Faster Sparse Suffix Sorting

Tomohiro I[1], Juha Kärkkäinen[2], and Dominik Kempa[3]

1 **Kyushu University, Japan**
`tomohiro.i@inf.kyushu-u.ac.jp`
2 **University of Helsinki, Finland**
`juha.karkkainen@cs.helsinki.fi`
3 **University of Helsinki, Finland**
`dominik.kempa@cs.helsinki.fi`

Abstract

The sparse suffix sorting problem is to sort $b = o(n)$ arbitrary suffixes of a string of length n using $o(n)$ words of space in addition to the string. We present an $\mathcal{O}(n)$ time Monte Carlo algorithm using $\mathcal{O}(b \log b)$ space and an $\mathcal{O}(n \log b)$ time Las Vegas algorithm using $\mathcal{O}(b)$ space. This is a significant improvement over the best prior solutions by Bille et al. (ICALP 2013): a Monte Carlo algorithm running in $\mathcal{O}(n \log b)$ time and $\mathcal{O}(b^{1+\epsilon})$ space or $\mathcal{O}(n \log^2 b)$ time and $\mathcal{O}(b)$ space, and a Las Vegas algorithm running in $\mathcal{O}(n \log^2 b + b^2 \log b)$ time and $\mathcal{O}(b)$ space. All the above results are obtained with high probability not just in expectation.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases string algorithms, sparse suffix sorting, sparse suffix trees, Karp–Rabin fingerprints, space–time tradeoffs

Digital Object Identifier 10.4230/LIPIcs.STACS.2014.386

1 Introduction

Given a string T of length n and a set of b positions in T, the *sparse suffix sorting problem* is to sort the set of suffixes of T starting at those positions. The problem can be easily solved in $\mathcal{O}(n)$ time and space by constructing the suffix array, i.e., by sorting the set of all suffixes of T [8]. However, if the space is restricted to $\mathcal{O}(b)$ words in addition to the string the problem becomes more difficult. A straightforward solution is to use plain string sorting without taking advantage of the overlaps between the suffixes, but this requires at least $\Omega(nb)$ time in the worst case since the total length of the suffixes as separate strings is $\Omega(nb)$. More generally, we are interested in space–time tradeoffs, i.e., solutions using $\mathcal{O}(s)$ words of extra space for $s \in [b..n]$.

An efficient sparse suffix sorting algorithm has many potential applications. In the space-efficient Burrows–Wheeler transform algorithm in [9], sparse suffix sorting is used for two purposes: to sort a random sample of suffixes in order to partition the suffix array into approximately equal parts, and then to sort each of those parts separately. A similar approach might be useful for external memory and distributed suffix array construction. A sorted random sample of suffixes might be useful for estimating various measures and statistics on the string, but this direction has not been studied much because, until recently, there was no efficient algorithm for sorting a random sample. The full suffix array has numerous applications in text indexing and mining [1]. If application specific information allows us to focus on a smaller set of text positions, a full text index can be replaced with a much smaller sparse text index [2], the construction of which requires sparse suffix sorting.

There are many results for efficiently sorting certain special types of sets of suffixes (see [2]), but the general case of sorting an arbitrary set of suffixes has proved to be much harder. We are aware of two previous results improving on the naive bounds mentioned above. One uses the technique of *difference cover sampling* [4, 8] for sorting the suffixes in $\mathcal{O}(n^2/s)$ time and $\mathcal{O}(s)$ extra space for any $s \in [b..n]$ [8, Sect. 8]. The other result, by Bille et al. [2], is a Monte Carlo algorithm running in $\mathcal{O}(n \log^2 b/ \log(1+s/b))$ time and $\mathcal{O}(s)$ extra space. In particular, it runs in $\mathcal{O}(n \log^2 b)$ time when using $\mathcal{O}(b)$ space and in $\mathcal{O}(n \log b)$ time when using $\mathcal{O}(b^{1+\epsilon})$ space for any constant $\epsilon > 0$. Bille et al. also describe a deterministic verification algorithm to obtain a Las Vegas algorithm running in $\mathcal{O}(n \log^2 n + b^2 \log b)$ time using $\mathcal{O}(b)$ words of extra space.

In this paper, we improve on the results of Bille et al. by presenting a randomized Monte Carlo algorithm that runs in $\mathcal{O}(n + (nb/s) \log s)$ time using $\mathcal{O}(s)$ words of extra space for any $s \in [b..n]$. In particular, the time is $\mathcal{O}(n \log b)$ when using $\mathcal{O}(b)$ space and $\mathcal{O}(n)$ when using $\mathcal{O}(b \log b)$ space. As with the algorithm of Bille et al., our algorithm may produce an incorrect answer but the probability of this happening can be made smaller than n^{-c} for any constant c. In addition, we obtain a faster Las Vegas algorithm by describing a deterministic verification algorithm running in $\mathcal{O}(n \log b)$ time using $\mathcal{O}(b)$ words of extra space.

