Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching

(extended abstract)

Roberto Grossi*

Dipartimento di Informatica
Università di Pisa
56125 Pisa, Italy
grossi@di.unipi.it

Jeffrey Scott Vitter†

Department of Computer Science
Duke University
Durham, NC 27708–0129, USA
jsv@cs.duke.edu

Abstract

The proliferation of online text, such as on the World Wide Web and in databases, motivates the need for space-efficient index methods that support fast search. Consider a text T of n binary symbols to index. Given any query pattern P of m binary symbols, the goal is to search for P in T quickly, with T being fully scanned only once, namely, when the index is created. All indexing schemes published in the last thirty years support searching in $\Theta(m)$ worst-case time and require $\Theta(n)$ memory words (or $\Theta(n \log n)$ bits), which is significantly larger than the text itself. In this paper we provide a breakthrough both in searching time and index space under the same model of computation as the one adopted in previous work. Based upon new compressed representations of suffix arrays and suffix trees, we construct an index structure that occupies only $O(n)$ bits and compares favorably with inverted lists in space. We can search any binary pattern P, stored in $O(m/\log n)$ words, in only $o(m)$ time.

Specifically, searching takes $O(1)$ time for $m = o(\log n)$, and $O(m/\log n + \log^\epsilon n) = o(m)$ time for $m = \Omega(\log n)$ and any fixed $0 < \epsilon < 1$. That is, we achieve optimal $O(m/\log n)$ search time for sufficiently large $m = \Omega(\log^{1+\epsilon} n)$. We can list all the *occ* pattern occurrences in optimal $O(occ)$ additional time when $m = \Omega(\text{polylog}(n))$ or when $occ = \Omega(n^\epsilon)$; otherwise, listing takes $O(occ \log^\epsilon n)$ additional time.

*Supported in part by the Italian MURST project "Algorithms for Large Data Sets: Science and Engineering" and by the United Nations Educational, Scientific and Cultural Organization under contract UVO-ROSTE 875.631.9.

†Part of this work was done while the author was on sabbatical at I.N.R.I.A. in Sophia Antipolis, France. Supported in part by Army Research Office MURI grant DAAH04–96–1–0013 and by National Science Foundation research grants CCR–9522047 and CCR–9877133.

1 Introduction

A great deal of textual information is available in electronic form in databases and on the World Wide Web, and consequently, devising indexing methods to support fast search is a relevant research topic. Inverted lists and signature files are efficient indexes for texts that are structured as long sequences of words or keys. Inverted lists are theoretically and practically superior to signature files [49]. Their versatility allows for several kinds of queries (exact, boolean, ranked, and so on) whose answers have a variety of output formats.

Searching unstructured text for string matching queries, however, adds a new difficulty to text indexing. The set of candidate keys is much larger than that of structured texts because it consists of all possible substrings of the text. String matching queries look for the occurrences of a pattern string P of length m as *any* substring of a long text T of length n. We are interested in three types of queries: existential, counting, and enumerative. An existential query returns a boolean value that says if P is contained in T. A counting query computes the number *occ* of occurrences of P in T. An enumerative query outputs the list of *occ* positions where P occurs in T. In the rest of the paper, we assume that the strings are defined over a binary alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. Our results extend to an alphabet of $\sigma \geq 2$ symbols by the standard trick of encoding each symbol with $\lceil \log \sigma \rceil$ bits. (The implied base of the log function is 2.)

The prominent data structures widely used in string matching, such as suffix arrays [35; 24], suffix trees [37; 46] and similar tries or automata [15], are more powerful than inverted lists and signature files when used in text indexing. The *suffix tree* for text $T = T[1, n]$ is a compact trie whose leaves store pointers to the n suffixes in the binary text—$T[1, n]$, $T[2, n]$, ... , $T[n, n]$—and whose internal nodes each have two children. The *suffix array* stores the pointers to the n suffixes in lexicographic order. It also keeps another array of longest common prefixes to speed up the search [35]. In this paper we refer to the suffix array as the plain array of pointers. Both data structures occupy $\Theta(n)$ memory words (or $\Theta(n \log n)$ bits) in the unit cost RAM model. We can do existential and counting queries of P in T in $O(m)$ time (using automata or suffix trees and their variations) and in $O(m + \log n)$ time (using suffix arrays along with longest common prefixes). Enumerative queries take an additional additive output sensitive cost $O(occ)$.

Indexes based upon suffix trees and suffix arrays and related data structures are especially efficient when several searches are to be performed, since the text T needs to be fully scanned only once, namely, when the indexes are created. The importance of suffix arrays and suffix trees is witnessed by numerous references to a great variety of applications besides string searching [4; 25; 35]. Their range of applications is growing in molecular biology, data compression, and text retrieval.

A major criticism that limits the applicability of indexes based upon suffix arrays and suffix trees is that they occupy significantly more space than inverted lists. Space occupancy is especially crucial for large texts. For a text of n binary symbols, suffix arrays use n words of $\log n$ bits each (a total of $n \log n$ bits), while suffix trees require between $4n$ and $5n$ words (or between $4n \log n$ and $5n \log n$ bits) [35]. In contrast, inverted lists require less than $0.1\,n/\log n$ words (or $0.1n$ bits) in many practical cases [38] in order to index a set of words consisting of a total of n bits. However, as previously mentioned, inverted files have less functionality than suffix arrays and suffix trees since only the words are indexed, whereas suffix arrays and suffix trees index all substrings of the text.

No data structures with the functionality of suffix trees and suffix arrays published in the literature to date use $o(n)$ words (or $o(n \log n)$ bits) and support fast queries in the worst case. In order to remedy the space problem, we introduce *compressed suffix arrays*, which are abstract data structures supporting two basic operations:

1. *compress*: Given a suffix array SA, compress SA so as to represent it succinctly.

2. *lookup*(i): Given the compressed representation mentioned above, return $SA[i]$, the pointer to the ith suffix in T in lexicographic order.

The primary measures of performance are the query time to do *lookup*, the amount of space occupied by the compressed suffix array, and the preprocessing time taken by *compress*.

Our main result is that we can implement operation *compress* in only $O(n)$ bits and $O(n)$ preprocessing time, so that each call to *lookup* takes sublogarithmic worst-case time, that is, $O(\log^\epsilon n)$ time for any fixed constant $\epsilon > 0$. We can also achieve $O(n \log\log n)$ bits and $O(n)$ preprocessing time, so that calls to *lookup* can be done in $O(\log\log n)$ time.

Our findings have several important implications:

- To the best of our knowledge, ours is the first result successfully breaking the space barrier of $n \log n$ bits (or n words) for a full text index while retaining fast lookup in the worst case. We refer the reader to the literature described in Section 2.

- Our compressed suffix arrays are provably as good as inverted lists in terms of space usage, at least theoretically. No previous result supported this finding. In the worst case, both types of indexes require asymptotically the *same* number of bits; however, compressed suffix arrays have more functionality because they support search for arbitrary substrings.

