Sparse Suffix Tree Construction in Optimal Time and Space

Paweł Gawrychowski\textsuperscript{1} and Tomasz Kociumaka\textsuperscript{2}

\textsuperscript{1}University of Haifa, Israel
\textsuperscript{2}Institute of Informatics, University of Warsaw, Poland
\{gawry,kociumaka\}@mimuw.edu.pl

Abstract

Suffix tree (and the closely related suffix array) are fundamental structures capturing all substrings of a given text essentially by storing all its suffixes in the lexicographical order. In some applications, such as sparse text indexing, we work with a subset of \(b\) interesting suffixes, which are stored in the so-called sparse suffix tree. Because the size of this structure is \(\Theta(b)\), it is natural to seek a construction algorithm using only \(O(b)\) words of space assuming read-only random access to the text. We design a linear-time Monte Carlo algorithm for this problem, hence resolving an open question explicitly stated by Bille et al. [TALG 2016]. The best previously known algorithm by I et al. [STACS 2014] works in \(O(n \log b)\) time. As opposed to previous solutions, which were based on the divide-and-conquer paradigm, our solution proceeds in \(n/b\) rounds. In the \(r\)-th round, we consider all suffixes starting at positions congruent to \(r\) modulo \(n/b\). By maintaining rolling hashes, we can lexicographically sort all interesting suffixes starting at such positions, and then we can merge them with the already considered suffixes. For efficient merging, we also need to answer LCE queries efficiently (and in small space). By plugging in the structure of Bille et al. [CPM 2015] we obtain \(O(n + b \log b)\) time complexity. We improve this structure by a recursive application of the so-called difference covers, which then implies a linear-time sparse suffix tree construction algorithm.

We complement our Monte Carlo algorithm with a deterministic verification procedure. The verification takes \(O(n \sqrt{\log b})\) time, which improves upon the bound of \(O(n \log b)\) obtained by I et al. [STACS 2014]. This is obtained by first observing that the pruning done inside the previous solution has a rather clean description using the notion of graph spanners with small multiplicative stretch. Then, we are able to decrease the verification time by applying difference covers twice. Combined with the Monte Carlo algorithm, this gives us an \(O(n \sqrt{\log b})\)-time and \(O(b)\)-space Las Vegas algorithm.

\textsuperscript{*}Work done while the author held a post-doctoral position at Warsaw Center of Mathematics and Computer Science.

\textsuperscript{†}Supported by Polish budget funds for science in 2013-2017 as a research project under the ‘Diamond Grant’ program.
1 Introduction

In many if not all algorithms operating on texts one needs a compact representation of all substrings. A well-known data structure capturing all substrings of a given text is the suffix tree, which is a compacted trie storing all suffixes. The size of the suffix tree is linear in the length of the text and it provides efficient indexing, that is, locating all occurrences of a given pattern. The first linear-time suffix tree construction algorithm was given by Weiner \[26\]. Later, McCreight \[23\] provided a simpler procedure, and Ukkonen showed a different approach that allows the text to be maintained under appending characters \[25\]. All these algorithms work in linear time assuming constant-size alphabet. However, such assumption is not always justified. Farach developed a different construction method based on the divide-and-conquer paradigm that takes only linear time as long as the alphabet is linear-time sortable \[9\]. In particular, his algorithm works in linear time for polynomially-bounded integer alphabets.

While the suffix tree provides a lot of information about the structure of the text and hence is a very convenient building block in more complicated algorithms, its large memory footprint is often prohibitive in practice. Hence there has been a lot of interest in suffix arrays \[22\]. A suffix array is just a lexicographically sorted list of all suffixes of the text. The list is usually augmented with the so-called LCP table, which stores the length of the longest common prefix of every two adjacent suffixes. This leads to a very memory efficient representation that is still capable of providing enough information about the text to replace suffix trees in all applications with no or very small penalty in the time complexity \[1\]. Furthermore, a suffix array can be constructed in linear time for any linear-time sortable alphabet \[18\].

Even though the suffix array occupies linear space when measured in words, this might be larger than the encoding of the text. This started a long line of work on compressed suffix arrays \[10,14,15\], which take space proportional to the entropy of the text. However, in some applications even smaller space usage is desired. In particular, in the last few years there has been a lot of interest among the string algorithms community in designing sublinear space solutions, where one assumes a read-only random access to the input text and measures the working space. Of course, the running time should still be linear or close to linear.

A natural idea for text indexing in sublinear space is to use some additional knowledge about the structure of the queries to consider only a (small) subset of all suffixes of the text, and provide indexing only for occurrences starting at the corresponding positions. This was first explored by Kärkkäinen and Ukkonen \[19\], who introduced the sparse suffix tree. They showed that the evenly space sparse suffix tree, which is the compacted trie storing every \(k\)-th suffix of the text, can be constructed in linear time and working space proportional to the number of suffixes. However, the question of construct a general sparse suffix tree (or sparse suffix array, which is a lexicographically sorted list of the chosen subset of suffixes together with their LCP information) for an arbitrary subset of \(b\) suffixes of a text \(T[1..n]\) using \(O(b)\) working space, remained open. Recently, Bille et al. \[4\] were able to make a significant progress towards resolving this question by developing an \(O(n \log^2 b)\)-time Monte Carlo algorithm. They also provided a verification procedure implying an \(O(n \log^2 b + b^2 \log b)\) Las Vegas algorithm. Then, I et al. \[17\] improved the complexity to \(O(n \log b)\) (both for Monte Carlo and Las Vegas randomization). They also gave an \(O(n)\)-time Monte Carlo solution using \(O(b \log b)\) space. Very recently, Fischer et al. \[12\] gave an \(O(n \sqrt{\log n} + b \log b \log n \log^* n)\)-time and \(O(b)\)-space deterministic algorithm in a stronger model of rewritable text which needs to be restored before termination.

Another natural problem in the model of read-only random access to the text is LCE queries,
where we are to preprocess a text subject to queries \( \text{LCE}(i, j) \) returning the longest common prefix of two suffixes \( T[i..] \) and \( T[j..] \). Several trade-off between query time, data structure size, construction time and space usage have been obtained [5, 6, 24]. The queries are typically deterministic, but construction algorithms range from Monte Carlo randomization via Las Vegas randomization to deterministic solutions. State-of-the-art Monte Carlo data structures of \( \mathcal{O}(b) \) size have \( \mathcal{O}(n) \)-time and \( \mathcal{O}(b) \)-space construction with \( \mathcal{O}(n/b + \log b) \)-time queries, or \( \mathcal{O}(n \log b) \)-time and \( \mathcal{O}(b) \)-space construction with \( \mathcal{O}(n/b) \)-time queries [5].

**Our contribution.** We design an \( \mathcal{O}(n) \)-time Monte Carlo algorithm for sorting an arbitrary subset of \( b \) suffixes of a text \( T[1..n] \) using \( \mathcal{O}(b) \) working space. We also show how to verify the answer in \( \mathcal{O}(n \sqrt{\log b}) \) time and \( \mathcal{O}(b) \) working space, which implies a Las Vegas algorithm with such complexity. Hence, for Monte Carlo algorithms we close the problem, while for Las Vegas algorithm we are able to make a substantial progress towards the desired linear time complexity.

As an auxiliary result, we also develop an \( \mathcal{O}(b) \)-space LCE data structure with \( \mathcal{O}(n/b) \)-time queries and \( \mathcal{O}(n) \)-time and \( \mathcal{O}(b) \)-space Monte Carlo construction.

**Model.** We are given read-only random access to a text \( T[1..n] \) consisting of characters from \( \Sigma = \{1, 2, \ldots, n^{\Theta(1)}\} \). We assume the standard word RAM model with word size \( \Theta(\log n) \), where basic arithmetic and bit-wise operations on \( \mathcal{O}(\log n) \)-bit integers take constant time. Our randomized algorithms succeed with high probability, i.e., \( 1 - n^{-c} \) for any user-specified constant \( c \).

**Previous and our techniques.** The Monte Carlo algorithm of I et al. [17] is based on the notion of \( \ell \)-strict sparse suffix trees. Intuitively, they are approximate variants of the sparse suffix tree operating on blocks of length \( \ell \) rather with single-character precision. The algorithm starts with a trivial \( n \)-strict sparse suffix tree and performs \( \lceil \log n \rceil \) steps, each of which halves the block length.

Our algorithm, described in Section 3, performs just two steps. Its intermediate result, the coarse compacted trie, is basically the same as the \( [n/b] \)-strict sparse suffix tree. The second phase, building the sparse suffix tree from the coarse compacted trie, is relatively easy. For the more challenging first step, we employ Karp–Rabin fingerprints stored in a rolling fashion. More precisely, we proceed in \( [n/b] \) rounds; in the \( r \)-th round, we store fingerprints of all suffixes starting at positions congruent to \( r \) modulo \( [n/b] \). This way, while inserting such a suffix to the coarse compacted trie, we can guarantee that almost all fingerprints needed can be accessed in constant time. The last step of every insertion reduces to a longest common extension (LCE) query. If we apply a recent data structure by Bille et al. [5], the total running time becomes \( \mathcal{O}(n + b \log b) \).

To obtain \( \mathcal{O}(n) \) time, we improve the LCE query time to \( \mathcal{O}(n/b) \). Bille et al. [5] already applied difference covers for that purpose, but this was at the expense of superlinear preprocessing time since a certain sparse suffix tree had to be constructed. Our main insight is that one can build a sequence of larger and larger sparse suffix trees, each of them providing faster LCE queries used to construct the next sparse suffix tree.

Our complementary result, an \( \mathcal{O}(\sqrt{\log b}) \)-time Las Vegas algorithm, is provided in Section 4. In short, our solution is based on an \( \mathcal{O}(n \log^2 b) \)-time algorithm by I et al. [17] for verifying \( b \) substring equations. We provide a few improvements by exploiting a slightly cleaner (though more complex) formalization of the underlying ideas. First, we avoid eagerly checking some constraints, which lets us reduce the running time to \( \mathcal{O}(n \log b + b \log^2 n) \). Next, we use difference covers to restrict the
set of starting positions of fragments involved in equations; this technique independently speeds up two steps of the algorithm.

2 Preliminaries

We consider finite strings over an integer alphabet $\Sigma = \{1, \ldots, n^{O(1)}\}$. For a string $T = T[1] \cdots T[n]$, its length is $|T| = n$. For $1 \leq i \leq j \leq n$, a string $T[i] \cdots T[j]$ is called a substring of $T$. By $T[i..j]$ we denote its occurrence at position $i$, called a fragment of $T$. A fragment with $i = 1$ is called a prefix (also denoted $T[..j]$) and a fragment with $j = n$ is called a suffix (denoted $T[i..]$).

