each LCE query takes $\mathcal{O}(\delta)$ time, and the algorithm computes $|P|-1=\mathcal{O}(\frac{n}{\tau})$ LCE queries, the total time for this phase is $\mathcal{O}(n\cdot\frac{\delta}{\tau})$. With the SSA and the corresponding LCP array, the algorithm computes the SST in $\mathcal{O}(|P|)=\mathcal{O}(\frac{n}{\tau})$ time using the algorithm of Kasai et al. [31].

Concluding, given a set of suffixes B, one can build the SST of B using only $\mathcal{O}(|B|)$ working space, by setting $\tau=\frac{n}{|B|}$. Using the randomized selection, this method yields a randomized Las-Vegas algorithm with $\mathcal{O}(n)$ running time with high probability for $|B|\geq\Omega(\log^2 n)$, and $\mathcal{O}(n)$ expected running time for $|B|<o(\log^2 n)$ completing the proof of Theorem 1.3. Using the deterministic selection this method yields a deterministic algorithm with $\mathcal{O}(n\log\tau)$ running time, for any $|B|\geq\Omega(\log n)$, completing the proof of Theorem 1.4.

7 Even Better Deterministic LCE

In this section we show how to use the deterministic selection technique from Section 5, combining with concepts from the so-called difference covers [37] technique and the SST construction from Section 6 (Theorem 1.4), to improve the LCE query time from $\mathcal{O}(\tau\log^* n)$ to $\mathcal{O}(\tau\sqrt{\log^* n})$ without any asymptotic penalty in the space usage or construction time. At the very high level, we consider a partitioning set for $\tau'=\frac{\tau}{\sqrt{\log^* n}}$. By applying the algorithm of Section 5, one can find a $(\mathcal{O}(\tau'),\mathcal{O}(\tau'\cdot\log^* n))=\left(\mathcal{O}\left(\frac{\tau}{\sqrt{\log^* n}}\right),\mathcal{O}(\tau\cdot\sqrt{\log^* n})\right)$-partitioning set P of size $\mathcal{O}(\frac{n}{\tau'})=\mathcal{O}\left(\frac{n}{\tau}\cdot\sqrt{\log^* n}\right)$. Using this partitioning set, one can build a data structure that answers LCE queries in $\mathcal{O}(\tau\cdot\sqrt{\log^* n})$ time, using the construction of Section 3. However, the size of this data structure will be $\Theta\left(\frac{n}{\tau}\cdot\sqrt{\log^* n}\right)$ which is too large in our settings. Therefore, our goal is to use a special subset of P that has only $\mathcal{O}(1/\sqrt{\log^* n})$ fraction of P's positions, and with these positions, the algorithm still will be able to answer LCE queries fast enough.

Formally, we consider P as the set of positions induced by the $\mu=\log_{3/2}\left(\frac{\tau}{\sqrt{\log^* n}}\right)$ level of the hierarchical decomposition. As in previous sections we postpone the discussion about periodic large blocks to the full version of the paper [8], and focus here on the case where all the blocks in the partitioning set are regular (of type 2a in Definition 1), so their length is at most τ'. In addition, we assume for now that $\tau\geq\sqrt{\log^* n}$ (and therefore $\tau'\geq 1$), and the discussion on the case where $\tau<\sqrt{\log^* n}$ also appears in [8].

Combine SST to simplify the LCE computation. Recall that the computation of $\mathsf{LCE}(i,j)=\ell$ query, in the data structure of Section 3 was composed of three phases. In the first phase the algorithm reads $\mathcal{O}(\delta)$ characters, then the algorithm finds a length k such both $i+k$ and $j+k$ are positions in the partitioning set. The second phase treats a large common substring from $i+k$ and $j+k$ until at most δ characters prior to $i+\ell$ and $j+\ell$ which are the end of the common substring. Notice that, using the SST construction from Section 6 on the positions of the partitioning set, with LCA support, after finding the common k such that $i+k,j+k\in P$ and $S[i..i+k]=S[j..j+k]$, one can compute $\mathsf{LCE}(i,j)$ by adding k to the common extension of $i+k$ and $j+k$. Therefore, one can merge the second and third phases to be done in constant time. In our construction we will use this idea, and focus on finding an offset k such that $S[i..i+k]=S[j..j+k]$ and both $i+k$ and $j+k$ are in the set of positions.

