Construction of Sparse Suffix Trees and LCE Indexes in Optimal Time and Space

Dmitry Kosolobov
Ural Federal University, Ekaterinburg, Russia

Nikita Sivukhin
Ural Federal University, Ekaterinburg, Russia

Abstract

The notions of synchronizing and partitioning sets are recently introduced variants of locally consistent parsings with a great potential in problem-solving. In this paper we propose a deterministic algorithm that constructs for a given readonly string of length n over the alphabet $\{0, 1, \ldots, n^{\mathcal{O}(1)}\}$ a variant of a τ-partitioning set with size $\mathcal{O}(b)$ and $\tau = \frac{n}{b}$ using $\mathcal{O}(b)$ space and $\mathcal{O}(\frac{1}{\epsilon}n)$ time provided $b \ge n^{\epsilon}$, for $\epsilon > 0$. As a corollary, for $b \ge n^{\epsilon}$ and constant $\epsilon > 0$, we obtain linear time construction algorithms with $\mathcal{O}(b)$ space on top of the string for two major small-space indexes: a sparse suffix tree, which is a compacted trie built on b chosen suffixes of the string, and a *longest common extension* (LCE) index, which occupies $\mathcal{O}(b)$ space and allows us to compute the longest common prefix for any pair of substrings in $\mathcal{O}(n/b)$ time. For both, the $\mathcal{O}(b)$ construction storage is asymptotically optimal since the tree itself takes $\mathcal{O}(b)$ space and any LCE index with $\mathcal{O}(n/b)$ query time must occupy at least $\mathcal{O}(b)$ space by a known trade-off (at least for $b \ge \Omega(n/\log n)$). In case of arbitrary $b \ge \Omega(\log^2 n)$, we present construction algorithms for the partitioning set, sparse suffix tree, and LCE index with $\mathcal{O}(n \log_b n)$ running time and $\mathcal{O}(b)$ space, thus also improving the state of the art.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases (τ, δ)-partitioning set, longest common extension, sparse suffix tree

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.20

Related Version *Full Version*: https://arxiv.org/abs/2105.03782

Funding Supported by the Russian Science Foundation (RSF), project 20-71-00050.

1 Introduction

Indexing data structures traditionally play a central role in algorithms on strings and in information retrieval. Due to constantly growing volumes of data in applications, the attention of researchers in the last decades was naturally attracted to small-space indexes. In this paper we study two closely related small-space indexing data structures: a sparse suffix tree and a longest common extension (LCE) index. We investigate them in the general framework of (deterministic) locally consistent parsings that was developed by Cole and Vishkin [7], Jeż [21, 20, 22, 19], and others [1, 11, 12, 13, 15, 28, 29, 32] (the list is not exhausting) and was recently revitalized in the works of Birenzwige et al. [5] and Kempa and Kociumaka [25] where two new potent concepts of partitioning and synchronizing sets were introduced.

The sparse suffix tree (*SST*) for a given set of b suffixes of a string is a compacted trie built on these suffixes. It can be viewed as the suffix tree from which all suffixes not from the set were removed (details follow). The tree takes $\mathcal{O}(b)$ space on top of the input string and can be easily constructed in $\mathcal{O}(n)$ time from the suffix tree, where n is the length of the string. One can build the suffix tree in $\mathcal{O}(n)$ time [10] provided the letters of the string are sortable in linear time. However, if at most $\mathcal{O}(b)$ space is available on top of the input, then in general there is not enough memory for the full suffix tree and the problem, thus, becomes much more difficult. The $\mathcal{O}(b)$ bound is optimal since the tree itself takes $\mathcal{O}(b)$

35th Annual Symposium on Combinatorial Pattern Matching (CPM 2024).
Editors: Shunsuke Inenaga and Simon J. Puglisi; Article No. 20; pp. 20:1–20:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

space. The construction problem with restricted $\mathcal{O}(b)$ space naturally arises in applications of the sparse suffix tree and the sparse suffix array (which is easy to retrieve from the tree) where we have to index data in the setting of scarce memory. As is common in algorithms on strings, it is assumed that the input string is readonly, its letters are polynomially bounded integers $\{0, 1, \ldots, n^{\mathcal{O}(1)}\}$, and the space is at least polylogarithmic, i.e., $b \geq \log^{\Omega(1)} n$. We note, however, that in supposed usages the memory restrictions can often be relaxed even more to $b \geq n^{\epsilon}$, for constant $\epsilon > 0$.

The $\mathcal{O}(b)$-space construction problem was posed by Kärkkäinen and Ukkonen [24] who showed how to solve it in linear time for the case of evenly spaced b suffixes. In a series of works [2, 5, 11, 14, 18, 23], the problem was settled for the case of randomized algorithms: an optimal linear $\mathcal{O}(b)$-space Monte Carlo construction algorithm for the sparse suffix tree was proposed by Gawrychowski and Kociumaka [14] and an optimal linear $\mathcal{O}(b)$-space Las-Vegas algorithm was described by Birenzwige et al. [5]. The latter authors also presented the best up-to-date deterministic solution that builds the sparse suffix tree within $\mathcal{O}(b)$ space in $\mathcal{O}(n \log \frac{n}{b})$ time [5] (log is in base 2 unless explicitly stated otherwise). All these solutions assume (as we do too) that the input string is readonly and its alphabet is $\{0, 1, \ldots, n^{\mathcal{O}(1)}\}$; the case of rewritable inputs is apparently very different, as was shown by Prezza [30].

The LCE index, crucial in string algorithm applications, preprocesses a readonly input string of length n so that one can answer queries $\mathsf{lce}(p, q)$, for any positions p and q, computing the length of the longest common prefix of the suffixes starting at p and q. The now classical result of Harel and Tarjan states that the LCE queries can be answered in $\mathcal{O}(1)$ time provided $\mathcal{O}(n)$ space is used [16]. In [3] Bille et al. presented an LCE index that, for any given user-defined parameter b, occupies $\mathcal{O}(b)$ space on top of the input string and answers queries in $\mathcal{O}(\frac{n}{b})$ time. In [26] it was proved that this time-space trade-off is optimal provided $b \geq \Omega(n/\log n)$ (it is conjectured that the same trade-off lower bound holds for a much broader range of values b; a weaker trade-off appears in [4, 6]). In view of these lower bounds, it is therefore natural to ask how fast one can construct, for any parameter b, an LCE index that can answer queries in $\mathcal{O}(\frac{n}{b})$ time using $\mathcal{O}(b)$ space on top of the input. The space $\mathcal{O}(b)$ is optimal for this query time and the construction algorithm should not exceed it. The issue with the data structure of [3] is that its construction time is unacceptably slow, which motivated a series of works trying to solve this problem. As in the case of sparse suffix trees, the problem was completely settled in the randomized setting: an optimal linear $\mathcal{O}(b)$-space Monte Carlo construction algorithm for an LCE index with $\mathcal{O}(\frac{n}{b})$-time queries was presented by Gawrychowski and Kociumaka [14] and a Las-Vegas construction with the same time and space was proposed by Birenzwige et al. [5] provided $b \geq \Omega(\log^2 n)$. The best deterministic solution is also presented in [5] and runs in $\mathcal{O}(n \log \frac{n}{b})$ time answering queries in slightly worse time $\mathcal{O}(\frac{n}{b}\sqrt{\log^* n})$ provided $b \geq \Omega(\log n)$ (the previous best solution was from [34] and it runs in $\mathcal{O}(n \cdot \frac{n}{b})$ time but, for some exotic parameters b, has slightly better query time). The input string is readonly in all these solutions and the alphabet is $\{0, 1, \ldots, n^{\mathcal{O}(1)}\}$.

For a broad range of values b, we settle both construction problems, for sparse suffix trees and LCE indexes, in $\mathcal{O}(b)$ space in the deterministic case. Specifically, given a readonly string of length n over the alphabet $\{0, 1, \ldots, n^{\mathcal{O}(1)}\}$, we present two algorithms: one that constructs the sparse suffix tree, for any user-defined set of b suffixes such that $b \geq \Omega(\log^2 n)$, in $\mathcal{O}(n \log_b n)$ time using $\mathcal{O}(b)$ space on top of the input; and another that constructs an LCE index with $\mathcal{O}(\frac{n}{b})$-time queries, for any parameter b such that $b \geq \Omega(\log^2 n)$, in $\mathcal{O}(n \log_b n)$ time using $\mathcal{O}(b)$ space on top of the input. This gives us optimal $\mathcal{O}(b)$-space solutions with $\mathcal{O}(\frac{1}{\epsilon} n) = \mathcal{O}(n)$ time when $b \geq n^{\epsilon}$, for constant $\epsilon > 0$, which arguably includes most interesting cases. As can be seen in Table 1, our result beats the previous best solution in virtually all settings since $n \log_b n = o(n \log \frac{n}{b})$, for $b = o(n)$.

Table 1 LCE indexes deterministically constructible in $\mathcal{O}(b)$ space on a readonly input, for $b \ge \Omega(\log^2 n)$.

Algorithm	Tanimura et al. [34]	Birenzwige et al. [5]	Theorem 4
Query time	$\mathcal{O}(\frac{n}{b}\log\min\{b, \frac{n}{b}\})$	$\mathcal{O}(\frac{n}{b}\sqrt{\log^* n})$	$\boldsymbol{\mathcal{O}(\frac{n}{b})}$
Construction in $\mathcal{O}(b)$ space	$\mathcal{O}(n \cdot \frac{n}{b})$	$\mathcal{O}(n\log\frac{n}{b})$	$\boldsymbol{\mathcal{O}(n\log_b n)}$

In order to achieve these results, we develop a new algorithm that, for any given parameter $b \ge \Omega(\log^2 n)$, constructs a so-called τ-partitioning set of size $\mathcal{O}(b)$ with $\tau = \frac{n}{b}$. This result is of independent interest.

We note that there is another natural model where the input string is packed in memory in such a way that one can read in $\mathcal{O}(1)$ time any $\Theta(\log_\sigma n)$ consecutive letters of the input packed into one $\Theta(\log n)$-bit machine word, where $\{0, 1, \dots, \sigma{-}1\}$ is the input alphabet. In this case the $\mathcal{O}(n)$ construction time is not necessarily optimal for the sparse suffix tree and the LCE index and one might expect to have $\mathcal{O}(n/\log_\sigma n)$ time. As was shown by Kempa and Kociumaka [25], this is indeed possible for LCE indexes in $\mathcal{O}(n/\log_\sigma n)$ space. It remains open whether one can improve our results for the $\mathcal{O}(b)$-space construction in this setting; note that the lower bound of [26] does not apply here due to its assumption of single-letter input memory cells.

Techniques. The core of our solution is a version of locally consistent parsing developed by Birenzwige et al. [5], the so-called τ-partitioning sets (unfortunately, we could not adapt the more neat τ-synchronizing sets from [25] for the deterministic case). It was shown by Birenzwige et al. that the $\mathcal{O}(b)$-space construction of a sparse suffix tree or an LCE index can be performed in linear time provided a τ-partitioning set of size $\mathcal{O}(b)$ with $\tau = \frac{n}{b}$ is given. We define a variant of τ-partitioning sets and, for completeness, repeat the argument of Birenzwige et al. with minor adaptations to our case. The main bulk of the text is devoted to the description of an $\mathcal{O}(b)$-space algorithm that builds a (variant of) τ-partitioning set of size $\mathcal{O}(b)$ with $\tau = \frac{n}{b}$ in $\mathcal{O}(n\log_b n)$ time provided $b \ge \Omega(\log^2 n)$, which is the main result of the present paper. In comparison Birenzwige et al.'s algorithm for their τ-partitioning sets runs in $\mathcal{O}(n)$ *expected* time (so that it is a Las Vegas construction) and $\mathcal{O}(b)$ space; their deterministic algorithm takes $\mathcal{O}(n\log\tau)$ time but the resulting set is only $\tau\log^* n$-partitioning. Concepts very similar to partitioning sets appeared also in [31, 33].