- Compressed suffix trees can be implemented in $O(n)$ bits by using compressed suffix arrays and the techniques for compact representation of Patricia tries presented in [42]. As a result, they occupy asymptotically the *same* space as that of the text string being indexed.

- A text index on T can be built in only $O(n)$ bits by a suitable combination of our compressed suffix trees and previous techniques [12; 30; 42; 39]. This is the first result obtaining existential and counting queries of any binary pattern string of length m in $o(m)$ time and $O(n)$ bits. Specifically, searching takes $O(1)$ time for $m = o(\log n)$, and $O(m/\log n + \log^\epsilon n) = o(m)$ time for $m = \Omega(\log n)$ and any fixed $0 < \epsilon < 1$. That is, we achieve optimal $O(m/\log n)$ search time for sufficiently large $m = \Omega(\log^{1+\epsilon} n)$. For enumerative queries, retrieving all *occ* occurrences has optimal cost $O(occ)$ time when $m = \Omega((\log^3 n) \log\log n)$ or when $occ = \Omega(n^\epsilon)$; otherwise, it takes $O(occ \log^\epsilon n)$ time.

Outline of the paper. In the next section we review related work on string searching and text indexing. In Section 3 we describe the ideas behind our new data structure for compressed suffix arrays. In Section 4 we show how to use compressed suffix arrays to construct compressed suffix trees and a general space-efficient indexing mechanism for text search. Details of our compressed suffix array construction are given in Section 5. We adopt the standard unit cost RAM for the analysis of our algorithms, as does the previous work that we compare with. We use standard arithmetic and boolean operations on words of $O(\log n)$ bits, each operation taking constant time and each word read or written in constant time. We give final conclusions and comments in Section 6.

2 Previous Work on String Searching and Text Indexing

The seminal paper by Knuth, Morris, and Pratt [33] presented the first string matching solution taking $O(m+n)$ time and $O(m)$ words to scan the text. The space complexity was remarkably lowered to $O(1)$ words in [22; 14]. A seminal paper by Weiner [46] introduced the suffix tree for solving the text indexing problem in string matching. Since then, a plethora of papers have studied the problem in several contexts and sometimes using different terminology [7; 8; 13; 19; 26; 37; 35; 45]; for more references see [4; 15; 25]. Although very efficient, the resulting index data structures are greedy of space, at least n words or $\Omega(n \log n)$ bits.

Numerous papers faced the problem of saving space in these data structures, both in practice and in theory. Many of the papers were aimed at improving the lower-order terms, as wells the constants in the higher order term, or at achieving tradeoff between space requirements and search time complexity. Some authors improved the multiplicative constants in the $O(n \log n)$-bit practical implementations. For the analysis of constants, we refer the reader to [3; 10; 23; 29; 34; 35]. Other authors devised several variations of sparse suffix trees to store a subset of the suffixes [2; 24; 32; 31; 36; 39]. Some of them wanted queries to be efficient when the occurrences are aligned with the beginnings of the indexed suffixes. Sparsity saves much space but makes the search for arbitrary substrings difficult and, in the worst case, as expensive as scanning the whole text in $O(m+n)$ time. Another interesting index, the Lempel-Ziv index of Kärkkäinen and Sutinen [30], occupies $O(n)$ bits and takes $O(m)$ time to search patterns shorter than $\log n$; for longer patterns, it may occupy $\Theta(n \log n)$ bits.

A recent line of research has been built upon Jacobson's succinct representation of trees in $2n$ bits, with navigational operations [27]. That representation was extended in [11] to represent a suffix tree in $n \log n$ bits plus an extra $O(n \log \log n)$ expected number of bits. A solution requiring $n \log n + O(n)$ bits and $O(m + \log \log n)$ search time was described in [12]. Munro et al. [42] used it along with an improved succinct representation of balanced parentheses [41] in order to get $O(m)$ search time with only $n \log n + o(n)$ bits.

3 Compression of Suffix Arrays

The compression of suffix arrays falls into the general framework presented by Jacobson [28] for the abstract optimization of data structures. We start from the specification of our data structure as an abstract data type with its supported operations. We take the time complexity of the "natural" (and space inefficient) implementation of the data structure. Then, we define the class C_n of all distinct data structures storing n elements. A simple combinatorial argument implies that each such data structure can be canonically identified by $\log |C_n|$ bits. We try to give a succinct implementation of the same data structure in $O(\log |C_n|)$ bits, while supporting the operations within time complexity comparable with that of the natural implementation. However, the combinatorial argument does not guarantee that the operations can be supported efficiently.

We define the suffix array SA for a binary string T as an abstract data type that supports the two operations *compress* and *lookup* described in the introduction. We will adopt the convention that T is a binary string of length $n-1$ over the alphabet $\{\mathtt{a}, \mathtt{b}\}$, and it is terminated in the nth position by a special end-of-string symbol #, such that $\mathtt{a} < \# < \mathtt{b}$.[1] The suffix array SA is a permutation of $\{1, 2, \ldots, n\}$ that corresponds to the lexicographic ordering of the suffixes in T; that is, $SA[i]$ is the starting position in T of the ith suffix in lexicographic order. In the example below are the suffix arrays corresponding to the 16 binary strings of length 4:

aaaa#	aaab#	aaba#	aabb#
1 2 3 4 5	1 2 3 5 4	1 4 2 5 3	1 2 5 4 3
abaa#	abab#	abba#	abbb#
3 4 1 5 2	1 3 5 2 4	4 1 5 3 2	1 5 4 3 2
baaa#	baab#	baba#	babb#
2 3 4 5 1	2 3 5 1 4	4 2 5 3 1	2 5 1 4 3
bbaa#	bbab#	bbba#	bbbb#
3 4 5 2 1	3 5 2 4 1	4 5 3 2 1	5 4 3 2 1

[1] Usually an end-of-symbol character is not explicitly stored in T, but rather is implicitly represented by a blank symbol ␣, with the ordering ␣ $<$ a $<$ b. However, our use of # is convenient for showing the explicit correspondence between suffix arrays and binary strings.

The natural explicit implementation of suffix arrays requires $O(n \log n)$ bits and supports the *lookup* operation in constant time. The abstract optimization discussed above suggests that there is a canonical way to represent suffix arrays in $O(n)$ bits. This observation follows from the fact that the class C_n of suffix arrays has no more than 2^{n-1} distinct members, as there are 2^{n-1} binary strings of length $n-1$.

We use the intuitive correspondence between suffix arrays of length n and binary strings of length $n-1$. According to the correspondence, given a suffix array SA, we can infer its associated binary string T and vice versa. To see how, let x be the entry in SA corresponding to the last suffix # in lexicographic order. Then T must have the symbol a in each of the positions pointed to by $SA[1]$, $SA[2]$, ..., $SA[x-1]$, and it must have the symbol b in each of the positions pointed to by $SA[x+1]$, $SA[x+2]$, ..., $SA[n]$. For example, in the suffix array $\langle 45321 \rangle$ (the 15th of the 16 examples above), the suffix # corresponds to the second entry 5. The preceding entry is 4, and thus the string T has a in position 4. The subsequent entries are 3, 2, 1, and thus T must have bs in positions 3, 2, 1. The resulting string T, therefore, must be bbba#.