The filtered positions. To reduce the number of positions from the set P the algorithm uses the difference covers technique. The following lemma states the properties that we will use.

LEMMA 7.1. ([37]) *Let m and t be integers such that $t<m$. There exists a set $DC\subseteq[m]$ of size $|DC|=\mathcal{O}\left(\frac{m}{\sqrt{t}}\right)$ such that, for every $i,j\in[m-t]$ there exists $k\leq t$ such that both $i+k\in DC$ and $j+k\in DC$.*

We will use this technique to reduce the size of P, with m as the size $|P|$ and $t=\log^* n$. We begin with the set P, which is a $(\tau',\mathcal{O}(\tau'\log^* n))$-partitioning set of size $\mathcal{O}(\frac{n}{\tau'})$. Let $p_1,p_2\ldots,p_h$ be the positions in P sorted in ascending order. We define the set Q as the set that contains all the position p_i such that $i\mod\sqrt{\log^* n}=0$ or $i\bmod\log^* n<\sqrt{\log^* n}$. It is straightforward that $|Q|=\left(\frac{|P|}{\sqrt{\log^* n}}\right)$. Since $|P|=\mathcal{O}(\frac{n}{\tau}\sqrt{\log^* n})$ we have that $|Q|=\mathcal{O}(\frac{n}{\tau})$. Our data structure will build the SST of the set Q, using the deterministic algorithm from Section 6. Notice that, one can decide whether a position from P is also in Q without actually store P, since the decision depends only on the *rank* of the position in P. Hence, the algorithm computes the set Q with only $\mathcal{O}(\frac{n}{\tau})$ words of working space.

The query computation. To compute $\mathsf{LCE}(i,j)$ query the algorithm simply reads the suffixes $S[i..n]$ and $S[j..n]$ simultaneously where for each offset $k=0,1,\ldots,$ when comparing $S[i+k]$ and $S[j+k]$ the algorithm checks whether both $i+k\in Q$ and $j+k\in Q$. This

process is stopped when one of the following occurs. If a mismatch is reached in offset k ($S[i+k] \neq S[j+k]$), then k is reported as $\mathsf{LCE}(i,j)$. Otherwise, if the algorithm reaches a common offset in Q ($i+k, j+k \in Q$), then $k+\mathsf{LCE}(i+k, j+k)$ is reported as $\mathsf{LCE}(i,j)$. Notice that after spending $\mathcal{O}(\tau)$ time, the algorithm can check in additional constant time whether $i+k \in Q$, or $j+k \in Q$, using the linked list of positions and the auxiliary array, as described for the successor computation. In the following lemma we prove that this process will never take more than $\mathcal{O}(\tau \cdot \sqrt{\log^* n})$ time.

LEMMA 7.2. *There exists a constant c such that for any $i, j \in [n]$ with $\mathsf{LCE}(i,j) = \ell > c\tau\sqrt{\log^* n}$ there exists $k < c\tau\sqrt{\log^* n}$ such that both $i+k \in Q$ and $j+k \in Q$.*

Proof. Let $c' > 1$ be a constant such that P is a $(\tau', c'\tau' \log^* n)$-partitioning set (of size $\mathcal{O}(\frac{n}{\tau'})$), in the terms of Section 3 we have $\delta = c'\tau' \log^* n = c'\tau\sqrt{\log^* n}$. We will prove the lemma holds for $c = 4c'$. Let $i, j \in [n]$ be positions such that $\mathsf{LCE}(i,j) > c\tau\sqrt{\log^* n}$. By Lemma 3.1, we have:

$$\{p - i \,|\, p \in (P \cap [i+\delta..i+\ell-\delta-1])\} = \{p - j \,|\, p \in (P \cap [j+\delta..j+\ell-\delta-1])\}$$

In particular, since $\ell > 4\delta$ we have that:

$$\begin{aligned}\Delta &= \{p - i \,|\, p \in (P \cap [i+\delta..i+3\delta-1])\} \\ &= \{p - j \,|\, p \in (P \cap [j+\delta..j+3\delta-1])\}.\end{aligned}$$