Our solution combines two well-known approaches to deterministic locally consistent parsings: the *deterministic coin tossing* introduced by Cole and Vishkin [7] and developed in [1, 11, 12, 13, 15, 28, 29, 32], and the *recompression* invented by Jeż [19] and studied in [17, 21, 20, 22]. The high level idea is first to use Cole and Vishkin's technique that constructs a τ-partitioning set of size $\mathcal{O}(b\log^* n)$ where $\tau = \frac{n}{b}$ (in our algorithm the size is actually $\mathcal{O}(b\log\log\log n)$ since we use a "truncated" version of Cole and Vishkin's bit reductions); second, instead of storing the set explicitly, which is impossible in $\mathcal{O}(b)$ space, we construct a string R of length $\mathcal{O}(b\log^* n)$ in which every letter corresponds to a position of the set and occupies $o(\log\log n)$ bits so that R takes $o(b\log^* n\log\log n)$ bits in total and, thus, can be stored into $\mathcal{O}(b)$ machine words of size $\mathcal{O}(\log n)$ bits; third, Jeż's recompression technique is iteratively applied to the string R until R is shortened to length $\mathcal{O}(b)$; finally, the first technique generating a τ-partitioning set is performed again but this time we retain and store explicitly those positions that correspond to surviving letters of the string R. There are many hidden obstacles on this path and because of them our solution is only of purely theoretical value in its present form due to numerous internal complications in the actual scheme (in contrast, randomized results on synchronizing sets [9, 25] seem quite practical).

The paper is organized as follows. In Section 2 we define τ-partitioning sets and show how one can use them to build an LCE index. Section 3 describes the first stage of the construction of a τ-partitioning set that is based on a modification of Cole and Vishkin's technique. Section 4 improves the running time of this stage from $\mathcal{O}(n \log \tau)$ to $\mathcal{O}(n \log_b \tau)$. In Section 5 the second stage based on a modification of Jeż's recompression technique is presented. Appendix C in the full version [27] describes separately the case of very small τ.

2 Partitioning Sets with Applications

Let us fix a readonly string s of length n whose letters $s[0], s[1], \ldots, s[n{-}1]$ are from a polynomially bounded alphabet $\{0, 1, \ldots, n^{\mathcal{O}(1)}\}$. We use s as the input in our algorithms. As is standard, the algorithms are in the word-RAM model, their space is measured in $\Theta(\log n)$-bit machine words, and each $s[i]$ occupies a separate word. We write $s[i..j]$ for the *substring* $s[i]s[i{+}1]\cdots s[j]$, assuming it is empty if $i > j$; $s[i..j]$ is called a *suffix* (resp., *prefix*) of s if $j = n-1$ (resp., $i = 0$). For any string t, let $|t|$ denote its length. We say that t *occurs* at position i in s if $s[i..i{+}|t|{-}1] = t$. Denote $[i..j] = \{k \in \mathbb{Z}\colon i \le k \le j\}$, $(i..j] = [i..j] \setminus \{i\}$, $[i..j) = [i..j] \setminus \{j\}$, $(i..j) = [i..j) \cap (i..j]$. A number $p \in [1..|t|]$ is called a *period* of t if $t[i] = t[i-p]$ for each $i \in [p..|t|)$. For brevity, denote $\log\log\log n$ by $\log^{(3)} n$. We assume that n, the length of s, is sufficiently large: larger than $2^{\max\{16,c\}}$, where c is a constant such that n^c upper-bounds the alphabet.

Given an integer $\tau \in [4..n/2]$, a set of positions $S \subseteq [0..n)$ is called a *τ-partitioning set* if it satisfies the following properties:

(a) if $s[i{-}\tau..i{+}\tau] = s[j{-}\tau..j{+}\tau]$ for $i, j \in [\tau..n{-}\tau)$, then $i \in S$ iff $j \in S$;

(b) if $s[i..i{+}\ell] = s[j..j{+}\ell]$, for $i, j \in S$ and some $\ell \ge 0$, then, for each $d \in [0..\ell{-}\tau)$, $i + d \in S$ iff $j + d \in S$;

(c) if $i, j \in S \cup \{0, n{-}1\}$ with $j{-}i > \tau$ and $(i..j) \cap S = \emptyset$, then the period of $s[i..j]$ is at most $\tau/4$.

Our definition is inspired by the *forward synchronized* (τ, τ)*-partitioning sets* from [5, Def. 3.1 and 6.1] but slightly differs; nevertheless, we retain the term "partitioning" to avoid inventing unnecessary new terms for very close concepts. In the definition, (a), (b), and (c) state, respectively, that S is locally consistent, forward synchronized, and dense: the choice of positions depends only on short substrings around them, long enough equal right "contexts" of positions from S are "partitioned" identically, and S has a position every τ letters unless a long range with small period is encountered. In our construction of S a certain converse of (c) will also hold: whenever a substring $s[i..j]$ has a period at most $\tau/4$, we will have $S \cap [i + \tau..j - \tau] = \emptyset$ (see Lemma 17). This converse is not in the definition since it is unnecessary for our applications and we will use auxiliary τ-partitioning sets not satisfying it. The definition also implies the following convenient property of "monotonicity".

▶ **Lemma 1.** *For any $\tau' \ge \tau$, every τ-partitioning set is also τ'-partitioning.*

Due to (c), all τ-partitioning sets in some strings have size at least $\Omega(n/\tau)$. In the remaining sections we devise algorithms that construct a τ-partitioning set of s with size $\mathcal{O}(n/\tau)$ (matching the lower bound) using $\mathcal{O}(n/\tau)$ space on top of s; for technical reasons, we assume that $\Omega(\log^2 n)$ space is always available, i.e., $n/\tau \ge \Omega(\log^2 n)$, which is a rather mild restriction. Thus, we shall prove the following main theorem.

▶ **Theorem 2.** *For any string of length n over an alphabet $[0..n^{\mathcal{O}(1)}]$ and any $\tau \in [4..\mathcal{O}(n/\log^2 n)]$, one can construct in $\mathcal{O}(n \log_b n)$ time and $\mathcal{O}(b)$ space on top of the string a τ-partitioning set of size $\mathcal{O}(b)$, for $b = n/\tau$.*

Let us sketch how one can construct indexes with the τ-partitioning set of Theorem 2.

LCE index and sparse suffix tree. An LCE index is a data structure on s that, given a pair of positions p and q, answers the *LCE query* $\mathsf{lce}(p, q)$ computing the length of the longest common prefix of $s[p..n{-}1]$ and $s[q..n{-}1]$. Such index can be stored in $\mathcal{O}(b)$ space on top of s with $\mathcal{O}(\frac{n}{b})$ query time [3] and this trade-off is optimal, at least for $b \ge \Omega(\frac{n}{\log n})$ [26].

Given b suffixes $s[i_1..n{-}1], s[i_2..n{-}1], \ldots, s[i_b..n{-}1]$, their *sparse suffix tree* [24] is a compacted trie on these suffixes in which all edge labels are stored as pointers to corresponding substrings of s. Thus, the tree occupies $\mathcal{O}(b)$ space.

Our construction scheme for these two indexes is roughly as follows: given a τ-partitioning set S with $\tau = \frac{n}{b}$ and size $\mathcal{O}(b) = \mathcal{O}(n/\tau)$, we first build the sparse suffix tree T for the suffixes $s[j..n{-}1]$ with $j \in S$, then we use it to construct an LCE index, and, using the index, build the sparse suffix tree for arbitrarily chosen b suffixes. We elaborate on this scheme below; our exposition, however, is rather sketchy and some details are omitted since the scheme is essentially the same as in [5] and is given here mostly for completeness.

To construct the sparse suffix tree T for all $s[j..n{-}1]$ with $j \in S$, we apply the following lemma. Its cumbersome formulation is motivated by its subsequent use in Section 4. In the special case when $m = n$ and $\sigma = n^{\mathcal{O}(1)}$, which is of primary interest for us now, the lemma states that T can be built in $\mathcal{O}(n)$ time: this case implies that $m \log_b \sigma = \mathcal{O}(n \log_b n)$ is $\mathcal{O}(n)$ if $b > n/\log n$, and $b \log b$ is $\mathcal{O}(n)$ if $b \le n/\log n$, and, therefore, $\min\{m \log_b \sigma, b \log b\} = \mathcal{O}(n)$. The proof essentially follows arguments of [5] and is given in Appendix A in the full version [27].

▶ **Lemma 3.** *Given an integer $\tau \ge 4$ and a read-only string s of length m over an alphabet $[0..\sigma)$, let S be an "almost" τ-partitioning set of size $b = \Theta(m/\tau)$: it satisfies properties (a) and (b), but not necessarily (c). The sparse suffix tree T for all suffixes $s[j..m{-}1]$ with $j \in S$ can be built in $\mathcal{O}(m + \min\{m \log_b \sigma, b \log b\})$ time and $\mathcal{O}(m/\tau)$ space on top of the space required for s.*

For our LCE index, we equip T with the lowest common ancestor (LCA) data structure [16], which allows us to compute $\mathsf{lce}(p, q)$ in $\mathcal{O}(1)$ time for $p, q \in S$, and we preprocess an array $N[0..b{-}1]$ such that $N[i] = \min\{j \ge i\tau \colon j \in S\}$ for $i \in [0..b)$, which allows us to calculate $\min\{j \ge p \colon j \in S\}$, for any p, in $\mathcal{O}(\tau)$ time by traversing $j_k, j_{k+1}, \ldots$ in S, for $j_k = N[\lfloor p/\tau \rfloor]$. In order to answer an arbitrary query $\mathsf{lce}(p, q)$, we first calculate $p' = \min\{j \ge p + \tau \colon j \in S\}$ and $q' = \min\{j \ge q + \tau \colon j \in S\}$ in $\mathcal{O}(\tau)$ time. If either $p' - p \le 2\tau$ or $q' - q \le 2\tau$, then by the local consistency of S, $s[p..n{-}1]$ and $s[q..n{-}1]$ either differ in their first 3τ positions, which is checked naïvely, or $s[p..p'] = s[q..q']$ and the answer is given by $p' - p + \mathsf{lce}(p', q')$ using T. If $\min\{p' - p, q' - q\} > 2\tau$, then the strings $s[p{+}\tau..p']$ and $s[q{+}\tau..q']$ both have periods at most $\tau/4$ due to property (c); we compare $s[p..p{+}2\tau]$ and $s[q..q{+}2\tau]$ naïvely and, if there are no mismatches, therefore, due to periodicity, $s[p{+}\tau..p']$ and $s[q{+}\tau..q']$ have a common prefix of length $\ell = \min\{p' - p, q' - q\} - \tau$; hence, the problem is reduced to $\mathsf{lce}(p + \ell, q + \ell)$, which can be solved as described above since either $p' - (p + \ell) \le 2\tau$ or $q' - (q + \ell) \le 2\tau$. We thus have proved the following theorem.