The abstract optimization does not say anything regarding the efficiency of the supported operations. By the correspondence above, we can define a trivial *compress* operation that transforms SA into a sequence of $n-1$ bits, namely, string T. The drawback, however, is the unaffordable cost of *lookup*. It takes $\Omega(n)$ time to decompress a single pointer in SA as it must build the whole suffix array on T from scratch. In other words, the trivial method proposed so far does not support efficient *lookup* operations.

In this paper we give an elegant and efficient method to represent suffix arrays in $O(n)$ bits. Our breakthrough idea is to distinguish among the permutations of $\{1, \ldots, n\}$ by relating them to the suffixes of the corresponding strings, instead of studying them alone. We mimic a simple divide-and-conquer "de-construction" of the suffix arrays to define the permutations recursively in terms of shorter permutations. For some examples of divide-and-conquer construction of suffix arrays and suffix trees, see [5; 16; 17; 18; 35; 44]. We reverse the construction process to compress the permutations.

Our decomposition scheme is by a simple recursion mechanism. Let SA be the suffix array for binary string T. In the base case, we denote SA by SA_0, and let $n_0 = n$ be the number of its entries. For simplicity in exposition, we assume that n is a power of 2.

In the inductive phase $k \geq 0$, we start with suffix array SA_k, which is available by induction. It has $n_k = n/2^k$ entries and stores a permutation of $\{1, \ldots, n_k\}$. We run four main steps to transform SA_k into an equivalent but more succinct representation:

Step 1. Produce a bit vector B_k of n_k bits, such that $B_k[i] = \mathbf{1}$ if $SA_k[i]$ is even and $B_k[i] = \mathbf{0}$ if $SA_k[i]$ is odd.

Step 2. Map each $\mathbf{0}$ in B_k onto its companion $\mathbf{1}$. (We say that a certain $\mathbf{0}$ is the *companion* of a certain $\mathbf{1}$ if the odd entry in SA associated with the $\mathbf{0}$ is 1 less than the even entry in SA associated with the $\mathbf{1}$.) We can denote this correspondence by a partial function Ψ_k, where $\Psi_k(i) = j$ if and only if $SA_k[i]$ is odd and $SA_k[j] = SA_k[i] + 1$. When defined, $\Psi_k(i) = j$ implies that $B_k[i] = \mathbf{0}$ and $B_k[j] = \mathbf{1}$. It is

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T:	a	b	b	a	b	b	a	b	b	a	b	b	a	b	a	a
SA_0:	15	16	31	13	17	19	28	10	7	4	1	21	24	32	14	30
B_0:	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**1**
$rank_0$:	0	1	1	1	1	1	2	3	3	4	4	4	5	6	7	8
Ψ_0:	2	2	14	15	18	23	7	8	28	10	30	31	13	14	15	16

	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
T:	a	b	a	b	a	b	b	a	b	b	b	a	b	b	a	#
SA_0:	12	18	27	9	6	3	20	23	29	11	26	8	5	2	22	25
B_0:	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**0**
$rank_0$:	9	10	10	10	11	11	12	12	12	12	13	14	14	15	16	16
Ψ_0:	17	18	7	8	21	10	23	13	16	17	27	28	21	30	31	27

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
SA_1:	8	14	5	2	12	16	7	15	6	9	3	10	13	4	1	11

Figure 1: The effect of a single application of Steps 1–4.

convenient to make Ψ_k a total function by setting $\Psi_k(i) = i$ when $SA_k[i]$ is even (i.e., when $B_k[i] = \mathbf{1}$). In summary, for $1 \le i \le n_k$, we have

$$\Psi_k(i) = \begin{cases} j & \text{if } SA_k[i] \text{ is odd and } SA_k[j] = SA_k[i] + 1; \\ i & \text{otherwise.} \end{cases}$$

Step 3. Compute the number of **1**s for each prefix of B_k. We use function $rank_k$ for this purpose, such that $rank_k(j)$ counts how many **1**s there are in the first j bits of B_k.

Step 4. Pack together the even values from SA_k and divide each of them by 2. The resulting values form a permutation of $\{1, 2, \ldots, n_{k+1}\}$, where $n_{k+1} = n_k/2 = n/2^{k+1}$. Store them into a new suffix array SA_{k+1} of n_{k+1} entries, and remove the old suffix array SA_k.

The example in Fig. 1 illustrates the effect of a single application of Steps 1–4. The next lemma shows that these steps preserve the information originally kept in suffix array SA_k:

LEMMA 1. *Given suffix array SA_k, let B_k, Ψ_k, $rank_k$ and SA_{k+1} be the result of the transformation performed by Steps 1–4 of phase k. We can reconstruct SA_k from SA_{k+1} by the following formula, for $1 \le i \le n_k$,*

$$SA_k[i] = 2 \cdot SA_{k+1}\big[rank_k\big(\Psi_k(i)\big)\big] + (B_k[i] - 1).$$

PROOF. Suppose $B_k[i] = \mathbf{1}$. By Step 3, there are $rank_k(i)$ **1**s among $B_k[1], B_k[2], \ldots, B_k[i]$. By Step 1, $SA_k[i]$ is even, and by Step 4, $SA_k[i]/2$ is stored in the $rank_k(i)$th entry of SA_{k+1}. In other words, $SA_k[i] = 2 \cdot SA_{k+1}\big[rank_k(i)\big]$. As $\Psi_k(i) = i$ by Step 2, and $B_k[i] - 1 = 0$, we obtain the claimed formula.

Next, suppose $B_k[i] = \mathbf{0}$ and let $j = \Psi_k(i)$. By Step 2, we have $SA_k[i] = SA_k[j] - 1$ and $B_k[j] = \mathbf{1}$. Consequently, we can apply the previous case of our analysis to index j, and we get $SA_k[j] = 2 \cdot SA_{k+1}\big[rank_k(j)\big]$. The claimed formula follows by replacing j with $\Psi_k(i)$ and by noting that $B_k[i] - 1 = -1$. $\square$

We now give the main ideas to perform the compression of suffix array SA and support the lookup operations on its compressed representation.

Procedure *compress*. We represent SA succinctly by executing Steps 1–4 of phases $k = 0, 1, \ldots, \ell - 1$, where the exact value of $\ell = \Theta(\log\log n)$ will be determined in Section 5. As a result, we have $\ell + 1$ levels of information, numbered $0, 1, \ldots, \ell$, which form the *compressed representation* of suffix array SA:

- Level k, for each $0 \le k < \ell$, stores B_k, Ψ_k, and $rank_k$. We do not store SA_k, but we refer to it for the sake of discussion. The arrays Ψ_k and $rank_k$ are not stored explicitly, but are stored in a specially compressed form to be described later.
- The last level $k = \ell$ stores SA_ℓ explicitly because it is sufficiently small to fit in $O(n)$ bits. The ℓth level functionality of structures B_ℓ, Ψ_ℓ, and $rank_\ell$ are not needed as a result.