Tries and compacted tries. Recall that a trie is a rooted tree whose nodes correspond to prefixes of strings in a given set of strings $S$. The prefix corresponding to a node $u$ is denoted $\text{str}(u)$, and the node $u$ is called the locus of $\text{str}(u)$. We extend this notion as follows: the locus of an arbitrary string $s$ is the node $u$ such that $\text{str}(u)$ is prefix of $s$ and and $|\text{str}(u)|$ is maximized.

The parent-child relation in the trie is defined so that the root is the locus of $\varepsilon$, while the parent $u$ of a node $v$ is the locus of $\text{str}(v)$ without the last character. This character is the label of the edge from $u$ to $v$. The order on the alphabet naturally yields an order on the edges outgoing from any node of the trie, so tries are often assumed to be ordered rooted trees.

A node $u$ is branching if it has at least two children and terminal if $\text{str}(u) \in S$. A compacted trie is obtained from the underlying trie by dissolving all nodes except the root, branching nodes, and terminal nodes. The dissolved nodes are called implicit while the preserved nodes are called explicit. The compacted trie takes $O(|S|)$ space provided that edge labels are stored as pointers to fragments of strings in $S$. In some applications, the first character is kept explicitly, however.

Note that the suffix tree of a string $T$ is precisely the compacted trie of the set of all suffixes of $T$. Similarly, the sparse suffix tree of an arbitrary set $B$ of suffixes of $T$ is the compacted trie of $B$. Given the lexicographic order on $S$ along with the lengths of the longest common prefixes between any two consecutive (in this order) elements of $S$, one can easily compute the compacted trie in $O(|S|)$ time; see e.g. [8]. Thus, the problem of constructing the sparse suffix tree is equivalent to that of building the sparse suffix array along with the LCP values.

LCA queries. For two nodes $u, v$ of a trie, we denote their lowest common ancestor by $\text{lca}(u, v)$. Since $\text{str}(\text{lca}(u, v))$ is the longest common prefix of $\text{str}(u)$ and $\text{str}(v)$, the data structures for LCA queries are often applied to efficiently determine longest common prefixes.

Lemma 2.1 ([3, 16]). The compacted trie of a set of strings $S$ can be preprocessed in $O(|S|)$ time to compute the length of the longest common prefix of any two strings in $S$ in constant time.

Karp–Rabin fingerprints. For a prime number $p$, an integer $x \in \mathbb{Z}_p$, the Karp–Rabin fingerprint [20] of a string $w$ is $(\sum_{i=1}^{n} w[i] \cdot x^{i-1}) \mod p$. For efficiency, we augment it as follows:

$$\phi(w) = \left( \left( \sum_{i=1}^{n} w[i] \cdot x^{i-1} \right) \mod p, x^{\text{|w|}} \mod p, x^{-|w|} \mod p, |w| \right).$$

Observation 2.2. Let $u, v, w$ be strings such that $uv = w$. Given two out of three fingerprints $\phi(u), \phi(v), \phi(w)$, the third one can be computed in constant time.
For a text $T$, we say that the fingerprints are collision-free if $\phi(T[i..j]) = \phi(T[i'..j'])$ implies $T[i..j] = T[i'..j']$. Randomization lets us construct such fingerprints with high probability:

**Fact 2.3.** Let $T$ be a text of length $n$ over alphabet $\Sigma$, and let $p$ be a prime number such that $p \geq \max(|\Sigma|, n^{3+c})$. If $x$ is uniformly random, $\phi$ is collision-free with probability at least $1 - n^{-c}$.

This is the only source of randomization in this paper. The original argument by Karp and Rabin [20] used random $p$ and fixed $x$, but we use the more modern approach; see e.g. [4,5,17].

**LCE queries.** For a text $T$ of length $n$ and two positions $i, j$ ($1 \leq i, j \leq n$), we define LCE($i, j$) as the length of the longest common prefix of $T[i..]$ and $T[j..]$. We use the following recent result as a building block in our algorithms:

**Lemma 2.4** (Bille et al. [5]). Given read-only random access to a text $T$ of length $n$ and a parameter $b$, $1 \leq b \leq n$, it is possible to construct in $O(n)$ time and $O(b)$ space a structure of size $O(b)$, which answers LCE queries in $O(n/b + \log(b \cdot \ell/n)) = O(n/b + \log b)$ time, where $\ell$ is the result of the query. The data structure assumes collision-free Karp–Rabin fingerprints.

**Difference covers.** We say that a set $DC \subseteq \{1, \ldots, n\}$ is a $t$-difference-cover (of $\{1, \ldots, n\}$) if for each $i, j \in \{1, \ldots, n - t\}$ there is a value $h(i, j)$, $0 \leq h(i, j) < t$, such that $i + h(i, j) \in DC$ and $j + h(i, j) \in DC$. We say that $DC$ can be indexed efficiently, if there is a bijection $f : DC \to \{1, \ldots, |DC|\}$ such that $f$ and $f^{-1}$ are computable in constant time.

**Lemma 2.5** (Maekawa [21], Burkhard and Kärkkäinen [7]). For every positive integers $t$ and $n$, $t \leq n$, there exists a $t$-difference-cover $DC$ of size $O(\frac{n}{\sqrt{t}})$, such that $h$ can be evaluated in constant time and $DC$ can be efficiently indexed.

## 3 Monte Carlo Algorithm

In this section, we provide an $O(n)$-time and $O(b)$-space Monte Carlo algorithm for computing the sparse suffix tree. In Section 3.1, we introduce the coarse compacted trie, which is an intermediate byproduct of our algorithm, and we show how to use it to build the sought sparse suffix tree. Then we concentrate on computing the coarse compacted trie. Section 3.2 provides an $O(n + b \log b)$-time algorithm whose bottleneck are LCE queries. We overcome this in Section 3.3 by providing an improved data structure for LCE queries. We conclude the exposition in Section 3.4.

### 3.1 Coarse Compacted Tries

The coarse compacted trie of a collection of strings is defined as follows. We conceptually partition each string into blocks consisting of $\lceil n/b \rceil$ characters (the last block might be shorter). Then we consider each block to be a single supercharacter, and we form the compacted trie of the resulting set of strings. We use fingerprints to represent the supercharacters; hence, the order on the edges outgoing from the same node in a coarse compacted trie corresponds to the order of the fingerprints of the first block on every edge, not to the lexicographical order of the blocks.

Below, we show that the sought sparse suffix tree can be quite easily derived from the corresponding coarse compacted trie. The key building block is sorting $O(b)$ strings of length $\lceil n/b \rceil$. For $b \leq \sqrt{n}$, a simple comparison-based algorithm achieves $O(n)$ time and $O(b)$ space complexity.
Fact 3.1. Given random access to $b$ strings of length $\ell$, the strings can be sorted in $O(b(b+\ell))$ time and $O(b)$ space.

Proof. We shall prove that a single string $S$ can be inserted into a sorted array of $b$ strings in $O(b+\ell)$ time. We scan the consecutive letters of $S$. Having read $S[j]$, we maintain a partition of the strings $T$ in the array into three classes depending on whether $T[\ldots j]$ is smaller, equal, or greater than $S[\ldots j]$. Note that these classes form consecutive ranges. Moreover, in order to update the partition, it suffices to scan the ‘equal’ class from both ends and remove all leading strings $T$ satisfying $T[j+1] < S[j+1]$, and all trailing strings $T$ satisfying $T[j+1] > S[j+1]$. The running time is proportional to the number of strings removed plus $O(1)$, which gives $O(b+\ell)$ in total over all steps. After we scan the whole $S$, the partition determines the position where it should be inserted.

For $b \geq \sqrt{n}$, we have enough space to use an algorithm based on counting sort.

Fact 3.2. Assume we are given random access to $b$ strings of length $\ell$ over alphabet $\{1, \ldots, \sigma\}$. For any positive integer $S \leq \sigma$, the strings can be sorted in $O((b+S)\ell \log_S \sigma)$ time and $O(b+S)$ space.

Proof. We treat every character as a log-$\sigma$-bit integer and partition it into chunks of $\lceil \log S \rceil$ bits. We radix sort the resulting collection of $b$ strings of length $\Theta(\ell \log_S \sigma)$ over a smaller alphabet $\{1, 2, \ldots, S\}$. With $\Theta(\ell \log_S \sigma)$ iterations of counting sort, we obtain the claimed bounds.

Now, we can give the procedure building the compacted trie from of the coarse compacted trie.

Lemma 3.3. Given the coarse compacted trie of a set of $b$ suffixes, we can construct their compacted trie in $O(n)$ time and $O(b)$ space.

Proof. Consider a branching node $u$ of the coarse compacted trie and let $v_1, v_2, \ldots, v_k$ be its implicit children, that is, $(u, v_i)$ is an edge labelled with a single supercharacter representing a string $s_i$ of length $\lceil n/b \rceil$. To obtain the compacted trie, we remove the edges $(u, v_i)$ and paste the compacted trie of the strings $s_1, s_2, \ldots, s_k$ to connect $u$ with $v_1, v_2, \ldots, v_k$. Such a trie is easy to construct in $O(k[\lceil n/b \rceil])$ time by inserting strings in the lexicographic order: we follow the rightmost path while its label matches the inserted string, and we create a new branch as soon as it does not match. Hence, we only need to show how sort the strings $s_1, s_2, \ldots, s_k$.

We gather such collections of strings from all branching nodes, sort the disjoint union of all these collections, and then recover the sorted collections with a single scan. Note that the total number of strings to be sorted is bounded by the number of edges in the coarse compacted trie, which is $O(b)$. If $b \leq \sqrt{n}$, we sort using Fact 3.1, while for $b > \sqrt{n}$ we apply Fact 3.2 with $\sigma = n^{O(1)}$ and $S = b$. In both cases, sorting is performed is $O(n)$ time and $O(b)$ space. This is also the overall time and space necessary to construct compacted tries based on the sorted collections.

3.2 Construction of Coarse Compacted Tries

Again, we provide slightly different construction algorithms for $b \leq \sqrt{n}$ and $b > \sqrt{n}$. Both algorithms use Lemma 2.4 and rolling Karp–Rabin fingerprints, which we describe below. Moreover, while constructing the coarse compacted trie, at each explicit node $v$ we always store the fingerprint of the corresponding string $\text{str}(v)$. Similarly, each edge stores the fingerprint of the first block of its label, and a reference to a fragment of $T$ representing the whole label. For each node, the outgoing edges are kept in a doubly-linked list (ordered by the fingerprints of their first blocks).
For an integer \( r \), we say a position \( i \) is \( r \)-aligned if \( i \equiv r \pmod{n/b} \). A fragment \( T[i..j] \) is called an \( r \)-aligned fragment of \( T \) if \( i \) is an \( r \)-aligned position and \( j = n \) or \( j+1 \) is also \( r \)-aligned. The key idea of our algorithm is to proceed in \( [n/b] \) rounds so that in the \( r \)-th round, while inserting \( r \)-aligned suffixes, we can compute the fingerprint of any \( r \)-aligned fragment in constant time. More formally, we denote the set of all \( r \)-aligned suffixes by \( S_r \), and we define a component \( \Phi_r \) consisting of the fingerprint of every suffix in \( S_r \), as well as the fingerprint of the whole text \( T \). The following fact, stating all the necessary properties of \( \Phi_r \), easily follows from Observation 2.2.