▶ **Theorem 4.** *For any string of length n over an alphabet $[0..n^{\mathcal{O}(1)}]$ and any $b \ge \Omega(\log^2 n)$, one can construct in $\mathcal{O}(n \log_b n)$ time and $\mathcal{O}(b)$ space on top of the string an LCE index that can answer LCE queries in $\mathcal{O}(n/b)$ time.*

Let us consider the construction of the SST for b suffixes $s[i_1..n{-}1]$, $s[i_2..n{-}1], \ldots, s[i_b..n{-}1]$. Denote by j_k the kth position in a given τ-partitioning set S of size $\mathcal{O}(b)$ with $\tau = \frac{n}{b}$ (so that $j_1 < \cdots < j_{|S|}$). For each suffix $s[i_\ell..n{-}1]$, we compute in $\mathcal{O}(\tau)$ time using the array N an index k_ℓ such that $j_{k_\ell} = \min\{j \ge i_\ell + \tau : j \in S\}$. It takes $\mathcal{O}(b\tau) = \mathcal{O}(n)$ time in total. Then, we sort all strings $s[i_\ell..i_\ell{+}4\tau]$ in $\mathcal{O}(n)$ time as in the proof of Lemma 3 and assign to them ranks r_ℓ (equal strings are of equal ranks). For each $k \in [1..|S|]$, we obtain from the tree T the rank $\bar{r}_k$ of $s[j_k..n{-}1]$ among the suffixes $s[j..n{-}1]$ with $j \in S$. Suppose that $j_{k_\ell} \le i_\ell + 3\tau$, for all $\ell \in [1..b]$. By property (a), the equality $r_\ell = r_{\ell'}$, for any $\ell, \ell' \in [1..b]$, implies that $j_{k_\ell} - i_\ell = j_{k_{\ell'}} - i_{\ell'}$ when $j_{k_\ell} - i_\ell \le 3\tau$. Then, we sort the suffixes $s[i_\ell..n{-}1]$ with $\ell \in [1..b]$ in $\mathcal{O}(b)$ time using the radix sort on the corresponding pairs $(r_\ell, \bar{r}_{j_{k_\ell}})$. The SST can be assembled from the sorted suffixes in $\mathcal{O}(b\tau) = \mathcal{O}(n)$ time using the LCE index to calculate longest common prefixes of adjacent suffixes.

The argument is more intricate when the condition $j_{k_\ell} > i_\ell + 3\tau$ does not hold. Suppose that $j_{k_\ell} > i_\ell + 3\tau$, for some $\ell \in [1..b]$. Then, by property (c), the minimal period of $s[i_\ell{+}\tau..j_{k_\ell}]$ is at most $\tau/4$. Denote this period by p_ℓ. We compute p_ℓ in $\mathcal{O}(\tau)$ time using a linear $\mathcal{O}(1)$-space algorithm [8] and, then, we find the leftmost position $t_\ell > j_{k_\ell}$ breaking this period: $s[t_\ell] \neq s[t_\ell{-}p_\ell]$. As $j_{k_\ell}{-}p_\ell > i_\ell{+}2\tau > j_{k_{\ell-1}}$, we obtain $s[j_{k_\ell}{-}\tau..j_{k_\ell}{+}\tau] \neq s[j_{k_\ell}{-}p_\ell{-}\tau..j_{k_\ell}{-}p_\ell{+}\tau]$ (since otherwise $j_{k_\ell}{-}p_\ell \in S$ by property (a)) and, hence, $t_\ell \in (j_{k_\ell}..j_{k_\ell}{+}\tau]$. Therefore, the computation of t_ℓ takes $\mathcal{O}(\tau)$ time. Thus, all p_ℓ and t_ℓ can be calculated in $\mathcal{O}(b\tau) = \mathcal{O}(n)$ total time. We then sort the strings $s[t_\ell..t_\ell{+}\tau]$ in $\mathcal{O}(n)$ time and assign to them ranks $\tilde{r}_\ell$. For each suffix $s[i_\ell..n{-}1]$ with $\ell \in [1..b]$, we associate the tuple $(r_\ell, 0, 0, \bar{r}_{j_{k_\ell}})$ if $j_{k_\ell} \le i_\ell + 3\tau$, and the tuple $(r_\ell, d_\ell, \tilde{r}_\ell, \bar{r}_{j_{k_\ell}})$ if $j_{k_\ell} > i_\ell{+}3\tau$, where $d_\ell = \pm(t_\ell{-}i_\ell - n)$ with plus if $s[t_\ell] < s[t_\ell{-}p_\ell]$ and minus otherwise. We claim that the order of the suffixes $s[i_\ell..n{-}1]$ is the same as the order of their associated tuples and, hence, the suffixes can be sorted by sorting the tuples in $\mathcal{O}(n)$ time using the radix sort. We then assemble the SST as above using the LCE index. We do not dive into the proof of the claim since it essentially repeats similar arguments in [5]; see [5] for details.

▶ **Theorem 5.** *For any string of length n over an alphabet $[0..n^{\mathcal{O}(1)}]$ and any $b \ge \Omega(\log^2 n)$, one can construct in $\mathcal{O}(n \log_b n)$ time and $\mathcal{O}(b)$ space on top of the string the sparse suffix tree for arbitrarily chosen b suffixes.*

3 Refinement of Partitioning Sets

In this section we describe a process that takes the trivial partitioning set $[0..n)$ and iteratively refines it in $\lfloor \log \frac{\tau}{2^4 \log^{(3)} n} \rfloor$ phases removing some positions so that, after the kth phase, the set is $(2^{k+3}\lfloor \log^{(3)} n \rfloor)$-partitioning and has size $\mathcal{O}(n/2^k)$; moreover, it is "almost" 2^{k+3}-partitioning, satisfying properties (a) and (b) but not necessarily (c) (for $\tau = 2^{k+3}$). In particular, the set after the last phase is $\frac{\tau}{2}$-partitioning (and, thus, τ-partitioning by Lemma 1) and has size $\mathcal{O}(\frac{n}{\tau} \log^{(3)} n)$. Each phase processes all positions of the currently refined set from left to right and, in an almost online fashion, chooses which of them remain in the set. Rather than performing the phases one after another, which requires $\mathcal{O}(n)$ space, we run them simultaneously feeding the positions generated by the kth phase to the $(k{+}1)$th phase. Thus, the resulting set is produced in one pass. The set, however, has size $\mathcal{O}(\frac{n}{\tau} \log^{(3)} n)$, which is still too large to be stored in $\mathcal{O}(n/\tau)$ space; this issue is addressed in Section 5. Let us elaborate on the details of this process.

Throughout this section, we assume that $\tau \ge 2^5 \log^{(3)} n$ and, hence, the number of phases is non-zero; the case $\tau < 2^5 \log^{(3)} n$ is addressed in Appendix E in the full version [27]. Consider the kth phase, for $k \ge 1$. Its input is a set S_{k-1} produced by the $(k{-}1)$th phase; for $k = 1$, $S_0 = [0..n)$. Denote by j_h the hth position in S_{k-1} (so that $j_1 < \cdots < j_{|S_{k-1}|}$). The phase processes $j_1, j_2, \ldots$ from left to right and decides which of them to put into the new set $S_k \subseteq S_{k-1}$ under construction. The decision for j_h is based on the distances $j_h - j_{h-1}$ and $j_{h+1} - j_h$, on the substrings $s[j_{h+\ell}..j_{h+\ell}{+}2^k]$ with $\ell \in [-1..4]$, and on certain numbers v_{h-1}, v_h, v_{h+1} computed for j_{h-1}, j_h, j_{h+1}, which we define below. For technical reasons, we also assume $j_0 = -\infty$ and $j_{|S_{k-1}|+1} = \infty$, so $j_1 - j_0 = \infty$ and $j_{|S_{k-1}|+1} - j_{|S_{k-1}|} = \infty$.

For any distinct integers $x, y \geq 0$, denote by $\mathsf{bit}(x, y)$ the index of the lowest bit in which the bit representations of x and y differ (the lowest bit has index 0); e.g., $\mathsf{bit}(1, 0) = 0$, $\mathsf{bit}(2, 8) = 1$, $\mathsf{bit}(8, 0) = 3$. It is well known that $\mathsf{bit}(x, y)$ can be computed in $\mathcal{O}(1)$ time provided x and y occupy $\mathcal{O}(1)$ machine words [35]. Denote $\mathsf{vbit}(x, y) = 2\,\mathsf{bit}(x, y) + a$, where a is the bit of x with index $\mathsf{bit}(x, y)$; e.g., $\mathsf{vbit}(8, 0) = 7$ and $\mathsf{vbit}(0, 8) = 6$. Note that the bit representation of the number $\mathsf{vbit}(x, y)$ is obtained from that of $\mathsf{bit}(x, y)$ by appending a.

Let w be the number of bits in an $\mathcal{O}(\log n)$-bit machine word sufficient to represent letters from the alphabet $[0..n^{\mathcal{O}(1)}]$ of s. For each j_h, denote $s_h = \sum_{i=0}^{2^k} s[j_h{+}i]2^{wi}$. Each number s_h takes $(2^k{+}1)w$ bits and its bit representation coincides with that of the string $s[j_h..j_h{+}2^k]$, when we treat this string as a number stored in memory in the little endian format. The numbers s_h are introduced merely for convenience of the exposition, they are never discerned from their corresponding substrings $s[j_h..j_h{+}2^k]$ in the algorithm. For each j_h, define $v'_h = \mathsf{vbit}(s_h, s_{h+1})$ if $j_{h+1} - j_h \leq 2^{k-1}$ and $s_h \neq s_{h+1}$, and $v'_h = \infty$ otherwise. Observe that $\mathsf{bit}(s_h, s_{h+1}) = w\ell + \mathsf{bit}(s[j_h{+}\ell], s[j_{h+1}{+}\ell])$, where $\ell = \mathsf{lce}(j_h, j_{h+1})$; i.e., $\mathsf{bit}(s_h, s_{h+1})$ is given by an LCE query in the bit string of length wn obtained from s by substituting each letter with its w-bit representation. Define $v''_h = \mathsf{vbit}(v'_h, v'_{h+1})$, $v'''_h = \mathsf{vbit}(v''_h, v''_{h+1})$, $v_h = \mathsf{vbit}(v'''_h, v'''_{h+1})$, assuming $\mathsf{vbit}(x, y) = \infty$ if either $x = \infty$ or $y = \infty$.

For each j_h, denote by $R(j_h)$ a predicate that is true iff $j_{h+1}{-}j_h \leq 2^{k-1}$ and $s_h = s_{h+1}$; to verify whether $R(j_h)$ holds, we always check the former condition first and only then the latter if the former condition is satisfied.

Refinement rule. *The kth phase decides to put a position j_h into S_k either if $\infty > v_{h-1} > v_h$ and $v_h < v_{h+1}$ (i.e., $v_{h-1} \neq \infty$ and v_h is a local minimum of the sequence $v_1, v_2, \ldots$), or in three "boundary" cases: (i) $j_{h+1} - j_h > 2^{k-1}$ or $j_h - j_{h-1} > 2^{k-1}$; (ii) $R(j_{h-1})$ does not hold while $R(j_h)$, $R(j_{h+1})$, $R(j_{h+2})$ hold; (iii) $R(j_h)$ holds but $R(j_{h+1})$ does not.*

Note that we always have $j_1, j_{|S_{k-1}|} \in S_k$ since $j_1{-}j_0 = \infty > 2^{k-1}$ and $j_{|S_{k-1}|+1}{-}j_{|S_{k-1}|} = \infty > 2^{k-1}$. For now, assume that the numbers $\mathsf{bit}(s_h, s_{h+1})$, required to calculate v'_h and $R(j_h)$, are computed by the naïve comparison of $s[j_h..j_h{+}2^k]$ and $s[j_{h+1}..j_{h+1}{+}2^k]$ in $\mathcal{O}(2^k)$ time (we will change it later). Thus, the process is well defined. The trick with local minima and vbit reductions is, in essence, as in the deterministic approach of Cole and Vishkin to locally consistent parsings [7]. In what follows we derive some properties of this approach in order to prove that the kth phase indeed produces a $(2^{k+3}\lfloor\log^{(3)} n\rfloor)$-partitioning set.