Our algorithms use some of the same basic tools and techniques as those of Bille et al., but in a quite different way. Karp–Rabin fingerprints have a key role in both Monte Carlo algorithms, but our underlying sorting algorithm is the optimal radix sorting of Paige and Tarjan [11] rather than the quicksort of Bille et al. Similarly to Bille et al., our verification algorithm utilizes the transitivity of equality by finding cycles in graphs connecting substrings that are supposed be equal, but our algorithms for processing the graphs are entirely different.

2 Problem and Model of Computation

Let $T = T[1..n]$ be a string of length n. To simplify algorithms, we assume that $T[n]$ is a unique character, ensuring that the set of suffixes is prefix-free. For $i \in [1..n]$, let $T_i = T[i..n]$ denote the suffix of length $n - i + 1$ and let $lcp(i, j)$ denote the length of the *longest common prefix* between the suffixes T_i and T_j. Let $\mathcal{S} \subseteq [1..n]$ be a set of b positions in T. The problem we want to solve is to sort the set of suffixes $T_{\mathcal{S}} = \{T_i : i \in \mathcal{S}\}$ lexicographically.

The *sparse suffix array* $SSA[1..b]$ is the array containing the positions in $\mathcal{S}$ in the lexicographical order of the suffixes. The associated *LCP array* $LCP[2..b]$ contains the longest common prefix lengths for adjacent suffixes in the sorted set, that is, $LCP[i] = lcp(SSA[i], SSA[i-1])$ for all $i \in [2..b]$. Given the arrays SSA and LCP, we can easily compute the sparse suffix tree SST for $T_{\mathcal{S}}$ in $\mathcal{O}(b)$ time [2] (see Section 4 for a definition of SST). Conversely, SSA and LCP can be constructed in $\mathcal{O}(b)$ time by a depth first traversal of SST. If we are given SSA but not LCP, we can easily compute the LCP in $\mathcal{O}(n \log b)$ time and $\mathcal{O}(b)$ extra space using the techniques of Bille et al. [2] or the ones in this paper. Thus sorting the suffixes is the key problem. The Monte Carlo sorting algorithm of Section 4 outputs SST and the deterministic verification algorithm of Section 5 takes SSA and LCP as input.

The model of computation is the word RAM with word size $\Omega(\log n)$. For the most part, our algorithms operate in the comparison model, i.e., we only assume that characters can be compared in constant time. However, for fingerprint computation in the Monte Carlo algorithm, we need the stronger assumption that, for any character a, we can in constant time compute an integer representation $\rho(a)$ such that $\rho(a) = \rho(b)$ if and only if $a = b$. Note that we do not assume that the integer representation can be used for order comparison.

3 Basic Techniques

3.1 Karp–Rabin fingerprints

Let q be a prime and choose $r \in [0..q-1]$ uniformly at random.[1] The fingerprint for a substring of $T[i..j]$ is defined as

$$FP[i..j] = \sum_{k=i}^{j} \rho(T[k]) \cdot r^{j-k} \bmod q \ .$$

Clearly, if $T[i..i+\ell] = T[j..j+\ell]$ then $FP[i..i+\ell] = FP[j..j+\ell]$. On the other hand, if $T[i..i+\ell] \neq T[j..j+\ell]$ then $FP[i..i+\ell] \neq FP[j..j+\ell]$ with a probability at least $1-\ell/q$ [6]. Since we are comparing only substrings of equal length, the number of different possible substring comparisons is less than n^3. Thus, for any positive constant c, we can set q to be a prime larger than n^{c+4} (but still small enough to fit in $\mathcal{O}(1)$ words) to make the fingerprint function perfect with probability at least $1-n^{-c}$.