Notice that $|[i+\delta..i+3\delta-1]| = |[j+\delta..j+3\delta-1]| = 2\delta = 2c'\tau' \log^* n$. Hence, since we have assumed that the length of each block induced by P is at most τ', we have that the number of elements (offsets) in Δ is at least $\frac{2\delta}{\tau'} = 2c' \log^* n > 2\log^* n$. For every element $k \in \Delta$, we have $i+k \in P$ and $j+k \in P$. Let $p_\alpha, p_{\alpha+1}, \ldots, p_{\alpha+2\log^* n-1}$ be the first $2\log^* n$ elements in $\{p-i \,|\, p \in (P \cap [i+\delta..i+\ell-\delta-1])\}$. There must be an index $\alpha \leq \alpha' \leq \alpha + \log^* n$ such that $\alpha' \bmod \log^* n = 0$. Therefore, all the positions $p_{\alpha'}, p_{\alpha'+1}, \ldots, p_{\alpha'+\sqrt{\log^* n}-1}$ are positions in Q. Let β' be the index corresponding to α' such that $p_{\alpha'} - i = p_{\beta'} - j$. We consider the positions $p_{\beta'}, p_{\beta'+1}, \ldots, p_{\beta'+\sqrt{\log^* n}-1}$. Since these are $\sqrt{\log^* n}$ consecutive elements in P, it must be that at least one of them is taken into Q (since its index in P is an integer multiple of $\sqrt{\log^* n}$). Since for any $0 \leq x < \sqrt{\log^* n}$ we have $p_{\alpha'+x} - i = p_{\beta'+x} - j$ it must be that there exists at least one x such that both $p_{\alpha'+x} \in Q$ and $p_{\beta'+x} \in Q$ and therefore for $k = p_{\alpha'+x} - i$ we have $i+k \in Q$ and $j+k \in Q$, and $k = p_{\alpha'+x} - i < 3\delta \leq c\tau\sqrt{\log^* n}$. ∎

Complexities. The size of the set Q is $\mathcal{O}(\frac{n}{\tau})$, and using the SST — the data structure answers LCE queries in $\mathcal{O}(\tau\sqrt{\log^* n})$ time. The construction time of the data structure is $\mathcal{O}(n \log \tau)$ for finding P and Q, and for the construction of Q's SST due to the construction of Section 6. Thus, we have proved Theorem 1.2.

8 Ensuring Randomized Selection in Linear Time With High Probability

In Section 4, we introduced a randomized algorithm that selects a small $(2\tau, 2\tau)$-partitioning set in $\mathcal{O}(n)$ expected time. The algorithm uses a fingerprint hash function ϕ and an approximately-min-wise hash function h to compute an ID for each position, and selects positions with minimum values in a range of size τ for the partitioning set. Remark that only the partitioning set's size is in question —the expected size is $\mathcal{O}(\frac{n}{\tau})$ but the probability that the set size is indeed $\mathcal{O}(\frac{n}{\tau})$ is not guaranteed to be high. If we ensure with high probability that a pair of hash functions ϕ and h can be found fast such that $\mathrm{ID}_{\phi,h}$ will induce small partitioning set, then the running time of the algorithm is linear with high probability.

In this section, like in Section 4, we assume that S does not contain long periodic substrings with short periods. With very small modifications, as described in [8], the methods that appear next could work in the general case by "skipping" all long periodic substrings while running the algorithms.

We argue that a good selection of (ϕ, h) pair can be determined in $\mathcal{O}(n)$ time with high probability while not exceeding $\mathcal{O}(\frac{n}{\tau})$ words of working space. More precisely, since ϕ may render "bad" results only when it has collisions on substrings of S, and this event occurs with inverse polynomial probability, we ignore this case (this event is absorbed in the high probability of the algorithm). Thus, we focus on a fast method to pick $h \in \mathcal{F}$ that will make the average amount of partitioning positions per interval of size τ to be constant with high probability.

There are two distinct cases: either $\tau < \log^2 n$ or $\tau \geq \log^2 n$, and the algorithm treats each one of them separately. In the first case since τ is small we can use a sampling of not too many intervals and count the number of selected positions from P in each interval, and use that to approximate the total size of P with high probability. In the second case, we run the algorithm on $\mathcal{O}(\log n)$ different hash functions, which will ensure that with high probability at least one of those hash functions induces a small enough partitioning set. We pick the partitioning set as a subset of a larger $(\log n, \log n)$-partitioning set, instead of picking candidates from every position, to allow the parallel process to run in a total of $\mathcal{O}(n)$ time.

