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

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, N. C. 27708–0129
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 text indexing methods that support fast string searching. In this scenario, consider a text T that is made up of n symbols drawn from a fixed alphabet Σ and that is represented in $n \log |\Sigma|$ bits by encoding each symbol with $\log |\Sigma|$ bits. The goal is to support quick search queries of any string pattern P of m symbols, with T being fully scanned only once, namely, when the index is created.

Text indexing schemes published in the literature are greedy of space and require additional $\Omega(n \log n)$ bits in the worst case. For example, suffix trees and suffix arrays need $\Omega(n)$ memory words of $\Omega(\log n)$ bits in the standard unit cost RAM. These indexes are larger than the text itself by a factor of $\Omega(\log_{|\Sigma|} n)$, which is significant when Σ is of constant size, such as ASCII or UNICODE. On the other hand, they support fast searching either in $O(m \log |\Sigma|)$ time or in $O(m + \log n)$ time, plus an output-sensitive cost $O(occ)$ for listing the pattern occurrences.

We present a new text index that is based upon new compressed representations of suffix arrays and suffix trees. It achieves $O(m/\log_{|\Sigma|} n + \log^{\epsilon}_{|\Sigma|} n)$ search time in the worst case, for any constant $0 < \epsilon \leq 1$, with at most $\left(\epsilon^{-1} + O(1)\right) n \log |\Sigma|$ bits of storage; that is, the index size is comparable to the text size in the worst case. The above bounds improve *both* time and space of previous indexing schemes. Listing the pattern occurrences introduces a sublogarithmic slowdown factor in the output-sensitive cost, giving $O(occ \log^{\epsilon}_{|\Sigma|} n)$ time as a result. When the patterns are sufficiently long, we can use auxiliary data structures in $O(n \log |\Sigma|)$ bits to obtain a total search bound of $O(m/\log_{|\Sigma|} n + occ)$ time, which is optimal.

---

*A preliminary and shortened version of the results has been presented in the *Thirty-Second Annual ACM Symposium on the Theory of Computing* [GV00].

†Work supported in part by the United Nations Educational, Scientific and Cultural Organization (UNESCO) under contract UVO-ROSTE 875.631.9 and by the Italian Ministry of Research and Education (MURST).

‡Part of this work was done while the author was on sabbatical at I.N.R.I.A. in Sophia Antipolis, France. It was supported in part by Army Research Office MURI grant DAAH04–96–1–0013 and grant DAAD19–01–1–0725 and by National Science Foundation research grant CCR–9877133.

1 Introduction

A great deal of textual information is available in electronic form in databases and on the World Wide Web, and therefore devising efficient text indexing methods to support fast string searching is an important research topic. A basic search operation involves string matching [KMP77], in which one is interested in looking for the occurrences of a pattern string P of length m in a longer text string T of length n, both strings having their symbols drawn from a fixed alphabet Σ of size $|\Sigma| \leq n$. An occurrence of the pattern at position i identifies substring $T[i, i+m-1]$ equal to P, where $T[i,j]$ denotes the concatenation of the symbols in T at positions $i, i+1, \ldots, j$. In this paper, we consider 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, where $occ \leq n$. An enumerative query outputs the list of *occ* positions where P occurs in T. Efficient string matching algorithms, such as [KMP77], can answer each individual query in $O(m+n)$ time by an efficient text scan.

The large mass of presently existing text documents makes it infeasible to scan through all the documents for every query, because n is typically much larger than the pattern length m and the number of occurrences *occ*. In this scenario, text indexes are preferable, as they are especially efficient when several string searches are to be performed on the same set of text documents. The text T needs to be entirely scanned only once, namely, when the indexes are created. After that, searching is output-sensitive, that is, the time complexity of each search query is proportional to either $O(m \log |\Sigma| + occ)$ or $O(m + \log n + occ)$, instead of $O(m+n)$.[1]