Procedure *lookup*(i). We define $lookup(i) = rlookup(i, 0)$, where procedure $rlookup(i, k)$ is described recursively in Figure 2. If k is the last level ℓ, then it performs a direct lookup in $SA_\ell[i]$. Otherwise, it exploits Lemma 1 and the inductive hypothesis so that $rlookup(i, k)$ returns the value in $SA_k[i]$.

Further details on how to represent $rank_k$ and Ψ_k in compressed form and how to implement *compress* and $lookup(i)$ will be given in Section 5. Our main theorem below gives the resulting time and space complexity that we are able to achieve.

THEOREM 1. *Consider the suffix array SA built on a binary string of length $n - 1$.*

i. We can implement compress in $O(n \log\log n)$ bits and $O(n)$ preprocessing time, so that each call $lookup(i)$ takes $O(\log\log n)$ time.

ii. We can implement compress in $O(n)$ bits and $O(n)$ preprocessing time, so that each call $lookup(i)$ takes $O(\log^\epsilon n)$ time for any constant $\epsilon > 0$.

REMARK 1. In each of the cases stated in Theorem 1, we can batch together $j - i + 1$ procedure calls $lookup(i)$, $lookup(i + 1)$, $\ldots$, $lookup(j)$, so that the total cost is

- $O(j - i + (\log^2 n)\log\log n)$ time when the suffixes pointed to by $SA[i]$ and $SA[j]$ have the same first $\Omega(\log^2 n)$ binary symbols in common, or

```
procedure rlookup(i, k):
    if k = ℓ then
        return SA_ℓ[i]
    else
        return 2 × rlookup(rank_k(Ψ_k(i)), k + 1) + (B_k[i] − 1).
```

Figure 2: Recursive lookup of entry $SA_k[i]$ in a compressed suffix array.

- $O(j - i + n^{\alpha})$ time, for any constant $0 < \alpha < 1$, when the suffixes pointed to by $SA[i]$ and $SA[j]$ have the same first $\Omega(\log n)$ binary symbols.

4 Text Indexing, String Matching, and Compressed Suffix Trees

Text indexing is worthwhile when handling multiple queries on text collections. Inverted lists are versatile data structures for this purpose [49]. They keep a vocabulary of distinct keys, with a list of the occurrences for each distinct key. They support search of the form, "Given a query pattern, is the query one of the keys in the vocabulary, and where are the instances in the text where it appears?" We show that, despite their extra functionality, compressed suffix arrays require the same asymptotic space as inverted lists in the worst case.

LEMMA 2. *In the worst case, both inverted lists and compressed suffix arrays require $\Theta(n)$ bits for a binary string of length n, whereas compressed suffix arrays can search arbitrary substrings.*

PROOF. (Sketch) Let us take a De Bruijn sequence S of length n, in which each substring of $\log n$ characters is different from the others. Now let the keys in the inverted file be those obtained by partitioning S into $s = n/(2 \log n)$ disjoint substrings of length $k = 2 \log n$. Any data structure that implements inverted lists must be able to solve the static dictionary problem on the s keys, and so it requires at least $\log \binom{2^k}{s} = \Theta(n)$ bits. We build instead our compressed suffix array by setting our text $T = S$ (i.e., our keys are *all* the substrings) and the resulting space is still $\Theta(n)$ bits. $\square$

In practice, we expect several occurrences for each distinct key, and the experimental study in [49] shows that inverted lists are space-efficient. Moreover, inverted lists are dynamic while we do not know how to maintain our compressed suffix arrays in a dynamic setting.

We now focus on text indexing with string matching queries. Here, we are given a binary pattern string P of m symbols and we are interested in its occurrences (perhaps overlapping) in a binary text string T of n symbols (where # is the nth symbol).

THEOREM 2. *Given a binary text string T of length n, we can build an index data structure on T in $O(n)$ time such that the index occupies $O(n)$ bits and supports the following queries on a pattern string P of m bits packed into $O(m/\log n)$ words:*

i. Existential and counting queries can be done in $o(m)$ time; in particular, they take $O(1)$ time for $m = o(\log n)$, and $O(m/\log n + \log^{\epsilon} n) = o(m)$ time for $m = \Omega(\log n)$ and any fixed $0 < \epsilon < 1$;

ii. An enumerative query can be done in $O(m/\log n + occ)$ time, where occ is the number of occurrences of P in T, when $m = \Omega((\log^3 n) \log \log n))$ or when $occ = \Omega(n^{\epsilon})$ for fixed $0 < \epsilon < 1$; otherwise, it takes $O(m/\log n + occ \log^{\epsilon} n)$ time.

We prove Theorem 2 by employing our compressed suffix arrays and some techniques presented in [30; 39; 42] and briefly discussed below.

The Lempel-Ziv (LZ) index [30] is a powerful tool to search for q-grams (substrings of length q) in T. If we fix $q = \epsilon \log n$ for any fixed positive constant $\epsilon < 1$, we can build an LZ index on T in $O(n)$ time, such that the LZ index occupies $O(n)$ bits and any pattern of length $m \leq \epsilon \log n$ can be searched in $O(m + occ)$ time. In this special case, we can actually obtain $O(1 + occ)$ time by suitable table lookup. (Unfortunately, for longer patterns, the LZ index may take $\Theta(n \log n)$ bits.) The LZ index allows us to concentrate on patterns of length $m > \epsilon \log n$.

The Patricia trie [39] is another powerful tool in text indexing. It is a binary tree that stores a set of distinct strings, in which each internal node has two children and each leaf stores a string. Each internal node also keeps an integer (called skip value) to locate the position of the branching character while descending towards a leaf. The left arcs are implicitly labeled with one character of the alphabet, and the right arcs are implicitly labeled with the other character (recall we have a binary alphabet). Suffix trees are often implemented by building a Patricia trie on the suffixes of T [24]. Searching for P takes $O(m)$ time and retrieves only the suffix pointer in two leaves (i.e., the leaf reached by branching with the skip values, and the leaf corresponding to an occurrence). It requires only $O(1)$ calls to the *lookup* operation in the worst case.

Unfortunately, the Patricia trie storing s suffixes of T occupies $O(s \log n)$ bits. This amount of space usage is the result of three separate factors: the Patricia trie topology, the skip values, and the string pointers [10; 11]. Because of our compressed suffix arrays, the string pointers are no longer a problem. For the remaining two items, the space-efficient incarnation of Patricia tries in [42] cleverly avoids the overhead for the Patricia trie topology and the skip values. It is able to represent a Patricia trie storing s suffixes of T with only $O(s)$ bits, *provided that a suffix array is given separately* (which in our case is a compressed suffix array). Searching for query pattern P takes $O(m)$ time and accesses $O(s)$ suffix pointers in the worst case. For each traversed node, its corresponding skip value is computed in time $O(skip\_value)$ by accessing the suffix pointers in its leftmost and rightmost descendant leaves. Consequently, searching requires $O(s)$ calls to *lookup* in the worst case.