8.1 Case 1: $\tau \leq \log^2 n$ Fix a fingerprint function ϕ with no collisions on S. We will use sampling methods in order to evaluate the amount of partitioning positions that a given approximate min-wise independent hash function h renders when using $\mathrm{ID}_{\phi,h}$ as the ID function.

For any $0 \leq i \leq \frac{n-2}{\tau}$ denote $P_i = P \cap [i\tau+1..(i+1)\tau]$ and denote $C_i = |P_i|$. By using the algorithm described in the proof of Lemma 4.3 on the substring $S[i\tau+1..(i+1)\tau]$, C_i can be calculated in $\mathcal{O}(\tau + |P_i|\tau)$ time using $\mathcal{O}(1)$ words of space (by maintaining only the rightmost partitioning position picked so far and only counting the amount of selected positions). For every $\xi > 0$ we can pick $m = \frac{\xi}{2}\log^5 n$ intervals $[i_k\tau + 1..(i_k+1)\tau]$ for $1 \leq k \leq m$ uniformly and independently of the other chosen intervals. Denote $\overline{C} = \frac{1}{m}\sum_{k=1}^{m} C_{i_k}$.

Observation 1. *With high probability,* $\overline{C} \cdot \frac{n}{\tau} - \frac{n}{\tau} < |P| < \overline{C} \cdot \frac{n}{\tau} + \frac{n}{\tau}$.

Proof. Remark that $\mathbb{E}[C_{i_k}] = |P| \cdot \frac{\tau}{n}$ and therefore also $\mathbb{E}[\overline{C}] = |P| \cdot \frac{\tau}{n}$. By Hoeffding's inequality, since $0 \leq C_{i_k} \leq \tau$ and all C_{i_k} are independent then:

$$\begin{aligned}\Pr\left[|\overline{C} - \mathbb{E}[\overline{C}]| \geq 1\right] &\leq 2\exp\left(-\frac{2m^2}{m\tau^2}\right) \\ &\leq 2\exp\left(-\frac{\xi\log^5 n}{\log^4 n}\right) \\ &= \mathcal{O}\left(\frac{1}{n^{\xi}}\right)\end{aligned}$$

Thus, by sampling m intervals we can calculate $\overline{C}$ and then evaluate with high probability that $\overline{C} \cdot \frac{n}{\tau} - \frac{n}{\tau} < |P| < \overline{C} \cdot \frac{n}{\tau} + \frac{n}{\tau}$. ∎

Observation 2. *There exists a constant* $c > 0$*, such that* $\Pr_{\phi,h}\left(\overline{C} \leq c\right) > \frac{2}{3}$.

Proof. By Lemma 4.2, we know that $\mathbb{E}_{\phi,h}[|P|] = \mathcal{O}(\frac{n}{\tau})$ and thus by Markov's inequality for some $c' > 0$: $\Pr_{\phi,h}\left(|P| \leq \frac{c'n}{\tau}\right) > \frac{2}{3}$. But $|P| \leq \frac{c'n}{\tau} \Rightarrow \overline{C} \leq c' + 1$ with high probability. Therefore, $\Pr_{\phi,h}\left(\overline{C} \leq c'+1\right) > \Pr_{\phi,h}\left(|P| \leq \frac{c'n}{\tau}\right) > \frac{2}{3}$. ∎

Calculating $\overline{C}$ takes at most $\mathcal{O}(m\tau^2) = \mathcal{O}(\log^9 n)$ time since for each sampled interval there could be as much as τ selected partitioning positions, and it may cost up to $\mathcal{O}(\tau)$ time to select each of them in the worst case. With high probability, after at most $\xi\log_3 n$ repetitions of the algorithm for different approximately min-wise hash functions, $\overline{C}$ will be small enough, due to Observation 2. The partitioning set induced by the selected functions must be calculated and verified as small enough. This will happen with high probability as well due to Observation 1. Thus, with high probability after $\mathcal{O}(\mathrm{polylog}(n)+n)$ time a small partitioning set will be found. If the run will fail, the failure will be detected within $\mathcal{O}(n)$ time. Therefore, the total running time is $\mathcal{O}(n)$ time with high probability.