The fingerprint of a string of length ℓ can obviously be computed in $\mathcal{O}(\ell)$ time, but for Karp–Rabin fingerprints [10], we can additionally use the following equations to compute fingerprints more efficiently from existing fingerprints. For all $i \leq j \leq k$,

$$\begin{aligned} FP[i..j+1] &= FP[i..j] \cdot r + \rho(T[j+1]) \bmod q \\ FP[i..k] &= FP[i..j] \cdot r^{k-j} + FP[j+1..k] \bmod q \\ FP[j+1..k] &= FP[i..k] - FP[i..j] \cdot r^{k-j} \bmod q \end{aligned} \tag{1}$$

3.2 Grouping

By grouping, we mean the task of ordering a multiset so that equal elements are grouped together but the groups can be in arbitrary order. Suppose we have a zero-initialized array A of s words. Given $k = o(s)$ integers from a universe of size s, we can group but not sort the integers in $\mathcal{O}(k)$ time by a simple distribute-and-collect procedure while leaving A in the zero-initialized state ready to be reused. Thus d such grouping tasks can be executed in $\mathcal{O}(s+kd)$ time. Sorting could be done in $\mathcal{O}(s+kd)$ time offline (as a single batch) but online sorting would require $\mathcal{O}(sd)$ time. The online setting occurs in radix sorting. Thus, grouping k integers from a universe of size $u > s$ by a modified LSD radix sort with radix s can be done in $\mathcal{O}(s + k\log_s u)$ time while sorting would need $\mathcal{O}(s\log_s u)$ time. A prominent example of this technique is the string sorting algorithm of Paige and Tarjan [11]. It uses a similar modification of MSD radix sorting into a grouping algorithm to construct an *unsorted* compact trie of the strings, and then sorts the child edges of all nodes in one batch. Our suffix sorting algorithm has the same structure, though the unsorted trie construction is different and not based on MSD radix grouping (but does involve LSD radix grouping). Grouping and unsorted compact tries have a key role in our algorithm because fingerprints cannot be used for order comparisons.

3.3 String periodicity

An integer $p \in [0..m-1]$ is a *period* of a string $X[1..m]$ if $X[i] = X[i+p]$ for all $i \in [1..m-p]$. A basic result on periods is the following:

[1] The choice of r is the only random operation in the algorithm.

▶ **Lemma 1** (Weak Periodicity Lemma [7]). *If p_1 and p_2 are periods of a string $X[1..m]$ and $p_1 + p_2 \leq m$, then $\gcd(p_1, p_2)$ is a period of X too.*

Our verification algorithm uses this in the form of the following generalization to multiple periods:

▶ **Corollary 2.** *Integers $p_1, p_2, \ldots, p_k \in [0..\lceil m/2 \rceil]$ are periods of a string $X[1..m]$ if and only if $\gcd(p_1, p_2, \ldots, p_k)$ is a period of X.*

The value $\gcd(p_1, p_2, \ldots, p_k)$ can be computed in $\mathcal{O}(k + \log m)$ time using the Euclidean algorithm [3]. Note that zero is a period of all nonempty strings, and all of the above holds for 0-periods too if we define $\gcd(p, 0) = p$ for all p.

4 Monte Carlo Algorithm

In this section, we describe a Monte Carlo algorithm for sparse suffix sorting. The output is actually the sparse suffix tree SST of the b suffixes in $T_{\mathcal{S}}$, but this is equivalent to suffix sorting as explained above. The construction of SST uses a novel technique of gradual refinement.

We start by defining a relaxed form of compact tries. An *ℓ-strict compact trie* is a rooted tree, where each internal node has at least two children. The edges are labelled by strings, and induced by the edge labelling, each node v is labelled by the concatenation of the edge labels from the root to v. For any two edges with the same parent node, their labels must be different, one label cannot be a prefix of the other, and the labels cannot share a common prefix of length ℓ (ℓ-strictness). The trie is ordered if the child edges of each node are lexicographically ordered by their labels. The standard compact trie is the (unique) ordered 1-strict compact trie.

An *ℓ-strict sparse suffix tree ℓ-SST* for the set $T_{\mathcal{S}}$ is an ℓ-strict compact trie with b leaves representing exactly the suffixes in $T_{\mathcal{S}}$. The edge labels are substrings of T and can be represented in constant space by pointers to T. Thus the size of an ℓ-SST is $\mathcal{O}(b)$ words. The standard sparse suffix tree SST (the desired output of the algorithm) is the ordered 1-SST.

The algorithm begins by constructing ℓ-SST for $\ell = 2^{\lceil \log n \rceil}$, which is just the root and b leaves with the edges labelled by the suffixes. Then, in each round of the computation, the algorithm halves the value of ℓ and computes an ℓ-SST from the 2ℓ-SST until $\ell = 1$. Given the 1-SST, we can sort the child edges of all nodes using $\mathcal{O}(b \log b)$ character comparisons to obtain SST.

