Sparse Text Indexing in Small Space*

Philip Bille[1,†] phbi@dtu.dk
Johannes Fischer[2] johannes.fischer@cs.tu-dortmund.de
Inge Li Gørtz[1,†] inge@dtu.dk
Tsvi Kopelowitz[3] kopelot@gmail.com
Benjamin Sach[4] ben@cs.bris.ac.uk
Hjalte Wedel Vildhøj[1] hwv@hwv.dk

Abstract

In this work we present efficient algorithms for constructing sparse suffix trees, sparse suffix arrays and sparse positions heaps for b arbitrary positions of a text T of length n while using only $O(b)$ words of space during the construction.

Attempts at breaking the naive bound of $\Omega(nb)$ time for constructing sparse suffix trees in $O(b)$ space 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 b suffixes were evenly spaced in T. In this paper there is no constraint on the locations of the suffixes.

Our main contribution is to show that the sparse suffix tree (and array) can be constructed in $O(n \log^2 b)$ time. To achieve this we develop a technique, 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.

Finally, we show that at the expense of slower pattern queries, it is possible to construct sparse position heaps in $O(n + b \log b)$ time and $O(b)$ space.

[1] Technical University of Denmark, DTU Compute
[2] TU Dortmund, Department of Computer Science
[3] Weizmann Institute of Science, Faculty of Mathematics and Computer Science
[4] University of Bristol, Department of Computer Science

[†] Supported partly by a grant from the Danish Council for Independent Research | Natural Sciences. Supported by the Danish Research Council (DFF4005-00267, DFF 1323-00178) and the Advanced Technology Foundation.

[*] Preliminary version appeared in *Proceedings of the 40th International Colloquium on Automata, Languages and Programming*

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 [23], suffix arrays [19] and positions heaps [8] 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 [14]—the origins of string indexing in 1968 [20]. First results were only obtained in 1996, where Andersson et al. [2, 3] and Kärkkäinen and Ukkonen [14] 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, 13], and the result was recently generalised to variable length codes such as Huffman code [22]. The second restricted case [14] 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 [14] 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. [18]. 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 present the first improvements over 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 in $O(n\log^2 b + b^2\log b)$ time, using $O(b)$ space. We show that this verifier can be used to obtain the sparse suffix tree with certainty within the same time and space bounds. 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.

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.

Finally, in Section 7 we show that an entirely different data structure, the *position heap* of Ehrenfeucht et al. [8], yields a completely different tradeoff for indexing a sparse set of positions. Position heaps are in a sense "easier" to compute than suffix trees or suffix arrays, since it is not necessary to sort the *entire* suffixes. The price is that, in their plain form, pattern matching is slower than with suffix trees, namely $O(m^2)$ for a pattern of length m. Using this approach, we show how to index b positions from a text of length n using $O(n + b\log b)$ time and $O(b)$ space, such that subsequent pattern matching queries (finding the k occurrences starting at one of the b positions) can be answered in $O(m^2 + k)$ time, for patterns of length m. Again, this algorithm is Monte-Carlo and outputs correct answers with high probability.

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*. Logarithms are given in base two.

We assume the reader is familiar with both the suffix tree data structure [23] as well as suffix and LCP arrays [19].

Fingerprinting We make use of the fingerprinting techniques of Karp and Rabin [16]. 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 $n^{-\Omega(1)}$ [16]. 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 $(i, j) \in P_k$ whether $LCP(i, j) \leq 2^{\log n-k}$ 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 to 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 from $FP[1,i-1]$ and $FP[1, i+2^{\log n-k}]$, which we have stored. 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 $P_{\mathit{final}} = \{(i+LCP(i,j), j+LCP(i,j)) : (i,j) \in P_{\log q}\}$. For each $(i,j) \in P_{\mathit{final}}$ denote by $(i_0, j_0) \in P$ the pair which originated (i,j). We claim that for any $(i,j) \in P_{\mathit{final}}$, $LCP(i_0, j_0) = i - i_0$.

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 $(i,j) \in P_{\mathit{final}}$, $LCP(i_0, j_0) = i - i_0$. 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 $(i, j) \in P_{\mathit{final}}$, $LCP(i_0, j_0) = i - i_0 (= j - j_0)$.*

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 $i_{\mathit{final}} = i_{\log q} + LCP(i_{\log q}, j_{\log q})$. Therefore, $LCP(i_0, j_0) = i_{\mathit{final}} - i_0$ 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 will group 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 [21]. 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 [17].

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.

In fact our main contribution here is an efficient algorithm which solves the general problem of verifying that b arbitrary pairs of substrings of T match. As we will see below, this suffices to verify the correctness of the sparse suffix and LCP arrays. A naive approach to verifying that b arbitrary substring pairs of T match would be to verify each pair separately in $\Omega(nb)$ time. However, by exploiting the way the pairs overlap in T, we show how to do much better. In the statement of our result in Lemma 4, each substring (w_i or w_i') in the input is provided by giving the indices of its leftmost and rightmost character (i.e. in $O(1)$ words).

Lemma 4. *Given (the locations of) b pairs of substrings $(w_1, w_1'), \ldots, (w_b, w_b')$ of a string T of length n, there is a deterministic algorithm that decides whether all pairs match, i.e. whether $w_i = w_i'$ for* all *$i = 1, \ldots, b$, in time $O(n \log^2 b + b^2 \log b)$ and $O(b)$ working space.*

Before we prove Lemma 4, we first discuss how it can be used to verify a batch of LCP queries and then in turn to verify the sparse suffix array. Consider some LCP query (i, j) for which the answer $LCP(i, j)$ has been computed (perhaps incorrectly). By definition, it suffices to check that $T_{i,i+LCP(i,j)-1} = T_{j,j+LCP(i,j)-1}$ and $t_{i+LCP(i,j)} \neq t_{j+LCP(i,j)}$. The latter check takes $O(1)$ time per query while the former is exactly the problem solved in Lemma 4. Lemma 5 then follows immediately from Lemma 4 and the Monte-Carlo algorithm for batched LCP queries we gave in Theorem 1.

Lemma 5. *There exists a randomized Las-Vegas algorithm that correctly answers a batch of b LCP queries on suffixes from a string of length n. The algorithm runs in $O(n \log^2 b + b^2 \log b)$ time with high probability and uses $O(b)$ space in the worst case.*

Finally 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 any two 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 and the result then follows.