8.2 Case 2: $\tau > \log^2 n$ The algorithm for this case is composed of two steps: In the first step the algorithm finds ϕ, h_0 such that the size of the $(\log n, \log n)$-partitioning set induced by ID_{ϕ,h_0}, denoted P_0, is at most $\mathcal{O}\left(\frac{n}{\log n}\right)$. In the second step the algorithm draws $\log n$ approximately min-wise independent hash functions $h_1, h_2 \ldots h_{\xi\log_3 n} \in \mathcal{F}$ in random, and calculates for each $1 \leq i \leq \log n$ a $(2\tau, 2\tau)$-partitioning set using ID_{ϕ,h_i} that will be a subset of P_0.

For each h_i and in each interval $[j..j+\tau-1]$ the algorithm selects the positions with the smallest ID value from $P_0 \cap [j..j+\tau-1]$. Hence,

$$P_i = \Big\{ k \in P_0 \mid \exists j \in [k-\tau+1..k] : \mathrm{ID}_{\phi,h_i}(k) = \min_{\ell \in [j..j+\tau-1] \cap P_0} \{\mathrm{ID}_{\phi,h_i}(\ell)\} \Big\}$$

We prove that P_i is $(2\tau, 2\tau)$-partitioning set and that the expected size of P_i is $\mathcal{O}(\frac{n}{\tau})$.

Lemma 8.1. *P_i is a $(2\tau, 2\tau)$-partitioning set.*

Proof. We prove that P_i has the properties of Definition 1:

Local Consistency Let $j, k \in [n-2\tau]$ be two indices such that $S[j-2\tau..j+2\tau] = S[k-2\tau..k+2\tau]$. Since $\tau > \log^2 n \geq \log n$ then for all $\ell \in [-\tau..\tau]$ it follows that $S[j+\ell-\log n..j+\ell+\log n] = S[k+\ell-\log n..k+\ell+\log n]$. Remark that since P_0 is a $(\log n, \log n)$-partitioning set then $j+\ell \in P_0 \Leftrightarrow k+\ell \in P_0$. Notice that because $S[j-2\tau..j+2\tau] = S[k-2\tau..k+2\tau]$ then it also holds that for all $\ell \in [-\tau..\tau]$: $\mathrm{ID}_{\phi,h_i}(j+\ell) = \mathrm{ID}_{\phi,h_i}(k+\ell)$. Since $j \in P_i$ if and only if $j \in P_0$ and it has the minimal ID in some interval, and since the same holds for k then from all the above it follows that $j \in P_i \Leftrightarrow k \in P_i$.

Compactness Remark that since $\tau > \log^2 n \geq \log n$ and since P_0 is a $(\log n, \log n)$-partitioning set then in any interval of size τ there is a position from P_0. Therefore, the minimal ID on positions from P_0 within an interval of size τ is well-defined. Hence, the proof of Property 2 from Lemma 4.1 holds the same for P_i. ∎

Lemma 8.2. $\mathbb{E}\left[\,|P_i|\,\right] = \mathcal{O}(\frac{n}{\tau})$

Proof. The proof is based on the proof of Lemma 4.2, with one change: positions in P_i are picked from P_0 which is a $(\log n, \log n)$-partitioning set, and not all the positions in the interval are candidates. Each interval of size $\frac{\tau}{2}$ will contain at least $\frac{\tau}{2\log n}$ candidates to P_i from P_0. As in the proof of Lemma 4.2, this implies that $\Pr(j \in P_i) = \mathcal{O}(\frac{\log n}{\tau})$. Hence, by linearity of expectation over all $\mathcal{O}(\frac{n}{\log n})$ positions in P_0: $\mathbb{E}[|P_i|] = \mathcal{O}(\frac{n}{\tau})$. ∎

From the above lemma and by Markov's inequality it follows that there exists some constant $c' > 0$ such that $\Pr(|P_i| < \frac{c'n}{\tau}) > \frac{2}{3}$. Therefore, with high probability at least one of the approximate min-wise independent hash functions in $\{h_i\}_{i=1}^{\xi \log_3 n}$ renders a partitioning set P_i such that $|P_i| = \mathcal{O}(\frac{n}{\tau})$.