**Fact 3.4.** The component \( \Phi_1 \) can be constructed in \( \mathcal{O}(n) \) time and \( \mathcal{O}(b) \) space. Moreover, given \( \Phi_r \), the fingerprint of any \( r \)-aligned fragment can be computed in constant time, and \( \Phi_{r+1} \) can be constructed in \( \mathcal{O}(b) \) time and space.

### 3.2.1 Small \( b \)

First, we show how to construct coarse compacted trie for a set of \( b \leq \sqrt{n} \) suffixes. Our procedure actually works for an arbitrary number of suffixes, but it is too slow for \( \omega(\sqrt{n}) \) suffixes.

**Theorem 3.5.** The coarse compacted trie of set of \( b \) suffixes of a text \( T \) of length \( n \) can be computed in \( \mathcal{O}(n + b^2) \) time.

**Proof.** We proceed in rounds corresponding to \( r = 1, \ldots, [n/b] \). In the \( r \)-th round, we insert \( r \)-aligned suffixes \( T[i..] \) one by one. For this, we use \( \Phi_r \) and the LCE-structure from Lemma 2.4.

We locate the locus of \( T[i..] \) in the current trie by a traversal starting at the root. If we are currently at an explicit node \( u \), we use \( \Phi_r \) to obtain the fingerprint of the next supercharacter to be followed. Next, we scan the edges going out of \( u \) comparing that fingerprint with the ones stored with the edges. If none of them matches, then \( u \) is the locus and we insert a new leaf with \( u \) as its parent. Otherwise, we have selected an edge leading to a child \( v \) of \( u \). We check if the locus is in the subtree of \( v \) by comparing \( \phi(\text{str}(v)) \) with the fingerprint of the corresponding \( r \)-aligned prefix of \( T[i..] \). If so, we continue the traversal at \( v \). Otherwise, we know that the locus is an implicit node on the edge \((u,v)\). In this case, we use one LCE query (rounded down to a multiple of \([n/b]\)) to calculate the exact position of the locus on the edge, and we attach a new leaf there. Then, we also need to spend \( \mathcal{O}(n/b) \) time to compute the fingerprint of the edge created by subdividing \((u,v)\) at the locus. All other fingerprints stored in the new nodes and edges are computed in constant time since the corresponding fragments are \( r \)-aligned.

The trie is of size \( \mathcal{O}(b) \), so in a single traversal we visit at most \( \mathcal{O}(b) \) explicit nodes and scan \( \mathcal{O}(b) \) edges in total, spending \( \mathcal{O}(1) \) time at each of them. The last step requires \( \mathcal{O}(n/b + \log b) \) time for an LCE query and fingerprint computation, so the overall insertion time is \( \mathcal{O}(b + n/b + \log b) = \mathcal{O}(b + n/b) \). Summing over all the rounds, this is \( \mathcal{O}(n + b^2) \) (including the time to maintain \( \Phi_r \) and build the LCE-structure). Space consumption remains \( \mathcal{O}(b) \) throughout the algorithm. \( \square \)

### 3.2.2 Large \( b \)

For larger \( b \), instead of processing suffixes one by one, we insert all suffixes from \( B_r = B \cap S_r \) in bulk. Our insertion algorithm requires the coarse compacted trie of \( B_r \), which is obtained from the coarse compacted trie of \( S_r \).

**Lemma 3.6.** For \( b = n^{\Omega(1)} \), given \( \Phi_r \) the coarse compacted trie of \( S_r \) can be computed in \( \mathcal{O}(b) \) time and space.
Proof. For each \( r \)-aligned position \( i \), we define its block as \( T[i..\min(n,i+[n/b]-1)] \). Then, we define a string \( T_r \) where \( T_r[j] \) is the fingerprint of the block corresponding to the \( j \)-th leftmost \( r \)-aligned position in \( T \). In other words, \( T_r \) is obtained from \( T \) by partitioning \( T[r..] \) into blocks of size \([n/b]\) and replacing each block by its fingerprint. Observe that this partition coincides with the partitions of \( r \)-aligned suffixes in the definition of the coarse compacted trie of \( S_r \). Thus, the coarse compacted trie of \( S_r \) is precisely the suffix tree of \( T_r \).

The first step of our construction algorithm is to compute \( T_r \). Using \( \Phi_r \), this takes \( \mathcal{O}(b) \) time since blocks are \( r \)-aligned fragments. Then, we sort the letters of \( T_r \) using Fact 3.2 with \( \ell = 1 \), \( \sigma = n^{\mathcal{O}(1)} \), and \( S = b \). This way, using \( \mathcal{O}(b \log_b n) = \mathcal{O}(b) \) time and \( \mathcal{O}(b) \) space, each character of \( T_r \) can be replaced by its rank, which is at most \( b \). After such normalization, we construct the suffix tree of \( T_r \) in \( \mathcal{O}(b) \) time and space. Finally, we replace the normalized characters on the edges by the original \( \mathcal{O}(\log n) \)-bit fingerprints.

Next, we implement the bulk insertion procedure. Note that coarse compacted trie of \( B_r \) can be extracted in \( \mathcal{O}(b) \) time from the coarse compacted trie of \( S_r \). We define \( B_{<r} = \bigcup_{j=1}^{r-1} B_j \).

Lemma 3.7. Assume that we have access to \( \Phi_r \) and an LCE-structure with \( T_{LCE} \) query time. Given the coarse compacted trie for \( B_{<r} \) and the coarse compacted trie for \( B_r \), we can construct the coarse compacted trie for \( B_{<r} \cup B_r \) in \( \mathcal{O}(b+|B_r|(T_{LCE}+n/b)) \) time and \( \mathcal{O}(b) \) space.

Proof. Intuitively, our aim is to traverse the Euler tour of the resulting coarse compacted trie of \( B_{<r} \cup B_r \). More precisely, we process consecutive strings \( s_1, \ldots, s_k \in B_r \) and for each \( s_i \) we find the locus of \( s_i \) in the trie of \( B_{<r} \cup \{s_1, \ldots, s_{i-1}\} \) and then add \( s_i \) to the trie. Strings \( s_i \) are processed according to the coarse lexicographic order, so we only need to move forward on the Euler tour to reach the next locus; actually, we can start from the new terminal node (representing \( s_{i-1} \)), possibly skipping some part of the Euler tour.

While visiting an edge \((u,v)\) whose label starts with a supercharacter \( c \), we shall find out if the locus of \( s_i \) is an implicit node on that edge. An equivalent condition is that \( \text{str}(u) \) is a prefix of \( s_i \) followed by a block represented by \( c \), but \( \text{str}(v) \) is not a prefix of \( s_i \). For this, we compute the fingerprints of the appropriate fragments of \( s_i \). Since \( s_i \in S_r \), these fragments are \( r \)-aligned in \( T \), so the fingerprints are determined in \( \mathcal{O}(1) \) time. If the locus turns out to be on the current edge, we make an LCE query to determine its exact depth; the result needs to be rounded down to full blocks.

Next, we introduce a new explicit node \( v' \) on the edge \((u,v)\) and a terminal node representing \( s_i \). Fingerprints stored at the new nodes and edges can be computed in \( \mathcal{O}(1) \) time, except for the one at \((v',v)\), which takes \( \mathcal{O}(n/b) \) time as the corresponding block may not be \( r \)-aligned.

Similarly, while at an explicit node \( v \), we first check if \( \text{str}(v) \) is a prefix of \( s_i \); otherwise, the subtree of \( v \) can be ignored. Next, we compute the supercharacter \( c \) representing the block of \( s_i \) following \( \text{str}(v) \) and compare it to the labels of edges from \( v \) to its children. Note that a vertex with \( d \) children appears on the Euler tour \( d+1 \) times, so during any visit to \( v \) we need to consider at most two edges whose labels are the lower bound and the upper bound on \( c \). If \( c \) turns out to be within the range, we simply introduce the leaf representing \( s_i \) as a child of \( v \). The new edge (labeled with \( c \)) is inserted between the two considered.

Hence, in either case it takes \( \mathcal{O}(1) \) time to make progress traversing the Euler tour, and at most \( \mathcal{O}(T_{LCE}+n/b) \) time to extend the trie. This sums up to \( \mathcal{O}(b+|B_r|(T_{LCE}+n/b)) \) in total.

Theorem 3.8. Assume that we have access to LCE-structure with \( T_{LCE} \) query time. The coarse compacted trie of any set \( B \) of \( b = n^{\mathcal{O}(1)} \) suffixes can be computed using \( \mathcal{O}(n+b \cdot T_{LCE}) \) time and \( \mathcal{O}(b) \) space.
Proof. We iterate over \( r = 1, \ldots, \lceil n/b \rceil \) while maintaining \( \Phi_r \). By Fact 3.4, this takes \( \mathcal{O}(n) \) time and \( \mathcal{O}(b) \) space in total. In the \( r \)-th round, we apply Lemma 3.6 to compute the coarse compacted trie of \( B_r \), and then we use Lemma 3.7 to merge it with the coarse compacted trie of \( B_{< r} \) to obtain the coarse compacted trie of \( B_{< r+1} \). The total cost of all applications of Lemma 3.7 is

\[
\sum_{r=1}^{\lceil n/b \rceil} \mathcal{O}(b + |B_r|(n/b + T_{\text{LCE}})) = \mathcal{O}(n + b \cdot (n/b + T_{\text{LCE}})) = \mathcal{O}(n + b \cdot T_{\text{LCE}}).
\]

\[\square\]

Corollary 3.9. For any set of \( b \) suffixes of \( T[1..n] \), the sparse suffix tree can be computed using \( \mathcal{O}(n + b \log b) \) time and \( \mathcal{O}(b) \) space. The resulting tree might be incorrect with probability \( n^{-c} \) for a user-defined constant \( c \).

Proof. The algorithm builds the coarse compacted trie of the suffixes. If \( b \leq \sqrt{n} \) is small, we use Theorem 3.5. Otherwise, we apply Theorem 3.8 with Lemma 2.4 to answer LCE queries. This results in \( T_{\text{LCE}} = \mathcal{O}(n/b + \log b) \), so the total running time is \( \mathcal{O}(n + b \log b) \). Finally, we construct the sparse suffix tree using Lemma 3.3. \[\square\]

3.3 More efficient LCE queries

In this section, we provide a faster data structure for LCE queries. Our solutions takes \( \mathcal{O}(b) \) space, has \( \mathcal{O}(n) \) construction time and \( \mathcal{O}(n/b) \) query time. When used in Theorem 3.8, the running time immediately improves to \( \mathcal{O}(n) \).