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.*

5.1 Proof of Lemma 4

As before, our algorithm performs $O(\log b)$ rounds of computation. The rounds occur in decreasing order. In round k the input is a set of (at most) b pairs of substrings to be verified. Every substring considered in round k has length $m_k = 2^k$. Therefore they can be described as a pair of indices $\{x, y\}$, 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 $T_{x,x+m_k-1} = T_{y,y+m_k-1}$. 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 in Lemma 4, substring pairs can have arbitrary lengths. This is resolved by inserting two overlapping pairs into the appropriate round. I.e. if the original input contains a pair of substrings $(T_{x,x+d-1}, T_{y,y+d-1})$ we insert two index pairs into round $k = \lfloor \log d \rfloor$:

$$\{x, y\} \text{ and } \{x + d - 1 - m_k, y + d - 1 - m_k\}\,.$$

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.

We now focus on an arbitrary round k and for brevity we let $m = m_k$ when k is clear from the context.

The Suffix Implication Graph We start each round by building a graph (V, E) which encodes the overlap structure of the pairs we are to verify. 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 $|V| \leq 2b$. It is simple to build the suffix implication graph in $O(b \log b)$ time by traversing the pairs in sorted order. As we will show next, $|E| \leq b$, so we can store the graph in $O(b)$ space.

See Figure 1 for an example of the suffix implication graph. 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. Where it is clear from context, we will abuse notation by using $\{x, y\}$ to refer to the match and the corresponding edge in the graph. Note that the graph can have multiple edges and self-loops, which we are going to treat as cycles of length 2 and 1, respectively.

We now discuss the structure of the graph constructed and show how it can be exploited to efficiently verify the pairs in a round. The following simple lemma will be essential to our algorithm and underpins the main arguments below. The result is folklore but we include a proof for completeness.

Lemma 6. *Let (V, E) be an undirected graph in which every vertex has degree at least three. There is a (simple) cycle in (V, E) of length at most $2 \log |V| + 1$.*

Proof. Let v be any vertex in V. First observe that as each vertex has degree at least three, there must be a cycle (keep walking until you get back to somewhere you've been before). Perform a breadth first search from v. Every time you increase the distance from v by one, either the number of different vertices seen doubles or a cycle is found. This is because each vertex has degree at least three (but you arrived via one of the edges). Two of these edges must lead to new, undiscovered vertices (or a cycle has been found). Therefore when a cycle is discovered the length of the path from v is at most $\log |V|$. Note that this cycle may not contain v. However as the distance from v to any vertex found is at most $\log |V'|$, the cycle length is at most $2 \log |V| + 1$. As v was arbitrary, the lemma follows. □

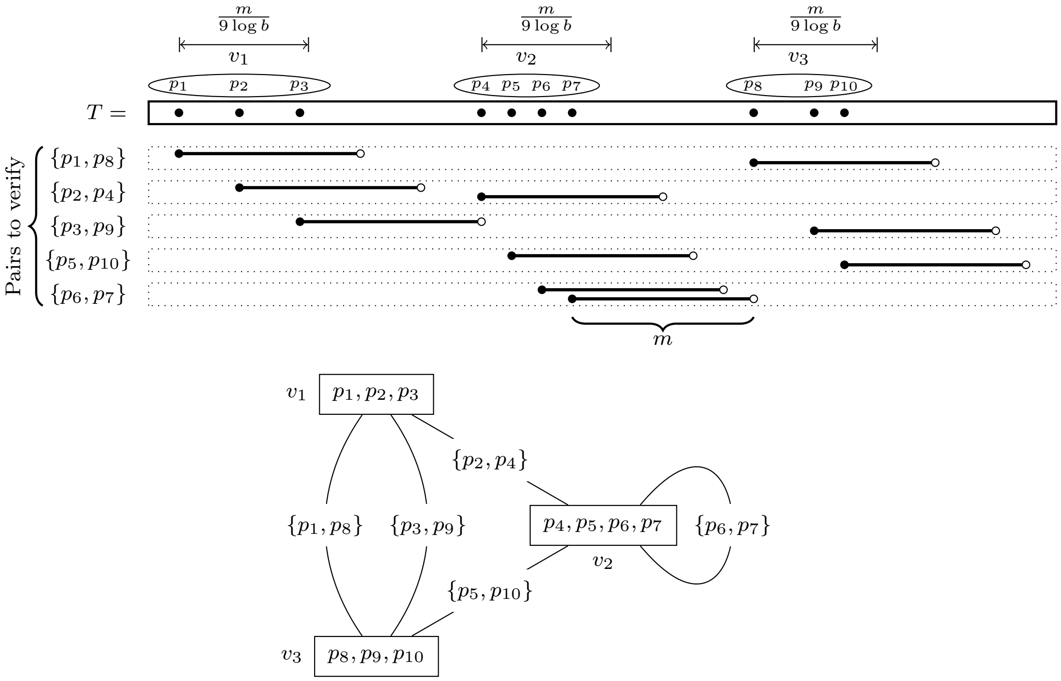


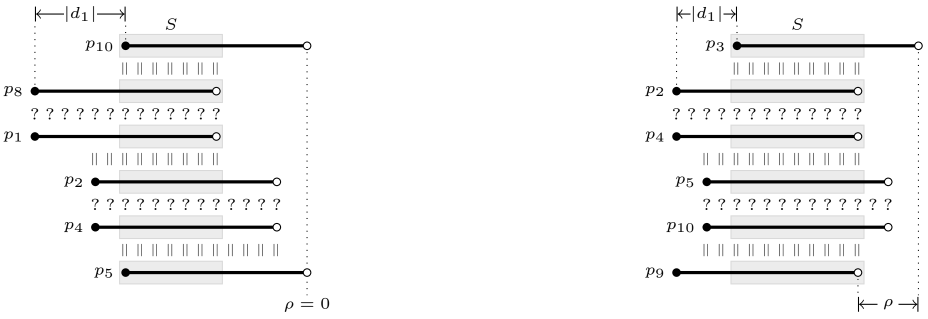
Figure 1: The suffix implication graph for a string T and 5 pairs of substrings to verify. In the case illustrated the vertex set constructed by the greedy algorithm has three vertices: v_1, v_2, v_3. Each edge in the suffix implication graph corresponds to a pair to be verified.