In order to pick a hash function such that $|P_i| < \frac{c'n}{\tau}$ we calculate $|P_i|$ for all h_i, and with high probability the minimal value will be below $\frac{c'n}{\tau}$. A naive approach would be to use a modification of the algorithm from the proof of Lemma 4.3, but on all the functions h_i simultaneously. When moving forward simultaneously, this will cost:

1. $\mathcal{O}(n)$ time in total for calculating positions of P_0 throughout the run.
2. $\mathcal{O}(n)$ time for the fingerprint calculation.
3. $\mathcal{O}(\frac{n}{\log n})$ hash function evaluations on the fingerprints of positions in P_0, for each hash function summing up to $\mathcal{O}(n)$ time, and an extra $\mathcal{O}(\log n)$ words of space for the positions counter and the rightmost picked position of all P_i.
4. $\mathcal{O}(\tau)$ for each step-back of the rolling interval algorithm.

All those operations sum up to $\mathcal{O}(n)$ total time and $\mathcal{O}(\frac{n}{\tau} + \log n)$ words of space, plus the amount of time consumed on step-backs for all h_i. In the algorithm described in the proof of Lemma 4.3, after each selected position a step-back may occur. This sums up to $\mathcal{O}(n \log n)$ time consumed for step-backs in total, and thus bounds the total work time asymptotically.

In [8], we present a method to calculate $|P_i|$ or determining that it is bigger than $\frac{c'n}{\tau}$ for all h_i using $\mathcal{O}(n)$ time and $\mathcal{O}(\frac{n}{\tau} + \log n)$ words of space. The main change is the way we handle step-backs. This improvement uses $\mathcal{O}(\frac{n}{\tau \log n})$ words of space for each hash function h_i to maintain "future" P_i candidates. By doing so, we reduce the total amount of cases where we have to step back and recalculate all the IDs to at most $\mathcal{O}(\log n)$ step-backs per hash function in the worst case. Thus, the amount of time spent on step-backs will sum up to $\mathcal{O}(\tau \log^2 n)$ time, which is $\mathcal{O}(n)$ for $\tau \leq \frac{n}{\log^2 n}$.

Therefore, the following lemmas will apply immediately:

LEMMA 8.3. *There exists an algorithm that for any $i = 1, 2, \ldots, \xi \log_3 n$, computes the sizes $|P_i|$ or determines that $|P_i| > \frac{c'n}{\tau}$. The algorithm takes $\mathcal{O}(n + \tau \log^2 n)$ time using $\mathcal{O}(\frac{n}{\tau} + \log n)$ words of space.*

LEMMA 8.4. *For $\log^2 n < \tau < \mathcal{O}(\frac{n}{\log^2 n})$, there exists an algorithm that computes a $(2\tau, 2\tau)$-partitioning set of size $\mathcal{O}(\frac{n}{\tau})$ in $\mathcal{O}(n)$ time with high probability using $\mathcal{O}(\frac{n}{\tau})$ words of space.*

Proof. After computing the size $|P_i|$ or determining that it is too big for all h_i in linear time, with high probability there exists at least one hash function h_k such that $|P_k| < \frac{c'n}{\tau}$. Then, the algorithm calculates P_k explicitly in $\mathcal{O}(n)$ time. ∎

8.3 Summary We have proved that for all $1 < \tau < \mathcal{O}(\frac{n}{\log^2 n})$ there exists an algorithm that computes a $(2\tau, 2\tau)$-partitioning set of size $\mathcal{O}(\frac{n}{\tau})$ using $\mathcal{O}(\frac{n}{\tau})$ words of working space that runs in linear time with high probability.

9 Overview on Methods for Long Periodic Blocks

In this section we survey the main idea that is used to treat periodic blocks, which are blocks of type 2b due to Definition 1. The complete details appear in [8].