The main idea is similar to that by Bille et al. [5]: we use a sparse suffix tree for a \( t \)-difference cover to process LCE queries with answer at least \( t \) in constant time. Bille et al. used the algorithm by I et al. [17] to build the tree, which resulted in \( \mathcal{O}(n \log b) \) construction time. We devise a recursive approach: we construct sparse suffix arrays for larger and larger difference covers, using the previous one to speed up the construction of the next. The largest of these difference covers consists of \( \mathcal{O}(b^2/n) \) suffixes but we can still use \( \mathcal{O}(b) \) space. Thus, having relatively more space, we can apply a much simpler variant of the algorithm of Theorem 3.8.

Formally, at the \( i \)-th level of recursion, we have a set \( B^{(i)} \) of \( b_i = \mathcal{O}(b(n/b)^i) \) suffixes, which is a \( t_i \)-difference cover for \( t_i = (n/b)^{2i} \) obtained using Lemma 2.5. The recursion terminates at \( i = i_{\text{max}} \) such that \( (n/b)^{2i_{\text{max}}+2} > n \). Hence, \( t_i \leq n \) for \( 1 \leq i \leq i_{\text{max}} \).

Lemma 3.10 (see Section 3.6 in [5]). After \( \mathcal{O}(n) \)-time and \( \mathcal{O}(b) \)-space preprocessing, for every \( i, 1 \leq i \leq i_{\text{max}} \), the sparse suffix tree of \( B^{(i)} \) can be processed in \( \mathcal{O}(b_i) \) time so that LCE queries can be answered in \( \mathcal{O}(n/b + i \log n/b) \) time.

Proof. We store the structure of Lemma 2.4 and the component \( \Phi_1 \) (to test equality of fragments in \( \mathcal{O}(n/b) \) time). Given the sparse suffix tree of \( B^{(i)} \), we build the data structure of Lemma 2.1 for longest common prefix queries on \( B^{(i)} \). Next, we exploit the fact that \( B^{(i)} \) forms a difference cover.

To find \( \text{LCE}(p, q) \), we first test whether the result is at least \( h(p, q) \). This involves a single substring equality check. If the answer is positive, we compute the result in constant time as \( \text{LCE}(p, q) = h(p, q) + \text{LCE}(p + h(p, q), q + h(p, q)) \) with the second summand determined using the component of Lemma 2.1. Otherwise, we use the data structure of Lemma 2.4. The running time is \( \mathcal{O}(n/b + \log(b \cdot h(p, q)/n)) = \mathcal{O}(n/b + \log((n/b)(2^{i+1}))) = \mathcal{O}(n/b + i \log(n/b)) \).

Next, we observe that the approach of Lemma 3.3 can also be used to process a coarse compacted trie of \( b' < b \) suffixes in the trie.
Lemma 3.11. Let \( b', b \) be positive integers such that \( b' \leq b \leq n \) and \( b' = n^{O(1)} \). Given the coarse compacted trie of an arbitrary set of \( b' \) suffixes, we can construct their compacted trie in \( O(nb'/b) \) time and \( O(b) \) space.

Recall that for \( 1 \leq r \leq \lceil n/b \rceil \), we defined \( S_r \) as the set of \( r \)-aligned suffixes. We denote \( B_r^{(i)} = B^{(i)} \cap S_r \).

Lemma 3.12. The sparse suffix arrays of \( B_r^{(i)} \) (for \( 1 \leq i \leq i_{\max} \) and \( 1 \leq r \leq \lceil n/b \rceil \)) can be computed in \( O(n) \) time and \( O(b) \) space in total if \( b = n^{O(1)} \).

Proof. First, we use Lemma 2.5 to generate \( B^{(i)} \) for each \( i \). This takes \( O(\sum_i b_i) = O(b^2/n) \) time and space. Next, we lexicographically sort triples \( (j \mod \lceil n/b \rceil, j, i) \) for \( j \in B^{(i)} \) in \( O(b^2/n \log_b n + b) \) time and \( O(b) \) space using Fact 3.2 for \( \ell = 3 \), \( S = b \), and \( \sigma = n \). Finally, we iterate over \( r = 1, \ldots, \lceil n/b \rceil \) and construct the sparse suffix trees of \( B_r^{(i)} \) based on the coarse compacted tries of \( S_r \). Building the latter takes \( O(n) \) time and \( O(b) \) space across all iterations by Lemma 3.6 and Fact 3.4. We also extend the tree with the data structure of Lemma 2.1 for LCP queries. We use the previously constructed list of triples to build an array indexed by elements \( j \in S_r \) storing in each entry a list of integers \( i \) such that \( j \in B^{(i)} \). Next, we extract coarse compacted tries of \( B_r^{(i)} \) from the coarse compacted trie of \( S_r \). Finally, we use Lemma 3.11 to obtain the sparse suffix arrays of \( B_r^{(i)} \).

Theorem 3.13. There is a data structure of size \( O(b) \) which can be constructed in \( O(n) \) time and answers LCE queries in \( O(n/b) \) time. The data structure might be corrupted with probability \( n^{-c} \) for a user-defined constant \( c \).

Proof. If \( b \geq n/2 \), we use the standard \( O(n) \)-space data structure. Similarly, if \( b \leq n/\log n \), then the query time of the data structure of Lemma 2.4 is always \( O(n/b + \log b) = O(n/b) \), so there is nothing to do as well. Thus, we assume \( n/\log n < b < n/2 \).

First, we use Lemma 3.12 to compute the sparse suffix arrays of \( B_r^{(i)} \) for \( 1 \leq i \leq i_{\max} \) and \( 1 \leq r \leq \lceil n/b \rceil \). Then, we iterate over \( i = i_{\max}, \ldots, 1 \) and, for each such \( i \), merge the suffix arrays of all \( B_r^{(i)} \) to obtain the sparse suffix array of \( B^{(i)} \). This is easily achieved using \( O(b_i \log \frac{n}{b}) \) LCE queries. We answer these queries using the component of Lemma 2.4 for \( i = i_{\max} \) and Lemma 3.10 (plugging in the sparse suffix tree constructed for \( i + 1 \)) for \( i < i_{\max} \). In either case, the cost of the LCE query is \( O(n/b + (i + 1) \log (n/b)) = O(n/b + i \log (n/b)) \).

The space consumption is clearly \( O(b) \) throughout the algorithm. The overall construction time is dominated by on LCE queries, leading to the total running time of

\[
O \left( \sum_{i=1}^{i_{\max}} b_i \left( \frac{b}{b_i} + i \log \frac{b}{b_i} \right) \right) = O \left( \sum_{i=1}^{i_{\max}} b_i \cdot \frac{b}{b_i} \cdot i \right) = O \left( n \sum_{i=1}^{\infty} i \cdot \left( \frac{b}{n} \right)^i \right) = O(n). \]

\]

3.4 Summary

Theorem 3.14. For any set of \( b \) suffixes of \( T[1..n] \), the sparse suffix tree can be computed using \( O(n) \) time and \( O(b) \) space. The resulting array might be incorrect with probability \( n^{-c} \) for a user-defined constant \( c \).

Proof. First, we apply Theorem 3.13 to construct an efficient data structure for LCE queries. Next, the algorithm builds the coarse compacted trie of the suffixes. We use Theorem 3.5 if \( b^2 \leq n \) and Theorem 3.8 otherwise. Finally, we construct the sparse suffix tree using Lemma 3.3.
4 Las Vegas Algorithm

A simple way to obtain a Las Vegas algorithm is to run a Monte Carlo procedure and then verify if the result is correct. Thus, for each two adjacent suffixes $T[i..]$ and $T[j..]$ in the claimed lexicographic order, we need to check if their longest common prefix $\ell$ has been computed correctly and if $T[i..]$ is indeed lexicographically smaller than $T[j..]$. Equivalently, these conditions can be stated as $T[i..i+\ell-1] = T[j..j+\ell-1]$ and $T[i+\ell] < T[j+\ell]$. The latter constraint is trivial to verify, so the problem boils down to checking whether $T$ satisfies a system of $b-1$ substring equations.

We provide a deterministic $O(n\sqrt{\log b})$-time $O(b)$-space algorithm for that problem, improving upon an $O(n \log b)$-time solution by I et al. [17]. Let us start by recalling a simpler $O(n \log^2 b)$-time version of their algorithm. A straightforward reduction allows to restrict the lengths of all equations to $3 \cdot 2^k$ for integer values $k$. Now, the main idea is to relax the problem: a YES answer is required if all equations are satisfied, but a NO—only if there is a mismatch within the first $2 \cdot 2^k$ positions of an equation. The exact problem easily reduces to two instances of the relaxed version: one on the original text and one on its reverse. Then, the algorithm of I et al. works in $O(\log b)$ phases. Each phase is given equations of length $3 \cdot 2^k$ and it is responsible for making sure that there are no mismatches within the middle $2^k$ positions. If so, the equations can be shortened to $3 \cdot 2^{k-1}$. Once the maximum length is sufficiently small, all equations are verified naively.

In each phase, a graph $G$ is constructed, with nodes corresponding to blocks of the text. Each edge represents an equation and connects nodes corresponding to blocks containing the starting positions of the involved fragments. Then, $O(|V(G)|)$ constraints are verified naively. These are original equations forming an $O(\log |V(G)|)$-spanner of the graph, as well as constraints stating that certain fragments of the text have a given period (such constraints can also be expressed as substring equations). This turns out to be sufficient to implement the phase.

We introduce a few important changes to the above algorithm. First, we avoid the eager verification of the $O(|V(G)|)$ equations. Instead, we split them into several equations of length $3 \cdot 2^{k-1}$ and process them in further phases as if they were given in the input. Secondly, we observe that we can create these equations so that they have slack on both ends: just the middle $2^{k-1}$ positions need to be checked. Thus, we change the original relaxation to work with such equations only. Some technical details (e.g., the reduction from the exact problem) become more complex, but now each phase simply converts a system of (relaxed) equations of length $3 \cdot 2^k$ to a system of (relaxed) equations of length $3 \cdot 2^{k-1}$. This way, we achieve $O(n \log b + b \log^2 n)$ running time.

The notion of difference covers lets us further exploit the slack at both sides of the equations and consequently reduce the running time to $O(n \sqrt{\log b})$. We perturb each of them so that the lengths are still uniform, but the starting positions of both fragments involved belong to a certain set. First, this approach lets us restrict the set of starting positions of long equations, and consequently remove most of them. (Detecting irrelevant equations also involves computing the maximum-weight spanning forest in a certain graph.) Secondly, we reduce the number of blocks where starting positions appear, i.e., the number of vertices of the graph $G$ for which the spanner is constructed. This results in fewer equations being created in each phase.