4.1 Speeding Up Patricia Trie Search

Before we discuss how to construct the index, we first need to show that search in Patricia tries, which normally proceeds one level at a time, can be improved to sublinear time by processing $\log n$ bits of the pattern at a time. Let us first consider an ordinary Patricia trie [39] PT. We will show how to reduce the search time for an m-bit pattern in PT from $O(m)$ to $O(m/\log n + \log^{\epsilon} n)$. Without loss of generality, we show how to achieve $O(m/\log n + \sqrt{\log n}\,)$ time as this bound extends to any exponent $\epsilon > 0$.[2] The point is that, in the worst case, we may have to traverse $\Theta(m)$ nodes, so we

[2] We can actually achieve a better bound with more sophisticated techniques that have applications to other problems not discussed here. Although interesting in itself, we do not discuss this bound as it does not improve the final search time obtained in this section.

need a tool to skip most of these nodes. Ideally, we would like to branch downward matching $\log n$ bits in constant time, independently of the number of traversed nodes.

We use a perfect hash function h for this purpose [21], on keys of length at most $2\log n$ bits each. First of all, we enumerate the nodes of PT in preorder starting from the root, numbered 1. Next, we build hash tables in which we store pair $\langle j, b\rangle$ at position $h(i,x)$, where node j is a descendent of node i, string x is of length less than or equal to $\log n$, and b is a nonnegative integer. These parameters must satisfy two conditions:

1. j is the node identified by starting out from node i and traversing downward PT according to the bits in x.

2. b is the unique integer such that the string corresponding to the path from i to j has prefix x and length $|x| + b$; this condition does not hold for any ancestor of j.

The rationale behind conditions 1–2 is that of defining shortcut links from nodes i to nodes j, so that each successful branching takes constant time, matches at least $|x|$ bits and skips no more than $|x|$ nodes downward. In order to handle border situations, we build PT on the text string padded with $\log n$ symbols #. We want to use a small number of shortcut links and, to this end, we set up two hash tables T_1 and T_2. The first table stores entries

$$T_1[h(i,x)] = \langle j, b\rangle$$

such that all strings x are of length $|x| = \log n$, and the shortcut links are defined by a simple top-down traversal of PT. Initially, we create all possible shortcut links from the root. This step links the root to a set of descendents. Each of these nodes is then recursively linked to its descendents in the same fashion. Note that the number of links does not exceed the number of nodes in PT, and PT is partitioned into subtries of maximum depth $\log n$.

We set up the second table T_2 analogously. We start from the root of each individual subtrie and use strings of length $|x| = \sqrt{\log n}$. Again, the number of these links is upper bounded by the number of nodes in PT. The mechanism that makes the above speedup possible is that we closely follow the Patricia topology, so that the strings that we hash are not all possible substrings of length $\log n$ (or $\sqrt{\log n}$), but only a subset of those that start at the nodes in the Patricia trie.

We are now ready to describe the search of a pattern in the Patricia trie PT augmented with the shortcut links. It suffices to match its longest prefix. We take the first $\log n$ bits in the pattern and branch quickly from the root by using T_1. If the hash lookup in T_1 succeeds and gives pair $\langle j, b\rangle$, we try to match the next b bits in the pattern in $O(b/\log n)$ time, and then recursively search in node j with the next $\log n$ bits in the pattern. Instead, if the hash lookup fails because there are fewer than $\log n$ bits left in the query pattern, we switch to T_2 and take only the next $\sqrt{\log n}$ bits in the pattern to branch further in PT. Here the scheme is the same, except that we compare $\sqrt{\log n}$ bits at a time. Finally, when we fail branching again, we have to match no more than $\sqrt{\log n}$ bits remaining in the pattern. We complete this task by branching in the standard way, one bit a time. This completes the description of the search in PT.

The speed up in the search of a space-efficient Patricia trie [42] is easier. In our case, we do not need to skip nodes, but just compare $\Theta(\log n)$ bits at a time in constant time by precomputing a suitable table. The search cost is therefore $O(m/\log n)$ plus a linear cost proportional to the number of traversed nodes.

4.2 Index Construction

We blend the previously mentioned tools with our compressed suffix arrays to design a multilevel index data structure, called the *compressed suffix tree*, following the multilevel scheme adopted in [12; 42]. It suffices to support searching of patterns of length $m > \epsilon \log n$ because of the LZ index. We assume that $0 < \epsilon < 1/2$ as the case $1/2 \leq \epsilon < 1$ requires minor modifications.

Given text T, we build its compressed suffix tree by starting out from the suffix array SA built on T in $O(n)$ time via a suffix tree. We have $O(1)$ levels, which we describe top down.

In the first level, we build a regular Patricia trie PT^1, augmented with the shortcut links as mentioned in Section 4.1, on the $s_1 = \Theta(n/\log n)$ suffixes pointed to by $SA[1]$, $SA[1 + \log n]$, $SA[1 + 2\log n]$, $\ldots$. This implicitly splits SA into s_1 subarrays of size $\log n$, except the last one. The size of PT^1 is $O(s_1 \log n) = O(n)$ bits.

In the second level, we process the s_1 subarrays created in the first level, and create s_1 space-efficient Patricia tries [42], denoted PT_1^2, PT_2^2, $\ldots$, $PT_{s_1}^2$. We build the ith Patricia PT_i^2 on the ith subarray. Assume without loss of generality that the subarray consists of $SA[h+1]$, $SA[h+2]$, $\ldots$, $SA[h+\log n]$. Then, PT_i^2 is built on the $s_2 = \log^{\epsilon/2} n$ suffixes pointed to by $SA[h+1]$, $SA[h+1+\log^{1-\epsilon/2} n]$, $SA[h+1+2\log^{1-\epsilon/2} n]$, $\ldots$. This further splits the subarrays into smaller subarrays, each of size $\log^{1-\epsilon/2} n$. The size of PT_i^2 is $O(s_2)$ bits without accounting for the suffix array.

We go on in this way with $O(1)$ further levels, each splitting every subarray into $s_2 = \log^{\epsilon/2} n$ smaller subarrays, until we are left with small subarrays of size at most s_2.

In the last level, we execute *compress* on the suffix array SA and store its compressed version, so that accessing a pointer takes $O(\log^{\epsilon/2} n)$ time.

We build each of the levels in $O(n)$ time. As for the space complexity, we have a total of $O(n)$ bits. The first level requires $O(n)$ bits for the single Patricia trie; the O(1) intermediate levels require $O(n)$ bits in total; the last level requires $O(n)$ bits by Theorem 1.

4.3 Search Algorithm

We now have to show that searching for a pattern P in the text T costs $O(m/\log n + \log^{\epsilon} n)$ time. The search locates the leftmost occurrence and the rightmost occurrence of P in SA, without having SA stored explicitly. A successful search determines two positions $i \leq j$, such that $SA[i]$, $SA[i+1]$, $\ldots$, $SA[j]$ contain all the pointers to the suffixes that begin with P. The counting query returns $j - i + 1$, and the existence checks whether there are any matches at all. The enumerative query executes the $j - i + 1$ queries *lookup*(i), *lookup*$(i+1)$, $\ldots$, *lookup*(j) to list all the occurrences.

