Sparse Suffix Tree Construction in Small Space

Philip Bille[1], Johannes Fischer[2], Inge Li Gørtz[1], Tsvi Kopelowitz[3], Benjamin Sach[4], and Hjalte Wedel Vildhøj[1]

[1] Technical University of Denmark, {phbi,inge}@dtu.dk,hwv@hwv.dk
[2] KIT, Institute of Theoretical Informatics, johannes.fischer@kit.edu
[3] Weizmann Institute of Science, kopelot@gmail.com
[4] University of Warwick, sach@dcs.warwick.ac.uk

Abstract. We consider the problem of constructing a sparse suffix tree (or suffix array) for b suffixes of a given text T of length n, using only $O(b)$ words of space during construction. Attempts at breaking the naive bound of $\Omega(nb)$ time for this problem can be traced back to the origins of string indexing in 1968. First results were only obtained in 1996, but only for the case where the suffixes were evenly spaced in T. In this paper there is no constraint on the locations of the suffixes.
We show that the sparse suffix tree can be constructed in $O(n \log^2 b)$ time. To achieve this we develop a technique, which may be of independent interest, that allows to efficiently answer b longest common prefix queries on suffixes of T, using only $O(b)$ space. We expect that this technique will prove useful in many other applications in which space usage is a concern. Our first solution is Monte-Carlo and outputs the correct tree with high probability. We then give a Las-Vegas algorithm which also uses $O(b)$ space and runs in the same time bounds with high probability when $b = O(\sqrt{n})$. Furthermore, additional tradeoffs between the space usage and the construction time for the Monte-Carlo algorithm are given.

1 Introduction

In the *sparse text indexing problem* we are given a string $T = t_1 \dots t_n$ of length n, and a list of b interesting positions in T. The goal is to construct an *index* for only those b positions, while using only $O(b)$ words of space during the construction process (in addition to storing the text T). Here, by index we mean a data structure allowing for the quick location of all occurrences of patterns *starting at interesting positions only*. A natural application comes from computational biology, where the string would be a sequence of nucleotides or amino acids, and additional biological knowledge rules out many positions where patterns could potentially start. Another application is indexing far eastern languages, where one might be interested in indexing only those positions where words start, but natural word boundaries do not exist.

Examples of suitable $O(b)$ space indexes include suffix trees [18] or suffix arrays [14] built on only those suffixes starting at interesting positions. Of course, one can always first compute a full-text suffix tree or array in linear time, and

then postprocess it to include the interesting positions only. The problem of this approach is that it needs $O(n)$ words of *intermediate working space*, which may be much more than the $O(b)$ words needed for the final result, and also much more than the space needed for storing T itself. In situations where the RAM is large enough for the string itself, but not for an index on *all* positions, a more space efficient solution is desirable. Another situation is where the text is held in read-only memory and only a small amount of read-write memory is available. Such situations often arise in embedded systems or in networks, where the text may be held remotely.

A "straightforward" space-saving solution would be to sort the interesting suffixes by an *arbitrary* string sorter, for example, by inserting them one after the other into a compacted trie. However, such an approach is doomed to take $\Omega(nb + n \log n)$ time [5], since it takes no advantage of the fact that the strings are suffixes of one large text, so it cannot be faster than a general string sorter.

Breaking these naive bounds has been a problem that can be traced back to—according to Kärkkäinen and Ukkonen [10]—the origins of string indexing in 1968 [15]. First results were only obtained in 1996, where Andersson et al. [2, 3] and Kärkkäinen and Ukkonen [10] considered *restricted* variants of the problem: the first [2, 3] assumed that the interesting positions coincide with natural *word boundaries* of the text, and the authors achieved *expected* linear running time using $O(b)$ space. The expectancy was later removed [9, 7], and the result was recently generalized to variable length codes such as Huffman code [17]. The second restricted case [10] assumed that the text of interesting positions is *evenly spaced*; i.e., every k^{th} position in the text. They achieved linear running time and optimal $O(b)$ space. It should be mentioned that the data structure by Kärkkäinen and Ukkonen [10] was not necessarily meant for finding only pattern occurrences starting at the evenly spaced indexed positions, as a large portion of the paper is devoted to recovering *all* occurrences from the indexed ones. Their technique has recently been refined by Kolpakov et al. [13]. Another restricted case admitting an $O(b)$ space solution is if the interesting positions have the same period ρ (i.e., if position i is interesting then so is position $i + \rho$). In this case the sparse suffix array can be constructed in $O(b\rho + b \log b)$ time. This was shown by Burkhardt and Kärkkäinen [6], who used it to sort *difference cover samples* leading to a clever technique for constructing the full suffix array in sublinear space. Interestingly, their technique also implies a time-space tradeoff for sorting b *arbitrary* suffixes in $O(v + n/\sqrt{v})$ space and $O(\sqrt{v}n + (n/\sqrt{v}) \log(n/\sqrt{v}) + vb + b \log b)$ time for any $v \in [2, n]$.