In the following sections, we formalize the above discussion. In Section 4.1 we introduce the notion of substring equation and we prove basic facts. In Sections 4.2 and 4.3, we formalize further tools already used by I et al., originating in combinatorics of strings and graph spanners, respectively. Next, in Section 4.4 we provide our first solution, whose running time $O(n \log b + b \log^2 n)$ is comparable to the previous state of the art, and in Section 4.5 we improve the time complexity to $O(n \sqrt{\log b})$. Both algorithms require $O(b)$ space.
4.1 Systems of Substring Equations

Consider texts of length \( n \). For any integers \( p, q, p', q' \) such that \( p, p' \geq 1, q, q' \leq n \), and \( q - p = q' - p' \), we say that \( e : T[p..q] = T[p'..q'] \) is a substring equation. The quantity \( q - p + 1 = q' - p' + 1 \) is called the length of the equation and \( p' - p = q' - q \) is called the shift. Note that shift is oriented: its sign is reversed when we write \( e \) as \( T[p'..q'] = T[p..q] \). A particular text \( T \) satisfies the equation whenever \( T[p..q] \) and \( T[p'..q'] \) are occurrences of the same string. Equations of length 0 or less are assumed to be trivially satisfied. For an integer \( S \) we say that \( T \) satisfies \( e : T[p..q] = T[p'..q'] \) with shortage \( S \) whenever \( T \) satisfies \( T[p + S..q - S] = T[p' + S..q' - S] \).

Intuitively, our algorithm internally works with relaxed equations: we can verify if they are satisfied (with no shortage), but it suffices to check if they are satisfied with some shortage \( S \), not necessarily uniform across all equations. This justifies the structure of most auxiliary results.

The following fact describes how a single relaxed equation can be replaced by a system of shorter relaxed equations. If \( p = p_1 \) and \( q = q_1 \), we can think of this procedure as splitting \( e \) into \( e_1, \ldots, e_m \).

**Fact 4.1.** Let \( e, e_1, \ldots, e_m \) (\( e : T[p..q] = T[p'..q'] \), \( e_i : T[p_i..q_i] = T[p_i'..q_i'] \)) be substring equations with the same shift such that \( p \leq p_i \) and \( q_i \leq q \) for \( 1 \leq i \leq m \). Moreover, let \( S_i \) (\( 1 \leq i \leq m \)) be non-negative integers such that \( p_{i+1} + S_{i+1} \leq q_i - S_i + 1 \) for \( 1 \leq i < m \).

(a) If \( e \) is satisfied, then \( e_1, \ldots, e_m \) are satisfied.

(b) If each \( e_i \) is satisfied with shortage \( S_i \), then \( e \) is satisfied with shortage \( S = \max(p_1 - p + S_1, q - q_m + S_m) \).

**Proof.** (a) For \( 1 \leq i \leq m \), \( T[p_i..q_i] \) is clearly a subfragment of \( T[p..q] \). Since \( e_i \) has the same shift as \( e \), \( T[p_i'..q_i'] \) is the corresponding subfragment of \( T[p'..q'] \). Thus, \( e_i \) must be satisfied if \( e \) is.

(b) We inductively prove that \( T[p_1 + S_1..q_1 - S_1], \ldots, T[p_m + S_m..q_m - S_m] \) cover \( T[p_1 + S_1..q_m - S_m] \). By definition of \( S_i \), the later covers \( T[p + S..q - S] \). Since all equations have the same shift, this yields (b).

A set of substring equations on the same text \( T \) is called a system. A particular text \( T \) satisfies the system (with shortage \( S \)) if it satisfies all its member equations (with shortage \( S \) resp.). A system is called uniform if all equations in the system have the same length. Uniform systems can be obtained as follows by splitting longer equations using Fact 4.1:

**Corollary 4.2.** Let \( e \) be an equation of length \( L \). For every positive integer \( \ell \), there exists a uniform system \( E \) of \( \Theta(L) \) equations of length \( \ell \) such that if \( e \) is satisfied, then \( E \) is satisfied, and if \( E \) is satisfied with shortage \( S \) (\( S \leq \frac{L}{3} \)), then \( e \) is also satisfied with shortage \( S \). Moreover, \( E \) can be computed in \( \Theta(L) \) time.

**Proof.** Let \( e : T[p..q] = T[p'..q'] \) and let \( r = \max(1, \lfloor \frac{L}{3} \rfloor) \). We include in \( E \) equations of length \( \ell \) and start positions \( (p + ir, p' + ir) \) for \( 0 \leq i < \frac{L - \ell}{\ell} \), as well as \((q - \ell + 1, q' - \ell + 1) \). It is easy to see that these equations satisfy Fact 4.1 with \( S_i = S \) provided that \( 2S \leq \ell - r \) (e.g., if \( S \leq \frac{L}{3} \)).

The following result, in a simpler version introduced by I et al. [17], relates a cycle of substring equations with periods of certain fragments.
Lemma 4.3. Let $E = \{e_1, \ldots, e_m\}$ be a uniform system of substring equations with $e_i : T[p_i..q_i] = T[p'_i..q'_i]$. Assuming $p_{m+1} = p_1$, let us define $r = \left| \sum_{i=1}^m (p_{i+1} - p'_i) \right|$ and $R = \sum_{i=1}^m |p_{i+1} - p'_i|.$

(a) If $E$ is satisfied, then $r$ is a period of $T[p_1 + R..q_1 - R]$. 

(b) For a positive integer $S$, if $r$ is a period of $T[p_1 + S..q_1 - S]$ and $E \setminus \{e_1\}$ is satisfied with shortage $S$, then $e_1$ is satisfied with shortage $R + S$.

Proof. For $i \in \{1, \ldots, m\}$ let us define $r_i = \sum_{j=1}^i (p_{j+1} - p'_j)$; we also have $r_i = \sum_{j=1}^i (q_{j+1} - q'_j)$, since the system is uniform. Note that $r = |r_m|$ and that $|r_i| \leq R$. Before we proceed, let us show the following auxiliary result:

Claim. If $E \setminus \{e_1\}$ is satisfied with shortage $S$, then $T[p'_i + R + S..q'_i - R - S] = T[p_i + r_i + R + S..q_i - r_i - R - S]$ for each $i \in \{1, \ldots, m\}$.

Proof of the claim. We provide an inductive proof. For $i = 1$, equalities $p_2 = p'_1 + r_1$ and $q_2 = q'_1 + r_1$ trivially yield the inductive base. Now, suppose that the claim is satisfied for $i - 1$, i.e., $T[p'_i + R + S..q'_i - R - S] = T[p_i - r_{i-1} + R + S..q_i - r_{i-1} - R - S]$. Since $|r_{i-1}| \leq R$, the fact that $e_i$ is satisfied with shortage $S$ implies $T[p'_i + R + S..q'_i - R - S] = T[p_i - r_{i-1} + R + S..q_i - r_{i-1} - R - S]$. Equalities $p_{i+1} = p'_i + r_i$ and $q_{i+1} = q'_i + r_i$ further yield $T[p'_i + R + S..q'_i - R - S] = T[p_i + r_i + R + S..q_i - r_i - R - S]$, which concludes the proof of the claim. \qed

Let us proceed to the proofs of (a) and (b).

(a) The claim implies $T[p'_1 + R..q'_1 - R] = T[p_1 - r_m + R..q_1 - r_m - R]$, while $e_1$ further yields $T[p_1 + R..q_1 - R] = T[p_1 - r_m + R..q_1 - r_m - R]$. Consequently, $r = |r_m|$ must be a period of $T[p_1 + R..q_1 - R]$. 

(b) The claim implies $T[p'_1 + R + S..q'_1 - R - S] = T[p_1 - r_m + R + S..q_1 - r_m - R - S]$. The fact that $T[p_1 + S..q_1 - S]$ has period $r = |r_m| \leq R$, yields $T[p_1 - r_m + R + S..q_1 - r_m - R - S] = T[p_1 + R + S..q_1 - R - S]$. Combining these two equalities, we conclude that $T[p'_1 + R + S..q'_1 - R - S] = T[p_1 + R + S..q_1 - R - S]$, i.e., that $e_1$ is satisfied with shortage $R + S$. \qed

4.2 Period constraints

The fact that a fragment $T[p..q]$ has period $r$ can be expressed as a substring equation $T[p..q - r] = T[p + r..q]$. This condition is particularly important if $2r \leq |T[p..q]|$, i.e., if $r$ does not exceed the length of the underlying equation. Thus, if for an equation $e : T[p..q] = T[p'..q']$ the absolute value of the shift does not exceed the length, we say that the equation is a (substring) period constraint which concerns $T[min(p, p')..max(q, q')]$ and enforces period $|p - p'|$. Periodicity Lemma, a classic tool of combinatorics of words, lets us essentially reduce several period constraints enforcing different periods with a single period constraint enforcing the greatest common divisor of these periods.

Lemma 4.4 (Periodicity Lemma, Fine and Wilf [11]). If a string $x$ has length $n$ and periods $p, q$ such that $n \geq p + q - \gcd(p, q)$, then it also has $\gcd(p, q)$ as a period.

Lemma 4.5. Let $E = \{e_1, \ldots, e_m\}$ be a system of period constraints concerning a fragment $T[p..q]$ and enforcing periods $r_1, \ldots, r_m$, and let $e$ be a constraint enforcing $r = \gcd(r_1, \ldots, r_m)$ as a period of $T[p..q]$.

(a) $E$ is satisfied if and only if $e$ is satisfied.
(b) For every positive integer $S$, if $e$ is satisfied with shortage $S$, then $E$ is satisfied with shortage $S$.

Proof. Note that $e : T[p..q − r] = T[p + r..q]$ is satisfied with shortage $S$ whenever $T[p + S..q − S]$ has period $r$ (not necessarily proper). Thus, the fact that $e$ is satisfied with shortage $S$ means that $T[p + S..q − S]$ has period $r$ and consequently all periods $r_i$ (as $r \mid r_i$). Consequently, $E$ is satisfied with storage $S$, as claimed in (b). The ‘if’ part of (a) is a special case for $S = 0$, so let us proceed to the proof of the ‘only if’ part. Suppose that $E$ is satisfied. Iteratively applying Periodicity Lemma (Lemma 4.4) to $w = T[p..q]$ with periods $r_i$ and gcd$\left(r_1, \ldots, r_{n-1}\right)$, we conclude that gcd$\left(r_1, \ldots, r_m\right)$ is a period of $T[p..q]$, i.e., that $e$ is satisfied. Note that gcd$\left(r_1, \ldots, r_{n-1}\right) + r_i \leq r_{n-1} + r_i \leq |T[p..q]|$ due to the fact that $r_i \leq \frac{1}{m}|T[p..q]|$ by definition of a period constraint.

4.3 Graph Spanners

In this section, we recall a classic construction of a $(2\lceil \log n \rceil - 1)$-multiplicative graph spanner of size at most $2n$. The underlying original idea is by Awerbuch [2]. These spanners have already been implicitly applied in [17] to verify substring equations. Our setting requires some extra information concerning the paths witnessing that each edge has low enough stretch. Thus, we formulate the algorithm in a non-standard way, formalizing the contribution of [17].