It is convenient to interpret the kth phase as follows (see Fig. 1): the sequence $j_1, j_2, \ldots$ is split into maximal disjoint contiguous regions such that, for any pair of adjacent positions j_h and j_{h+1} inside each region, the distance $j_{h+1} - j_h$ is at most 2^{k-1} and $R(j_h) = R(j_{h+1})$. Thus, the regions are of two types: all-R ($\{j_{16}, \ldots, j_{20}\}$ in Fig. 1) and all-non-R ($\{j_1, \ldots, j_{15}\}$ or $\{j_{21}, \ldots, j_{25}\}$ in Fig. 1). By case (i), for each long gap $j_{h+1} - j_h > 2^{k-1}$ between regions, we put both j_h and j_{h+1} into S_k. In each all-R region, we put into S_k its last position due to case (iii) and, if the length of the region is at least 3, its first position by case (ii). In each all-non-R region, we put into S_k all local minima v_h such that $v_{h-1} \neq \infty$. Only all-non-R regions have positions j_h with $v_h \neq \infty$; moreover, as it turns out, only the last three or four their positions j_h have $v_h = \infty$ whereas, for other j_h, $v_h \neq \infty$ and $v_h \neq v_{h+1}$. Lemmas 8, 9 describe all this formally; their proof is deferred to Appendix B.1 in the full version [27].

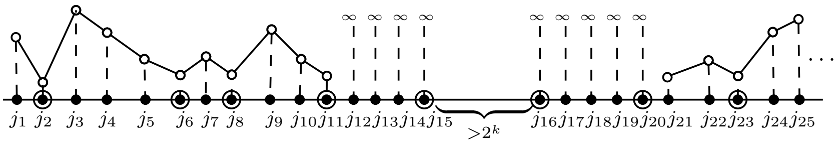
∞ ∞ ∞ ∞
∞ ∞ ∞ ∞ ∞
...
j_1 j_2 j_3 j_4 j_5 j_6 j_7 j_8 j_9 j_{10} j_{11} j_{12} j_{13} j_{14} j_{15}
$>2^k$
j_{16} j_{17} j_{18} j_{19} j_{20} j_{21} j_{22} j_{23} j_{24} j_{25}

Figure 1 The kth phase. The heights of the dashed lines over j_h are equal to v_h. Encircled positions are put into S_k: they are local minima of v_h, or are at the "boundaries" of all-R regions, or form a gap of length $>2^k$. In the figure $R(j_{16}), \dots, R(j_{20})$ hold and $R(j_{21})$ does not hold.

The goal of the fourfold vbit reduction for v_h is to make v_h small enough so that local minima occur often and, thus, the resulting set S_k is not too sparse. This is the key observation of Cole and Vishkin [7] and it is stated in Lemma 7 and directly follows from the construction of v_h and Lemma 6.

▶ **Lemma 6** (see [7]). *Given a string $a_1 a_2 \cdots a_m$ over an alphabet $[0..2^u)$ such that $a_i \neq a_{i+1}$ for any $i \in [1..m)$, the string $b_1 b_2 \cdots b_{m-1}$ such that $b_i = \mathsf{vbit}(a_i, a_{i+1})$, for $i \in [1..m)$, satisfies $b_i \neq b_{i+1}$, for any $i \in [1..m{-}1)$, and $b_i \in [0..2u)$.*

Proof. Consider b_i and b_{i+1}. Denote $\ell = \mathsf{bit}(a_i, a_{i+1})$ and $\ell' = \mathsf{bit}(a_{i+1}, a_{i+2})$. As $a_i, a_{i+1} \in [0..2^u)$, we have $\ell \in [0..u)$. Hence, $b_i \le 2\ell + 1 \le 2u - 1$, which proves $b_i \in [0..2u)$. If $b_i = b_{i+1}$, then $\ell = \ell'$ and the bits with indices ℓ and $\ell' = \ell$ in a_i and a_{i+1} coincide; however, by the definition of $\ell = \mathsf{bit}(a_i, a_{i+1})$, a_i and a_{i+1} must differ in this bit, which is a contradiction. ◀

▶ **Lemma 7** (see [7]). *For any $v_h \neq \infty$ in the kth phase, we have $v_h \in [0..2\log^{(3)} n{+}3)$.*

Proof. Since $v'_h \in [0..2nw) = [0..\mathcal{O}(n \log n))$, we deduce from Lemma 6 that $v''_h \in [0..\mathcal{O}(\log n))$, $v'''_h \in [0..2\log\log n{+}\mathcal{O}(1))$, and, due to the inequality $\log(x{+}\delta) \le \log x {+} \frac{\delta \log e}{x}$, we finally obtain $v_h \in [0..2\log^{(3)} n + 3)$, for sufficiently large n. ◀

The refinement rule implies that, for contiguous regions $j_p, j_{p+1}, \dots, j_q$ where $R(j_h)$ holds, only j_p and j_q may be in S_k and the period of $s[j_p..j_q + 2^k]$ is $\le 2^{k-1}$; for "dense" contiguous regions $j_p, j_{p+1}, \dots, j_q$ where $R(j_h)$ does not hold, Lemma 6 ensure frequent local minima. This is summarized in Lemmas 8, 9 (the proofs are in Appendix B.1 in the full version [27]).

▶ **Lemma 8.** *Let $j_p, j_{p+1}, \dots, j_q$ be a maximal contiguous region of $j_1, j_2, \dots$ such that, for all $h \in [p..q]$, $R(j_h)$ holds. Then, we have $j_q \in S_k$. Further, if $q - p \ge 2$ or $j_p - j_{p-1} > 2^{k-1}$, we have $j_p \in S_k$. All other positions j_h in the region do not belong to S_k. The string $s[j_p..j_q{+}2^k]$ has a period at most 2^{k-1}.*

▶ **Lemma 9.** *Let $j_p, j_{p+1}, \dots, j_q$ be a maximal contiguous region of $j_1, j_2, \dots$ such that, for all $h \in [p..q]$, $R(j_h)$ does not hold and, for $h \in [p..q)$, we have $j_{h+1} - j_h \le 2^{k-1}$. Then, $v_h \neq \infty$ for $h \in [p..q{-}4]$, $v_h = \infty$ for $h \in (q{-}3..q]$, and v_{q-3} may be ∞ or not. Further, for $h \in [p..q{-}3]$, we have $v_h \neq v_{h+1}$ whenever $v_h \neq \infty$. For $h \in (p..q)$, $j_h \in S_k$ iff $\infty > v_{h-1} > v_h$ and $v_h < v_{h+1}$; $j_p \in S_k$ iff $j_p - j_{p-1} > 2^{k-1}$; $j_q \in S_k$ iff $j_{q+1} - j_q > 2^{k-1}$.*

By Lemmas 9 and 7, any sequence of $8\log^{(3)} n + 12$ numbers v_h all of which are not ∞ contains a local minimum v_h and j_h will be put in S_k. Thus, we obtain the following lemma.

▶ **Lemma 10.** *Let S_{k-1} and S_k be the sets generated by the $(k{-}1)$th and kth phases. Then, any range $j_\ell, j_{\ell+1}, \dots, j_m$ of at least $8\log^{(3)} n + 12$ consecutive positions from S_{k-1} such that $v_h \neq \infty$, for all $h \in [\ell..m]$, has a position from S_k.*

The following intuitive lemma is very non-trivial; see Appendix B.2 in the full version [27].

▶ **Lemma 11.** *For any $i, i' \in [0..n]$, $|S_k \cap [i..i')| \le 2^6 \lceil (i'-i)/2^k \rceil$; in particular, $|S_k| \le n/2^{k-6}$.*

Now we are able to prove that S_k is a $(2^{k+3}\lfloor \log^{(3)} n \rfloor)$-partitioning set and, moreover, it is almost a 2^{k+3}-partitioning set, in a sense. The proof technique is very similar to the one in [5]; for brevity, we defer its detailed proof to Appendix B.2 in the full version [27].

▶ **Lemma 12.** *The kth phase generates a $(2^{k+3}\lfloor \log^{(3)} n \rfloor)$-partitioning set S_k. Moreover, S_k is almost 2^{k+3}-partitioning: for $\tau = 2^{k+3}$, it satisfies properties (a) and (b) but not (c), i.e., if $(i..j) \cap S_k = \emptyset$, for $i, j \in S_k$ such that $2^{k+3} < j-i \le 2^{k+3}\lfloor \log^{(3)} n \rfloor$, then $s[i..j]$ does not necessarily have period $\le 2^{k+2}$.*

4 Speeding up the Refinement Procedure

Since, for any k, $|S_k| \le n/2^{k-6}$ by Lemma 11, it is evident that the algorithm of Section 3 takes $\mathcal{O}(|S_0| + |S_1| + \cdots) = \mathcal{O}(n)$ time plus the time needed to calculate the numbers v'_h, for all positions (from which the numbers v_h are derived). For a given $k \ge 1$, denote by j_h the hth position in S_{k-1}. For each j_h, the number v'_h can be computed by checking whether $j_{h+1} - j_h > 2^{k-1}$ (in this case $v'_h = \infty$), and, if $j_{h+1} - j_h \le 2^{k-1}$, by the naïve comparison of $s[j_h..j_h{+}2^k]$ and $s[j_{h+1}..j_{h+1}{+}2^k]$ in $\mathcal{O}(2^k)$ time. Thus, all numbers v'_h for the set S_{k-1} can be computed in $\mathcal{O}(2^k|S_{k-1}|) = \mathcal{O}(n)$ time, which leads to $\mathcal{O}(n \log \tau)$ total time for the whole algorithm. This naïve approach can be sped up if one can perform the LCE queries that compare $s[j_h..j_h{+}2^k]$ and $s[j_{h+1}..j_{h+1}{+}2^k]$ faster; in fact, if one can do this in $\mathcal{O}(1)$ time, the overall time becomes linear. To this end, we exploit the online nature of the procedure. Let us briefly outline the procedure again on a high level.

The algorithm runs simultaneously $\lfloor \log \frac{\tau}{2^4 \log^{(3)} n} \rfloor$ phases: the kth phase takes positions from the set S_{k-1} produced by the $(k{-}1)$th phase and decides which of them to feed to the $(k{+}1)$th phase, i.e., to put into S_k (the "top" phase feeds the positions to an external procedure described in the next section). To make the decision for $j_h \in S_{k-1}$, the kth phase needs to know the distance $j_h{-}j_{h-1}$ and the distances $j_{h+\ell}{-}j_h$ to the positions $j_{h+\ell}$ with $\ell \in [1..5]$ such that $j_{h+\ell}{-}j_h \le 5 \cdot 2^{k-1}$. Then, the kth phase calculates $\min\{2^k{+}1, \mathsf{lce}(j_{h+\ell-1}, j_{h+\ell})\}$, for all $\ell \in [0..5]$ such that $j_{h+\ell}{-}j_{h+\ell-1} \le 2^{k-1}$ and $j_{h+\ell}{-}j_h \le 5 \cdot 2^{k-1}$, and, based on the distances and the LCE values, computes v_{h-1}, v_h, v_{h+1} and decides the fate of j_h.