The graph we have constructed may have vertices with degree less than three, preventing us from applying Lemma 6. For each vertex $v(x)$ with degree less than three, we verify every index pair $\{x, y\}$ (which corresponds to a unique edge). 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 Consider the graph (V, E). As every vertex has degree at least three, there is a *short cycle* of length at most $2 \log |V| + 1 \leq 2 \log b + 1$ by Lemma 6. 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 6). 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, contributing an $O(b^2)$ time additive term. In the remainder we will explain the two cycle cases in more detail and prove that the time complexity for round k is upper bounded by $O(n \log b)$ (excluding finding the cycles). Recall we only perform $O(\log b)$ rounds (after which $m = O(n/b)$ and we verify all pairs naively), so 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 the pairs $\{x_i, y_i\}$ for $i \in \{1, \ldots, \ell\}$ define a cycle in the graph of length at most $2 \log b + 1$, i.e. $v(y_i) = v(x_{i+1})$ for $1 \leq i < \ell$ and $v(y_\ell) = v(x_1)$. Let $d_i = x_i - y_{i-1}$ for $1 < i < \ell$, $d_1 = x_1 - y_\ell$ 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. See Figure 2 for an example of a lock-stepped and an unlocked cycle in the suffix implication graph of Figure 1. We will discuss these examples in more detail below.

Observe that the definition given is well-defined in the event of multi-edges ($\ell = 2$) and self-loops ($\ell = 1$). For example in Figure 1, there is multi-edge cycle formed by $\{x_1, y_1\} = \{p_1, p_8\}$ and $\{x_2, y_2\} = \{p_9, p_3\}$ where $v(p_8) = v(p_9) = v_3$ and $v(p_3) = v(p_1) = v_1$. There is also a self-loop cycle formed by $\{x_1, y_1\} = \{p_6, p_7\}$ where $v(p_6) = v(p_7) = v_2$ which conforms with the definition as $\ell = 1$. Note that self-loops are always unlocked since $\rho = d_1 = x_1 - y_1 \neq 0$ (we can assume that $x_1 \neq y_1$, otherwise a match is trivial).



(a) A lock-stepped cycle.

(b) An unlocked cycle.

Figure 2: Illustrating two cases of cycles in the suffix implication graph shown in Figure 1. The lock-stepped cycle consists of the pairs $\{p_5, p_{10}\}$, $\{p_8, p_1\}$ and $\{p_2, p_4\}$. The unlocked cycle consists of the pairs $\{p_9, p_3\}$, $\{p_2, p_4\}$ and $\{p_5, p_{10}\}$. The symbol $\shortparallel$ is used between the characters that are known to be equal because the strings overlap in T, and ? is used between characters that do not necessarily match.

As discussed above our algorithm proceeds by repeatedly finding a short cycle in the suffix implication graph. Let $\{x_i, y_i\}$ for $i = 1 \ldots \ell$ define a cycle uncovered in (V, E) by the algorithm. There are now two cases, either the cycle is lock-stepped or, by definition, it is unlocked. We now discuss the key properties we use in each case to process this cycle. We begin with the simpler, lock-stepped cycle case.

5.1.1 Lock-stepped Cycles

For a lock-stepped cycle the key property we use is given in Lemma 7 below. In the lemma we prove that in any lock-stepped cycle there is a particular pair $\{x_j, y_j\}$ such that if every other pair $\{x_i, y_i\}$ matches (i.e. when $i \neq j$) then the left half of $\{x_j, y_j\}$ matches. Therefore to determine whether $\{x_j, y_j\}$ matches we need only check whether the right half of $\{x_j, y_j\}$ matches.

Before formally proving Lemma 7 we give an overview of the proof technique with reference to the example in Figure 2(a). The Figure 2(a) gives an illustration of the structure of the text underlying the cycle given by the nodes v_1, v_2 and v_3 in the suffix implication graph in Figure 1. More formally, it illustrates the cycle $\{x_1, y_1\} = \{p_5, p_{10}\}$, $\{x_2, y_2\} = \{p_8, p_1\}$ and $\{x_3, y_3\} = \{p_2, p_4\}$ where $v(p_{10}) = v(p_8) = v_3$, $v(p_1) = v(p_2) = v_1$ and $v(p_4) = v(p_5) = v_2$. Notice that in Figure 1 the substrings T_{p_8, p_8+m-1} and $T_{p_{10}, p_{10}+m-1}$ overlap in T and so overlap by the same amount in Figure 2(a). Further because they overlap in T we know that a portion of them is equal - this is indicated with $||$ symbols (drawn as a rotated =). Next consider the substrings T_{p_8, p_8+m-1} and T_{p_1, p_1+m-1} which correspond to a pair $\{p_8, p_1\}$ which should be verified for equality. To illustrate this we draw them with one directly above the other with ? symbols. The diagram then proceeds in this fashion for the other edges. Notice that because $\rho = 0$ it follows that the top substring $T_{p_{10}, p_{10}+m-1}$ is aligned directly above the bottom substring T_{p_5, p_5+m-1} and also forms a pair $\{p_5, p_{10}\}$ to be verified.

Consider the string S (the grey box in the diagram), which is a prefix of $T_{p_{10},p_{10}+m-1}$. As $T_{p_{10},p_{10}+m-1}$ and T_{p_8,p_8+m-1} overlap in T, the string S also occurs as a suffix of T_{p_8,p_8+m-1}. Now assume (as in the Lemma below) that both $\{p_8, p_1\}$ and $\{p_2, p_4\}$ match. This is equivalent to saying that all the ? symbols are equal. We therefore have (as illustrated) that S occurs as a substring of T_{p_1,p_1+m-1} as well. Continuing this argument we conclude that S is a prefix of T_{p_5,p_5+m-1}. As we demonstrate in the proof, $|S| > m/2$ and thus we have, for free, that the first half of $\{p_5, p_{10}\}$ matches. Lemma 7 formalises this intuition:

Lemma 7. *Let $\{x_i, y_i\}$ for $i = 1 \ldots \ell$ be the edges in a lock-stepped cycle. Further let $j = \arg\max \sum_{i=1}^{j} d_i$. If $\{x_i, y_i\}$ match for all $i \neq j$ then $\{x_j, y_j\}$ matches iff*

$$T_{x_j+m/2,x_j+m-1} = T_{y_j+m/2,y_j+m-1}\,.$$

Proof. In the following we will work with indices modulo ℓ, i.e. $x_{\ell+1}$ is x_1. As observed above, as the cycle is lock-stepped, it cannot be a self-loop and thus $\ell \geq 2$.