1.1 Our Results

We are the first to break the naive $O(nb)$ time algorithm for general sparse suffix trees, by showing how to construct a sparse suffix tree in $O(n \log^2 b)$ time, using only $O(b)$ words of space. To achieve this, we develop a novel technique for performing efficient batched longest common prefix (LCP) queries, using little space. In particular, in Section 3, we show that a batch of b LCP queries can be answered using only $O(b)$ words of space, in $O(n \log b)$ time. This technique may

be of independent interest, and we expect it to be helpful in other applications in which space usage is a factor. Both algorithms are Monte-Carlo and output correct answers with high probability, i.e., at least $1 - 1/n^c$ for any constant c.

In Section 5 we give a Las-Vegas version of our sparse suffix tree algorithm. This is achieved by developing a deterministic verifier for the answers to a batch of b longest common prefix queries. We show that this verifier can be used to obtain the sparse suffix tree with certainty in $O(n \log^2 b + b^2 \log b)$ time with high probability using only $O(b)$ space. For example for $b = O(\sqrt{n})$ we can construct the sparse suffix tree correctly in $O(n \log^2 b)$ time with high probability using $O(b)$ space in the worst case. This follows because, for verification, a single batch of b LCP queries suffices to check the sparse suffix tree. The verifier we develop encodes the relevant structure of the text in a graph with $O(b)$ edges. We then exploit novel properties of this graph to verify the answers to the LCP queries efficiently.

Finally, in Section 6, we show some tradeoffs of construction time and space usage of our Monte-Carlo algorithm, which are based on time-space tradeoffs of the batched LCP queries. In particular we show that using $O(b\alpha)$ space the construction time is reduced to $O\left(n\frac{\log^2 b}{\log \alpha} + \frac{\alpha b \log^2 b}{\log \alpha}\right)$. So, for example, the cost for constructing the sparse suffix tree can be reduced to $O(n \log b)$ time, using $O(b^{1+\varepsilon})$ words of space where $\varepsilon > 0$ is any constant.

2 Preliminaries

For a string $T = t_1 \cdots t_n$ of length n, denote by $T_i = t_i \cdots t_n$ the i^{th} suffix of T. The LCP of two suffixes T_i and T_j is denoted by $LCP(T_i, T_j)$, but we will slightly abuse notation and write $LCP(i, j) = LCP(T_i, T_j)$. We denote by $T_{i,j}$ the substring $t_i \cdots t_j$. We say that $T_{i,j}$ has period $\rho > 0$ iff $T_{i+\rho,j} = T_{i,j-\rho}$. Note that ρ is *a period* of $T_{i,j}$ and not necessarily the unique minimal period of $T_{i,j}$, commonly referred to as *the period*.

We assume the reader is familiar with both the suffix tree data structure [18] as well as suffix and LCP arrays [14].

Fingerprinting We make use of the fingerprinting techniques of Karp and Rabin [11]. Our algorithms are in the word-RAM model with word size $\Theta(\log n)$ and we assume that each character in T fits in a constant number of words. Hence each character can be interpreted as a positive integer, no larger than $n^{O(1)}$. Let p be a prime between n^c and $2n^c$ (where $c > 0$ is a constant picked below) and choose $r \in \mathbb{Z}_p$ uniformly at random. A fingerprint for a substring $T_{i,j}$, denoted by $FP[i, j]$, is the number $\sum_{k=i}^{j} r^{j-k} \cdot t_k \mod p$. Two equal substrings will always have the same fingerprint, however the converse is not true. Fortunately, as each character fits in $O(1)$ words, the probability of any two different substrings having the same fingerprint is at most by $n^{-\Omega(1)}$ [11]. By making a suitable choice of c and applying the union bound we can ensure that with probability at least $1 - n^{-\Omega(1)}$, all fingerprints of substring of T are collision free. I.e. for every pair of substrings T_{i_1,j_1} and T_{i_2,j_2} we have that $T_{i_1,j_1} = T_{i_2,j_2}$

iff $FP[i_1, j_1] = FP[i_2, j_2]$. The exponent in the probability can be amplified by increasing the value of c. As c is a constant, any fingerprint fits into a constant number of words.

We utilize two important properties of fingerprints. The first is that $FP[i, j+1]$ can be computed from $FP[i, j]$ in constant time. This is done by the formula $FP[i, j+1] = FP[i, j] \cdot r + t_{j+1} \mod p$. The second is that the fingerprint of $T_{k,j}$ can be computed in $O(1)$ time from the fingerprint of $T_{i,j}$ and $T_{i,k}$, for $i \leq k \leq j$. This is done by the formula $FP[k, j] = FP[i, j] - FP[i, k] \cdot r^{j-k} \mod p$. Notice however that in order to perform this computation, we must have stored $r^{j-k} \mod p$ as computing it on the fly may be costly.