To compute ℓ-SST from 2ℓ-SST, we perform the following steps:

1. For each edge with a label of length at least ℓ, compute the fingerprint $FP[i..i + \ell - 1]$, where i is the starting position of the edge label in T.
2. Use LSD radix grouping to group all edges from Step 1 using the key (u, f), where u is the parent node of the edge and f is the fingerprint from Step 1.
3. For each group of at least two edges with the same key (u, f), add a new node v and an edge (u, v) to the tree. The new node v becomes the parent node of the edges in the group. The label of (u, v) has length ℓ and the starting position is taken from one of the old edges. The labels of the old edges are shortened by ℓ by moving the starting positions forward by ℓ positions. If v is the only child of u after the stage, we remove u by concatenating its parent and child edges.

It is easy to check that the tree resulting from the above steps (i) is still a valid compact trie, (ii) retains the labels of all nodes inherited from 2ℓ-SST, and (iii) satisfies the ℓ-strictness condition. It is thus a valid ℓ-SST.

The remaining detail is how to compute the fingerprints. We assume that we can use $\mathcal{O}(s)$ words of extra space for some $s \in [b..n]$. We divide the string T into blocks of size $h = n/s$. For all $i \in [1..n/h]$, we compute and store the values $FP[1..ih]$ and $r^{ih} \bmod q$. We can then compute the fingerprint of any substring of T of length ℓ in $\mathcal{O}(\min(\ell, n/s))$ time using Equations (1).

▶ **Theorem 3.** *Any set of b suffixes of a string of length n can be sorted correctly with high probability in $\mathcal{O}(n + (bn/s)\log s)$ time using $\mathcal{O}(s)$ words of space in addition to the string for any $s \in [b..n]$.*

Proof. With high probability, there are no fingerprint collisions, which is enough to ensure the correctness of the algorithm by the above discussion.

Most of the data structures including the tree fit in $\mathcal{O}(b)$ space during the whole computation. We need $\mathcal{O}(s)$ space for the precomputed fingerprints and powers of r, as well as for the bucket array in LSD radix grouping. Thus the total space requirement is $\mathcal{O}(s)$ words in addition to the string T.

The time complexity is dominated by the fingerprint computation. The initialization of the precomputed fingerprints needs $\mathcal{O}(n)$ time. In the first $\log(s)$ rounds (i.e., while $\ell \geq n/s$), each fingerprint computation in Step 1 takes $\mathcal{O}(n/s)$ time, and in the later rounds $\mathcal{O}(\ell)$ time. Thus the total time for fingerprint computation is $\mathcal{O}(n + (bn/s)\log s)$. In Step 2 of each round, we are grouping $\mathcal{O}(b)$ integers of $\mathcal{O}(\log n)$-bits each using a bucket array of size s, which takes $\mathcal{O}(b \log_s n)$ time. The total grouping time is $\mathcal{O}(b \log^2 n/\log s) = \mathcal{O}((bn/s)\log s)$. Everything else can be done in $\mathcal{O}(b \log n) = \mathcal{O}((bn/s)\log s)$ time. ◀

By choosing $s = b$ and $s = b \log b$, we obtain the following results.

▶ **Corollary 4.** *Any set of b suffixes of a string of length n can be sorted correctly with high probability in $\mathcal{O}(n \log b)$ time using $\mathcal{O}(b)$ words of space in addition to the string.*

▶ **Corollary 5.** *Any set of $b = \mathcal{O}(n/\log n)$ suffixes of a string of length n can be sorted correctly with high probability in $\mathcal{O}(n)$ time using $\mathcal{O}(b \log b)$ words of space in addition to the string.*

Note that if $b \log b = \omega(n)$, we can use a full suffix array construction algorithm to sort $T_{\mathcal{S}}$ in $\mathcal{O}(n)$ time and space (assuming a proper integer alphabet where strings can be radix sorted).

5 Las Vegas Algorithm

In this section we describe a deterministic algorithm to check if the arrays SSA and LCP are correct. Assuming SSA contains a permutation of set $\mathcal{S}$ and besides the trivial sanity checks of LCP, we need to verify if the conditions

$$\begin{aligned} T[a_i..a_i + \ell_i - 1] &= T[b_i..b_i + \ell_i - 1] \\ T[a_i + \ell_i] &< T[b_i + \ell_i] \end{aligned} \tag{2}$$

are satisfied for $i \in [2..b]$, where $a_i = SSA[i-1]$, $b_i = SSA[i]$ and $\ell_i = LCP[i]$.

Checking the second condition takes only $\mathcal{O}(b)$ time. A naive verification of the first one, however, requires $\mathcal{O}(bn)$ operations in the worst-case. We now describe a faster algorithm.

The first version runs in $\mathcal{O}(n \log^2 b)$ time. A refinement yielding $\mathcal{O}(n \log b)$ time is presented next.