By assumption we have that $\{x_i, y_i\}$ matches for all $i \neq j$. This means that $T_{x_i,x_i+m-1} = T_{y_i,y_i+m-1}$ for all $i \neq j$. Let $\gamma = \sum_{i=1}^{j} d_i$ and observe that as $\rho = \sum_{i=1}^{\ell} d_i = 0$, by the maximality of j, we have that $\gamma \geq 0$. More generally we define $\gamma'_i = \gamma - \sum_{i'=1}^{i} d_{i'}$. We first show that for all $i \in \{1, 2, \ldots, \ell\}$ we have that $\gamma'_i + d_i = \gamma'_{i-1}$. This fact will be required below. Observe that $\gamma'_i \geq 0$ for all $i \neq j$ (or the maximality of γ is contradicted) and $\gamma'_j = 0$. For $i > 1$, it is immediate that,

$$\gamma'_i + d_i = \left(\gamma - \sum_{i'=1}^{i} d_{i'}\right) + d_i = \gamma - \sum_{i'=1}^{i-1} d_{i'} = \gamma'_{i-1}\,.$$

For $i = 1$ we have $\gamma'_\ell = \gamma - \sum_{i'=1}^{\ell} d_{i'} = \gamma$ and $\gamma'_1 = \gamma - d_1$ therefore, $\gamma'_1 + d_1 = \gamma = \gamma'_\ell$ (which is γ'_0 as we are working with indices modulo ℓ).

For notational simplicity let $S = T_{x_j,x_j+m-\gamma-1}$. We will show that for all $i \in \{1, 2, \ldots, \ell\}$, there is an occurrence of S in T_{y_i,y_i+m-1} starting at offset γ'_i, i.e. that $T_{y_i+\gamma'_i,y_i+\gamma'_i+m-\gamma-1} = S$. The result then follows almost immediately as $\gamma'_j = 0$ and we will show that $\gamma \leq m/4$.

We proceed by induction on i *in decreasing order modulo ℓ*, starting with $i = (j-1) \mod \ell$ and ending with $i = j \mod \ell$. That is we consider $i = (j-1)$ down to $i = 1$ followed by $i = \ell$, down to $i = j$.

We first show that in both the base case and the inductive step there is an occurrence of S in $T_{x_{i+1},x_{i+1}+m-1}$ starting at offset γ'_{i+1}. For the base case, $i = (j-1) \mod \ell$, by the definition of S, we immediately have that $T_{x_{i+1}+\gamma'_{i+1},x_{i+1}+\gamma'_{i+1}+m-\gamma-1} = S$ as $i+1 = j$ and $\gamma'_j = 0$. For the inductive step where $i \neq (j-1) \mod \ell$, by the inductive hypothesis we have that $T_{y_{i+1}+\gamma'_{i+1},y_{i+1}+\gamma'_{i+1}+m-\gamma-1} = S$. Further as $i+1 \neq j$, $\{x_{i+1}, y_{i+1}\}$ matches and therefore,

$$T_{x_{i+1}+\gamma'_{i+1},x_{i+1}+\gamma'_{i+1}+m-\gamma-1} = T_{y_{i+1}+\gamma'_{i+1},y_{i+1}+\gamma'_{i+1}+m-\gamma-1} = S\,.$$

Both the base case and the inductive step are now proven in an identical manner.

As the edges $\{x_i, y_i\}$ form a cycle, we have that have that $v(x_{i+1}) = v(y_i)$ and further that $x_{i+1} = y_i + d_{i+1}$. This means that T_{y_i,y_i+m-1} and $T_{x_{i+1},x_{i+1}+m-1}$ overlap in T by $m - |d_{i+1}|$

characters. I.e. there is a substring of T of length $m - |d_{i+1}|$ which is a prefix of $T_{x_{i+1},x_{i+1}+m-1}$ and a suffix of T_{y_i,y_i+m-1} or visa-versa (depending on the sign of d_{i+1}). In particular this implies that,

$$S = T_{x_{i+1}+\gamma'_{i+1},\, x_{i+1}+\gamma'_{i+1}+m-\gamma-1} = T_{(y_i+d_{i+1})+\gamma'_{i+1},\, (y_i+d_{i+1})+\gamma'_{i+1}+m-\gamma-1} = T_{y_i+\gamma'_i,\, y_i+\gamma'_i+m-\gamma-1} \,.$$

The first equality follows because $x_{i+1} = y_i + d_{i+1}$ and the second because $\gamma'_{i+1} + d_{i+1} = \gamma'_i$. This completes the inductive argument.

Finally observe that when $i = j$, we have that S occurs in T_{y_j,y_j+m-1} starting at offset $\gamma'_i = 0$. I.e. $T_{x_j,x_j+m-\gamma-1} = T_{y_j,y_j+m-\gamma-1}$. As $|d_j| < m/(9 \cdot \log b)$ and further $\ell \leq 2\log b + 1$ we have that $\gamma = \sum_{i=1}^{j} d_i \leq \sum_{i=1}^{\ell} |d_i| \leq m/4$, completing the proof. □

Processing lock-stepped Cycles We use Lemma 7 as follows. First we identify the edge $\{x_j, y_j\}$. This can be achieved by calculating $\sum_{i=1}^{j} d_i$ for all i by traversing the cycle in $O(\log b)$ time. We then delete this edge from the graph, breaking the cycle. However we still need to check that $T_{x_j+m/2,x_j+m-1} = T_{y_j+m/2,y_j+m-1}$. This is achieved by inserting a new pair, $\{x_j + m/2, y_j + m/2\}$ into the next round where $m \leftarrow m_{k-1} = m_k/2$. Processing all lock-stepped cycles in this way takes $O(b \log b)$ time in total as we remove an edge each time.