We clarify the difficulty caused by long periodic blocks in the partitioning of S. We will focus now on the LCE computation in such a case; however, the difficulty is similar on the deterministic construction of Section 7. When comparing two suffixes, $S[i..n]$ and $S[j..n]$ that share a large common prefix, of length larger than 2δ, we are guaranteed that their partitioning into blocks is identical except for at most δ positions at the margins of the common substrings. This guarantee is based on the locality property of the partitioning set (Property 1 in Definition 1), and is formulated in Lemma 3.1. When we consider the case of only short blocks, we exploit this property to find selected positions with the same distance from i and j efficiently. Since the length of each block is at most τ in this case, such positions must exist in an offset of at most $\delta + \tau$ positions following i and j. These corresponding positions are very useful in order to skip a large common substring, which is composed of full blocks and should end at most $\delta + \tau$ characters prior to the first mismatch.

However, in the general case with long periodic blocks, it is possible that the first selected positions with

the same offset from i and j will be in $\omega(\delta)$ positions after i and j. In such a case we cannot compare all the characters until the synchronized positions since it will take too much time. Thus, we utilize the fact that long blocks are corresponding to periodic substrings of S. Given two blocks of type 2b that share a common substring of length at least 2τ, we do not need to compare the rest of the characters in order to verify that the corresponding characters match each other. This is true because the common substrings guarantee that both blocks have the same period, and therefore all the corresponding characters match each other. The claim is formalized in Lemma 2.2.

Hence, after the algorithm verifies that two substrings of length $\delta + 2\tau$ are the same, while these substrings are fully contained in long blocks the algorithm skips all the pairs of corresponding characters in the two blocks. If the two blocks end in exactly the same offset from i and j then the beginning of the following blocks are correlated positions which we use for skipping over the large middle part of the common substrings. Otherwise, if the two blocks end in different offsets from i and j, then due to the locality of the partitioning set, it must be that the first mismatch between the suffixes is at most δ positions after the last synchronized offset inside the long blocks. Then, in such a case we can compare at most additional δ characters until we find the first mismatch.

References

[1] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Pattern matching in dynamic texts. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 819–828, 2000.

[2] Amihood Amir and Igor Nor. Real-time indexing over fixed finite alphabets. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 1086–1095, 2008.

[3] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium*, pages 88–94, 2000.

[4] Philip Bille, Johannes Fischer, Inge Li Gørtz, Tsvi Kopelowitz, Benjamin Sach, and Hjalte Wedel Vildhøj. Sparse text indexing in small space. *ACM Transactions on Algorithms*, 12(3):39:1–39:19, 2016.

[5] Philip Bille, Inge Li Gørtz, Mathias Bæk Tejs Knudsen, Moshe Lewenstein, and Hjalte Wedel Vildhøj. Longest common extensions in sublinear space. In *Combinatorial Pattern Matching - 26th Annual Symposium, CPM*, pages 65–76, 2015.

[6] Philip Bille, Inge Li Gørtz, Benjamin Sach, and Hjalte Wedel Vildhøj. Time-space trade-offs for longest common extensions. *Journal of Discrete Algorithms*, 25:42–50, 2014.

[7] Timo Bingmann, Johannes Fischer, and Vitaly Osipov. Inducing suffix and lcp arrays in external memory. In *Proceedings of the 15th Meeting on Algorithm Engineering and Experiments, ALENEX*, pages 88–102, 2013.

[8] Or Birenzwige, Shay Golan, and Ely Porat. Locally consistent parsing for text indexing in small space. *CoRR*, abs/1812.00359, 2018.

[9] Dany Breslauer and Zvi Galil. Real-time streaming string-matching. *ACM Transactions on Algorithms*, 10(4):22:1–22:12, 2014.

[10] Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking. In *Combinatorial Pattern Matching, 14th Annual Symposium, CPM*, pages 55–69, 2003.

[11] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986.

[12] Graham Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Transactions on Algorithms*, 3(1):2:1–2:19, 2007.

[13] Rene De La Briandais. File searching using variable length keys. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, pages 295–298. ACM, 1959.

[14] Martin Dietzfelbinger, Joseph Gil, Yossi Matias, and Nicholas Pippenger. Polynomial hash functions are reliable (extended abstract). In *Proceedings of Automata, Languages and Programming, 19th International Colloquium, ICALP92*, pages 235–246, 1992.

[15] Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS*, pages 137–143, 1997.

[16] Paolo Ferragina and Roberto Grossi. Improved dynamic text indexing. *Journal of Algorithms*, 31(2):291–319, 1999.