3 Batched LCP Queries

3.1 The Algorithm

Given a string T of length n and a list of q pairs of indices P, we wish to compute $LCP(i, j)$ for all $(i, j) \in P$. To do this we perform $\log q$ rounds of computation, where at the k^{th} round the input is a set of q pairs denoted by P_k, where we are guaranteed that for any $(i, j) \in P_k, LCP(i, j) \leq 2^{\log n-(k-1)}$. The goal of the k^{th} iteration is to decide for any $(i, j) \in P_k$ if $LCP(i, j) \leq 2^{\log n-k}$ or not. In addition, the k^{th} round will prepare P_{k+1}, which is the input for the $(k+1)^{\text{th}}$ round. To begin the execution of the procedure we set $P_0 = P$, as we are always guaranteed that for any $(i, j) \in P$, $LCP(i, j) \leq n = 2^{\log n}$. We will first provide a description of what happens during each of the $\log q$ rounds, and after we will explain how the algorithm uses $P_{\log q}$ to derive $LCP(i, j)$ for all $(i, j) \in P$.

A Single Round The k^{th} round, for $1 \leq k \leq \log q$, is executed as follows. We begin by constructing the set $L = \bigcup_{(i,j)\in P_k} \{i-1, j-1, i+2^{\log n-k}, j+2^{\log n-k}\}$ of size $4q$, and construct a perfect hash table for the values in L, using a 2-wise independent hash function into a world of size q^c for some constant c (which with high probability guarantees that there are no collisions). Notice if two elements in L have the same value, then we store them in a list at their hashed value. In addition, for every value in L we store which index created it, so for example, for $i-1$ and $i+2^{\log n-k}$ we remember that they were created from i.

Next, we scan T from t_1 till t_n. When we reach t_ℓ we compute $FP[1, \ell]$ in constant time from $FP[1, \ell-1]$. In addition, if $\ell \in L$ then we store $FP[1, \ell]$ together with ℓ in the hash table. Once the scan of T is completed, for every $(i, j) \in P_k$ we compute $FP[i, i+2^{\log n-k}]$ in constant time, as we stored $FP[1, i-1]$ and $FP[1, i+2^{\log n-k}]$. Similarly we compute $FP[j, j+2^{\log n-k}]$. Notice that to do this we need to compute $r^{2^{\log n-k}} \mod p = r^{\frac{n}{2^k}}$ in $O(\log n - k)$ time, which can be easily afforded within our bounds, as one computation suffices for all pairs.

If $FP[i, i+2^{\log n-k}] \neq FP[j, j+2^{\log n-k}]$ then $LCP(i, j) < 2^{\log n-k}$, and so we add (i, j) to P_{k+1}. Otherwise, with high probability $LCP(i, j) \geq 2^{\log n-k}$ and so we add $(i+2^{\log n+k}, j+2^{\log n+k})$ to P_{k+1}. Notice there is a natural bijection between pairs in P_{k-1} and pairs in P following from the method of constructing

the pairs for the next round. For each pair in P_{k+1} we will remember which pair in P originated it, which can be easily transferred when P_{k+1} is constructed from P_k.

LCP on Small Strings After the $\log q$ rounds have taken place, we know that for every $(i,j) \in P_{\log q}$, $LCP(i,j) \leq 2^{\log n - \log q} = \frac{n}{q}$. For each such pair, we spend $O(\frac{n}{q})$ time in order to exactly compute $LCP(i,j)$. Notice that this is performed for q pairs, so the total cost is $O(n)$ for this last phase. We then construct $P_{final} = \{(i + LCP(i,j), j + LCP(i,j)) : (i,j) \in P_{\log q}\}$. For each $(i,j) \in P_{final}$ denote by $(i_0, j_0) \in P$ the pair which originated (i,j). We claim that for any $(i,j) \in P_{final}$, $LCP(i_0, j_0) = i - i_0$.

3.2 Runtime and Correctness

Each round takes $O(n+q)$ time, and the number of rounds is $O(\log q)$ for a total of $O((n+q)\log q)$ time for all rounds. The work executed for computing P_{final} is an additional $O(n)$.

The following lemma on LCPs, which follows directly from the definition, will be helpful in proving the correctness of the batched LCP query.

Lemma 1. *For any $1 \leq i, j \leq n$, for any $0 \leq m \leq LCP(i,j)$, it holds that $LCP(i+m, j+m) + m = LCP(i,j)$.*

We now proceed on to prove that for any $(i,j) \in P_{final}$, $LCP(i_0, j_0) = i - i_0$. Lemma 2 shows that the algorithm behaves as expected during the $\log q$ rounds, and Lemma 3 proves that the work done in the final round suffices for computing the LCPs.