5.1 $\mathcal{O}(n \log^2 b)$ time algorithm

Let $m_0 = 3 \cdot 2^{\lfloor \log(n/3) \rfloor}$ and $m_h = m_0/2^h$ for $h > 0$. A substring of T of length m_h for some h is called a *segment*. Any substring can be covered by at most two (possibly overlapping) segments. Thus we can replace the b substring equalities with at most $2b$ segment equalities. A pair of segments whose equality we need to verify is called a *twin pair*. A twin pair of segments of length m_h is called an *h-pair* and its two segments are called *h-segments*.

The algorithm maintains two lists of segments, L and R. Initially, both lists contain all twin pair segments, with L sorted by the left endpoint (i.e., the starting position of the substring in T) and R sorted by right endpoint. Each segment is linked to its twin in the same list. We sort the lists once in $\mathcal{O}(b \log b)$ time in the beginning and maintain the sorted order afterwards. During the algorithm the size of each list never grows beyond the initial size of at most $4b$. The lists L and R are processed symmetrically (left–right symmetry) and completely independently of each other. We focus on describing the algorithm for L.

5.1.1 Overview

The algorithm operates in $\log b$ rounds. In the h-th round, $h \in [0..\lfloor \log b \rfloor]$, we process all h-pairs in L. The set of all h-pairs is partitioned into two subsets, *fully checked* pairs and *partially checked* pairs. The former, relatively small subset is verified by a direct brute-force comparison of the substrings. If a mismatch is detected, the verifier exits with a negative answer. Otherwise, we use the knowledge gained during these naive checks to indirectly verify the equality of the *middle third* of every partially checked pair. All fully checked pairs are removed from L and each partially checked h-pair is replaced by its left half (i.e., the $(h+1)$-pair with the same left endpoints), to obtain the list for the next round. After $\log b$ rounds, all segments on the list have length $\mathcal{O}(n/b)$ and can be checked by brute-force in $\mathcal{O}(n)$ total time.

Consider an initial twin pair that is partially checked and replaced by its first half, which in turn is partially checked and replaced etc., until eventually a pair is fully checked. The verified middle thirds of these pairs together with the final, fully checked pair completely cover the leftmost two thirds of the initial pair. Therefore, if the verifier did not exit prematurely due to a mismatch, the leftmost two thirds of each segment is confirmed to be equal to the leftmost two thirds of its twin. The symmetric processing of the list R will similarly verify all rightmost two thirds resulting in a complete verification.

5.1.2 A single round in detail

Suppose we are at round h and denote $m = m_h$. We start by constructing an undirected multigraph $G = G_h$ from the list L. Consider a partition of T into blocks of size $B = B_h = m/\lceil 6 \log b + 18 \rceil$. Each block corresponds to one vertex in G. We associate every h-segment in L with the block containing the left endpoint. For any h-pair in L we create an edge between the vertices associated with the two segments of the pair. Isolated vertices are omitted during the construction, thus the resulting graph has $|V(G)| = \mathcal{O}(\min(b, (n/m) \log b))$ vertices and $|E(G)| \leq 2b$ edges. This graph is essentially the same as the one in [2].

We start a breadth-first search from an arbitrary vertex v of G. The search continues as long as each new BFS layer (the subset of vertices with the same shortest distance to v)

at least doubles the total size of the BFS tree. Denote the constructed tree as F and the subgraph of G containing all edges inspected during the BFS as G'. Note that F and G' contain the vertices of the last layer but no edges with both ends in the last layer. The h-pairs associated with the tree edges $E(F)$ are chosen as fully checked pairs and the brute force comparisons are performed. The h-pairs associated with non-tree edges $E(G') \setminus E(F)$ become partially checked pairs. We will next describe how to indirectly verify the equality for the middle $m/3$ positions of those pairs.

Let A be an arbitrary h-segment associated with the root v of F. By M we denote the substring of A of length $2m/3$ centered exactly in the middle of A.

▶ **Lemma 6.** *Suppose all h-pairs corresponding to edges of F were successfully verified. Then every h-segment associated with some vertex of G' contains M as a substring.*

Proof. First note that the height of F is at most $\log b + 2$. Let U be an arbitrary segment associated with a vertex $u \in V(G')$. Consider the sequence of segments starting with A, containing all twin pairs associated with the edges on the path from v to u in F, and ending with U. Each adjacent pair of segments in this sequence is either a fully checked pair or is associated with the same vertex. In the former case, the segments are verified to be identical. In the latter case, the segments overlap by more than $m - B$ and the relative position of M inside the segments differ by less than B. Over the whole sequence, the relative position of M changes by less than $B(\log b + 3) \leq m/6$. Since M starts at a distance of $m/6$ from both ends of A, all segments in the sequence including U must contain M. ◀

We exploit this fact as follows. Let $\{u_i, u_j\} \in E(G') \setminus E(F)$ be an arbitrary edge associated with a twin pair $\{U_i, U_j\}$. Let d_i (d_j) be the relative position of M inside U_i (U_j).