We restrict our discussion to how to find the leftmost occurrence of P; finding the rightmost is analogous. We search in each level from scratch. That is, we perform the search with shortcut links described in Section 4.1 on PT^1 in the first level. We locate a subarray in the second level, say, the i_1th subarray. We go on and search in $PT^2_{i_1}$ according to the method described in Section 4.1 for space-efficient Patricia tries. We repeat a similar search for all the intermediate levels. While searching in the levels, we execute *lookup*(i) whenever we need the ith pointer in the last level.

The complexity of the search procedure is $O(m/\log n + \log^\epsilon n)$ total time for all the levels plus the cost of the *lookup* operations. In the first level, we call *lookup* $O(1)$ times; in the intermediate levels we call *lookup* $O(s_2)$ times. Multiplying these calls by the $O(\log^{\epsilon/2} n)$ cost of *lookup* as given in Theorem 1 (using $\epsilon/2$ in place of ϵ), we obtain $O(\log^\epsilon n)$ time in addition to $O(m/\log n + \log^\epsilon n)$. Finally, the cost of retrieving all the occurrences is the one stated in Remark 1 after Theorem 1, because the suffixes pointed to by $SA[i]$ and $SA[j]$ share at least $m = \Omega(\log n)$ symbols. This argument completes the proof of Theorem 2.

5 Algorithms for Compressed Suffix Arrays

In this section we constructively prove Theorem 1 by showing two ways to implement the recursive decomposition of suffix arrays discussed in Lemma 1 of Section 3. For brevity we defer the discussion of the preprocessing steps and time analysis to the full paper. Multiple occurrences can be reported in an optimal time under certain cases.

5.1 Compressed Suffix Arrays in $O(n \log\log n)$ Bits and $O(\log\log n)$ Access Time

The first method achieves $O(\log\log n)$ lookup time with a total space usage of $O(n \log\log n)$ bits. We perform the recursive decomposition of Steps 1–4 described in Section 3, for $0 \le k < \ell - 1$, where $\ell = \lceil \log\log n \rceil$. The decomposition below shows the result on the example of Section 3:

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*
SA_1:	8	14	5	2	12	16	7	15	6	9	3	10	13	4	1	11
B_1:	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**0**
$rank_1$:	1	2	2	3	4	5	5	5	6	6	6	7	7	8	8	8
Ψ_1:	1	2	9	4	5	6	1	6	9	12	14	12	2	14	4	5

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
SA_2:	4	7	1	6	8	3	5	2
B_2:	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**
$rank_2$:	1	1	1	2	3	3	3	4
Ψ_2:	1	5	8	4	5	1	4	9

	1	*2*	*3*	*4*
SA_3:	2	3	4	1

The resulting suffix array SA_ℓ on level ℓ contains at most $n/\log n$ entries and can thus be stored explicitly in n bits. We store the bit vectors $B_0, B_1, \ldots, B_{\ell-1}$ in explicit form, using less than $2n$ bits, as well as *implicit* representations of $rank_0, rank_1, \ldots, rank_{\ell-1}$ and $\Psi_0, \Psi_1, \ldots, \Psi_{\ell-1}$. If the implicit representations of $rank_k$ and Ψ_k can be accessed in constant time, the procedure described in Lemma 1 shows how to achieve the desired lookup in constant time per level, for a total of $O(\log\log n)$ time.

All that remains is to investigate how to represent $rank_k$ and Ψ_k, for $0 \le k \le \ell-1$, in $O(n)$ bits and support constant-time access. Given the bit vector B_k of $n_k = n/2^k$ bits, Jacobson [27] shows how to support constant-time access to $rank_k$ using only $o(n_k)$ bits. For the Ψ_k function, we use the following representation: For each $1 \le i \le n_k/2$, let j be the index of the ith **1** in B_k. Consider the 2^k symbols $T[2^k \cdot SA_k[j] - 2^k], \ldots, T[2^k \cdot SA_k[j] - 1]$; these 2^k symbols immediately precede the $(2^k \cdot SA_k[j])$th suffix in T, as the suffix pointer in $SA_k[j]$ was 2^k times larger before the compression. For each bit pattern of 2^k symbols that appears, we keep an ordered list of the indices j that correspond to it, and we record the number of items in each list. Continuing the example above, we get the following lists for level 0:

`a` list:	$\langle 2, 14, 15, 18, 23, 28, 30, 31\rangle$,	$\lvert$`a` list$\rvert = 8$
`b` list:	$\langle 7, 8, 10, 13, 16, 17, 21, 27\rangle$,	$\lvert$`b` list$\rvert = 8$

Level 1:

`aa` list:	$\emptyset$,	$\lvert$`aa` list$\rvert = 0$
`ab` list:	$\langle 9\rangle$,	$\lvert$`ab` list$\rvert = 1$
`ba` list:	$\langle 1, 6, 12, 14\rangle$,	$\lvert$`ba` list$\rvert = 4$
`bb` list:	$\langle 2, 4, 5\rangle$,	$\lvert$`bb` list$\rvert = 3$

Level 2:

`aaaa` list:	$\emptyset$,	$\lvert$`aaaa` list$\rvert = 0$
`aaab` list:	$\emptyset$,	$\lvert$`aaab` list$\rvert = 0$
`aaba` list:	$\emptyset$,	$\lvert$`aaba` list$\rvert = 0$
`aabb` list:	$\emptyset$,	$\lvert$`aabb` list$\rvert = 0$
`abaa` list:	$\emptyset$,	$\lvert$`abaa` list$\rvert = 0$
`abab` list:	$\emptyset$,	$\lvert$`abab` list$\rvert = 0$
`abba` list:	$\langle 5, 8\rangle$,	$\lvert$`abba` list$\rvert = 2$
`abbb` list:	$\emptyset$,	$\lvert$`abbb` list$\rvert = 0$
`baaa` list:	$\emptyset$,	$\lvert$`baaa` list$\rvert = 0$
`baab` list:	$\emptyset$,	$\lvert$`baab` list$\rvert = 0$
`baba` list:	$\langle 1\rangle$,	$\lvert$`baba` list$\rvert = 1$
`babb` list:	$\langle 4\rangle$,	$\lvert$`babb` list$\rvert = 1$
`bbaa` list:	$\emptyset$,	$\lvert$`bbaa` list$\rvert = 0$
`bbab` list:	$\emptyset$,	$\lvert$`bbab` list$\rvert = 0$
`bbba` list:	$\emptyset$,	$\lvert$`bbba` list$\rvert = 0$
`bbbb` list:	$\emptyset$,	$\lvert$`bbbb` list$\rvert = 0$

Suppose we want to compute $\Psi_k(i)$. If $B_k[i] = \mathbf{1}$, we trivially have $\Psi_k(i) = i$; therefore, let's consider the harder case in which $B_k[i] = \mathbf{0}$, which means that $SA_k[i]$ is odd. We have to determine the index j such that $SA_k[j] = SA_k[i]+1$. We can determine the number h of **0**s in B_k up to index i using the approach of [27] for $rank_k$. Consider the 2^k lists concatenated together in lexicographic order of the 2^k prefixes. What we need to find now is the hth entry in the concatenated list. For example, to determine $\Psi_0(25)$ in the example above, we find that there are $h = 13$ **0**s in the first 25 slots of B_0. There are eight entries in the `a` list and eight entries in the `b` list; hence, the 13th entry in the concatenated lists is the fifth entry above in the `b` list, namely, index 16. Hence, we have $\Psi_0(25) = 16$ as desired; note that $SA_0[25] = 29$ and $SA_0[16] = 30$ are consecutive values.