Lemma 2. *At round k, for any $(i_k, j_k) \in P_k$, $i_k - i_0 \leq LCP(i_0, j_0) \leq i_k - i_0 + 2^{\log n - k}$, assuming the fingerprints do not give a false positive.*

Proof. The proof is by induction on k. For the base, $k = 0$ and so $P_0 = P$ meaning that $i_k = i_0$. Therefore, $i_k - i_0 = 0 \leq LCP(i_0, j_0) \leq 2^{\log n} = n$, which is always true. For the inductive step, we assume correctness for $k-1$ and we prove for k as follows. By the induction hypothesis, for any $(i_{k-1}, j_{k-1}) \in P_{k-1}$, $i - i_0 \leq LCP(i_0, j_0) \leq i - i_0 + 2^{\log n - k + 1}$. Let (i_k, j_k) be the pair in P_k corresponding to (i_{k-1}, j_{k-1}) in P_{k-1}. If $i_k = i_{k-1}$ then $LCP(i_{k-1}, j_{k-1}) < 2^{\log n - k}$. Therefore,

$$\begin{aligned} i_k - i_0 = i_{k-1} - i_0 &\leq LCP(i_0, j_0) \\ &\leq i_{k-1} - i_0 + LCP(i_{k-1}, j_{k-1}) \leq i_k - i_0 + 2^{\log n - k}. \end{aligned}$$

If $i_k = i_{k-1} + 2^{\log n - k}$ then $FP[i, i + 2^{\log n - k}] = FP[j, j + 2^{\log n - k}]$, and because we assume that the fingerprints do not produce false positives, $LCP(i_{k-1}, j_{k-1}) \geq 2^{\log n - k}$. Therefore,

$$\begin{aligned} i_k - i_0 = i_{k-1} + 2^{\log n - k} - i_0 &\leq i_{k-1} - i_0 + LCP(i_{k-1}, j_{k-1}) \\ &\leq LCP(i_0, j_0) \leq i_{k-1} - i_0 + 2^{\log n - k + 1} \\ &\leq i_k - i_0 + 2^{\log n - k}, \end{aligned}$$

where the third inequality holds from Lemma 1, and the fourth inequality holds as $LCP(i_0, j_0) = i_{k-1} - i_0 + LCP(i_{k-1}, j_{k-1})$ (which is the third inequality), and $LCP(i_{k-1}, j_{k-1}) \leq 2^{\log n - k + 1}$ by the induction hypothesis. □

Lemma 3. *For any* $(i, j) \in P_{final}$, $LCP(i_0, j_0) = i - i_0 (= j - j_0)$.

Proof. Using Lemma 2 with $k = \log q$ we have that for any $(i_{\log q}, j_{\log q}) \in P_{\log q}$, $i_{\log q} - i_0 \leq LCP(i_0, j_0) \leq i_{\log q} - i_0 + 2^{\log n - \log q} = i_{\log q} - i_0 + \frac{n}{q}$. Because $LCP(i_{\log q}, j_{\log q}) \leq 2^{\log n - \log q}$ it must be that $LCP(i_0, j_0) = i_{\log q} - i_0 + LCP(i_{\log q}, j_{\log q})$. Notice that $i_{final} = i_{\log q} + LCP(i_{\log q}, j_{\log q})$. Therefore, $LCP(i_0, j_0) = i_{final} - i_0$ as required. □

Notice that the space used in each round is the set of pairs and the hash table for L, both of which require only $O(q)$ words of space. Thus, we have obtained the following. We discuss several other time/space tradeoffs in Section 6.

Theorem 1. *There exists a randomized Monte-Carlo algorithm that with high probability correctly answers a batch of q LCP queries on suffixes from a string of length n. The algorithm uses $O((n+q)\log q)$ time and $O(q)$ space in the worst case.*

4 Constructing the Sparse Suffix Tree

We now describe a Monte-Carlo algorithm for constructing the sparse suffix tree on any b suffixes of T in $O(n \log^2 b)$ time and $O(b)$ space. The main idea is to use batched LCP queries in order to sort the b suffixes, as once the LCP of two suffixes is known, deciding which is lexicographically smaller than the other takes constant time by examining the first two characters that differ in said suffixes.

To arrive at the claimed complexity bounds, we are interested in grouping the LCP queries into $O(\log b)$ batches each containing $q = O(b)$ queries on pairs of suffixes. One way to do this is to simulate a sorting network on the b suffixes of depth $\log b$ [1]. Unfortunately, such known networks have very large constants hidden in them, and are generally considered impractical [16]. There are some practical networks with depth $\log^2 b$ such as [4], however, we wish to do better.

Consequently, we choose to simulate the quick-sort algorithm by each time picking a random suffix called the pivot, and lexicographically comparing all of the other $b - 1$ suffixes to the pivot. Once a partition is made to the set of suffixes which are lexicographically smaller than the pivot, and the set of suffixes which are lexicographically larger than the pivot, we recursively sort each set in the partition with the following modification. Each level of the recursion tree is performed concurrently using one single batch of $q = O(b)$ LCP queries for the entire level. Thus, by Theorem 1 a level can be computed in $O(n \log b)$ time and $O(b)$ space. Furthermore, with high probability, the number of levels in the randomized quicksort is $O(\log b)$, so the total amount of time spent is $O(n \log^2 b)$ with high probability. The time bound can immediately be made worst-case by aborting if the number of levels becomes too large, since the algorithm is still guaranteed to return the correct answer with high probability.