The most popular indexes currently in use are inverted lists and signature files [Knu98]. Inverted lists are theoretically and practically superior to signature files [ZMR98]. Their versatility allows for several kinds of queries (exact, boolean, ranked, and so on) whose answers have a variety of output formats. They are efficient indexes for texts that are structured as long sequences of terms or keywords obtained by partitioning T into non-overlapping substrings $T[i_k, j_k]$ (the terms), where $1 \leq i_k \leq j_k < i_{k+1} \leq n$. We refer to the set of terms as the vocabulary. For each distinct term in the vocabulary, the index maintains the (inverted or position) list $\{i_k\}$ of the occurrences of that term in T. As a result, search queries must be limited to terms or portions of them; it is not efficient to search for arbitrary substrings of the text as in the string matching problem. For this reason, inverted files are sometimes referred to as term-level or word-level text indexes.

Searching *unstructured* text to answer string matching queries adds a new difficulty to text indexing. This case arises with DNA sequences and in some Eastern languages (Burmese, Chinese, Taiwanese, Tibetan, etc.), which do not have a well-defined notion of terms. Here, the set of successful search keys is possibly much larger than the set of terms in structured texts, because it consists of all feasible substrings of T; that is, we can have $\binom{n}{2} = \Theta(n^2)$ distinct substrings in the worst case, while the number of distinct terms is at most n (considered as nonoverlapping substrings). Suffix arrays [MM93, GBS92], suffix trees [McC76, Wei73] and similar tries or automata [CR94] are among the prominent data structures widely used for unstructured texts. Since they can handle all the search keys in $O(n)$ memory words, they are sometimes referred to as full-text indexes.

The *suffix tree* for text $T = T[1,n]$ is a compact trie whose leaves store the text suffixes $T[1,n]$, $T[2,n]$, $\ldots$, $T[n,n]$ represented with their positions $1, 2, \ldots, n$, and whose internal nodes each have at least two children. The *suffix array* stores a permutation of the positions $1, 2, \ldots, n$, so that a sequential scan of the array corresponds to listing the suffixes in lexicographic order. Associated with the array of positions is another array storing the lengths of the longest common prefixes of a subset of the suffixes to speed up the search [MM93]. Suffix trees and suffix arrays

[1] In this paper, we use the notation $\log_b^c n = (\log n / \log b)^c$ to denote the cth power of the base-b logarithm of n. If no base is specified, the implied base is 2.

organize the suffixes so as to support the efficient search of their prefixes. In order to find an occurrence $T[i, i+m-1] = P$, they exploit the property that P must be the prefix of suffix $T[i, n]$, and this guarantees that the occurrence is found. In general, existential and counting queries take $O(m \log |\Sigma|)$ time using automata or suffix trees and their variations, and $O(m + \log n)$ time using suffix arrays along with longest common prefixes. Enumerative queries take an additive output-sensitive cost $O(occ)$. In this paper, we use the term "suffix array" to denote the array containing the permutation of positions 1, 2, ..., n, but without the longest common prefix information mentioned above. To see why full-text indexes such as suffix arrays are more powerful than term-level inverted lists, we can restrict the suffix arrays to store only the suffixes $T[i_k, n]$ that correspond to the occurrences of the terms. In this way, an efficient full-text index can also implement inverted lists efficiently.

The growing importance of suffix arrays and suffix trees is witnessed by numerous references to a great variety of applications besides string searching, such as in molecular biology, data compression, data mining, and text retrieval, to name a few [Apo85, Gus97, MM93]. More and more applications are based upon these powerful data structures that are no longer confined to the string matching community. However, the amount of data is continuously increasing, and space occupancy has become a critical issue. A major criticism that limits the applicability of indexes based upon suffix arrays and suffix trees is that they occupy significantly more space than do inverted lists.