The key for our optimization is the locality of the decision making in the phases that is straightforward for the described process: for any prefix $s[0..d]$, once the positions $S_{k-1} \cap [0..d]$ are known to the kth phase, it reports all positions from the set $S_k \cap [0..d{-}5 \cdot 2^{k-1}]$ and no position from the set $S_{k-1} \cap [0..d{-}6 \cdot 2^{k-1}]$ will be accessed by an LCE query of the kth phase in the future. Thus, we can discard all positions $S_{k-1} \cap [0..d{-}6 \cdot 2^{k-1}]$ and have to focus only on positions $S_{k-1} \cap (d{-}6 \cdot 2^{k-1}..\infty]$ and LCE queries on them in the future. We deduce from this that after processing the prefix $s[0..d]$ by the whole algorithm, the kth phase reports all positions from the set $S_k \cap [0..d{-}5\sum_{k'=0}^{k-1} 2^{k'}] \supseteq S_k \cap [0..d{-}5 \cdot 2^k]$ and no LCE query in the kth phase accesses positions from the set $S_{k-1} \cap [0..d{-}6 \cdot 2^k]$ in the future.

This locality of the decision procedure guarantees that, at the time we processed a length-ℓ prefix of the string s, for some $\ell \ge 0$, all positions from the set $S_k \cap [0..\ell{-}5 \cdot 2^k]$ are reported and no position from the set $S_{k-1} \cap [0..\ell{-}5 \cdot 2^k]$ will be accessed by an LCE query of the kth phase in the future. Let us summarize this as follows.

▶ **Lemma 13.** *Suppose we run the described $\lfloor \log \frac{\tau}{2^4 \log^{(3)} n} \rfloor$ phases on a string s of length n from left to right. Then, for any $k \ge 1$ and $d \ge 0$, after processing the prefix $s[0..d]$, the kth phase reports all positions from $S_k \cap [0..d{-}5 \cdot 2^k]$ to the $(k+1)$th phase and will not perform queries $\mathsf{lce}(j, j')$ on positions $j, j' \in S_{k-1}$ such that $\min\{j, j'\} \le d - 6 \cdot 2^k$ in the future.*

Recall that we have $\mathcal{O}(\log \tau)$ phases and at least $\Omega(\log \tau)$ space. Let us sketch main techniques to speed up the algorithm. Details are given in Appendix C in the full version [27].

Suppose that $\tau < \sqrt{n}$. We have $b = \Theta(\frac{n}{\tau}) \geq \Omega(\sqrt{n})$ additional space for the algorithm in this case. To answer all required LCE queries in constant time, when the algorithm processes a letter $s[d]$, the classical LCE data structure from [16] is maintained for the leftmost substring $C_i = s[i\lfloor\sqrt{n}\rfloor..(i+3)\lfloor\sqrt{n}\rfloor - 1]$ whose middle part contains the position d (i.e. $d \in (i+1)\lfloor\sqrt{n}\rfloor..(i+2)\lfloor\sqrt{n}\rfloor - 1]$). By Lemma 13, we can use the data structure to correctly handle all queries because all LCE queries performed by the algorithm at the step d lie within the substring C_i. Since we must build the LCE data structure for every C_i once, the overall running time is $\mathcal{O}(n + \sum_i |C_i|) = \mathcal{O}(n)$ and the occupied space is $\mathcal{O}(\sqrt{n}) = \mathcal{O}(b)$.

Let us generalize this idea to the case $\tau \geq \sqrt{n}$. Denote $b = \frac{n}{\tau}$. We have $\mathcal{O}(b) < \mathcal{O}(\sqrt{n})$ space and cannot use the scheme described above since LCE data structures for substrings of length $\mathcal{O}(b)$ are not enough to answer queries of the form $\min\{2^k{+}1, \mathsf{lce}(j, j')\}$ when $2^k > \Omega(b)$. The key idea is to group contiguous phases into "levels" and maintain SST for a sliding window of positions in each level (in the case $\tau < \sqrt{n}$ we had a single "level" and a sliding window of size $\mathcal{O}(\sqrt{n})$). We must choose "level" size to be large enough to build less SSTs and fit in the $\mathcal{O}(n \log_b n)$ running time, but also the "levels" must be small to efficiently reduce the number of positions in each level and fit all supporting data structures in the $\mathcal{O}(b)$ space. To achieve this, we split evenly all $\lfloor \log \frac{\tau}{2^4 \log^{(3)} n} \rfloor$ phases into "levels", each containing $\Theta(\log \hat{b})$ phases, where $\hat{b} = \lfloor \frac{b}{\log n} \rfloor$. For each "level", we maintain a window of $\mathcal{O}(\hat{b})$ positions from S_k, where k is the lowest phase in the "level"; one window spans a substring of length $\mathcal{O}(2^k \hat{b})$ and the windows change $\mathcal{O}(\frac{n}{2^k \hat{b}})$ times in total. Overall we use $\mathcal{O}(\hat{b} \log \hat{b}) = \mathcal{O}(b)$ space. By Lemma 12, the set S_k is "almost" 2^k-partitioning, so we can build SST for each "level" as in Lemma 3 in time $\mathcal{O}(2^k \hat{b} + \min\{2^k \hat{b} \log_{\hat{b}} n, \hat{b} \log \hat{b}\})$, which simplifies to $\mathcal{O}(\hat{b} \log_{\hat{b}} n)$ for the first "level" and to $\mathcal{O}(2^k \hat{b})$ for subsequent "levels". (Note that there are no vicious circles here: Lemma 3 is self-contained and builds its SST using only the radix sort for strings related to its input partitioning set.) Overall we can upperbound the running time with $\mathcal{O}(n \log_b n)$ for all $\mathcal{O}(\log_b n)$ "levels". Thus, the described routine builds partitioning sets S_k in time $\mathcal{O}(n \log_b n)$ and space $\mathcal{O}(b)$. The described sketch of the algorithm is elaborated in details in Appendix C in the full version [27].

5 Recompression

Let S be the set produced by the last phase of the procedure from Sections 3 and 4. By Lemma 12, S is a $\frac{\tau}{2}$-partitioning set of size $\mathcal{O}(\frac{n}{\tau} \log^{(3)} n)$. Throughout this section, we assume that $\tau \geq (\log^{(3)} n)^4$ so that the size of S is at most $\mathcal{O}(\frac{n}{(\log^{(3)} n)^3})$; the case $\tau < (\log^{(3)} n)^4$ is discussed in Appendix E in the full version [27]. In what follows we describe an algorithm that removes positions from S transforming it into a τ-partitioning set of size $\mathcal{O}(n/\tau)$.

Instead of storing S explicitly, which is impossible in $\mathcal{O}(n/\tau)$ space, we construct a related-to-S string R of length $\mathcal{O}(\frac{n}{\tau} \log^{(3)} n)$ over a small alphabet such that R can be packed into $\mathcal{O}(n/\tau)$ machine words. Positions of S are represented, in a way, by letters of R. The construction of R is quite intricate, which is necessary in order to guarantee that letters of R corresponding to close positions of S (namely, positions at a distance at most $\tau/2^5$) are necessarily distinct even if the letters are not adjacent in R. This requirement is stronger than the requirement of distinct adjacent letters that was seen, for instance, in Lemma 6 but it is achieved by similar means using vbit reductions as in Section 3. We then apply to R a variant of the iterative process called *recompression* [19] that removes some letters thus shrinking the length of R to $\mathcal{O}(n/\tau)$. Then, the whole procedure of Sections 3–4 that generated S is performed again but this time we discard all positions of S corresponding to removed positions of the string R and store the remaining positions explicitly in a set $S^* \subseteq S$. We show that S^* is τ-partitioning and has size $\mathcal{O}(n/\tau)$. Let us elaborate on the details.

The algorithm starts with an empty string R and receives positions of S from left to right appending to the end of R new letters corresponding to the received positions. It is more convenient to describe the algorithm as if it acted in two stages: the first stage produces a $\frac{3}{4}\tau$-partitioning set $S' \subseteq S$, for which a condition converse to property (c) holds (thus, some positions of S are discarded already in this stage), and the second stage, for each position of S', appends to the end of R a letter of size $\mathcal{O}((\log^{(3)} n)^2)$ bits. Both stages act in an almost online fashion and, hence, can be actually executed simultaneously in one pass without the need to store the auxiliary set S'. The separation is just for the ease of the exposition.

The first stage. The goal is to construct set $S' \subseteq S$ by excluding from S all positions h for which there exist $i, j \in S$ such that $i < h \le j$, $j - i \le \tau/4$, and $s[i..i{+}\tau/2] = s[j..j{+}\tau/2]$. The algorithm generating S' is as follows.

We consider all positions of S from left to right and, for each $i \in S$, process every $j \in (i..i{+}\tau/4] \cap S$ by comparing $s[i..i{+}\tau/2]$ with $s[j..j{+}\tau/2]$. If $s[i..i{+}\tau/2] = s[j..j{+}\tau/2]$, then we traverse all positions of the set $(i..j] \cap S$ from right to left marking them for removal until an already marked position is encountered. Since the marking procedure works from right to left, every position is marked at most once. The position i is put into S' iff it was not marked previously. During the whole process, we maintain a "look-ahead" queue that stores the positions $(i..i{+}\tau/4] \cap S$ and indicates which of them were marked for removal.

Due to Lemma 11, the size of the set $(i..i{+}\tau/4] \cap S$ is $\mathcal{O}(\log^{(3)} n)$. Therefore, the look-ahead queue takes $\mathcal{O}(\log^{(3)} n)$ space, which is $\mathcal{O}(n/\tau)$ since $n/\tau \ge \log^2 n$, and $\mathcal{O}(\log^{(3)} n)$ comparisons are performed for each i. Hence, if every comparison takes $\mathcal{O}(1)$ time, the set S' is constructed in $\mathcal{O}(|S| \log^{(3)} n) = \mathcal{O}(\frac{n}{\tau}(\log^{(3)} n)^2)$ time, which is $\mathcal{O}(n)$ since $\tau \ge (\log^{(3)} n)^4$. Thus, it remains to explain how the comparisons can be performed.

Similar to the algorithm of Section 4, we consecutively consider substrings $C'_i = s[i\tau..(i+3)\tau)$, for $i \in [0..n/\tau{-}3]$: when all positions from a set $S \cap [i\tau..(i{+}3)\tau)$ are collected, we use the algorithm of Lemma 3 to build a SST for all suffixes of the string C'_i whose starting positions are from S; the tree, endowed with an LCA data structure [16], is used in the procedure for deciding which of the positions from the set $S \cap [(i + \frac{1}{2})\tau..(i + \frac{3}{2})\tau)$ (or $S \cap [0..\frac{3}{2}\tau)$ if $i = 0$) should be marked for removal. Thus, after processing the last string C'_i, all positions of S are processed and S' is generated. By Lemma 11, the number of suffixes in the SST for C'_i is $\mathcal{O}(\log^{(3)} n)$ and, therefore, the tree occupies $\mathcal{O}(\log^{(3)} n) \le \mathcal{O}(n/\tau)$ space and its construction takes $\mathcal{O}(\tau + \log^{(3)} n \cdot \log\log^{(3)} n)$ time by Lemma 3, which is $\mathcal{O}(\tau)$ since $\tau \ge (\log^{(3)} n)^4$. Thus, the total construction time for all the trees in the stage is $\mathcal{O}(\frac{n}{\tau}\tau) = \mathcal{O}(n)$ and the space used is $\mathcal{O}(\log^{(3)} n)$ since, at every moment, at most one tree is maintained.

The following lemma shows that the transformation within the first stage does not break τ-partitioning properties. Its proof is deferred to Appendix D.1 in the full version [27].