Continuing the example, consider the next level of the recursive processing of *rlookup*, in which we need to determine $\Psi_1(8)$. (The previously computed value $\Psi_0(25) = 16$ has a $rank_0$ value of 8 (i.e., $rank_0(16) = 8$), so the *rlookup* procedure needs to determine $SA_1[8]$, which it does by first calculating $\Psi_1(8)$.) There are $h = 3$ **0**s in the first eight entries of B_1. The third entry in the concatenated lists for `aa`, `ab`, `ba`, and `bb` is the second entry in the `ba` list, namely, 6. Hence, we have $\Psi_1(8) = 6$ as desired; note that $SA_1[8] = 15$ and $SA_1[6] = 16$ are consecutive values.

Finding the appropriate list containing the hth entry can be determined in constant time by using a clever encoding of a prefix sum array storing the cumulated list sizes. We can find the second entry in the `b` list using a recursive encoding of the list coupled with table lookup into directories by using the ranking and selection operations defined in [10; 28; 40]. Details will appear in the full paper.

5.2 Compressed Suffix Arrays in $O(n)$ Bits and $O(\log^{\epsilon} n)$ Access Time

Each of the $\lceil \log \log n \rceil$ levels of the data structure discussed in the previous Section 5.1 uses $O(n)$ bits, so one way to reduce the space complexity is to store only a constant number of levels, at the cost of increased access time. We keep a total of three levels: level 0, level ℓ', and level ℓ, where $\ell' = \lceil \frac{1}{2} \log \log n \rceil$ and as before $\ell = \lceil \log \log n \rceil$. In the previous example of $n = 32$, the three levels chosen are levels 0, 2, and 3. The trick is to determine how to reconstruct SA_0 from $SA_{\ell'}$ and how to reconstruct $SA_{\ell'}$ from SA_{ℓ}.

We keep a bit vector at level 0 that marks the indices that correspond to the entries at level ℓ', and similarly we keep a bit vector at level ℓ' that marks the indices that correspond to the entries at level ℓ. We also have a data structure for $k = 0$ and $k = \ell'$ to support the function Ψ'_k, which is similar to Ψ_k, except that it maps **1**s to the next corresponding **0**. To determine $SA_0[i]$, we use the functions Ψ_0 and Ψ'_0 to walk along indices i', i'', ..., such that $SA_0[i] + 1 = SA_0[i']$, $SA_0[i'] + 1 = SA_0[i'']$, and so on, until we reach a marked index that corresponds to an entry at level ℓ'. We can reconstruct the entry at level ℓ' from the explicit representation of SA_{ℓ} at level ℓ by a similar walk along indices at level ℓ'. We defer details for reasons of brevity. The maximum length of each walk is $\sqrt{\log n}$, and thus the lookup procedure requires $O(\sqrt{\log n}\,)$ time. The method can be generalized by adding more levels to support *lookup* in $O(\log^{\epsilon} n)$ time for any fixed $\epsilon > 0$.

5.3 Output-Sensitive Reporting of Multiple Occurrences

If we want to output a contiguous set $SA_0[i]$, ..., $SA_0[j]$ of entries from the suffix array, one way to output the $j - i + 1$ entries is via a reduction to two-dimensional orthogonal range search. We output the entries in a sequence of *len* stages (for parameter value *len* to be discussed below), with one range search per stage. In the tth stage, for $0 \le t < len$, we output the entries containing suffix pointers that, in the text T, are t symbols to the left of the suffix pointers compressed and kept in the entries of SA_{ℓ}. We say that these entries have *mod value t*. We use a variant of the two-dimensional prefix matching problem [31] and define the points in the range search instance as follows: Consider the lists defined in Section 5.1 and build analogous lists for level $\ell - 1$, except that the suffixes in the lists are *len* positions apart in the text, and so prefix patterns are longer, namely, of length *len* rather than $2^{\ell-1}$. For each entry e in the list of prefix p, we associate it with a two-dimensional point (e, p^{r}), where p^{r} is the reversal of string p.

Assuming that the common prefix among the suffixes starting at positions $SA_0[i]$, ..., $SA_0[j]$ has length at least *len*, we can find the entries among $SA_0[i]$, ..., $SA_0[j]$ whose mod value is t by means of the functions Ψ_0 and Ψ'_0 and a certain two-dimensional orthogonal range query. The range search can be done, using $O(n)$ bits, either in $O(\log \log n + occ_t)$ time for $len = \log^2 n$ [43] or $O(n^{\beta} + occ_t)$ time for $len = \log n$ and any fixed $\beta > 0$ [6; 47]. The total running time for all *len* range searches is thus the cost of the pattern search plus $O((\log^2 n) \log \log n + occ)$, which is $O(m / \log n + occ)$ when the pattern is $\Omega((\log^3 n) \log \log n)$ bits long; for shorter patterns, the total running time for all *len* range searches is the cost of the pattern search plus $O(n^{\beta} \log n + occ)$, which is $O(occ)$ when $occ = \Omega(n^{\epsilon})$, because we can make the choice $0 < \beta < \epsilon$. The details are suppressed for lack of space. The requirement on the common prefix length of the suffixes can be further reduced from *len* to $\Omega(len)$. This requirement is satisfied for the application of text indexing, as noted at the end of Section 4.3.

6 Conclusions

We have presented the first index structure to break through both the time barrier of $O(m)$ time and the space barrier of $O(n \log n)$ bits for fast text searching. Our method, which is based upon notions of compressed suffix arrays and suffix trees, uses $O(n)$ bits to index a text string T of n bits. Given any pattern P of m bits, it can be used to count the number of occurrences of P in T in $o(m)$ time. Namely, searching takes $O(1)$ time for $m = o(\log n)$, and $O(m / \log n + \log^{\epsilon} n) = o(m)$ time for $m = \Omega(\log n)$ and any fixed $0 < \epsilon < 1$. We achieve optimal $O(m / \log n)$ search time for sufficiently large $m = \Omega(\log^{1+\epsilon} n)$. For an enumerative query, retrieving all *occ* occurrences has optimal cost $O(occ)$ time when $m = \Omega((\log^3 n) \log \log n)$ or when $occ = \Omega(n^{\epsilon})$; otherwise, it takes $O(occ \log^{\epsilon} n)$ time.