[17] Johannes Fischer and Volker Heun. Range median of minima queries, super-cartesian trees, and text indexing. In *Proceedings of the 19th International Workshop on Combinatorial Algorithms, IWOCA*, pages 239–252, 2008.

[18] Johannes Fischer, Tomohiro I, and Dominik Köppl. Deterministic sparse suffix sorting on rewritable texts. In *LATIN 2016: Theoretical Informatics - 12th Latin American Symposium*, pages 483–496, 2016.

[19] Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. Faster entropy-bounded compressed suffix trees. *Theor. Comput. Sci.*, 410(51):5354–5364, 2009.

[20] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[21] Zvi Galil and Raffaele Giancarlo. Data structures and algorithms for approximate string matching. *Journal of Complexity*, 4(1):33–72, 1988.

[22] Pawel Gawrychowski and Tomasz Kociumaka. Sparse suffix tree construction in optimal time and space. In

Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, pages 425–439, 2017.

[23] Andrew V. Goldberg, Serge A. Plotkin, and Gregory E. Shannon. Parallel symmetry-breaking in sparse graphs. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 315–324, 1987.

[24] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[25] Tomohiro I. Longest common extensions with recompression. In *28th Annual Symposium on Combinatorial Pattern Matching, CPM*, pages 18:1–18:15, 2017.

[26] Tomohiro I, Juha Kärkkäinen, and Dominik Kempa. Faster sparse suffix sorting. In *31st International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 386–396, 2014.

[27] Piotr Indyk. A small approximately min-wise independent family of hash functions. *Journal of Algorithms*, 38(1):84–90, 2001.

[28] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer, 2003.

[29] Juha Kärkkäinen and Esko Ukkonen. Sparse suffix trees. In *Computing and Combinatorics, Second Annual International Conference, COCOON*, pages 219–230, 1996.

[30] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[31] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial Pattern Matching, 12th Annual Symposium, CPM*, pages 181–192, 2001.

[32] Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC.*, pages 756–767, 2019.

[33] Tomasz Kociumaka. *Efficient Data Structuresfor Internal Queries in Texts.* PhD thesis, University of Warsaw, 10 2018.

[34] Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal pattern matching queries in a text and applications. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 532–551, 2015.

[35] Dmitry Kosolobov. Tight lower bounds for the longest common extension problem. *Inf. Process. Lett.*, 125:26–29, 2017.

[36] Gad M. Landau and Uzi Vishkin. Fast string matching with k differences. *Journal of Computer and System Sciences*, 37(1):63–78, 1988.

[37] Mamoru Maekawa. A square root N algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, 1985.

[38] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[39] Kurt Mehlhorn, Rajamani Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.

[40] Donald R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[41] Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In *Proceedings of 50th Annual IEEE Symposium on Foundations of Computer Science, FOCS*, pages 315–323, 2009.

[42] Simon J. Puglisi and Andrew Turpin. Space-time tradeoffs for longest-common-prefix array computation. In *Algorithms and Computation, 19th International Symposium, ISAAC 2008, Gold Coast, Australia, December 15-17, 2008. Proceedings*, pages 124–135, 2008.

[43] Michael Roberts, Wayne Hayes, Brian R. Hunt, Stephen M. Mount, and James A. Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004.

[44] Süleyman Cenk Sahinalp and Uzi Vishkin. Symmetry breaking for suffix tree construction. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, pages 300–309, 1994.

[45] Süleyman Cenk Sahinalp and Uzi Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm (extended abstract). In *37th Annual Symposium on Foundations of Computer Science, FOCS*, pages 320–328, 1996.

[46] Saul Schleimer, Daniel Shawcross Wilkerson, and Alexander Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 76–85, 2003.

[47] Yuka Tanimura, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, Simon J. Puglisi, and Masayuki Takeda. Deterministic sub-linear space LCE data structures with efficient construction. In *27th Annual Symposium on Combinatorial Pattern Matching, CPM*, pages 1:1–1:10, 2016.

[48] Yuka Tanimura, Takaaki Nishimoto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Small-space encoding LCE data structure with constant-time queries. *CoRR*, abs/1702.07458, 2017.

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

[50] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory*, pages 1–11. IEEE Computer Society, 1973.