▶ **Lemma 14.** *The set S' is τ-partitioning and satisfies a converse of property (c): if a substring $s[i..j]$ has a period at most $\tau/4$, then $S' \cap [i{+}\frac{3}{4}\tau..j{-}\frac{3}{4}\tau] = \emptyset$. Moreover, S' is almost $\frac{3}{4}\tau$-partitioning, meeting properties (a) and (b) with $\frac{3}{4}\tau$ in place of τ, but not necessarily (c).*

The second stage. We consider all positions of S' from left to right and, for each $p \in S'$, append to the end of the (initially empty) string R a new carefully constructed letter a_p occupying $\mathcal{O}((\log^{(3)} n)^2)$ bits. Thus, the string R will have length $|S'|$ and will take

$\mathcal{O}(|S'|(\log^{(3)} n)^2) = \mathcal{O}(\frac{n}{\tau}(\log^{(3)} n)^3)$ bits of space, which can be stored into $\mathcal{O}(n/\tau)$ machine words of size $\mathcal{O}(\log n)$ bits. The crucial property of R for us is that any two letters of R corresponding to close positions of S' are distinct, namely the following lemma will be proved:

▶ **Lemma 15.** *For any $p, \bar{p} \in S'$, if $0 < \bar{p} - p \le \tau/2^5$, then $a_p \neq a_{\bar{p}}$.*

Consider $p \in S'$. We are to describe an algorithm generating an $\mathcal{O}((\log^{(3)} n)^2)$-bit letter a_p for p that will be appended to the string R.

Denote by $p_1, p_2, \dots, p_m$ all positions of $S' \cap (p..p{+}\tau/2^5]$ in the increasing order. By Lemma 11, $m \le \mathcal{O}(\log^{(3)} n)$ and, hence, there is enough space to store them. By construction, $s[p..p{+}\frac{\tau}{2}] \neq s[p_j..p_j{+}\frac{\tau}{2}]$, for each $j \in [1..m]$. One can compute the longest common prefix of $s[p..p{+}\frac{\tau}{2}]$ and $s[p_j..p_j{+}\frac{\tau}{2}]$, for any $j \in [1..m]$, in $\mathcal{O}(1)$ time using a SST with an LCA data structure [16] built in the first stage for a substring $C'_i = s[i\tau..(i+3)\tau - 1]$ such that $p \in [i\tau..(i+\frac{3}{2})\tau)$. (In order to have $p_1, p_2, \dots, p_m$ prepared, we handle p, which was reported by the first stage after processing C'_i, only when C'_{i+1} was processed too; thus, the first stage maintains two SSTs: one for a substring C'_{i+1} currently under analysis and one for C'_i, retained for its use in the second stage.)

Denote $\ell = 2^6\lceil\log^{(3)} n\rceil$. Recall that S is produced by the kth phase of the procedure of Section 3, for $k = \lfloor\log\frac{\tau}{2^4\log^{(3)} n}\rfloor$, and hence, by Lemma 11, the size of any set $S \cap [i..j]$, for $i \le j$, is at most $2^6\lceil(j-i+1)/2^k\rceil$. Therefore, since $S' \subseteq S$ and m is the size of the set $S' \cap (p..p{+}\tau/2^5]$, we obtain $m \le 2^6(\tau/2^5)/\frac{\tau}{2\cdot 2^4\log^{(3)} n} \le \ell$.

Let w be the number of bits in an $\mathcal{O}(\log n)$-bit machine word sufficient to represent letters from the alphabet $[0..n^{\mathcal{O}(1)}]$ of s. For each p_j, denote $t_j = \sum_{i=0}^{\tau/2} s[p_j{+}i]2^{wi}$; similarly, for p, denote $t = \sum_{i=0}^{\tau/2} s[p{+}i]2^{wi}$. As in an analogous discussion in Section 3, we do not discern the numbers t_j and t from their corresponding substrings in s and use them merely in the analysis. The intuition behind our construction is that the numbers $t, t_1, t_2, \dots, t_m$, in principle, could have been used for the string R as letters corresponding to the positions $p, p_1, p_2, \dots, p_m$ since $t, t_1, t_2, \dots, t_m$ are pairwise distinct (due to the definition of S') but, unfortunately, they occupy too much space ($\mathcal{O}(w\tau)$ bits each). One has to reduce the space for the letters retaining the property of distinctness. The tool capable to achieve this was already developed in Section 3: it is the vbit reduction, a trick from Cole and Vishkin's deterministic locally consistent parsing [7].

We first generate for p a tuple of ℓ numbers $\langle w'_1, w'_2, \dots, w'_\ell\rangle$: for $j \in [1..\ell]$, $w'_j = \mathsf{vbit}(t, t_j)$ if $j \le m$, and $w'_j = \infty$ otherwise. Since the longest common prefix of substrings $s[p..p{+}\frac{\tau}{2}]$ and $s[p_j..p_j{+}\frac{\tau}{2}]$, for $j \in [1..m]$, can be calculated in $\mathcal{O}(1)$ time, the computation of the tuple takes $\mathcal{O}(\ell) = \mathcal{O}(\log^{(3)} n)$ time. By Lemma 6, each number w'_j occupies less than $\lceil\log w + \log\tau + 1\rceil$ bits. Thus, we can pack the whole tuple into $\ell\lceil\log w + \log\tau + 1\rceil$ bits encoding each value w'_j into $\lceil\log w + \log\tau + 1\rceil$ bits and representing ∞ by setting all bits to 1. We denote this chunk of $\ell\lceil\log w + \log\tau + 1\rceil$ bits by $\bar{t}$. In the same way, for each p_i with $i \in [1..m]$, we generate a tuple $\langle w'_{i,1}, w'_{i,2}, \dots, w'_{i,\ell}\rangle$ comparing $s[p_i..p_i{+}\tau/2]$ to $s[q..q{+}\tau/2]$, for each $q \in S' \cap (p_i..p_i{+}\tau/2^5]$, and using the vbit reduction; the tuple is packed into a chunk $\bar{t}_i$ of $\ell\lceil\log w + \log\tau + 1\rceil$ bits. See Figure 2. For each $j \in [1..m]$, the number w'_j is not equal to ∞ and, thus, due to Lemma 6, differs from the number $w'_{j,j}$ (the jth element of the tuple $\langle w'_{i,1}, w'_{i,2}, \dots, w'_{i,\ell}\rangle$). Therefore, all the tuples – and, hence, their corresponding numbers $\bar{t}, \bar{t}_1, \bar{t}_2, \dots, \bar{t}_m$ – are pairwise distinct.

The numbers $\bar{t}, \bar{t}_1, \bar{t}_2, \dots, \bar{t}_m$, like the numbers $t, t_1, t_2, \dots, t_m$, could have been used, in principle, as letters for the string R but they still are too large. We therefore repeat the same vbit reduction but now for the numbers $\bar{t}, \bar{t}_1, \bar{t}_2, \dots, \bar{t}_m$ in place of $t, t_1, t_2, \dots, t_m$ thus generating a tuple $\langle w''_1, w''_2, \dots, w''_\ell\rangle$: for $j \in [1..\ell]$, $w''_j = \mathsf{vbit}(\bar{t}, \bar{t}_j)$ if $j \le m$, and $w''_j = \infty$

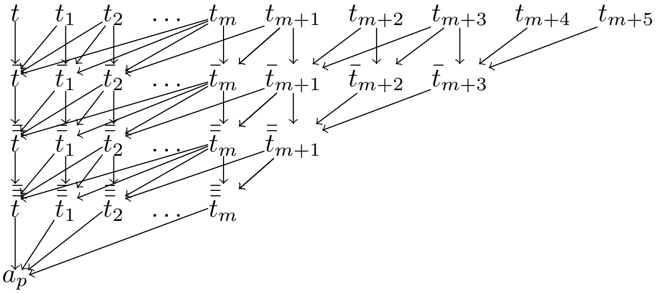

Figure 2 The scheme generating a_p via vbit reductions. If a node $\hat t$ has ingoing edges labeled with $\tilde t, \tilde t_1, \tilde t_2, \dots, \tilde t_r$ (from left to right), then $\hat t$ encodes a tuple $\langle \tilde w_1, \tilde w_2, \dots, \tilde w_\ell\rangle$ such that, for $j \in [1..r]$, $\tilde w_j = \mathsf{vbit}(\tilde t, \tilde t_j)$ and, for $j \in (r..\ell]$, $\tilde w_j = \infty$. In the figure, the numbers $t, t_1, t_2, \dots, t_{m+5}$ correspond to consecutive positions $p, p_1, p_2, \dots, p_{m+5}$ in the set S', respectively. By looking at which of the ingoing edges are present and which are not, one can deduce that here we have $S' \cap (p..p{+}\tau/2^5] = \{p_1, \dots, p_m\}$, $S' \cap (p_1..p_1{+}\tau/2^5] = \{p_2, \dots, p_m\}$, $S' \cap (p_2..p_2{+}\tau/2^5] = \{p_3, \dots, p_m, p_{m+1}\}$, $S' \cap (p_m..p_m{+}\tau/2^5] = \{p_{m+1}\}$, $S' \cap (p_{m+1}..p_{m+1}{+}\tau/2^5] = \{p_{m+2}, p_{m+3}\}$, $S' \cap (p_{m+2}..p_{m+2}{+}\tau/2^5] = \{p_{m+3}\}$, $S' \cap (p_{m+3}..p_{m+3}{+}\tau/2^5] = \{p_{m+4}, p_{m+5}\}$.

otherwise. The computation of $\mathsf{vbit}(\bar t, \bar t_j)$ takes $\mathcal{O}(\ell)$ time since $\bar t$ occupies ℓ machine words of size $\mathcal{O}(\log n)$ bits. It follows from Lemma 6 that the tuple $\langle w''_1, w''_2, \dots, w''_\ell\rangle$ can be packed into a chunk $\bar{\bar t}$ of $\ell\lceil\log\ell + \log\lceil\log w + \log\tau + 1\rceil + 1\rceil$ bits (i.e., $\mathcal{O}(\log^{(3)} n \cdot \log\log n)$ bits), which already fits into one machine word. We perform analogous reductions for the positions $p_1, p_2, \dots, p_m$ generating m tuples $\langle w''_{i,1}, w''_{i,2}, \dots, w''_{i,\ell}\rangle$, for $i \in [1..m]$, packed into new chunks $\bar{\bar t}_1, \bar{\bar t}_2, \dots, \bar{\bar t}_m$, respectively. Note that, in order to produce a tuple $\langle w''_{i,1}, w''_{i,2}, \dots, w''_{i,\ell}\rangle$, for $i \in [1..m]$, that is packed into $\bar{\bar t}_i$, we use not only the numbers $\bar t_i, \bar t_{i+1}, \dots, \bar t_m$ corresponding to positions $p_i, p_{i+1}, \dots, p_m$ but also similarly computed numbers at other positions from $S' \cap (p_i..p_i{+}\tau/2^5]$, if any. See Figure 2 for a clarification: it can be seen that the "top" numbers include not only $t, t_1, \dots, t_m$ precisely because of this.

By the same argument that proved the distinctness of $\bar t, \bar t_1, \bar t_2, \dots, \bar t_m$, one can easily show that $\bar{\bar t}, \bar{\bar t}_1, \bar{\bar t}_2, \dots, \bar{\bar t}_m$ are pairwise distinct. But they are still too large to be used as letters of R. Then again, we repeat the same reductions at positions $p, p_1, p_2, \dots, p_m$ but now for the numbers $\bar{\bar t}, \bar{\bar t}_1, \bar{\bar t}_2, \dots, \bar{\bar t}_m$ in place of $\bar t, \bar t_1, \bar t_2, \dots, \bar t_m$, thus generating new chunks $\bar{\bar{\bar t}}, \bar{\bar{\bar t}}_1, \bar{\bar{\bar t}}_2, \dots, \bar{\bar{\bar t}}_m$. Finally, once more, we do the vbit reduction for $\bar{\bar{\bar t}}, \bar{\bar{\bar t}}_1, \bar{\bar{\bar t}}_2, \dots, \bar{\bar{\bar t}}_m$ generating a tuple $\langle w_1, w_2, \dots, w_\ell\rangle$ such that, for $j \in [1..\ell]$, w_j is $\mathsf{vbit}(\bar{\bar{\bar t}}, \bar{\bar{\bar t}}_j)$ if $j \le m$, and ∞ otherwise.