Let $G = (V, E)$ be an undirected multigraph. We interpret each edge of $G$ as a pair of inverse arcs. For a subset $F \subseteq E$, we define $\tilde{F}$ to be the set of arcs corresponding to the edges in $F$. For an arc $e \in \tilde{E}$, we define $e^R$ as the arc inverse to $e$ (this way $(e^R)^R = e$). We say that $w : \tilde{E} \to \mathbb{Z}$ in an oriented weight function if $w(e^R) = -w(e)$ for each $e \in \tilde{E}$. The oriented weight function can be naturally extended to (oriented) paths and cycles: we set $w(C) = \sum_{e \in C} w(e)$.

**Lemma 4.6 ([17]).** Let $G = (V, E)$ be an undirected multigraph with oriented weight function $w$. In $O(|V| + |E|)$ time, one can compute a subset $E' \subseteq E$ of size at most $2|V|$ and an oriented weight function $c$ which satisfy the following condition: for each arc $e \in \tilde{E}$, there exists a directed cycle $C \subseteq \tilde{E}' \cup \{e\}$ containing $e$ such that $|C| \leq 2 \log |V|$ and $c(e) = w(C)$.

Proof. Let $v$ be an arbitrary vertex and, for a non-negative integer $i$, let $V_i$ be the set of vertices whose (unweighted) distance from $v$ is at most $i$. Let $d$ be the smallest positive such that $|V_d| \leq 2|V_{d-1}| + 1$. Note that $|V_i| \geq 2|V_{i-1}| + 2$ for $i < d$ and thus $d \leq \lceil \log |V| \rceil$.

Observe that breadth-first search starting from $v$ can be used to determine $d$ and $V_d$. Moreover, it results in a spanning tree of $G[V_d]$ rooted at $v$; we shall include the tree-edges in $E'$. Each arc $e$ incident to at least one vertex in $V_{d-1}$ can be completed to a cycle $C$ using a tree-path of length at most $2d - 1$. Note that after linear-time preprocessing of the tree (to compute the weight $w(P_u)$ of the tree-path from $u$ to $v$ for every vertex $u \in V_d$), the weight $w(C)$ can be computed in constant time (as $w(e) + w(P_{u'}) - w(P_u)$ for an arc $e$ from $u$ to $u'$). We shall set $c(e) = w(C)$ for these arcs.

The procedure above (including BFS) runs in time proportional to the number of arcs incident to $V_{d-1}$, and it lets us remove $V_{d-1}$ from the graph. The number of edges added to $E'$ is at most $|V_d| - 1 \leq 2|V_{d-1}|$, i.e., twice as much as the number of removed vertices. Thus, if we recursively continue the same procedure, we get a set $E'$ of size at most $2|V|$ and an oriented weight function $c$ satisfying the desired condition.
4.4 $\mathcal{O}(n \log b + b \log^2 n)$-time Solution

For a uniform system $E$ of equations of length $L$ and block size $B$, we construct a multigraph graph $G_{E,B}$ with $\mathcal{O}\left(\frac{n}{B}\right)$ vertices and edge set identified with $E$, and an oriented weight $w$ function corresponding to the shift of the equations.

Vertices of $G_{E,B}$ correspond to consecutive blocks of $T$ starting at *aligned* positions of the form $1 + kB$, and are represented by these positions. For a position $p$, we denote the preceding aligned position by $\text{pred}(p)$ ($p - B < \text{pred}(p) \leq p$). An equation $e : T[p..q] = T[p'..q']$ is represented by an edge between $\text{pred}(p)$ and $\text{pred}(p')$. The weights of the underlying arcs are $p' - p$ (for the arc from $\text{pred}(p)$) and $p - p'$ (for the arc from $\text{pred}(p')$). We discard isolated vertices from $G_{E,B}$, which lets us store this graph in $\mathcal{O}(|E|)$ space.

**Observation 4.7.** The graph $G_{E,B}$ and the weight function $w$ can be constructed in $\mathcal{O}(|E| + \frac{n}{B})$ time.

A spanner of the graph $G_{E,B}$, combined with Lemmas 4.3 and 4.5, can be used to reduce a uniform system of relaxed substring equations to a system of shorter relaxed substring equations. This result generalizes the main concept of the $\mathcal{O}(n \log^2 b)$-time algorithm of I et al. [17].

**Lemma 4.8.** Let $E$ be a system of substring equations of length $L$ on a text of length $n$. For a fixed block-size $B$, let $N$ be the number of vertices of the graph $G_{E,B}$. Moreover, let $S$ be a positive integer such that $B(2\lceil \log N \rceil + 1) \leq S \leq \frac{L}{4}$. There is a system $\tilde{E}$ of $\mathcal{O}(N)$ equations of length between $L - 3S$ and $L$ such that:

(a) if $E$ is satisfied, then $\tilde{E}$ is satisfied,

(b) if $\tilde{E}$ is satisfied with shortage $S'$, then $E$ is satisfied with shortage $S' + 2S$.

Moreover, such $\tilde{E}$ can be computed in $\mathcal{O}(|E| \log S)$ time and $\mathcal{O}(|E|)$ space given the graph $G_{E,B}$.

**Proof.** For every vertex (represented by aligned position $p$), we define a canonical fragment $T_p = T[p + S..p + L - 1 - S]$. We apply Lemma 4.6 to $G_{E,B}$, which results in a subset $E' \subseteq E$ of size $2N$ and an oriented weight function $c$. For every (oriented) equation $e : T[p..q] = T[p'..q']$ in $E$, we consider a period constraint $e''$ enforcing $|c(e)|$ as a period of $T_{\text{pred}(p)}$.

**Claim.** If $E$ is satisfied, then $e''$ is also satisfied. Moreover, if $E' \cup \{e''\}$ is satisfied with shortage $S'$, then $e$ is satisfied with shortage $S' + 2S$.

**Proof.** Let $C = (e_1, \ldots, e_m)$ be the witness cycle for $e$ with $e = e_1$ and $e_i : T[p_i..q_i] = T[p'_i..q'_i]$. We shall apply Lemma 4.3 for equations in $C$. Note that $c(e) = \sum_{i=1}^{m} (p'_i - p_i) = \sum_{i=1}^{m} (p'_i - p_{i+1})$ and thus $|c(e)| = r$. Moreover, since $\text{pred}(p_{i+1}) = \text{pred}(p'_i)$, we have $|p'_i - p_{i+1}| \leq B$, i.e., $r \leq R \leq |C|B$. Since $|C| \leq 2\lceil \log N \rceil$, this yields $S \geq R + B$.

Let $T_{\text{pred}(p)} = T[p..q]$. We shall first prove that $T_{\text{pred}(p)}$ is contained in $T[p + R..q - R]$ and contains $T[p + 2S - R..q - 2S + R]$. First, note that $\bar{p} = \text{pred}(p) + S \leq p + S \leq p + 2S - R$ and $\bar{q} = \text{pred}(p) + L - 1 - S \leq p + L - 1 - S = q - S \leq q - R$ since $S \geq R$. Similarly, $\bar{p} = \text{pred}(p) + S \geq p - B + S \geq p + R$ and $\bar{q} = \text{pred}(p) + L - 1 - S \geq p + L - 1 - B - S = q - S - B \geq q - 2S + R$ since $S \geq B + R$.

Suppose that $E$ is satisfied. Since $C \subseteq E$, Lemma 4.3(a) implies that $T[p + R..q - R]$, and thus also $T_{\text{pred}(p)}$, has period $r$. Hence, $e''$ is satisfied.

14
Now, suppose that \( E' \cup \{e''\} \) is satisfied with shortage \( S' \). This yields that \( C \setminus \{e\} \) is satisfied with shortage \( S' + 2S - R \) and that \( T[p + S' .. q - S'] \), i.e., also \( T[p + (S' + 2S - R) .. q - (S' + 2S - R)] \), has period \( r \). Consequently, Lemma 4.3(b) implies that \( e \) is satisfied with shortage \( R + (S' + 2S - R) = S' + 2S \).

Finally, note that the shift of the equation \( e'' \) is \( |c(e)| \leq S \), while the length is \( L - 2S - c(e) \leq L - 3S \). Since we assumed that \( S \leq \frac{L}{4} \), the equation \( e'' \) is indeed a period constraint concerning a canonical fragment.

Hence, the system \( E' \cup E'' \) with \( E'' = \{e'' : e \in E\} \) satisfies the claimed conditions on \( \bar{E} \) except that that its size is \( |E| + O(g) \) rather than \( O(g) \). For each canonical fragment, Lemma 4.5 lets us replace all period constraints concerning this fragment with a single constraint. As far as the implementation is concerned, this requires a single gcd query per period constraint. Since the periods are bounded by \( S \), Euclid’s algorithm takes \( O(\log S) \) time, which gives \( O(|E| \log S) \) total running time.

**Corollary 4.9.** Let \( k \geq 2 \) be a positive integer, and let \( E \) be a system of substring equations of length \( 3 \cdot 2^k \) on a text \( T \) of length \( n \). There exists a system \( E' \) of \( O(\frac{n}{2^k} \log \frac{n}{2^k}) \) substring equations of length \( 3 \cdot 2^{k-1} \) such that:

(a) if \( E \) is satisfied, then \( E' \) is satisfied;

(b) if \( E' \) is satisfied with shortage \( 2^{k-1} \), then \( E \) is satisfied with shortage \( 2^k \).

Moreover, such \( E' \) can be computed in \( O(|E|k + \frac{n}{2^k} \log \frac{n}{2^k}) \) time and \( O(|E| + \frac{n}{2^k} \log \frac{n}{2^k}) \) space.

**Proof.** We set \( L = 3 \cdot 2^k \), \( S = 2^{k-1} \), and the block size \( B \) as the largest integer such that \( B(2\lceil \log \frac{n}{2^k} \rceil + 1) \leq S \). We construct \( G_{E,B} \) using Observation 4.7. The number of vertices satisfies \( N = \lceil \frac{n}{2^k} \rceil = \Theta(\frac{n}{2^k} \log \frac{n}{2^k}) \). We have \( B(2\lceil \log N \rceil + 1) \leq S \leq \frac{L}{4} \), so we can apply Lemma 4.8 to obtain a set system \( E \) containing \( O(N) \) equations of length between \( L - 3S = 9 \cdot 2^{k-2} \) and \( L = 12 \cdot 2^{k-2} \).

Next, we use Corollary 4.2 for each equation in \( \bar{E} \) (with \( \ell = 3 \cdot 2^{k-1} \)) to obtain a system \( E' \) consisting of equations of length \( 3 \cdot 2^{k-1} \). Clearly, if \( \bar{E} \) is satisfied, then also \( \bar{E} \) and \( E' \) are. On the other hand, if \( E' \) is satisfied with shortage \( 2^{k-1} \), then \( \bar{E} \) is satisfied with shortage \( 2^{k-1} \), and consequently \( E \) is satisfied with shortage \( 2^{k-1} + 2 \cdot 2^{k-2} = 2^{k} \), as claimed.