▶ **Lemma 7.** *If M has period $p = |d_i - d_j|$, then a substring of length $m/3$ centered in the middle of U_i matches the corresponding substring in U_j. Otherwise $U_i \neq U_j$.*

Proof. If we align U_i and U_j, the occurrences of M inside the segments overlap by $k = |M| - p \geq m/3$. M has period p if and only if that overlapping part is equal in both segments. The lemma follows from the facts that the overlapping parts contain the middle thirds and are contained in the whole segments. ◀

Each non-tree edge of G' generates one periodicity query for M. In order to efficiently answer all queries we observe that p cannot exceed $m/3 = |M|/2$, thus all queries can be reduced to a single one using Corollary 2 (in Section 3). If the reduced periodicity query returns true, we obtain the equality of middle third for all h-pairs associated with $E(G') \setminus E(F)$. Otherwise, Lemma 7 implies that there must be a mismatch in some pair, and hence the verifier returns a negative answer and exits.

After processing G', the edges $E(G')$ are deleted from G along with the vertices that become isolated. The procedure is then repeated for the remaining part of the graph and this continues until the graph is empty.

▶ **Lemma 8.** *Round h of verification takes $\mathcal{O}(b + |V(G_h)|m_h)$ time and $\mathcal{O}(b)$ extra space.*

Proof. The list L is sorted, thus building G takes $\mathcal{O}(b)$ time and space. All BFS searches require $\mathcal{O}(|V(G)| + |E(G)|) = \mathcal{O}(b)$ time in total. The stopping criterion for the BFS implies that deleting the set $E(G')$ along with the introduced isolated vertices removes at least $|V(G')|/2$ vertices from G, thus the BFS trees have altogether $\mathcal{O}(|V(G)|)$ edges. The brute-force checking of all twin pairs associated with such edges therefore takes $\mathcal{O}(|V(G)|m)$ time.

The relative position of M can be easily tracked during the BFS, hence the generation of the periodicity queries of Lemma 7 for all non-tree edges can be done in $\mathcal{O}(b)$ time. Reducing the number of periodicity queries using Corollary 2 consumes $\mathcal{O}(b + g \log m)$ time in total, where $g \le |V(G)|$ is the number of subgraphs G' processed during the round, and the reduced periodicity queries can be executed in $\mathcal{O}(gm)$ time. Thus total time for all partial checks is $\mathcal{O}(b + gm) = \mathcal{O}(b + |V(G)|m)$. ◀

Note that $\mathcal{O}(b + |V(G_h)|m_h) = \mathcal{O}(n \log b)$ for all h. We perform $\mathcal{O}(\log b)$ rounds, after which we verify the remaining segments in $\mathcal{O}(n)$ time, hence we obtain the following result.

▶ **Theorem 9.** *The correctness of a sparse suffix tree constructed for a set of b suffixes of a string of length n can be deterministically verified in $\mathcal{O}(n \log^2 b)$ time and $\mathcal{O}(b)$ words of space in addition to the string.*

5.2 $\mathcal{O}(n \log b)$ time algorithm

The time complexity of the verification algorithm is dominated by the $\mathcal{O}(|V(G_h)|m_h)$ time spent in each round h for brute force comparisons and partial checks. The number of vertices is bounded by $\mathcal{O}(n \log b / m_h)$ but can be smaller. In this section we show how to modify the algorithm to reduce the number of vertices in the graphs, which decreases the overall verification time to $\mathcal{O}(n \log b)$. The reduction is accomplished by moving segments in a way that concentrates them on certain areas while other areas become empty. The empty areas contribute no vertices to the graphs.

The movement of segments happens at the end of each round. Consider the end of round h when all h-pairs on the list L have been removed or replaced. Let i and j be the left endpoints of the segments in some h-pair that was fully checked during the round, i.e., we know that $T[i..i + m_h - 1] = T[j..j + m_h - 1]$. If an h'-segment in L for some $h' > h$ is completely inside $T[i..i + m_h - 1]$ it can be moved to the corresponding position in $T[j..j + m_h - 1]$ without affecting the correctness of the verification. If we move all segments in L that are inside $T[i..i + m_h - 1]$, there are no left endpoints in the region $[i..i + m_h/2 - 1]$ anymore and we say that the region $[i..i + m_h/2 - 1]$ has been *cleared*.