5.1.2 Unlocked Cycles

The remaining case is when we find an unlocked cycle in the graph (V, E). For an unlocked cycle, the key property is given in Lemma 8. This lemma is similar to the previous lemma for lock-stepped cycles in that it identifies a particular pair, $\{x_j, y_j\}$ such that if every other pair $\{x_i, y_i\}$,$i \neq j$ matches then $\{x_j, y_j\}$ matches if and only if two conditions hold. The first condition is the same as the condition in the previous Lemma. The second condition requires that the first three-quarters of both the strings have a small period. This second condition may seem nonintuitive but, when viewed in the correct light, follows fairly straight-forwardly from the fact that the cycle is unlocked.

Again, we begin with an overview of the proof technique. We focus on the forward direction, that is we assume that all $\{x_i, y_i\}$ match (including $i = j$) and show that the two properties required indeed hold. The reverse direction follows from the observation that $T_{x_j+m/2,x_j+m-1}$ contains a full period from $T_{x_j+m/2,x_j+m-1}$. This overview is again given with reference to the example in Figure 2(b). This illustration is constructed in the same manner as the illustration for a lock-stepped cycle in Figure 2(a). However this time it illustrates the unlocked cycle $\{p_9, p_3\}$, $\{p_2, p_4\}$ and $\{p_5, p_{10}\}$ where $v(p_3) = v(p_2) = v_1$, $v(p_4) = v(p_5) = v_2$ and $v(p_{10}) = v(p_9) = v_3$. See Section 5.1.1 for an explanation of the diagram.

Again consider the string S, which is a prefix of T_{p_3,p_3+m-1}. Assume that all three pairs $\{p_9, p_3\}$, $\{p_2, p_4\}$ and $\{p_5, p_{10}\}$ match. Similarly to for unlocked cycles, we can then show (as is illustrated) that the string S occurs as a substring of each of T_{p_2,p_2+m-1}, T_{p_4,p_4+m-1}, T_{p_5,p_5+m-1}, $T_{p_{10},p_{10}+m-1}$ and in particular T_{p_9,p_9+m-1}. Further as (by assumption), T_{p_9,p_9+m-1} equals T_{p_3,p_3+m-1} and so we have found two occurrences of S in T_{p_3,p_3+m-1}. Crucially we show in the proof that as the cycle is unlocked these are two distinct occurrences of S, which are $|\rho|$ characters apart. This in turn implies that T_{p_3,p_3+m-1} (and hence also T_{p_9,p_9+m-1}) has a long, periodic prefix as required.

Lemma 8. *Let $\{x_i, y_i\}$ for $i = 1 \ldots \ell$ be the edges in an unlocked cycle of length ℓ. Further let $j = \arg\max \sum_{i=1}^{j} d_i$. If $\{x_i, y_i\}$ match for all $i \neq j$ then $\{x_j, y_j\}$ matches iff both the following hold:*

1. $T_{x_j+m/2,x_j+m-1} = T_{y_j+m/2,y_j+m-1}$
2. $T_{x_j,x_j+3m/4-1}$ *and* $T_{y_j,y_j+3m/4-1}$ *both have period* $|\rho| = |\sum_{i=1}^{\ell} d_i| \leq m/4$

Proof. In the following we will work with indices modulo ℓ, i.e. $x_{\ell+1}$ equals x_1. We begin proving the forward direction. That is we assume that $\{x_i, y_i\}$ matches for $i = j$ (as well as for all $i \neq j$) and prove that both conditions hold. The first condition is immediate and hence we focus on the second.

As in the proof of Lemma 7, let $\gamma = \sum_{i=1}^{j} d_i$ and for all i, let $\gamma'_i = \gamma - \sum_{i'=1}^{i} d_{i'}$. Recall that $\gamma'_j = 0$ and $\gamma'_i \geq 0$ for all $i \neq j$ (or the maximality of γ is contradicted). For $i > 1$, it is easy to check that, as in the proof of Lemma 7, $\gamma'_i + d_i = \gamma'_{i-1}$. However, unlike in Lemma 7, we do not have that $\gamma'_1 + d_1 = \gamma'_\ell$. In Lemma 7 this followed because $\delta = \sum_{i=1}^{\ell} d_i = 0$, which is not true here. The first portion of this proof is similar to the proof of Lemma 7 but with some small modifications.

We will begin by showing that for all $i \in \{1, 2, \ldots, \ell\}$, $S = T_{x_j,x_j+m-\gamma-1}$ occurs in each T_{x_i,x_i+m-1} at offset γ_i i.e. that the string $T_{x_i+\gamma'_i,x_i+\gamma'_i-\gamma-1} = S$. We first proceed by induction on i *in decreasing order*, starting with $i = j$ and halting with $i = 1$. The base case, $i = j$ is immediate as $\gamma'_j = 0$.

We now assume by the inductive hypothesis that $T_{x_i+\gamma'_i,x_i+\gamma'_i+m-\gamma-1} = S$. As the edges $\{x_i, y_i\}$ form a cycle, we have that $v(x_i) = v(y_{i-1})$ for all i. By the definition of $v(x_i)$ and $v(y_{i-1})$ we have that $d_i = x_i - y_{i-1}$. In other words, $T_{y_{i-1},y_{i-1}+m-1}$ and T_{x_i,x_i+m-1} overlap in T by $m - |d_i|$ characters. Analgously to in the proof of Lemma 7, this implies an occurrence of S in $T_{y_{i-1},y_{i-1}+m-1}$ starting at $\gamma'_i + d_i = \gamma'_{i-1} \geq 0$. Finally observe that for all i, we have that $T_{y_{i-1},y_{i-1}+m-1} = T_{x_{i-1},x_{i-1}+m-1}$ so S occurs in $T_{x_{i-1},x_{i-1}+m-1}$ starting at γ'_i, completing the inductive case.

We now repeat the induction on i *in increasing order*, starting with $i = j$ and halting with $i = \ell$. We now assume by the inductive hypothesis that $T_{x_i+\gamma'_i,x_i+\gamma'_i+m-\gamma-1} = S$. The argument is almost identical but in reverse. We provide the argument for completeness.