Notice that once the suffixes have been sorted, then we have in fact computed the sparse suffix array for the b suffixes. Moreover, the corresponding sparse LCP array can be obtained as a by-product or computed subsequently by a answering a single batch of $q = O(b)$ LCP queries in $O(n \log b)$ time. Hence we have obtained the following.

Theorem 2. *There exists a randomized Monte-Carlo algorithm that with high probability correctly constructs the sparse suffix array and the sparse LCP array for any b suffixes from a string of length n. The algorithm uses $O(n \log^2 b)$ time and $O(b)$ space in the worst case.*

Having obtained the sparse suffix and LCP arrays, the sparse suffix tree can be constructed deterministically in $O(b)$ time and space using well-known techniques, e.g. by simulating a bottom-up traversal of the tree [12].

Corollary 1. *There exists a randomized Monte-Carlo algorithm that with high probability correctly constructs the sparse suffix tree on b suffixes from a string of length n. The algorithm uses $O(n \log^2 b)$ time and $O(b)$ space in the worst case.*

5 Verifying the Sparse Suffix and LCP Arrays

In this section we give a deterministic algorithm which verifies the correctness of the sparse suffix and LCP arrays constructed in Theorem 2. This immediately gives a Las-Vegas algorithm for constructing either the sparse suffix array or sparse suffix tree with certainty. For space reasons some proofs are omitted.

First observe that as lexicographical ordering is transitive it suffices to verify the correct ordering of each pair of indices which are adjacent in the sparse suffix array. The correct ordering of suffixes T_i and T_j can be decided deterministically in constant time by comparing $t_{i+LCP(i,j)}$ to $t_{j+LCP(i,j)}$. Therefore the problem reduces to checking the LCP of each pair of indices which are adjacent in the sparse suffix array. These LCPs are computed as a by-product of our Monte-Carlo algorithm, and there is a small probability that they are incorrect.

Therefore our focus in this section is on giving a deterministic algorithm which verifies the correctness of the answers to a batch of b LCP queries. As before, to do this we perform $O(\log b)$ rounds of computation. The rounds occur in decreasing order. In the k^{th} round the input is a set of (at most) b index pairs to be verified. Let $\{x, y\}$ be such a pair of indices, corresponding to a pair of substrings $T_{x,x+m_k-1}$ and $T_{y,y+m_k-1}$ where $m_k = 2^k$. We say that $\{x, y\}$ *matches* iff $T_{x,x+m_k-1} = T_{y,y+m_k-1}$. In round k we will replace each pair $\{x, y\}$ with a new pair $\{x', y'\}$ to be inserted into round $(k-1)$ such that $T_{x,x+m_k-1} = T_{y,y+m_k-1}$ iff $T_{x',x'+m_{k-1}-1} = T_{y',y'+m_{k-1}-1}$. Each new pair will in fact always correspond to substrings of the old pair. In some cases we may choose to directly verify some $\{x, y\}$, in which case no new pair is inserted into the next round. The initial, largest value of k is the largest integer such that $m_k < n$. We perform $O(\log b)$ rounds, halting when $n/b < m_k < 2n/b$ after which point we can verify all pairs by scanning T in $O(m_k \cdot b) = O(n)$ time.

Of course an original query pair $\{x, y\}$ may not have $LCP(T_x, T_y) = m_k$ for any k. This is resolved by inserting two overlapping pairs into round k where $m_{k-1} < LCP(T_x, T_y) < m_k$. If the verifier succeeds, for each original pair we have that $T_{x,x+LCP(T_x,T_y)-1}$ equals $T_{y,y+LCP(T_x,T_y)-1}$. We also need to check that $t_{x+LCP(T_x,T_y)}$ does not equal $t_{x+LCP(T_x,T_y)}$ - otherwise the true LCP value is larger than was claimed. Where it is clear from context, for simplicity, we abuse notation by letting $m = m_k$. We now focus on an arbitrary round k.

The Suffix Implication Graph We now build a graph (V, E) which will encode the structure in the text. We build the vertex set V greedily. Consider each text index $1 \leq x \leq n$ in ascending order. We include index x as a vertex in V iff it occurs in some pair $\{x, y\}$ (or $\{y, x\}$) and the last index included in V was at least $m/(9 \cdot \log b)$ characters ago. Observe that $|V| \leq 9 \cdot (n/m) \log b$ and also varies between $9 \cdot \log b \leq |V| \leq b$ as it contains at most one vertex per index pair.

Each pair of indices $\{x, y\}$ corresponds to an edge between vertices $v(x)$ and $v(y)$. Here $v(x)$ is the unique vertex such that $v(x) \leq x < v(x)+m/(9 \cdot \log b)$. The vertex $v(y)$ is defined analogously. This may create multiple edges between two vertices $v(x)$ and $v(y)$. Any multi-edges imply a two-cycle and can be handled first using a simplification of the main algorithm without increasing the overall time or space complexity. For brevity we omit this case and continue under the assumption that there are no multi-edges. It is simple to build the graph in $O(b \log b)$ time by traversing the pairs. As $|E| \leq b$ we can store the graph in $O(b)$ space.