However, a region that has been cleared may not stay cleared as other moves in the same round or later rounds may reintroduce left endpoints to the region. In the case of the moves in the same round, this is easy to avoid by doing all moves either leftwards or rightwards. As shown later, we cannot fix the direction in advance but have to choose the better direction separately for each round. However, in a single round, all moves are made in the same direction, and it is easy to see that by processing the fully checked pairs in the appropriate order, all cleared regions will stay cleared. Moves in later rounds are more problematic. The cleared region $[i..i + m_h/2 - 1]$ may be almost completely overlapped by a $(h + 1)$-segment with the left endpoint at $i - 1$ and moves to this $(h + 1)$-segment can undo the clearance. To avoid this, we will be more selective with the moves.

Let $h_{\text{last}} = \lfloor \log b \rfloor$ be the last round in the algorithm, and recall that $B_h = m_h / \lceil 6 \log b + 18 \rceil$. Define $d_h = (h_{\text{last}} - h + 2) B_h / 2$ for all $h \in [0..h_{\text{last}}]$. If we are moving from $T[i..i + m_h - 1]$ to $T[j..j + m_h - 1]$, we will move segments if and only if their left endpoint is in the *source region* $[i..i + d_h]$, i.e., only the source region will be cleared. The left endpoints are moved to the *target region* $[j..j + d_h]$. Source and target regions are both called *move regions*. The next two lemmas establish the key properties of move regions. The first lemma shows that the source region is a subregion of $[i..i + m_h/2 - 1]$ and thus can indeed be cleared. The second one shows that a sufficiently large part of a cleared source region is *permanently cleared*.

▶ **Lemma 10.** $d_h < m_h/12$ *for all* $h \in [0..h_{\text{last}}]$.

Proof. We just need to note that $h_{\text{last}} - h + 2 < \log b + 3$ and $B_h \leq m_h/(6(\log b + 3))$. ◀

▶ **Lemma 11.** *If a source region* $[i..i + d_h]$ *is cleared in round* h*, then the region* $[i + d_h - B_h..i + d_h]$ *cannot be uncleared in later rounds.*

Proof. Consider the worst case scenario with regard to unclearing $[i..i + d_h]$. Assume there is an $(h+1)$-segment with left endpoint at $i - 1$ and a target region extending to $i - 1 + d_{h+1}$. An $(h + 2)$-segment moved to the end of the target region has a target region extending to $i - 1 + d_{h+1} + d_{h+2}$, an $(h + 3)$-segment moved to the end of that target region has a target region extending to $i - 1 + d_{h+1} + d_{h+2} + d_{h+3}$ and so on. Thus no more than $D_h = \sum_{h'=h+1}^{h_{\text{last}}} d_{h'}$ leftmost positions of $[i..i + d_h]$ can be uncleared. We will show by induction that $D_h = d_h - B_h$, which proves the lemma.

Clearly, $D_{h_{\text{last}}} = 0 = d_{h_{\text{last}}} - B_{h_{\text{last}}}$. Assume then that $D_{h+1} = d_{h+1} - B_{h+1}$ and note that $B_h = 2B_{h+1}$. Thus, $D_h = D_{h+1} + d_{h+1} = 2d_{h+1} - B_{h+1} = (h_{\text{last}} - (h + 1) + 1)B_{h+1} = (h_{\text{last}} - h)B_h/2 = d_h - B_h$. ◀

In order to perform the segment moves efficiently, we will implement the list L as a list of sets, each of which contains segments with the same left endpoint and is implemented as a linked list. This representation supports all the operations we need in the basic algorithm, but additionally we can now remove a full set from the list, insert the set in a different place in the list or merge the set with a different set, all in constant time. When a segment belonging to a fully checked pair is removed from L, we remove it from the corresponding set but add a special node to L in front of the set and store a pointer to the special node in the corresponding move region. If a set contains multiple fully checked segments, they share a single special node. By moving the sets (but not the special nodes) we can execute all the moves from one source region in $\mathcal{O}(d_h) = \mathcal{O}(m_h)$ time. Thus the cost of the move operations is no higher than the cost of the brute force checks.

Due to possibly overlapping move regions, the order in which the moves are made has to be chosen with some care, but it is still possible to completely clear all the source regions in one round as long as all moves are made in the same direction. The direction, leftwards or rightwards, is chosen in each round so that the total area of the permanently cleared regions is maximized.