An interesting open problem is to improve upon our $O(n)$-bit compressed suffix array so that each call to *lookup* takes constant time. Such an improvement would decrease the output-sensitive time of the enumerative queries to $O(occ)$ in all cases. A related question is to characterize combinatorially the permutations that correspond to suffix arrays. A better understanding of the correspondence may lead to more efficient compression methods. Ideally we'd like to find a text index that uses as few as bits as possible and that supports enumerative queries for each query pattern in sublinear time in the worst case. The interplay between compression and indexing is the subject of current investigation in [20]. Additional open problems are listed in [42].

The kinds of queries examined in this paper are very basic and involve exact occurrences of the pattern strings. They are often used as preliminary filters so that more sophisticated queries can be performed on a smaller amount of text. An interesting extension would be to support some sophisticated queries directly, such as those that tolerate a small number of errors in the pattern match [1; 9; 48].

7 References

[1] A. Amir, D. Keselman, G. M. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh. Indexing and dictionary matching with one error. *Lecture Notes in Computer Science*, 1663:181–190, 1999.

[2] A. Andersson, N. J. Larsson, and K. Swanson. Suffix trees on words. *Algorithmica*, 23(3):246–260, 1999.

[3] A. Andersson and S. Nilsson. Efficient implementation of suffix trees. *Software Practice and Experience*, 25(2):129–141, Feb. 1995.

[4] A. Apostolico. The myriad virtues of suffix trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume 12 of *NATO Advanced Science Institutes, Series F*, pages 85–96, Springer-Verlag, berlin, 1985.

[5] A. Apostolico, C. Iliopoulos, G. M. Landau, B. Schieber, and U. Vishkin. Parallel construction of a suffix tree with applications. *Algorithmica*, 3:347–365, 1988.

[6] J. L. Bentley and H. A. Maurer. Efficient worst-case data structures for range searching. *Acta Informatica*, 13:155–168, 1980.

[7] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40(1):31–55, Sept. 1985.

[8] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, July 1987.

[9] G. S. Brodal and L. Gąsieniec. Approximate dictionary queries. In D. S. Hirschberg and E. W. Myers, editors, *Proc. 7th Annual Symp. Combinatorial Pattern Matching, CPM*, volume 1075 of *Lecture Notes in Computer Science, LNCS*, pages 65–74, Springer-Verlag, 10–12 June 1996.

[10] D. Clark. Compact Pat trees. PhD Thesis, Department of Computer Science, University of Waterloo, 1996.

[11] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage (extended abstract). In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391, Atlanta, Georgia, 28–30 Jan. 1996.

[12] L. Colussi and A. De Col. A time and space efficient data structure for string searching on large texts. *Information Processing Letters*, 58(5):217–222, Oct. 1996.

[13] M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45(1):63–86, 1986.

[14] M. Crochemore and D. Perrin. Two–way string matching. *Journal of the Association for Computing Machinery*, 38:651–675, 1991.

[15] M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, 1994.

[16] M. Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science*, pages 137–143, Miami Beach, Florida, 20–22 Oct. 1997. IEEE.

[17] M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In *IEEE Symposium on Foundations on Computer Science (to appear in J.ACM)*, 1998.

[18] M. Farach and S. Muthukrishnan. Optimal logarithmic time randomized suffix tree construction. In F. M. auf der Heide and B. Monien, editors, *Automata, Languages and Programming, 23rd International Colloquium*, volume 1099 of *Lecture Notes in Computer Science*, pages 550–561, Paderborn, Germany, 8–12 July 1996, Springer-Verlag.

[19] P. Ferragina and R. Grossi. The String B-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, Mar. 1999.

[20] P. Ferragina and G. Manzini. Personal communication, 2000.

[21] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the Association for Computing Machinery*, 31(3):538–544, July 1984.

[22] Z. Galil and J. Seiferas. Time-space-optimal string matching. *Journal of Computer and System Sciences*, 26:280–294, 1983.

[23] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. In J. S. Vitter and C. D. Zaroliagis, editors, *Proceedings of the 3rd Workshop on Algorithm Engineering*, number 1668 in Lecture Notes in Computer Science, pages 30–42, London, UK, 1999, Springer-Verlag, Berlin.

[24] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In *Information Retrieval: Data Structures And Algorithms*, chapter 5, pages 66–82. Prentice-Hall, 1992.

[25] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[26] R. W. Irving. Suffix binary search trees. Technical Report TR-1995-7, Computing Science Department, University of Glasgow, 1995.

[27] G. Jacobson. Space-efficient static trees and graphs. In *IEEE Symposium on Foundations of Computer Science*, pages 549–554, 1989.

[28] G. Jacobson. Succinct static data structures. Technical Report CMU-CS-89-112, Dept. of Computer Science, Carnegie-Mellon University, Jan. 1989.

[29] J. Kärkkäinen. Suffix cactus: A cross between suffix tree and suffix array. In *Combinatorial Pattern Matching*, volume 937 of *Lecture Notes in Computer Science*, pages 191–204. Springer, 1995.

[30] J. Kärkkäinen and E. Sutinen. Lempel-Ziv index for q-grams. *Algorithmica*, 21(1):137–154, 1998.

[31] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In N. Ziviani, R. Baeza-Yates, and K. Guimarães, editors, *Proceedings of the 3rd South American Workshop on String Processing*, pages 141–155, Recife, Brazil, 1996. Carleton University Press.

[32] J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. *Lecture Notes in Computer Science*, 1090:219–230, 1996.

[33] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.

[34] S. Kurtz. Reducing the space requirement of suffix trees. Technical Report 98-03, Universität Bielefeld, 1998.

[35] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[36] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 23–32, 1994.

[37] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[38] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, Oct. 1996.

[39] D. R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded In Alphanumeric. *Journal of the ACM*, 15(4):514–534, Oct. 1968.

[40] J. I. Munro. Tables. *FSTTCS: Foundations of Software Technology and Theoretical Computer Science*, 16:37–42, 1996.

[41] J. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *38th Annual Symposium on Foundations of Computer Science*, pages 118–126, 1997.

[42] J. I. Munro, V. Raman, and S. Srinivasa Rao. Space efficient suffix trees. In *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, volume 1530 of *Lecture Notes in Computer Science*, pages 186–195, Berlin, Germany, 1998, Springer-Verlag.

[43] M. H. Overmars. Efficient data structures for range searching on a grid. *Journal of Algorithms*, 9(2):254–275, June 1988.

[44] S. C. Sahinalp and U. Vishkin. Symmetry breaking for suffix tree construction. In *Proceedings of the 26th Annual Symposium on the Theory of Computing*, pages 300–309, New York, May 1994, ACM Press.

[45] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, Sept. 1995.

[46] P. Weiner. Linear pattern matching algorithm. *Proc. 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[47] D. E. Willard. On the application of sheared retrieval to orthogonal range queries. In *Proceedings of the Second Annual Symposium on Computational Geometry*, pages 80–89, 1986.

[48] A. C. Yao and F. F. Yao. Dictionary look-up with small errors. *Lecture Notes in Computer Science*, 937:387–394, 1995.

[49] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490, Dec. 1998.