We have that $T_{y_i,y_i+m-1} = T_{x_i,x_i+m-1}$ so by the inductive hypothesis, S occurs in T_{y_i,y_i+m-1}, at offset γ'_i. As the edges $\{x_i, y_i\}$ form a cycle, we have that $v(x_{i+1}) = v(y_i)$ for all i. As $d_{i+1} = x_{i+1} - y_i$, the substrings T_{y_i,y_i+m-1} and $T_{x_{i+1},x_{i+1}+m-1}$ overlap in T by $m - |d_{i+1}|$ characters. Again, this implies an occurrence of S in $T_{x_{i+1},x_{i+1}+m-1}$ starting at $\gamma'_i - d_{i+1} = \gamma'_{i+1} \geq 0$. Finally observe that for all i, we have that $T_{y_{i+1},y_{i+1}+m-1} = T_{x_{i+1},x_{i+1}+m-1}$ so S occurs in $T_{x_{i+1},x_{i+1}+m-1}$ starting at γ'_i, completing the inductive case.

We now have that $S = T_{x_j,x_j+m-\gamma-1}$ occurs in each T_{x_i,x_i+m-1} at offset γ_i. In particular S occurs in T_{y_1,y_1+m-1} at offset γ'_1 - that is, starting at $x_1 + \gamma'_1$ in T. There is also an occurrence of S in $T_{x_\ell,x_\ell+m-1}$ at offset γ'_ℓ - that is, starting at $x_\ell + \gamma'_\ell$ in T. However we have that $\{x_i, y_i\}$ form a cycle, we have that $v(x_1) = v(y_\ell)$ and hence $d_1 = x_1 - y_\ell$. These two occurrences are therefore $|(x_1 + \gamma'_1) - (x_\ell + \gamma'_\ell)| = |d_1 + \gamma'_1 - \gamma'_\ell| = |\rho|$ characters apart in T. Therefore S has period $|\rho|$. As $|d_j| < m/(9 \cdot \log b)$ and further $\ell \leq 2 \log b + 1$ we have that $\gamma = \sum_{i=1}^{j} d_i \leq \sum_{i=1}^{\ell} |d_i| = |\rho| \leq m/4$. In conclusion, S has period $|\rho| \leq m/4$, length at least $3m/4$ and occurs at the start of T_{x_j,x_j+m-1} (and T_{y_j,y_j+m-1}) as required.

We now prove the reverse direction. That is that if both conditions hold that $\{x_j, x_j\}$ matches. By condition 2, both $T_{x_j,x_j+3m/4-1}$ and $T_{y_j,y_j+3m/4-1}$ are periodic with period at most $m/4$. Further, by condition 1 we have that $T_{x_j+m/2,x_j+m-1} = T_{y_j+m/2,y_j+m-1}$. Observe that $T_{x_j+m/2,x_j+m-1}$ contains at least a full period of characters from $T_{x_j,x_j+3m/4-1}$, and similarly with $T_{y_j+m/2,y_j+m-1}$ and $T_{y_j,y_j+3m/4-1}$ analogously. In other words, the first full period of $T_{x_j,x_j+3m/4-1}$ matches the first full period of $T_{y_j,y_j+3m/4-1}$. By the definition of periodicity we have that $T_{x_j,x_j+3m/4-1} = T_{y_j,y_j+3m/4-1}$ and hence that $T_{x_j,x_j+m-1} = T_{y_j,y_j+m-1}$, i.e. $\{x_j, y_j\}$ matches. □

Processing unlocked cycles We can again identify edge $\{x_j, y_j\}$ 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. We again delete the pair, $\{x_j, y_j\}$ (along with the edge in the graph) and insert a new pair, $\{x_j + m/2, y_j + m/2\}$ into the next round where $m \leftarrow m_{k-1} = m/2$. This checks the first property.

We also need to check the second property, i.e. that both strings $T_{x_j,x_j+3m/4-1}$ and $T_{y_j,y_j+3m/4-1}$ have $|\rho|$ as a 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 list uses $O(b)$ space as at most b edges are considered. 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 9 (see e.g. [10]). *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 9, 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 9, 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.

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.

7 Sparse Position Heaps

So far our focus has been on sparse suffix trees and arrays. In this section, we consider another sparse index, the sparse position heap, and show that it can be constructed much faster than the sparse suffix tree or array. However, the faster construction time comes at the cost of slower pattern matching queries.

7.1 Position Heaps

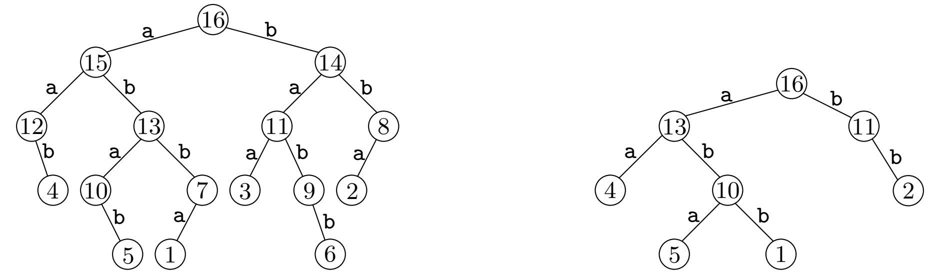
We start by reviewing position heaps. The position heap $\mathcal{H}_T$ over a text $T_{1,n}$ is a blend of a trie over T's suffixes and a heap over its indices [8]:

- The nodes are exactly the n indices (positions) from T such that the *max-heap property* is satisfied: a node i is *larger* than all nodes below i.

- The edges are labeled with single letters from T as in a usual trie such that the following constraint is satisfied: for every node i, the letters on the root-to-i path are a *prefix* of the suffix T_i.

See Figure 3 for an example. This definition almost directly results in the following *naive construction algorithm*: start with a tree consisting of a single node n only. Now assume that $\mathcal{H}_T^{i+1}$, the position heap for $T_{i+1,n}$ for some $i < n$, has already been constructed. Then $\mathcal{H}_T^i$ is obtained by first matching T_i in $\mathcal{H}_T^{i+1}$ as long as possible, thereby finding the longest prefix $T_{i,j}$ of T_i that is a root-to-node path in $\mathcal{H}_T^{i+1}$. Then a new node i is appended as a child to the node h where the search ended, and the new edge (h, i) is labeled with letter t_{j+1}. In the end, $\mathcal{H}_T^1$ is the desired result $\mathcal{H}_T$.