The running time \( O(|E| \log S + N) \) and the space consumption \( O(|E| + N) \) are as claimed.

The following result, complementary to Corollary 4.9, when applied recursively, lets us reduce a given system of substring equations to \( O(\log n) \) systems of uniform relaxed systems.

**Lemma 4.10.** Let \( k \) be a non-negative integer and let \( E \) be a system of \( b \) substring equations on a text \( T \), each of length at least \( 3 \cdot 2^k \). In \( O(b) \) time one can output systems \( E' \) and \( E'' \) such that:

(a) \( E' \) contains \( O(b) \) equations, each of length exactly \( 3 \cdot 2^k \);

(b) \( E'' \) contains at most \( b \) equations, each of length at least \( 3 \cdot 2^{k+1} \);

(c) if \( E \) is satisfied, then \( E' \) are \( E'' \) are also satisfied;

(d) if \( E' \) is satisfied with shortage \( S \), \( S \leq 2^k \), and \( E'' \) is satisfied with shortage \( 2^{k+1} \), then \( E \) is satisfied with shortage \( S \).
Proof. Note that it suffices to prove the lemma for \( b = 1 \); for \( b > 1 \), we construct \( E' \) and \( E'' \) as unions of the systems generated for each equation \( e \in E \). Thus, we assume that \( E \) consists of a single equation \( e \).

Let \( L \) be the length of \( e \). If \( L < 3 \cdot 2^{k+1} \), then we apply Corollary 4.2, return the resulting system as \( E' \), and set \( E'' = \emptyset \). Conditions (c) and (d) are satisfied because \( S \leq 2^k = \frac{1}{3} \cdot 3 \cdot 2^k \).

Thus, we may assume that \( e : T[p..q] = T'[p'..q'] \) has length \( L \geq 3 \cdot 2^{k+1} \). We consider a sequence of three equations \( e_1 : T[p..p + 3 \cdot 2^k - 1] = T[p'..p' + 3 \cdot 2^k - 1] \) holds for increasing values of \( c \).

Note that the systems \( E \) satisfied and each \( k > \ell \) are actually satisfied with no shortage. For \( k \leq \ell \), we have that both \( S_1 = S, S_2 = 2^{k+1} \), and \( S_3 = S \). Since \( S \leq 2^k \), the sequence clearly satisfies Fact 4.1, which yields (c) and (d).

Now, we are ready to describe the first version of our verification algorithm.

**Theorem 4.11.** A system \( E \) of \( b \) substring equations on a text \( T \) of length \( n \) can be verified \( O(n \log b + b \log^2 n) \) time using \( O(b) \) space.

**Proof.** We set \( \ell = \lfloor \log \frac{n \log b}{b} \rfloor \) and naively check all equations shorter than \( 3 \cdot 2^\ell \). This takes \( O(b \cdot \frac{n \log b}{b}) = O(n \log b) \) time. From now on, we assume that all equations are of length at least \( 3 \cdot 2^\ell \).

The equations are processed iteratively applying Lemma 4.10 for increasing values of \( k \), starting from \( k = \ell \), which results in systems \( E_\ell, \ldots, E_r \) \((r = \lfloor \log \frac{n}{b^k} \rfloor)\), such that each system \( E_k, \ell \leq k \leq r \), contains \( O(b) \) equations, each of length \( 3 \cdot 2^k \). Moreover, condition (c) of Lemma 4.10 implies that if the system \( E \) is satisfied, then all systems \( E_k \) is satisfied. Moreover, by condition (d), if \( E_\ell \) is satisfied and each \( E_k, \ell < k \leq r \), is satisfied with shortage \( 2^k \), then the input system \( E \) is satisfied.

Note that the systems \( E_k \) together do not fit in \( O(b) \) space, but each of them can be generated in \( O(b(\ell - \ell + 1)) = O(b \log n) \) time.

We process systems \( E_k \) for decreasing values \( k \). We define \( F_r = \emptyset \) and for \( \ell \leq k < r \), we define \( F_k \) as the effect of application of the reduction of Corollary 4.9 to \( E_{k+1} \cup F_{k+1} \). Note that \( F_k \) is a uniform system with \( O(b + \frac{n \log b}{2^k}) \) equations of length \( 3 \cdot 2^k \). Since \( \frac{n}{2^k} = \Theta(b \log^2 b) \), we actually have \( |F_k| = O(b) \) for each \( k \).

We shall prove that \( E \) is satisfied if and only if \( E_\ell \) and \( F_\ell \) are both satisfied. The ‘only if’ part is easier: we have already observed that that systems \( E_k \) are satisfied for \( \ell \leq k \leq r \). The fact that the systems \( F_k \) are satisfied is proved by induction for decreasing \( k \). For \( k = r \) this is trivial. For \( k < r \), we have that both \( E_{k+1} \) and \( F_{k+1} \) are satisfied, and Corollary 4.9 implies that \( F_k \) must also be satisfied. In the ‘if’ part we inductively prove that the systems \( E_k \) and \( F_k \) are satisfied with shortage \( 2^k \), this time for increasing values of \( k \). For \( k = \ell \) this is trivial, since they are actually satisfied with no shortage. For \( k > \ell \), we deduce from Corollary 4.9 that the fact that \( F_{k-1} \) is satisfied with shortage \( 2^{k-1} \) yields that \( F_k \cup E_k \) is satisfied with shortage \( 2^k \). Finally, by the analysis in the first part of the proof, we note that \( E \) must be satisfied, as claimed.

Consequently, in order to verify if \( E \) is satisfied, we can equivalently verify \( E_\ell \cup F_\ell \). The latter systems contains \( O(b) \) equations, each of length \( O(\frac{n \log b}{b}) \), so this takes \( O(n \log b) \) time and \( O(b) \) space. The construction of each \( F_k \) from \( F_{k+1} \) takes \( O(b) \) space and \( O(b \log n) \) time, which sums up to \( O(b \log^2 n) \) in total.

**4.5 \( O(n \sqrt{\log b}) \)-time solution**

We reduce the running time of the algorithm by applying difference covers to improve Corollary 4.9 and Lemma 4.10. We start with a counterpart of Corollary 4.9.
Lemma 4.12. Let \( k \geq 3 \) be a positive integer, and let \( E \) be a system of substring equations of length \( 3 \cdot 2^k \) on a text \( T \) of length \( n \). There exists a system \( E' \) of \( O\left(\frac{n}{2^k} \sqrt{\log \frac{2^k}{3}}\right) \) substring equations of length \( 3 \cdot 2^{k-1} \) such that:

(a) if \( E \) is satisfied, then \( E' \) is satisfied;

(b) if \( E' \) is satisfied with shortage \( 2^{k-1} \), then \( E \) is satisfied with shortage \( 2^k \).

Moreover, such \( E' \) can be computed in \( O(|E| \cdot \frac{n}{2^k} \sqrt{\log \frac{2^k}{3}}) \) time and \( O(|E| + \frac{n}{2^k} \sqrt{\log \frac{2^k}{3}}) \) space.

Proof. We set \( L = 3 \cdot 2^k \), \( S = 2^{k-3} \), and the block size \( B \) as the largest integer such that \( B(2[\log \frac{n}{SB} + 1] \leq S \). However, before building the graph \( G_{E,B} \), we construct a \([\frac{n}{SB}]\)-difference cover of \( \{1, \ldots, [\frac{n}{SB}]\} \) using Lemma 2.5; its size is \( O\left(\frac{n}{\sqrt{SB}}\right) \). We slightly abuse the notation and define the shift function \( h \) so that its arguments are aligned positions, and the values are multiples of \( B \).

We transform \( E \) into a system \( \tilde{E} \) of equations of length \( L - S \): for an equation \( e : T[p..q] = T[p'..q'] \) we compute \( h_e = h(\text{pred}(p), \text{pred}(p')) \) and construct \( e' : T[p + h_e..q + h_e - S] = T[p' + h_e..q' + h_e - S] \). Clearly, if \( e \) is satisfied, then \( e' \) is satisfied, and if \( e' \) is satisfied with shortage \( 2^{k-1} \), then \( e \) is satisfied with shortage \( 2^{k-1} + 2^{k-3} \).

After this operation, we are guaranteed that there are only \( N = O\left(\frac{n}{SB} \cdot \sqrt{\frac{n}{SB}}\right) \) (non-isolated) vertices in \( G_{E,B} \). Moreover, the fact that the difference cover can be indexed efficiently, implies that \( G_{E,B} \) can be constructed in \( O(|E| + \frac{n}{SB} \sqrt{\log \frac{2^k}{3}}) \) time. Note that \( B(2[\log \frac{n}{SB} + 1] \leq B(2[\log N] + 1) \leq S = \frac{L-S}{4} \), so we may apply Lemma 4.8 to \( \tilde{E} \), which results in a system \( \bar{E} \).

The lengths of equations in \( \bar{E} \) are between \( L - 4S = 20 \cdot 2^{k-3} \) and \( L - S = 23 \cdot 2^{k-3} \); we apply Corollary 4.2 to each of them in order to obtain a uniform system \( E' \) with equations of length \( 3 \cdot 2^{k-1} \).

If \( E \) is satisfied, then \( \bar{E}, \bar{E}, \) and \( E' \) are all satisfied. On the other hand, if \( E' \) is satisfied with shortage \( 2^{k-1} \), then \( \bar{E} \) is satisfied with shortage \( 2^{k-1} \), \( \bar{E} \) is satisfied with shortage \( 2^{k-1} + 2 \cdot 2^{k-3} \), and \( E \) is satisfied with shortage \( 2^{k-1} + 3 \cdot 2^{k-3} \leq 2^k \), as claimed. \( \square \)

Before we state a stronger version of Lemma 4.10, let us show an auxiliary result, based on the notion of the minimum spanning forest of a graph.

Lemma 4.13. Let \( E \) be a system of \( b \) substring equations on a text \( T \) of length \( n \) and let \( DC \) be a subset of \( \{1, \ldots, n\} \) which can be indexed efficiently and contains \( p \) and \( p' \) for each \( e : T[p..q] = T[p'..q'] \) in \( E \). In \( O(b + |DC|) \) time we can compute a subsystem \( E' \subseteq E \) of size at most \( |DC| \), such that for any shortage \( S \), \( E' \) is satisfied with shortage \( S \) if and only if \( E \) is satisfied with shortage \( S \).

Proof. We construct a weighted undirected multigraph \( G \) with vertex set \( DC \). Each equation \( e : T[p..q] = T[p'..q'] \) is represented as an edge between \( p \) and \( p' \) with weight \( q - p + 1 = q' - p' + 1 \) equal to the length of the equation. The graph can be built in \( O(b + |DC|) \) time, since \( DC \) can be indexed efficiently.

We compute a maximum-weight spanning forest of \( G \) (using a linear-time algorithm of Fredman and Willard [13]) and return the subsystem \( E' \) corresponding to the edges of the forest.