Suffix arrays store the positions of the suffixes of T as a permutation of 1, 2, ..., n by using $n \log n$ bits (kept in n words of $\log n$ bits each in the unit cost RAM), while suffix trees require between $4n \log n$ and $5n \log n$ bits (kept in $4n$–$5n$ words of the RAM) [MM93]. In contrast, inverted lists require approximately 10% of the text size [MZ96], and so suffix arrays and suffix trees require significantly more bits. (However, as previously mentioned, inverted lists have less functionality, especially on unstructured texts.) Here, we are assuming that each symbol is encoded by $\log |\Sigma|$ bits, so that a text T of n symbols occupies a total $n \log |\Sigma|$ bits. From a theoretical point of view, with large alphabets having $\log |\Sigma| = \Theta(\log n)$, suffix arrays require roughly the same number of bits as the text. Nevertheless, the alphabet Σ is typically of constant size in electronic documents (in ASCII or UNICODE). In this case, suffix arrays and suffix trees are larger than the text by an $O(\log_{|\Sigma|} n) = O(\log n)$ factor. For example, a DNA sequence of n symbols (with $|\Sigma| = 4$) can be stored with $2n$ bits in a computer. The suffix array for the sequence requires instead at least n words of 4 bytes each, or $32n$ bits, which is 16 times larger than the text itself. On the other hand, we cannot resort to inverted files since they are not efficient on unstructured sequences.

In this paper, we investigate whether it is possible to design a full-text index of $o(n \log n)$ bits that supports efficient searching. This problem is of both theoretical and practical interest. By assuming that each text is in one-to-one correspondence with a suffix array, we can derive a simple information-theoretic argument based upon the fact that there are $|\Sigma|^n$ different text strings of length n over the alphabet Σ. Hence, there are so many different suffix arrays, and each of them must require $\Omega(n \log |\Sigma|)$ bits to be distinguishable. It is an interesting problem to see if there is an efficient representation of suffix arrays that takes $n \log |\Sigma| + O(n)$ bits in the worst case.

In order to have an idea of the computational difficulty of the question, let us follow a simple approach that saves space. Let us consider binary alphabets. We bunch every $\log n$ bits together into a word (in effect, constructing a large alphabet) and create a text of length $n / \log n$ and a pattern of length $m / \log n$. The suffix array on the new text requires $O(n)$ bits. Searching for a pattern of length m must also consider situations where the pattern is not aligned at the precise word boundaries. What is the searching cost? It appears that we have to handle $\log n$ situations, with a slowdown factor of $\log n$ in the time complexity of the search. However, this is not really so; we actually have to pay a much larger slowdown factor of $\Omega(n)$ in the search cost, which makes querying the text index more expensive than running the $O(m + n)$-time algorithms such as [KMP77] from scratch. To see why, let us examine the situation in which the pattern occurs k positions to the right of a word boundary in the text. In order to query the index, we have to align the pattern to the boundary by padding k bits to the left of the pattern. Since we do not know a priori the correct k bits to prepend, we must try all 2^k possible ways to pad k binary symbols to the left of the pattern. When $k \approx \log n$, we have to query the index $2^k = \Omega(n)$ times in the worst case (see the sparse suffix trees [KU96b] cited in Section 1.2 to alleviate partially this drawback).

The above example shows that a small reduction in the index size can make querying the index useless in the worst case, as it can cost at least as much as performing a full scan of the text from scratch. In Section 1.2, we describe previous results motivated by the need to find an efficient solution to the problem of designing a full-text index that saves space and time in the worst case. No data structures with the functionality of suffix trees and suffix arrays that have appeared to date in the literature use $\Theta(n \log |\Sigma|) + o(n \log n)$ bits and support fast queries in $o(m \log |\Sigma|)$ or $o(m + \log n)$ worst-case time. Our goal in this paper is to simultaneously reduce *both* the space bound *and* the query time bound.

1.1 Our results

In this paper, we assume for simplicity that the alphabet Σ is of bounded size (i.e., ASCII or UNICODE/UTF8). We recall that the suffix array SA for text T stores the suffixes of T in lexicographic order, and refer the reader to Section 2 for some examples of suffix arrays. We represent SA in the form of a permutation of the starting positions $1, 2, \ldots, n$ in T of the suffixes. For all $1 \leq i < j \leq n$, we have $T\bigl[SA[i], n\bigr] < T\bigl[SA[j], n\bigr]$ in lexicographic order. We call *suffix pointers* the entries in SA.