Using the same reasoning as in the proof of Lemma 7, one can deduce from Lemma 6 that the tuple $\langle w_1, w_2, \dots, w_\ell\rangle$ fits into a chunk of $\ell \cdot 2\log\log^{(3)} n \le 2^6\lceil\log^{(3)} n\rceil^2$ bits (the inequality holds provided $n > 2^{16}$) encoding each value w_j into $\lceil\log^{(3)} n\rceil$ bits and representing ∞ by setting all $\lceil\log^{(3)} n\rceil$ bits to 1. Denote by a_p this chunk of $2^6\lceil\log^{(3)} n\rceil^2$ bits that encodes the tuple. We treat a_p as a new letter of R that corresponds to the position p and we append a_p to the end of R. Lemma 15 follows then straightforwardly by construction.

Given $p \in S'$, the calculation of the numbers $\bar t, \bar{\bar{\bar t}}, a_p$ takes $\mathcal{O}(\ell^2)$ time. The calculation of $\bar{\bar t}$ requires $\mathcal{O}(\ell^3)$ time since each reduction $\mathsf{vbit}(\bar t, \bar t_j)$ for it takes $\mathcal{O}(\ell)$ time. Hence, the total time for the construction of R is $\mathcal{O}(|S'|\ell^3) = \mathcal{O}(\frac{n}{\tau}(\log^{(3)} n)^4)$, which is $\mathcal{O}(n)$ as $\tau \ge (\log^{(3)} n)^4$.

Recompression. If the distance between any pair of adjacent positions of S' is at least $\tau/2^6$, then $|S'| \le 2^6 n/\tau$ and, by Lemma 14, S' can be used as the resulting τ-partitioning set of size $\mathcal{O}(n/\tau)$. Unfortunately, in general, this is not the case and we have to "sparsify" S'.

There is a one-to-one correspondence between S' and positions of R. Using a technique of Jeż [19] called *recompression* , we can remove in $\mathcal{O}(|R|)$ time some letters of R reducing by a fraction $\frac{4}{3}$ the number of pairs of adjacent letters $R[i], R[i{+}1]$ whose corresponding positions in S' are at a distance at most $\tau/2^6$. We perform such reductions until the length of R becomes at most $2^{14} \cdot n/\tau$. The positions of S' corresponding to remaining letters will constitute a τ-partitioning set of size $\mathcal{O}(n/\tau)$. In order to guarantee that this subset of S' is τ-partitioning, we have to execute the recompression reductions gradually increasing the distances that are of interest for us: first, we get rid of adjacent pairs with distances at most $\tau/\log^{(3)} n$ between them, then the threshold is increased to $2\tau/\log^{(3)} n$, then $2^2\tau/\log^{(3)} n$, and so on until (most) adjacent pairs with distances at most $2^{\log^{(4)} n-6}\tau/\log^{(3)} n = \tau/2^6$ between them are removed in last recompression reductions. The details follow.

Since it is impossible to store in $\mathcal{O}(n/\tau)$ space the precise distances between positions of S', the information about distances needed for recompression is encoded as follows. For each $i \in [0..|R|)$ and a position $p \in S'$ corresponding to the letter $R[i]$, we store an array of numbers $M_i[0..\lceil\log^{(4)} n\rceil]$ such that, for $j \in [0..\lceil\log^{(4)} n\rceil]$, $M_i[j]$ is equal to the size of the set $S' \cap (p..p{+}\tau/2^j]$. By Lemma 11, we have $|S' \cap (p..p{+}\tau]| \le \mathcal{O}(\log^{(3)} n)$ and, hence, each number $M_i[j]$ occupies $\mathcal{O}(\log^{(4)} n)$ bits. Therefore, all the arrays M_i can be stored in $\mathcal{O}(|R|(\log^{(4)} n)^2) \le \mathcal{O}(\frac{n}{\tau}\log^{(3)} n \cdot (\log^{(4)} n)^2)$ bits, which fits into $\mathcal{O}(\frac{n}{\tau})$ machine words of size $\mathcal{O}(\log n)$ bits. All arrays M_i are constructed in a straightforward way in $\mathcal{O}(|R|\log^{(3)} n) = \mathcal{O}(\frac{n}{\tau}(\log^{(3)} n)^2)$ time (which is $\mathcal{O}(n)$ since $\tau \ge (\log^{(3)} n)^4$) during the left-to-right pass over S' that generated the string R.

Our algorithm consecutively considers all numbers $j \in [6..\lceil\log^{(4)} n\rceil]$ in decreasing order, starting from $j = \lceil\log^{(4)} n\rceil$. For each j, it iteratively performs a recompression procedure reducing the number of adjacent letters $R[i], R[i{+}1]$ whose corresponding positions from S' are at a distance at most $\tau/2^j$, until R shrinks to a length at most $2^{j+10}\cdot\frac{n}{\tau}$. Thus, $|R| \le 2^{16}\cdot\frac{n}{\tau}$ after last recompression reductions for $j = 6$. Let us describe the recompression procedure.

Fix $j \in [6..\lceil\log^{(4)} n\rceil]$. To preserve property (c) of the τ-partitioning set S' during the sparsifications, we impose an additional restriction: a letter $R[i]$ cannot be removed if either $i = 0$ or the distance between the position $p \in S'$ corresponding to $R[i]$ and the predecessor of p in S' is larger than $\tau/2^5$, i.e., if $M_{i-1}[5] = 0$. The rationale is as follows: the position p might be the right boundary of a gap in S' of length $> \tau$ and it is dangerous to break the gap since, once p is removed, the gap might not satisfy property (c) (the range of the string s corresponding to the gap should have a period that is at most $\tau/4$).

The processing of the number j starts with checking whether $|R| \le 2^{j+10}\cdot\frac{n}{\tau}$. If so, we skip the processing of j and move to $j-1$ (provided $j > 6$). Suppose that $|R| > 2^{j+10}\cdot\frac{n}{\tau}$. Denote $\sigma = 2^{2^6\lceil\log^{(3)} n\rceil^2}$, the size of the alphabet $[0..\sigma)$ of R. Then, the algorithm creates an array $P[0..\sigma{-}1][0..\sigma{-}1]$ filled with zeros, which occupies $\mathcal{O}(\sigma^2) = \mathcal{O}(2^{2^7(\log^{(3)} n)^2}) = o(\log n)$ space, and collects in P statistics on pairs of adjacent letters of R whose corresponding positions in S' are at a distance at most $\tau/2^j$ and whose first letter may be removed: namely, we traverse all $i \in [1..|R|)$ and, if $M_i[j] \ne 0$ and $M_{i-1}[5] \ne 0$, then we increase by one the number $P[R[i]][R[i{+}1]]$. By Lemma 15, $R[i] \ne R[i+1]$ when $M_i[j] \ne 0$.

The core tool of the recompression technique proposed by Jeż [19] is an algorithm for multidigraph without self-loops $G = (V, E)$ that constructs a directed cut of size at least $\lceil\frac{|E|}{4}\rceil$ edges in time $\mathcal{O}(|V|^2)$ if the graph is given by an adjacency matrix. If we interpret P as an adjacency matrix, we can use Jeż's technique (there are no self-loops because $R[i] \ne R[i+1]$ when $M_i[j] \ne 0$ due to Lemma 15) and split the alphabet into two disjoint subsets correspoding to the cut: $[0..\sigma) = \acute{\Sigma} \sqcup \grave{\Sigma}$. After that we mark for removal from R

all indices $i \in [1..|R|{-}1)$ for which the following conditions hold: $M_i[j] \neq 0$, $M_{i-1}[5] \neq 0$, $R[i] \in \acute{\Sigma}$, and $R[i{+}1] \in \grave{\Sigma}$. Once the sets $\acute{\Sigma}$ and $\grave{\Sigma}$ are computed in time $\mathcal{O}(\sigma^2) = o(\log^2 n)$, the marking takes $\mathcal{O}(|R|)$ time and can be organized using a bit array of length $|R|$.

After the marking step we update values in all arrays M_i according to removal marks in one right to left pass: for each $i \in [0..|R|)$ and $j' \in [0..\lceil \log^{(4)} n \rceil]$, the new value for $M_i[j']$ is the number of indices $i+1, i+2, \ldots, i+M_i[j']$ that were not marked for removal, i.e., $M_i[j']$ is the number of positions in the set $S' \cap (p..p{+}\tau/2^{j'}]$ whose corresponding letters $R[i']$ will remain in R, where $p \in S'$ is the position corresponding to $R[i]$. Since $M_i[j'] \leq M_{i+1}[j'] + 1$, for $i \in [0..|R|{-}1)$, the pass updating M can be executed in $\mathcal{O}(|R| \log^{(4)} n)$ time.

Finally, we delete letters $R[i]$ and arrays M_i, for all indices i marked for removal, thus shrinking the length of R and the storage for M_i. We call this procedure, which marks letters of R and removes them and their corresponding arrays M_i, the recompression. One recompression iteration takes $\mathcal{O}(|R| \log^{(4)} n)$ time, where $|R|$ is the length of R before shrinking.

The next lemma states that the recompression shrinks the string R by a constant factor.

▶ **Lemma 16.** *If, for $j \in [6..\lceil \log^{(4)} n \rceil]$, before the recompression procedure there were d non-zero numbers $M_i[j]$ with $i \in [1..|R|)$ such that $M_{i-1}[5] \neq 0$, then the arrays M_i modified by the procedure, for all i corresponding to unremoved positions of R, contain at most $\frac{3}{4}d$ non-zero numbers $M_i[j]$ such that $M_{i-1}[5]{\neq}0$.*

Proof. The proof repeats an argument from [19] and [17, Lemma 7]. Consider an undirected weighted graph G corresponding to the digraph encoded in the adjacency matrix P. By construction of P, we have $d = \sum_{a \neq b} P[a][b]$, which follows from Lemma 15 that guarantees $R[i] \neq R[i+1]$ when $M_i[j] \neq 0$. Thus, d is the sum of weights of all edges in G. Putting a letter a into either $\acute{\Sigma}$ or $\grave{\Sigma}$, we add to the cut at least half of the total weight of all edges connecting a to the letters $0, 1, \ldots, a{-}1$. Therefore, the cut of G induced by $\acute{\Sigma}$ and $\grave{\Sigma}$ has a weight at least $\frac{1}{2}d$. The edges in the cut might be directed both from $\acute{\Sigma}$ to $\grave{\Sigma}$ and in the other direction. Switching $\acute{\Sigma}$ and $\grave{\Sigma}$, if needed, we ensure that the direction from $\acute{\Sigma}$ to $\grave{\Sigma}$ has a maximal total weight, which is obviously at least $\frac{1}{4}d$. According to this cut, we mark for removal from R at least $\frac{1}{4}d$ letters $R[i]$ such that $M_i[j] \neq 0$. Hence, the number of non-zero values $M_i[j]$ such that $M_{i-1}[5] \neq 0$ is reduced by $\frac{1}{4}d$, which gives the result of the lemma since new non-zero values could not appear after the deletions. ◀

Suppose, for a fixed $j \in [6..\lceil \log^{(4)} n \rceil]$, the algorithm has performed one iteration of the recompression. Denote by S'' the set of all positions from S' that "survived" the recompression for $j \in [6..\lceil \log^{(4)} n \rceil]$ and, thus, have a corresponding letter in the updated string R. There is a one-to-one correspondence between S'' and letters of R. For each $i \in [0..|R|)$ and $j' \in [0..\lceil \log^{(4)} n \rceil]$, the number $M_i[j']$ in the modified arrays M_i is the size of the set $S'' \cap (p..p{+}\tau/2^{j'}]$, for a position $p \in S''$ corresponding to i. We therefore can again apply the recompression procedure thus further shrinking the length of R. The algorithm first again checks whether $|R| > 2^{j+10} \cdot \frac{n}{\tau}$ and, if so, repeats the recompression. For the given fixed j, we do this iteratively until $|R| \leq 2^{j+10} \cdot \frac{n}{\tau}$. During this process, the number of zero values $M_i[j]$ in the arrays M_i is always at most $2^j \cdot \frac{n}{\tau}$ since the equality $M_i[j] = 0$ implies that $S''' \cap (p..p{+}\tau/2^j] = \emptyset$, for a set $S''' \subseteq S'$ of size $|R|$ defined by analogy to the definition of S'' and for a position $p \in S'''$ corresponding to i. Therefore, due to Lemma 16, the condition $|R| \leq 2^{j+10} \cdot \frac{n}{\tau}$ eventually should be satisfied. Furthermore, as we are to show, for each j, the condition $|R| \leq 2^{j+10} \cdot \frac{n}{\tau}$ holds after at most three recompression iterations.