It suffices to prove that equations in \( E \setminus E' \) can be dropped. For such an equation \( e : T[p..q] = T[p'..q'] \) of length \( L \), there is a sequence of equations \( e_1, \ldots, e_m \in E' \) (\( e_i : T[p_i..q_i] = T[p'_i..q'_i] \)) of length at least \( L \) such that \( p_1 = p, p'_m = p' \), and \( p_{i+1} = p_i \) for \( 1 \leq i < m \). Clearly, if \( E' \) is satisfied with shortage \( S \), then \( e_1, \ldots, e_m \) are all satisfied with shortage \( S \), and consequently
Lemma 4.14. Let $k$ be a non-negative integer and let $E$ be a system of $b$ substring equations on a text $T$, each of length at least $3 \cdot 2^k$. In $O(b)$ time one can output systems $E'$ and $E''$ such that

(a) $E'$ contains $O(b)$ equations, each of length exactly $3 \cdot 2^k$;
(b) $E''$ contains $\min(b, O(\frac{n}{\sqrt{2^{k+1}}}))$ equations, each of length at least $3 \cdot 2^{k+1}$;
(c) if $E$ is satisfied, then $E'$ are $E''$ are also satisfied;
(d) if $E'$ is satisfied with shortage $S$, $S \leq 2^k$, and $E''$ is satisfied with shortage $2^{k+1}$, then $E$ is satisfied with shortage $S$.

Proof. If $b \leq \frac{n}{\sqrt{2^{k+1}}}$, we simply fall back to Lemma 4.10. Otherwise, like in the proof of Lemma 4.10, we use a different approach for and short long equations $e \in E$. This time, however, we set a larger threshold of $4 \cdot 2^{k+1}$ to distinguish between the two cases. If the length $L$ of the equation $e \in E$ is below this value, we apply Corollary 4.2 and insert the resulting equations of length $3 \cdot 2^k$ to $E'$.

To handle longer equations $e : T[p..q] = T[p'..q']$, we use a $2^{k+1}$-difference-cover $DC$ of $\{1, \ldots, n\}$ constructed according to Lemma 2.5. We generate a sequence of substring equations $e_1, \ldots, e_5$. The equation $e_1 : T[p + h(p,p')..q] = T[p' + h(p,p')..q]$ is inserted to $E''$, and the remaining equations, inserted to $E'$, are defined to have length $3 \cdot 2^k$ and start positions $(p, p')$, $(p + 2^k, p' + 2^k)$, $(p + 2^{k+1}, p' + 2^{k+1})$, and $(q - 3 \cdot 2^k + 1, q' - 3 \cdot 2^k + 1)$, respectively. Since $p + h(p,p') \leq p + 2^{k+1} \leq 1 + p + 2^{k+1} + (3 \cdot 2^k - 1) - S - 2^{k+1}$, we can use Fact 4.1 to prove that (c) and (d) are satisfied.

Finally, we reduce $E''$ using Lemma 4.13. Since the starting positions belong to $DC$, this results in $O(\frac{n}{\sqrt{2^{k+1}}})$ equations. The running time of the reduction is $O(b)$, as claimed.

Having developed all necessary tools, we are ready to prove the main results of this section.

Theorem 4.15. A system $E$ of $b$ substring equations on a text $T$ of length $n$ can be verified in $O(n \sqrt{\log b})$ time using $O(b)$ space.

Proof. We set $\ell = \lfloor \log \frac{n \sqrt{\log b}}{b} \rfloor$ and naively check all equations shorter then $3 \cdot 2^\ell$. This takes $O(b \cdot n \sqrt{\log b}) = O(n \sqrt{\log b})$ time. From now on, we assume that all equations are of length at least $3 \cdot 2^\ell$.

The equations are processed iteratively applying Lemma 4.14 for increasing values of $k$, starting from $k = \ell$. This results in systems $E_\ell, \ldots, E_r$ ($r = \lfloor \log \frac{n}{b} \rfloor$), such that each system $E_k$, $\ell \leq k \leq r$, contains $O(\min(b, \frac{n}{\sqrt{2^k}}))$ equations, each of length $3 \cdot 2^k$. Moreover, condition (c) of Lemma 4.14 implies that if the system $E$ is satisfied, then all systems $E_k$ is satisfied. By condition (d), if $E_\ell$ is satisfied and each $E_k$, $\ell < k \leq r$, is satisfied with shortage $2^k$, then the input system $E$ is satisfied.

Let $m = 2 \lfloor \log \frac{n}{b} \rfloor$. Note that the systems $E_k, \ldots, E_m$ together do not fit in $O(b)$ space, but each of them can be generated in $O(b(m - \ell + 1)) = O(b \log \frac{n}{b})$ time. On the other hand, the bound $O(\frac{n}{\sqrt{2^m}})$ on the size of $E_{m+1}, \ldots, E_r$ decreases geometrically an thus their total size is $O(\frac{n}{\sqrt{2^m}}) = O(b)$. The time required to generate these systems is also $O(b)$.

We process systems $E_k$ for decreasing values $k$. We define $F_r = \emptyset$ and for $\ell \leq k < r$, we define $F_k$ as the effective application of the reduction of Lemma 4.12 to $E_{k+1} \cup F_{k+1}$. Note that $F_k$ is a
uniform system with $O(\min(b, \frac{n}{\sqrt{2k}}) + \frac{n}{\sqrt{2k}} \sqrt{\log \frac{n}{2k}})$ equations of length $3 \cdot 2^k$. Since $\frac{n}{\sqrt{2k}} = \Theta(\frac{b}{\sqrt{\log b}})$, we actually have $|F_k| = O(\min(b, \frac{n}{\sqrt{2k}}))$ for each $k$.

The proof that $E$ is satisfied if and only if $E_\ell$ and $F_\ell$ are both satisfied is the same as in Theorem 4.11; we just use Lemma 4.12 instead of Corollary 4.9. The system $F_\ell \cup E_\ell$ contains $O(b)$ equations, each of length $O(\frac{n}{\sqrt{\log b}})$, so its verification takes $O(n \sqrt{\log b})$ time and $O(b)$ space. The construction of each $F_k$ from $F_{k+1}$ takes $O(b)$ space and $O(b \log \frac{n}{b})$ time for $k \leq m$, and $O(b \log \frac{n}{b})$ time in total for $k \geq m$. This gives overall running time of $O(b \log \frac{n}{b}) = O(n)$.

**Corollary 4.16.** The sparse suffix tree of any $b$ suffixes of a text of length $n$ can be computed using $O(n \sqrt{\log b})$ time and $O(b)$ space. The algorithm returns $\perp$ w. prob. $n^{-c}$ for user-defined constant $c$.

**References**

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004. doi:10.1016/S1570-8667(03)00065-0.

[2] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, 1985. doi:10.1145/4221.4227.

[3] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *Latin American Symposium on Theoretical Informatics, LATIN 2000*, volume 1776 of LNCS, pages 88–94. Springer Berlin Heidelberg, 2000. doi:10.1007/10719839_9.

[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, April 2016. doi:10.1145/2836166.

[5] Philip Bille, Inge Li Gørtz, Mathias Bæk Tejs Knudsen, Moshe Lewenstein, and Hjalte Wedel Vildhøj. Longest common extensions in sublinear space. In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Combinatorial Pattern Matching, CPM 2015*, volume 9133 of LNCS, pages 65–76. Springer, 2015. doi:10.1007/978-3-319-19929-0_6.

[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. doi:10.1016/j.jda.2013.06.003.

[7] Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking. In Ricardo A. Baeza-Yates, Edgar Chávez, and Maxime Crochemore, editors, *Combinatorial Pattern Matching, CPM 2003*, volume 2676 of LNCS, pages 55–69. Springer, 2003. doi:10.1007/3-540-44888-8_5.

[8] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, New York, NY, USA, 2007.

[9] Martin Farach. Optimal suffix tree construction with large alphabets. In *38th IEEE Annual Symposium on Foundations of Computer Science, FOCS 1997*, pages 137–143. IEEE Computer Society, 1997. doi:10.1109/SFCS.1997.646102.
[10] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In 41st Annual Symposium on Foundations of Computer Science, FOCS 2000, pages 390–398. IEEE Computer Society, 2000. doi:10.1109/SFCS.2000.892127.

[11] Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. Proceedings of the American Mathematical Society, 16(1):109–114, 1965. doi:10.2307/2034009.

[12] Johannes Fischer, Tomohiro I, and Dominik Köppl. Deterministic sparse suffix sorting on rewratable texts. In Evangelos Kranakis, Gonzalo Navarro, and Edgar Chávez, editors, Latin American Symposium on Theoretical Informatics, LATIN 2016, volume 9644 of LNCS, pages 483–496. Springer, 2016. doi:10.1007/978-3-662-49529-2_36.

[13] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. Journal of Computer and System Sciences, 48(3):533–551, 1994. doi:10.1016/S0022-0000(05)80064-9.

[14] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In 14th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2003, pages 841–850, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=644108.644250.

[15] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM Journal on Computing, 35(2):378–407, 2005. doi:10.1137/S0097539702402354.

[16] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. SIAM Journal on Computing, 13(2):338–355, 1984. doi:10.1137/0213024.

[17] Tomohiro I, Juha Kärkkäinen, and Dominik Kempa. Faster sparse suffix sorting. In Ernst W. Mayr and Natacha Portier, editors, Symposium on Theoretical Aspects of Computer Science, STACS 2014, volume 25 of LIPIcs, pages 386–396. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2014. doi:10.4230/LIPIcs.STACS.2014.386.

[18] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. Journal of the ACM, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.

[19] Juha Kärkkäinen and Esko Ukkonen. Sparse suffix trees. In Jin-yi Cai and C. K. Wong, editors, Computing and Combinatorics, COCOON 1996, volume 1090 of LNCS, pages 219–230. Springer, 1996. doi:10.1007/3-540-61332-3_155.

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

[21] Mamoru Maekawa. A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. ACM Transactions on Computer Systems, 3(2):145–159, May 1985. doi:10.1145/214438.214445.

[22] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. SIAM Journal on Computing, 22(5):935–948, 1993. doi:10.1137/0222058.

[23] Edward M. McCreight. A space-economical suffix tree construction algorithm. Journal of the ACM, 23(2):262–272, April 1976. doi:10.1145/321941.321946.
[24] 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 Roberto Grossi and Moshe Lewenstein, editors, Combinatorial Pattern Matching, CPM 2016, volume 54 of LIPIcs, pages 1:1–1:10. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CPM.2016.1.

[25] Esko Ukkonen. On-line construction of suffix trees. Algorithmica, 14(3):249–260, 1995. doi:10.1007/BF01206331.

[26] Peter Weiner. Linear pattern matching algorithms. In 14th Annual Symposium on Switching and Automata Theory, SWAT 1973, pages 1–11, Washington, DC, USA, 1973. IEEE Computer Society. doi:10.1109/SWAT.1973.13.