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 it to obtain its succinct representation.

2. *lookup*(i): Given the compressed representation mentioned above, return $SA[i]$, which is the suffix pointer in T of the ith suffix $T\bigl[SA[i], n\bigr]$ 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*.

In this paper, we exploit the "structure" of the suffix pointers stored in SA by observing that the permutation of positions is not arbitrary. For any fixed value of $0 < \epsilon \leq 1$, we show how to implement operation *compress* in $(1+\epsilon^{-1})\, n \log |\Sigma| + o(n \log |\Sigma|)$ bits and $O(n \log |\Sigma|)$ preprocessing time, so that each call to *lookup* takes sublogarithmic worst-case time, that is, $O(\log^{\epsilon}_{|\Sigma|} n)$ time. We can also achieve $(1 + \frac{1}{2} \log \log_{|\Sigma|} n)\, n \log |\Sigma| + O(n)$ bits and $O(n \log |\Sigma|)$ preprocessing time, so that calls to *lookup* can be done in $O(\log \log_{|\Sigma|} n)$ time. Our findings have several implications:

- We break the space barrier of $\Omega(n \log n)$ bits for a suffix array while retaining $o(\log n)$ lookup time in the worst case. We refer the reader to the literature described in Section 1.2.

- We can implement compressed suffix trees in $2n \log |\Sigma| + O(n)$ bits by using compressed suffix arrays (with $\epsilon = 1$) and the techniques for compact representation of Patricia tries presented in [MRS01]. They occupy asymptotically the *same* space as that of the text string being indexed.

- Our compressed suffix arrays and compressed suffix trees are provably as good as inverted lists in terms of space usage, at least theoretically. In the worst case, they require asymptotically the *same* number of bits.

- We can build a full-text index on T in at most $\big(\epsilon^{-1} + O(1)\big)\, n \log|\Sigma|$ bits by a suitable combination of our compressed suffix trees and previous techniques [CD96, KS98, MRS01, Mor68]. We can answer existential and counting queries of any pattern string of length m in $O(m/\log_{|\Sigma|} n + \log^{\epsilon}_{|\Sigma|} n)$ search time in the worst case, which is $o\big(\min\{m \log|\Sigma|, m + \log n\}\big)$. For enumerative queries, we introduce a sublogarithmic slowdown factor in the output-sensitive cost, giving $O(occ \log^{\epsilon}_{|\Sigma|} n)$ time as a result. When the patterns are sufficiently long, namely, for $m = \Omega\big((\log^{2+\epsilon} n)(\log_{|\Sigma|} \log n)\big)$, we can use auxiliary data structures in $O(n \log|\Sigma|)$ bits to obtain a total search bound of $O(m/\log_{|\Sigma|} n + occ)$ time, which is optimal.

The bounds claimed in the last point need further elaboration. Specifically, searching takes $O(1)$ time for $m = o(\log n)$, and $O(m/\log_{|\Sigma|} n + \log^{\epsilon}_{|\Sigma|} n) = o(m \log|\Sigma|)$ time otherwise. That is, we achieve optimal $O(m/\log_{|\Sigma|} n)$ search time for sufficiently large $m = \Omega(\log^{1+\epsilon}_{|\Sigma|} n)$. For enumerative queries, retrieving all occ occurrences has cost $O(m/\log_{|\Sigma|} n + occ \log^{\epsilon}_{|\Sigma|} n)$ when both conditions $m \in \big[\epsilon \log n, o(\log^{1+\epsilon} n)\big]$ and $occ = o(n^{\epsilon})$ hold, and cost $O\big(m/\log_{|\Sigma|} n + occ + (\log^{1+\epsilon} n)(\log|\Sigma| + \log\log n)\big)$ otherwise.