We now discuss the structure of the graph constructed and show how it can be exploited to efficiently solve the problem. The following simple lemma will be essential to our algorithm and underpins the main arguments below.

Lemma 4. *Let (V', E') be a connected component of an undirected graph in which every vertex has degree at least three. There is a (simple) cycle in (V', E') of length at most $2 \log |V'| + 1$.*

The graph we have constructed may have vertices with degree less than three, preventing us from applying Lemma 4. For each vertex $v(x)$ with degree less than three, we verify every index pair $\{x, y\}$ (which corresponds to an edge $(v(x), v(y))$). By directly scanning the corresponding text portions this takes $O(|V|m)$ time. We can then safely remove all such vertices and the corresponding edges. This may introduce new low degree vertices which are then verified iteratively in the same manner. As $|V| \leq 9 \cdot (n/m) \log b$, this takes a total of $O(n \log b)$ time. In the remainder we continue under the assumption that every vertex has degree at least three.

Algorithm Summary The algorithm for round k processes each connected component separately. However the time complexity arguments will be amortized over all components. Consider a connected component (V', E'). As every vertex has degree at least three, any component has a short cycle of length at

most $2\log|V'|+1 \leq 2\log b+1$ by Lemma 4. We begin by finding such a cycle in $O(b)$ time by performing a BFS of (V', E') starting at any vertex (this follows immediately from the proof of Lemma 4). Having located such a cycle, we will distinguish two cases. The first case is when the cycle is lock-stepped (defined below) and the other when it is unlocked. In both cases we will show below that we can exploit the structure of the text to safely delete an edge from the cycle, breaking the cycle. The index pair corresponding to the deleted edge will be replaced by a new index pair to be inserted into the next round where $m \leftarrow m_{k-1} = m_k/2$. Observe that both cases reduce the number of edges in the graph by one. Whenever we delete an edge we may reduce the degree of some vertex to below three. In this case we immediately directly process this vertex in $O(m)$ time as discussed above (iterating if necessary). As we do this at most once per vertex (and $O(|V|m) = O(n\log b)$), this does not increase the overall complexity. We then continue by finding and processing the next short cycle. The algorithm therefore searches for a cycle at most $|E| \leq b$ times over all components, contributing an $O(b^2)$ time additive term. In the remainder we will explain the two cycle cases in more detail and prove that summed over all components, the time complexity for round k is upper bounded by $O(n\log b)$ (excluding finding the cycles). As there are $O(\log b)$ rounds the final time complexity is $O(n\log^2 b + b^2\log b)$ and the space is $O(b)$.

Cycles We now define a lock-stepped cycle. Let $(v(x_i), v(y_i))$ for $i = 1\ldots\ell$ be a cycle of length at most $2\log b+1$, i.e. $v(y_i) = v(x_{i+1})$ for $1 \leq i < \ell$ and $v(y_\ell) = v(x_1)$. Here $\{x_i, y_i\}$ for all i are the underlying text index pairs. Let $d_i = x_{i+1} - y_i$ for $1 \leq i < \ell$, $d_\ell = x_1 - y_\ell$ and let $\rho = \sum_{i=1}^{\ell} d_i$. We say that the cycle is *lock-stepped* iff $\rho = 0$ (and *unlocked* otherwise). Intuitively, lock-stepped cycles are ones where all the underlying pairs are in sync. Lemma 5 gives the key property of lock-stepped cycles which we will use.

Lemma 5. *Let $(v(x_i), v(y_i))$ for $i = 1\ldots\ell$ be the edges in a lock-stepped cycle. Further let $j = \arg\max \sum_{i=1}^{j} d_j$. If $\{x_i, y_i\}$ match for all $i \neq j$ then $T_{x_j, x_j+m/2-1} = T_{y_j, y_j+m/2-1}$.*

Case 1: Lock-stepped Cycles The first, simpler case is when we find a lock-stepped cycle in the connected component (V', E'). By Lemma 5, once we have found a lock-stepped cycle we can safely remove some single edge, $(v(x_j), v(y_j))$ from the cycle. When we remove a single edge, we still need to verify the right half of the removed pair, $\{x_j, y_j\}$. This is achieved by inserting a new pair, $\{x_j + m/2, y_j + m/2\}$ into the next round where $m \leftarrow m_{k-1} = m_k/2$. We can determine which edge can be deleted by traversing the cycle in $O(\log b)$ time. Processing all lock-stepped cycles (over all components) takes $O(b\log b)$ time in total.

Case 2: Unlocked Cycles The remaining case is when we find an unlocked cycle in the connected component (V', E'). Lemma 6 tells us that in this case (if all pairs match) then one of the pairs, $\{x_j, y_j\}$ corresponding to an edge, $(v(x_j), v(y_j))$ in

the cycle must have a long, periodic prefix. We can again determine the suitable pair $\{x_j, y_j\}$ as well as ρ in $O(\log b)$ time by inspecting the cycle. This follows immediately from the statement of the lemma and the definition of ρ.