Given $j \in [6..\lceil \log^{(4)} n \rceil)$, the length of R before the first iteration of the recompression for j is at most $2^{j+11} \cdot \frac{n}{\tau}$ since this is a condition under which shrinking iterations stopped for $j+1$. The same bound holds for $j = \lceil \log^{(4)} n \rceil$: the initial length of R is at most $2^{11} \cdot \frac{n}{\tau} \log^{(3)} n$ (which is upper-bounded by $2^{j+11} \cdot \frac{n}{\tau}$) since $S' \subseteq S$ and S is produced by the kth phase of the procedure of Section 3, for $k = \lfloor \log \frac{\tau}{2^4 \log^{(3)} n} \rfloor$, so that the size of S, by Lemma 11, is at most $2^6 \lceil n/2^k \rceil \le 2^6 n / \frac{\tau}{2 \cdot 2^4 \log^{(3)} n} = 2^{11} \frac{n}{\tau} \log^{(3)} n$. Fix $j \in [6..\lceil \log^{(4)} n \rceil]$. Since the number of zero values $M_i[j]$ is always at most $2^j \cdot n/\tau$ and the number of zero values $M_{i-1}[5] = 0$ is at most $2^5 \cdot \frac{n}{\tau}$, three iterations of the recompression for j performed on a string R with initial length r shrink the length of R to a length at most $(\frac{3}{4})^3 r + 2^j \cdot \frac{n}{\tau} + 2^5 \cdot \frac{n}{\tau} \le (\frac{3}{4})^3 r + 2 \cdot 2^j \cdot \frac{n}{\tau}$, by Lemma 16. Putting $r = 2^{j+11} \cdot \frac{n}{\tau}$, we estimate the length of R after three iterations for j from above by $((\frac{3}{4})^3 2^{11} + 2) 2^j \cdot \frac{n}{\tau} < 2^{j+10} \cdot \frac{n}{\tau}$. That is, for each j, three iterations are enough to reduce the length of R to at most $2^{j+10} \cdot \frac{n}{\tau}$.

Thus, the total running time of all recompression procedures is $\mathcal{O}(\sum_{j=\lceil \log^{(4)} n \rceil}^{6} 2^{j+11} \cdot \frac{n}{\tau} \log^{(4)} n) = \mathcal{O}(\frac{n}{\tau} \log^{(4)} n)$, which is $\mathcal{O}(n)$ since $\tau \ge (\log^{(3)} n)^4$. Observe that the most time consuming part is in recalculations of the arrays M_i, each taking $\mathcal{O}(|R| \log^{(4)} n)$ time, all other parts take $\mathcal{O}(|R|)$ time, i.e., $\mathcal{O}(\sum_{j=\lceil \log^{(4)} n \rceil}^{6} 2^{j+11} \cdot \frac{n}{\tau}) = \mathcal{O}(\frac{n}{\tau})$ time is needed for everything without the recalculations. The length of R in the end is at most $2^{16} \cdot n/\tau$, which is a condition under which shrinking iterations stopped for $j = 6$.

Finally, we create a bit array E of the same length as the original string R that marks by 1 those letters that survived all iterations. Additional navigational structures for linear-time E construction are straightforward. We then re-run whole "semi-online" algorithm that generates the set S' (from which the string R was constructed) but, in this time, we discard all positions of S' that correspond to unmarked indices in E and we store all positions corresponding to marked indices of E explicitly in an array S^*. Since at most $2^{16} \cdot n/\tau$ indices in E are marked by 1, the size of S^* is $\mathcal{O}(n/\tau)$.

Finally, we have all required instruments to prove the main lemma. The proof is rather technical and, in a way, similar to the proof of Lemma 12; it is detailed in Appendix D.2 in the full version [27].

▶ **Lemma 17.** *The set S^* is τ-partitioning; also a converse of property (c) holds for S^*: if a substring $s[i..j]$ has a period at most $\tau/4$, then $S^* \cap [i + \tau..j - \tau] = \emptyset$.*

References

1 S. Alstrup, G. S. Brodal, and T. Rauhe. Pattern matching in dynamic texts. In *Proc. 11th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 819–828. SIAM, 2000.

2 P. Bille, J. Fischer, I.L. Gørtz, T. Kopelowitz, B. Sach, and H. W. Vildhøj. Sparse text indexing in small space. *ACM Transactions on Algorithms*, 12(3):1–19, 2016. doi:10.1145/2836166.

3 P. Bille, I. L. Gørtz, M. B. T. Knudsen, M. Lewenstein, and H. W. Vildhøj. Longest common extensions in sublinear space. In *Proc. 26th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 9133 of *LNCS*, pages 65–76. Springer, 2015. doi:10.1007/978-3-319-19929-0_6.

4 P. Bille, I. L. Gørtz, B. Sach, and H. W. Vildhøj. Time-space trade-offs for longest common extensions. *Journal of Discrete Algorithms*, 25:42–50, 2014. doi:10.1016/j.jda.2013.06.003.

5 O. Birenzwige, S. Golan, and E. Porat. Locally consistent parsing for text indexing in small space. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 607–626. SIAM, 2020. doi:10.1137/1.9781611975994.37.

6 G. S. Brodal, P. Davoodi, and S. S. Rao. On space efficient two dimensional range minimum data structures. In *Proc. 18th Annual European Symposium on Algorithms (ESA)*, volume 6347 of *LNCS*, pages 171–182. Springer, 2010. doi:10.1007/s00453-011-9499-0.

7 R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986. doi:10.1016/S0019-9958(86)80023-7.

8 M. Crochemore and W. Rytter. Squares, cubes, and time-space efficient string searching. *Algorithmica*, 13(5):405–425, 1995. doi:10.1007/BF01190846.

9 P. Dinklage, J. Fischer, A. Herlez, T. Kociumaka, and F. Kurpicz. Practical performance of space efficient data structures for longest common extensions. In *Proc. 28th Annual European Symposium on Algorithms (ESA)*, volume 173 of *LIPIcs*, pages 39:1–39:20, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2020.39.

10 M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 137–143. IEEE, 1997. doi:10.1109/SFCS.1997.646102.

11 J. Fischer, T. I, and D. Köppl. Deterministic sparse suffix sorting on rewritable texts. In *Proc. 12th Latin American Symposium on Theoretical Informatics (LATIN)*, volume 9644 of *LNCS*, pages 483–496. Springer, 2016. doi:10.1007/978-3-662-49529-2_36.

12 M. Gańczorz, P. Gawrychowski, A. Jeż, and T. Kociumaka. Edit distance with block operations. In *Proc. 26th Annual European Symposium on Algorithms (ESA)*, volume 112 of *LIPIcs*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.ESA.2018.33.

13 P. Gawrychowski, A. Karczmarz, T. Kociumaka, J. Łącki, and P. Sankowski. Optimal dynamic strings. In *Proc. 29th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1509–1528. SIAM, 2018. doi:10.1137/1.9781611975031.99.

14 P. Gawrychowski and T. Kociumaka. Sparse suffix tree construction in optimal time and space. In *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 425–439. SIAM, 2017. doi:10.1137/1.9781611974782.27.

15 A. Goldberg, S. Plotkin, and G. Shannon. Parallel symmetry-breaking in sparse graphs. In *Proc. 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 315–324. ACM, 1987. doi:10.1145/28395.28429.

16 D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984. doi:10.1137/0213024.

17 T. I. Longest common extensions with recompression. In *Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 78 of *LIPIcs*, pages 18:1–18:15, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CPM.2017.18.

18 T. I, J. Kärkkäinen, and D. Kempa. Faster sparse suffix sorting. In *Proc. 31st International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 25 of *LIPIcs*, pages 386–396, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.STACS.2014.386.

19 A. Jeż. Approximation of grammar-based compression via recompression. *Theoretical Computer Science*, 592:115–134, 2015. doi:10.1016/j.tcs.2015.05.027.

20 A. Jeż. Faster fully compressed pattern matching by recompression. *ACM Transactions on Algorithms*, 11(3):20, 2015. doi:10.1145/2631920.

21 A. Jeż. A really simple approximation of smallest grammar. *Theoretical Computer Science*, 616:141–150, 2016. doi:10.1016/j.tcs.2015.12.032.

22 A. Jeż. Recompression: a simple and powerful technique for word equations. *Journal of the ACM*, 63(1):4, 2016. doi:10.1145/2743014.

23 J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.

24 J. Kärkkäinen and E. Ukkonen. Lempel–Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd South American Workshop on String Processing (WSP)*, pages 141–155. Carleton University Press, 1996.

25 D. Kempa and T. Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In *Proc. 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 756–767. ACM, 2019. doi:10.1145/3313276.3316368.

26 D. Kosolobov. Tight lower bounds for the longest common extension problem. *Information Processing Letters*, 125:26–29, 2017. doi:10.1016/j.ipl.2017.05.003.

27 D. Kosolobov and N. Sivukhin. Construction of sparse suffix trees and LCE indexes in optimal time and space. *arXiv preprint arXiv:2105.03782*, 2021.

28 K. Mehlhorn, R. Sundar, and C. Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997. doi:10.1007/BF02522825.

29 T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda. Dynamic index and LZ factorization in compressed space. *Discrete Applied Mathematics*, 274:116–129, 2020. doi:10.1016/j.dam.2019.01.014.

30 N. Prezza. Optimal substring equality queries with applications to sparse text indexing. *ACM Transactions on Algorithms*, 17(1):1–23, 2020. doi:10.1145/3426870.

31 M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004. doi:10.1093/bioinformatics/bth408.

32 S. C. Sahinalp and U. Vishkin. Symmetry breaking for suffix tree construction. In *Proc. 26th Annual ACM Symposium on Theory of Computing (STOC)*, pages 300–309. ACM, 1994. doi:10.1145/195058.195164.

33 S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 76–85, 2003. doi:10.1145/872757.872770.

34 Y. Tanimura, T. I, H. Bannai, S. Inenaga, S.J. Puglisi, and M. Takeda. Deterministic sub-linear space LCE data structures with efficient construction. In *Proc. 27th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 54 of *LIPIcs*, pages 1:1–1:10, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CPM.2016.1.

35 D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(\mathrm{N})$. *Information Processing Letters*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.