Finding all k occurrences of a pattern $P_{1,m}$ using $\mathcal{H}_T$ works as follows: first try matching P as long as possible in $\mathcal{H}_T$, thereby finding the longest prefix $P_{1,j}$ of P that is a root-to-node path $i_1 \to i_2 \to \cdots \to i_j$ in $\mathcal{H}_T$. The indices $\{i_1, \ldots, i_j\}$ are potential candidates for matches of P; since $j \leq m$, they can be checked naively (meaning one-by-one character comparisons between P and the text) in total time $O(m^2)$. Further, if $j = m$ (the pattern has been fully matched), then all nodes *below* i_j match P *for sure*; they can be returned by a simple traversal of the subtree below i_j. The total time is $O(m^2 + k)$.



(a) The full position heap for T.

(b) The sparse position heap.

Figure 3: (a) Position heap for the text $T_{1,16} = \overset{1}{\texttt{a}}\,\overset{2}{\texttt{b}}\,\overset{3}{\texttt{b}}\,\overset{4}{\texttt{a}}\,\overset{5}{\texttt{a}}\,\overset{6}{\texttt{b}}\,\overset{7}{\texttt{a}}\,\overset{8}{\texttt{b}}\,\overset{9}{\texttt{b}}\,\overset{10}{\texttt{a}}\,\overset{11}{\texttt{b}}\,\overset{12}{\texttt{a}}\,\overset{13}{\texttt{a}}\,\overset{14}{\texttt{b}}\,\overset{15}{\texttt{a}}\,\overset{16}{\texttt{\$}}$. (b) Sparse position heap for positions $\mathcal{I} = \{1, 2, 4, 5, 10, 11, 13, 16\}$ in T.

The position heap can also be enhanced with *maximal reach pointers* that allow for optimal $O(m+k)$ search time; however, we do not review this technique here, since it does not seem to generalize for a sparse set of positions.

The definition suggests that position heaps should be "easier" to compute than suffix trees and suffix arrays, since they do not sort the entire suffixes, but still allow for fast pattern matching queries. This is particularly evident when only a sparse set $\mathcal{I}$ of b suffixes is to be indexed and those b suffixes are inserted using the naive construction algorithm above. Since the time of inserting a suffix $i \in \mathcal{I}$ can be upper bounded by the number of indexed suffixes to the right of i, the running time of the naive algorithm is $O(b^2)$, in particular optimal $O(n)$ for $b = O(\sqrt{n})$. Recall that the naive suffix tree insertion algorithm, on the other hand, would result in $O(nb)$ time, which is never linear unless $b = O(1)$.

Hence, from now on we can assume $n/b \leq b$.

7.2 A Monte-Carlo Construction Algorithm

Our basis is the naive construction algorithm sketched in Section 7.1: we scan T from right to left, and whenever we encounter an indexed suffix from $\mathcal{I}$, we insert it into the already existing heap to obtain the new one. To formalize this, assume that $\mathcal{I}$ is given as a sorted array $I[1,b]$. Assume that $\mathcal{H}_T^{k+1}$, the sparse position heap for suffixes $I[k+1,b]$, has already been constructed. Let $i = I[k]$ be the next position in I. Our aim is to insert the new suffix T_i and thereby obtain $\mathcal{H}_T^{k}$, faster than in $O(b)$ time.

Recall that the task upon insertion of T_i is to find the longest prefix $T_{i,j}$ of T_i that is already present in $\mathcal{H}_T^{k+1}$. Now instead of matching the suffix from start to end, we would like to *binary search* over all possible prefixes of T_i to find the longest prefix that already exists.[1] If such an existential test could be done in $O(1)$ time and since the longest possible such prefix is of length $O(b)$, the binary search would run in $O(\log b)$ time. This would result in $O(b \log b)$ running time.

The problem is to check (in constant time) if and where the prefixes $T_{i,i+\ell}$ occur in $\mathcal{H}_T^{k+1}$. This is where the fingerprints come into play, again. First, we use a hash table that stores the nodes of $\mathcal{H}_T^{k+1}$ and is indexed by the fingerprints of the respective root-to-node paths. Now assuming that we have the fingerprint for some prefix $T_{i,i+\ell}$, then we could easily check the existence of a node representing the same string in $O(1)$ time. If, finally, $T_{i,j}$ is the longest such prefix and i is inserted into $\mathcal{H}_T^{k+1}$ as a child of h, we can also insert i into the hash table (using the fingerprint $FP[i,j]$).

The remaining problem is how to compute the fingerprints. Storing the values $FP[1,i]$ for all $1 \leq i \leq n$ would do the trick, but is prohibitive due to the extra space of $O(n)$. Instead, we do the following indirection step: in a preprocessing step, store only the values $FP[1,i]$ at b regularly spaced positions $i \in \{n/b, 2n/b, \dots\}$; this takes $O(b)$ space and $O(n)$ overall time. We also precompute all powers $r^x \mod p$ for $x \leq b$ in $O(b)$ time and space (r and p are the constants needed for the fingerprints, see Section 2). Then finding the correct node h in $\mathcal{H}_T^{k+1}$ to which i should be appended amounts to the following steps (see Figure 4):

[1] This can be regarded as an x-fast trie-like search [24].

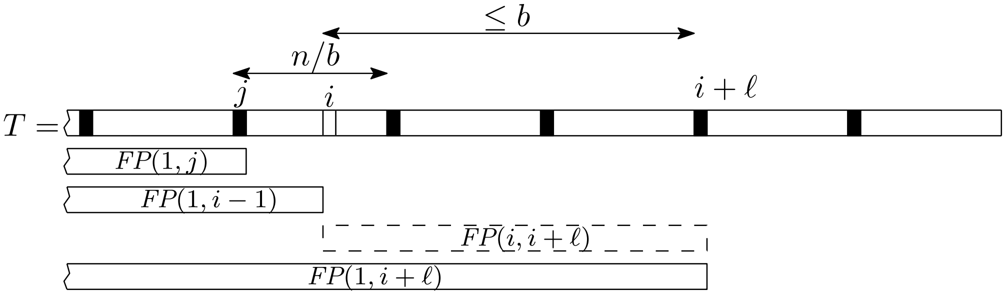


Figure 4: Sketch for computing $FP[i, i+\ell]$. Fingerprints for prefixes of T ending at solid positions are precomputed (here: $FP[1,j]$ and $FP[1,i+\ell]$).