Lemma 6. *Assume that the connected component, (V', E') contains an unlocked cycle denoted, $(v(x_i), v(y_i))$ for $i = 1 \dots \ell$. Further let $j = \arg\max \sum_{i=1}^{j} d_j$. If $\{x_i, y_i\}$ match for all $i = 1 \dots \ell$ then $T_{x_j, x_j+3m/4-1}$ has a period $|\rho| \leq m/4$.*

Consider some pair $\{x, y\}$ such that both $T_{x,x+3m/4-1}$ and $T_{y,y+3m/4-1}$ are periodic with period at most $m/4$. We have that $T_{x,x+m-1} = T_{y,y+m-1}$ iff $T_{x+m/2,x+m-1} = T_{y+m/2,y+m-1}$. This is because $T_{x+m/2,x+m-1}$ contains at least a full period of characters from $T_{x,x+3m/4-1}$, and similarly with $T_{y+m/2,y+m-1}$ and $T_{y,y+3m/4-1}$ analogously. So we have that if $\{x_i, y_i\}$ match for all $i \neq j$ then the chosen pair $\{x_j, y_j\}$ matches iff both $T_{x_j,x_j+3m/4-1}$ and $T_{y_j,y_j+3m/4-1}$ have period $|\rho| \leq m/4$ and $T_{x_j+m/2,x_j+m-1} = T_{y_j+m/2,y_j+m-1}$. We can therefore delete the pair $\{x_j, y_j\}$ (and the corresponding edge, $(v(x_j), v(y_j))$) and insert a new pair, $\{x_j + m/2, y_j + m/2\}$ into the next round where $m \leftarrow m_{k-1} = m/2$.

However, for this approach to work we still need to verify that both strings $T_{x_j,x_j+3m/4-1}$ and $T_{y_j,y_j+3m/4-1}$ have $|\rho|$ as period. We do not immediately check the periodicity, we instead delay computation until the end of round k, after all cycles have been processed. At the current point in the algorithm, we simply add the tuple $(\{x, y\}, \rho)$ to a list, Π of text substrings to be checked later for periodicity. This takes $O(b)$ space over all components. Excluding checking for periodicity, processing all unlocked cycles takes $O(b \log b)$ time in total.

Checking for Substring Periodicity The final task in round k is to scan the text and check that for each $(\{x, y\}, \rho) \in \Pi$, $|\rho|$ is a period of both $T_{x,x+3m/4-1}$ and $T_{y,y+3m/4-1}$. We process the tuples in left to right order. On the first pass we consider $T_{x,x+3m/4-1}$ for each $(\{x, y\}, \rho) \in \Pi$. In the second pass we consider y. The two passes are identical and we focus on the first.

We begin by splitting the tuples greedily into groups in left to right order. A tuple $(\{x, y\}, \rho)$ is in the same group as the previous tuple iff the previous tuple $(\{x', y'\}, \rho')$ has $x - x' \leq m/4$. Let $T_{z,z+m'-1}$ be the substring of T which spans every substring, $T_{x,x+3m/4-1}$ which appears in some $(\{x, y\}, \rho)$ in a single group of tuples. We now apply the classic periodicity lemma stated below.

Lemma 7 (see e.g. [8]). *Let S be a string with periods ρ_1 and ρ_2 and with $|S| > \rho_1 + \rho_2$. S has period $\gcd(\rho_1, \rho_2)$, the greatest common divisor of ρ_1 and ρ_2. Also, if S has period ρ_3 then S has period $\alpha \cdot \rho_3 \leq |S|$ for any integer $\alpha > 0$.*

First observe that consecutive tuples $(\{x, y\}, \rho)$ and $(\{x', y'\}, \rho')$ in the same group have overlap least $m/2 \geq |\rho| + |\rho'|$. Therefore by Lemma 7, if $T_{x,x+3m/4-1}$ has period $|\rho|$ and $T_{x',x'+3m/4-1}$ has period $|\rho'|$ then their overlap also has $\gcd(|\rho|, |\rho'|)$ as a period. However as their overlap is longer than a full period in each string, both $T_{x,x+3m/4-1}$ and $T_{x',x'+3m/4-1}$ also have period $\gcd(|\rho|, |\rho'|)$. By repeat application of this argument we have that if for every tuple $(\{x, y\}, \rho)$, the substring $T_{x,x+3m/4-1}$ has period $|\rho|$ then $T_{z,z+m'-1}$ has a period equal to

the greatest common divisor of the periods of all tuples in the group, denoted g. To process the entire group we can simply check whether $T_{z,z+m'-1}$ has period g in $O(m')$ time. If $T_{z,z+m'-1}$ does not have period g, we can safely abort the verifier. If $T_{z,z+m'-1}$ has period g then by Lemma 7, for each $(\{x, y\}, \rho)$ in the group, $T_{x,x+3m/4-1}$ has period $|\rho|$ as g divides $|\rho|$. As every $m' \geq 3m/4$ and the groups overlap by less than $m/2$ characters, this process takes $O(n)$ total time.