Finally, we need a minor modification to the way the algorithm deals with the special case, where the subgraph G' consists of a single vertex and one or more selfloop edges. In this case, one of the pairs associated with the edges is fully checked while others are partially checked. This does not affect the time complexity of the round, but ensures that every vertex is adjacent to at least one fully checked edge.

▶ **Theorem 12.** *The correctness of a sparse suffix tree constructed for a set of* b *suffixes of a string of length* n *can be deterministically verified in* $\mathcal{O}(n \log b)$ *time and* $\mathcal{O}(b)$ *words of space in addition to the string.*

Proof. Except for the segment moves, the algorithm operates as before and, as explained above, the segment moves do not compromise the correctness of the verification. Thus the algorithm is correct. The space complexity is clearly still $\mathcal{O}(b)$.

To prove the time complexity, we will show that the total area covered by the permanently cleared regions in round h is $\Theta(|V(G_h)|m_h/\log b)$. The costs of the form $\mathcal{O}(|V(G_h)|m_h)$ are distributed over the positions in the permanently cleared regions. Then every position gets charged for at most $\mathcal{O}(\log b)$ over the whole algorithm, and thus the total cost is $\mathcal{O}(n \log b)$.

Every vertex in the graph G_h is associated with a fully checked segment and each such segment contains a potential permanently cleared region (PPCR) of size B_h. The direction of moves decides whether a PPCR becomes a PCR. Two PPCRs can overlap only if they are associated with the same vertex or two vertices representing adjacent blocks. Thus there are at least $|V(G_h)|/2$ non-overlapping PPCRs and their total coverage is at least $B_h|V(G_h)|/2 = \Theta(|V(G_h)|m_h/\log b)$. The total coverage of the actual PCRs is at least half of that since we choose the direction to maximize the coverage. Note also that computing the coverage resulting from each direction can be done in $\mathcal{O}(b)$ time. ◀

▶ **Corollary 13.** *Any set of b suffixes of a string of length n can be sorted correctly using $\mathcal{O}(b)$ words of space in addition to the string in time that is $\mathcal{O}(n \log b)$ with high probability.*

6 Concluding Remarks

The time–space complexity of sparse suffix sorting has been a major open problem for a long time. Our new algorithms achieving the time–space product of $\mathcal{O}(nb \log b)$ are a major step towards a solution but open problems remain. Perhaps the main open problem is the deterministic time–space complexity: the best deterministic algorithms have a time–space product of $\mathcal{O}(n^2)$. Can our algorithms be made deterministic? Perhaps the use of fingerprints in the Monte Carlo algorithm can be replaced with a deterministic technique. Or perhaps the deterministic verification algorithm can be transformed into a sorting algorithm.

Some of the techniques developed in this paper, such as the ℓ-strict compact tries and their incremental construction, may have applications outside sparse suffix sorting. The concepts behind the verification algorithm could be useful not only as algorithmic tools but as analysis tools. For example, a major open problem is the size of the smallest grammar of a string particularly in comparison to the size of the Lempel–Ziv factorization of the same string [5]. This problem too involves pairs of identical substrings that overlap each other. The basic overlap graph was introduced in [2] but our algorithms reveal new combinatorial properties of this graph.

Acknowledgements. This work was supported by the Academy of Finland grant 118653 (ALGODAN) and by the Japan Society for the Promotion of Science.

References

1 Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algoritms*, 2(1):53–86, 2004.

2 Philip Bille, Johannes Fischer, Inge Li Gørtz, Tsvi Kopelowitz, Benjamin Sach, and Hjalte Wedel Vildhøj. Sparse suffix tree construction in small space. In *Proc. ICALP*, volume 7965 of *LNCS*, pages 148–159, 2013.

3 Gordon H. Bradley. Algorithm and bound for the greatest common divisor of n integers. *Commun. ACM*, 13(7):433–436, 1970.

4 Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking. In *Proc. CPM*, volume 2676 of *LNCS*, pages 55–69, 2003.

5 Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, April Rasala, Amit Sahai, and abhi shelat. Approximating the smallest grammar: Kolmogorov complexity in natural models. In *Proc. STOC*, pages 792–801. ACM, 2002.

6 Martin Dietzfelbinger, Joseph Gil, Yossi Matias, and Nicholas Pippenger. Polynomial hash functions are reliable. In *Proc. ICALP*, volume 623 of *LNCS*, pages 235–246, 1992.

7 Nathan J Fine and Herbert S Wilf. Uniqueness theorems for periodic functions. *Proc. Amer. Math. Soc.*, 16(1):109–114, 1965.

8 J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

9 Juha Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. *Theor. Comput. Sci.*, 387(3):249–257, 2007.

10 Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Develop.*, 31(2):249–260, 1987.

11 Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.