1. Compute $FP[1, i-1]$, by scanning from i backwards until finding the next multiple j of n/b to the left of i (for which $FP[1, j]$ has been precomputed), and using the formula

   $$FP[1, i-1] = FP[1, j] \cdot r^{i-j-1} + FP[j+1, i-1] \mod p \ .$$

   Note that $i - j \leq n/b \leq b$, hence all necessary powers of r are precomputed.

2. Perform a binary search over the prefixes $T_{i,i+\ell}$, for $i+\ell$ being a multiple of n/b. Using the result from step (1) and the precomputed fingerprints, the desired fingerprints can be computed in $O(1)$ time by

   $$FP[i, i+\ell] = FP[1, i+\ell] - FP[1, i-1] \cdot r^{\ell} \mod p \ ,$$

   and hence the binary search takes $O(\log b)$ time (note again $\ell \leq b$).

3. Let h' be the node where the binary search ended with prefix $T_{i,j'}$. From h', continue matching $t_{j'+1}, t_{j'+2}, \ldots$ in the trie until no further match is possible. This yields node h, the longest prefix of T_i that is present in $\mathcal{H}_T^{k+1}$.

The time for steps (1) and (3) is $O(n/b)$. Since there are b suffixes to be inserted, these steps take overall $O(n)$ time. The time for step (2) is $O(b \log b)$ in total. The fingerprint needed for the insertion of i into the hash table can be either computed along with step (3) in $O(n/b)$ time, or from the fingerprint of the parent node h in constant time. The claim follows.

Theorem 5. *There exists a randomized Monte-Carlo algorithm that with high probability correctly constructs the sparse position heap on b suffixes from a string of length n. The algorithm uses $O(n + b \log b)$ time and $O(b)$ space in the worst case, and finding the k occurrences of any pattern of length m takes $O(m^2 + k)$ time.*

8 Conclusions

The main open problem in sparse text indexing is whether we can obtain complexity bounds that completely generalise those of full text indexing. More specifically, is it possible to construct a sparse text index for b arbitrary positions in $O(b)$ space and $O(n)$ time (for integer alphabets) that support pattern matching queries in $O(m + k)$ time, where m is the length of the pattern and k is the number of occurrences?

In this paper we have shown an $O(n \log^2 b)$ time construction algorithm for sparse suffix trees and arrays. This makes significant progress towards the desired $O(n)$ bound, but closing the generalisation gap entirely remains an open question.

As an intermediate step, it might be advantageous to consider trade-offs between the construction time and space as well as the query time of the index. In this context we showed that the sparse position heap can be constructed in $O(n + b \log b)$ time while supporting pattern matching queries in $O(m^2 + k)$ time. This indicates that relaxing the query time constraint makes the problem more approachable, and thus it might be possible to construct a sparse text index in $O(n)$ time and $O(b)$ space for slower pattern matching queries.

Fingerprints play a fundamental role in our results, and it would be interesting if this technique can be further improved, e.g. by constructing a faster deterministic verifier for batched LCP queries – perhaps by generalizing the naming technique of Karp, Miller and Rosenberg [7, 15] to the sparse case. However, it would perhaps be of even greater interest if deterministic solutions similar to the well-known suffix tree and suffix array construction algorithms exist.

After the first appearance of our work, two interesting results have been published that make further progress on the problem.

I et al. [12] showed that the sparse suffix tree or array can be constructed in $O(n \log b)$ time using $O(b)$ space, thus improving our work by a factor of $\log b$. Their solution also relies on Karp-Rabin fingerprints, but they take an approach very different from ours. Interestingly, they also show that $O(n)$ time can be achieved if we allow $O(b \log b)$ space, and they significantly improve the verification step needed to obtain a Las-Vegas algorithm.

Fischer et al. [11] recently considered the problem on *rewritable texts*, i.e., where we with some restrictions can reuse the $O(n)$ memory cells holding the text. In this model they give a worst-case deterministic algorithm with a running time that is close to linear in the number of positions needed to distinguish the b suffixes, which in some cases can be sublinear in n.

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] M. Crochemore and W. Rytter. Usefulness of the Karp-Miller-Rosenberg algorithm in parallel computations on strings and arrays. *Theor. Comput. Sci.*, 88(1):59–82, 1991.

[8] A. Ehrenfeucht, R. M. McConnell, N. Osheim, and S.-W. Woo. Position heaps: A simple and dynamic text indexing data structure. *J. Discrete Algorithms*, 9(1):100–121, 2011.

[9] P. Ferragina and J. Fischer. Suffix Arrays on Words. In *Proc. 18th CPM (LNCS 4580)*, pages 328–339, 2007.

[10] N. J. Fine and H. S. Wilf. Uniqueness Theorems for Periodic Functions. *Proc. AMS*, 16(1):109–114, 1965.

[11] J. Fischer, T. I, and D. Köppl. Deterministic Sparse Suffix Sorting on Rewritable Texts. In *arXiv:1509.07417*, 2015.

[12] T. I, J. Kärkkäinen, and D. Kempa. Faster Sparse Suffix Sorting. In *Proc. 31st STACS*, pages 386–396, 2014.

[13] S. Inenaga and M. Takeda. On-line linear-time construction of word suffix trees. In *Proc. 17th CPM (LNCS 4009)*, pages 60–71, 2006.

[14] J. Kärkkäinen and E. Ukkonen. Sparse Suffix Trees. In *Proc. 2nd COCOON (LNCS 1090)*, pages 219–230, 1996.

[15] R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proc. 4th STOC*, pages 125–136. ACM, 1972.

[16] R. M. Karp and M. O. Rabin. Efficient Randomized Pattern-Matching Algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.

[17] 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.

[18] R. Kolpakov, G. Kucherov, and T. A. Starikovskaya. Pattern Matching on Sparse Suffix Trees. In *Proc. 1st CCP*, pages 92–97, 2011.

[19] U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[20] D. R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[21] M. Paterson. Improved Sorting Networks with $O(\log N)$ Depth. *Algorithmica*, 5(1):65–92, 1990.

[22] T. Uemura and H. Arimura. Sparse and truncated suffix trees on variable-length codes. In *Proc. 22nd CPM (LNCS 6661)*, pages 246–260, 2011.

[23] P. Weiner. Linear Pattern Matching Algorithms. In *Proc. 14th FOCS (SWAT)*, pages 1–11, 1973.

[24] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(n)$. *Inform. Process. Lett.*, 17(2):81–84, 1983.