Theorem 3. *There exists a randomized Las-Vegas algorithm that correctly constructs the sparse suffix array and the sparse LCP array for any b suffixes from a string of length n. The algorithm uses $O(n \log^2 b + b^2 \log b)$ time with high probability and $O(b)$ space in the worst case.*

6 Time-Space Tradeoffs for Batched LCP Queries

We provide an overview of the techniques used to obtain the time-space tradeoff for the batched LCP process, as it closely follows those of Section 3. In Section 3 the algorithm simulates concurrent binary searches in order to determine the LCP of each input pair (with some extra work at the end). The idea for obtaining the tradeoff is to generalize the binary search to an α-ary search. So in the k^{th} round the input is a set of q pairs denoted by P_k, where we are guaranteed that for any $(i, j) \in P_k, LCP(i, j) \leq 2^{\log n-(k-1)\log \alpha}$, and the goal of the k^{th} iteration is to decide for any $(i, j) \in P_k$ if $LCP(i, j) \leq 2^{\log n-k \log \alpha}$ or not. From a space perspective, this means we need $O(\alpha q)$ space in order to compute α fingerprints for each index in any $(i, j) \in P_k$. From a time perspective, we only need to perform $O(\log_\alpha q)$ rounds before we may begin the final round. However, each round now costs $O(n + \alpha q)$, so we have the following trade-off.

Theorem 4. *Let $2 \leq \alpha \leq n$. There exists a randomized Monte-Carlo algorithm that with high probability correctly answers a batch of q LCP queries on suffixes from a string of length n. The algorithm uses $O((n+\alpha q) \log_\alpha q)$ time and $O(\alpha q)$ space in the worst case.*

In particular, for $\alpha = 2$, we obtain Theorem 1 as a corollary. Consequently, the total time cost for constructing the sparse suffix tree in $O(\alpha b)$ space becomes

$$O\left(n\frac{\log^2 b}{\log \alpha} + \frac{\alpha b \log^2 b}{\log \alpha}\right).$$

If, for example, $\alpha = b^\varepsilon$ for a small constant $\varepsilon > 0$, the cost for constructing the sparse suffix tree becomes $O(\frac{1}{\varepsilon}(n \log b + b^{1+\varepsilon} \log b))$, using $O(b^{1+\varepsilon})$ words of space. Finally by minimizing with the standard $O(n)$ time, $O(n)$ space algorithm we achieve the stated result of $O(n \log b)$ time, using $O(b^{1+\varepsilon})$ space.

References

1. M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ Sorting Network. In *Proc. 15th STOC*, pages 1–9, 1983.

2. A. Andersson, N. J. Larsson, and K. Swanson. Suffix Trees on Words. In *Proc. 7th CPM (LNCS 1075)*, pages 102–115, 1996.
3. A. Andersson, N. J. Larsson, and K. Swanson. Suffix Trees on Words. *Algorithmica*, 23(3):246–260, 1999.
4. K. E. Batcher. Sorting Networks and Their Applications. In *Proc. AFIPS Spring JCC*, pages 307–314, 1968.
5. J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. 8th SODA*, pages 360–369, 1997.
6. S. Burkhardt and J. Kärkkäinen. Fast Lightweight Suffix Array Construction and Checking. In *Proc. 14th CPM (LNCS 2676)*, pages 55–69, 2003.
7. P. Ferragina and J. Fischer. Suffix Arrays on Words. In *Proc. 18th CPM (LNCS 4580)*, pages 328–339, 2007.
8. N. J. Fine and H. S. Wilf. Uniqueness Theorems for Periodic Functions. *Proc. AMS*, 16(1):109–114, 1965.
9. S. Inenaga and M. Takeda. On-line linear-time construction of word suffix trees. In *Proc. 17th CPM (LNCS 4009)*, pages 60–71, 2006.
10. J. Kärkkäinen and E. Ukkonen. Sparse Suffix Trees. In *Proc. 2nd COCOON (LNCS 1090)*, pages 219–230, 1996.
11. R. M. Karp and M. O. Rabin. Efficient Randomized Pattern-Matching Algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.
12. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In *Proc. 12th CPM (LNCS 2089)*, pages 181–192, 2001.
13. R. Kolpakov, G. Kucherov, and T. A. Starikovskaya. Pattern Matching on Sparse Suffix Trees. In *Proc. 1st CCP*, pages 92–97, 2011.
14. U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM J. Comput.*, 22(5):935–948, 1993.
15. D. R. Morrison. Patriciapractical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.
16. M. Paterson. Improved Sorting Networks with $O(\log N)$ Depth. *Algorithmica*, 5(1):65–92, 1990.
17. T. Uemura and H. Arimura. Sparse and truncated suffix trees on variable-length codes. In *Proc. 22nd CPM (LNCS 6661)*, pages 246–260, 2011.
18. P. Weiner. Linear Pattern Matching Algorithms. In *Proc. 14th FOCS (SWAT)*, pages 1–11, 1973.