The results described in this paper are theoretical, but they also have practical value. An implementation of our compressed suffix arrays, with the extensions described by Sadakane [Sad00], is able to index an ASCII text without the need to keep the text. Indeed, Sadakane showed that compressed suffix arrays are enough powerful to retrieve and to search any text substring without accessing the text itself. They require only $\epsilon^{-1} n H_0 + O(n)$ bits, where $H_0 \le \log|\Sigma|$ is the order-0 entropy of the text T. In this way, we have the new notion of *self-indexing text*. Given a text T of n symbols, we have seen that we can search T in $O(n)$ time with string matching algorithms [KMP77]. A self-indexing string S is a shorter representation of T that allows $o(n)$ time searching (see our $o(m)$ search bound for constant size alphabets). Hence, if we replace T with S, we do not only save space but we also get faster searching time. We have some preliminary experimental results on self-indexing texts showing that a 100-megabyte file of Associated Press news and its suffix array can be represented in a total of 30–40 megabytes. The less powerful inverted files would require roughly 110 megabytes as they must keep the text uncompressed. The (uncompressed) suffix array with the text would require 500 megabytes.

1.2 Related Work

The seminal paper by Knuth, Morris, and Pratt [KMP77] 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 [GS83, CP91]. A relevant paper by Weiner [Wei73] introduced a variant of the suffix tree for solving the text indexing problem in string matching. This paper pointed out the importance of text indexing as a tool to avoid a full scan of the text at each pattern search. This method takes $O(m \log|\Sigma|)$ search time plus the output-sensitive cost $O(occ)$ to report the occurrences. Since then, a plethora of papers have studied the text indexing problem in several contexts, sometimes using different terminology [BBH+85, BBH+87, Cro86, FG99, Irv95, McC76, MM93, Ukk95]; for more references see [Apo85, CR94, Gus97]. Although very efficient, the resulting index data structures are greedy in terms of space, using 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 well as 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 [AN95, Cla96, GKS99, Kär95, Kur99, Mäk00, MM93]. Other authors devised several variations of sparse suffix trees to store a subset of the suffixes [ALS99, GBS92, KU96b, KU96a, MW94, Mor68]. Some of them wanted queries to be efficient when the occurrences are aligned with the boundaries 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 [KS98], occupies $O(n)$ bits and takes $O(m)$ time to search patterns shorter than $\log n$; for longer patterns, it may occupy $\Omega(n \log n)$ bits. An efficient and practical compressed index is discussed in [SNZ97], but its searches are at word-level and not full-text (i.e., with arbitrary substrings).

A recent line of research has been built upon Jacobson's succinct representation of trees in $2n$ bits, with navigational operations [Jac89a]. That representation was extended in [CM96] 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 [CD96]. Munro et al. [MRS01] used it along with an improved succinct representation of balanced parentheses [MR97] in order to get $O(m \log |\Sigma|)$ search time with only $n \log n + o(n)$ bits. They also show in [MRS01] how to get $O(m)$ time and $O(n \log n / \log \log n)$ bits for existential queries in binary patterns.

The results described in this paper and its preliminary form [GV00] stimulated further work, and a number of interesting results appeared recently. A first question raised is about lower bounds. Assuming that the text is read-only, Demaine and López Ortiz [DO01] have shown that any text index with alphabet size $|\Sigma| = 2$ that supports fast queries by probing $O(m)$ bits in the text must use $\Omega(n)$ bits of storage space in the worst case under a stronger version of the bit-probe model. Thus, our index is space optimal in this sense. A second question is about compressible text. Ferragina and Manzini [FM00, FM01] have devised an index based upon the Burrows-Wheeler transform that asymptotically achieves the order-k empirical entropy of the text and allows them to obtain self-indexing texts. Their index appears to be theoretically more space efficient than compressed suffix arrays, whereas their $\Omega(m)$ search time is slower than compressed suffix trees. As previously mentioned, Sadakane [Sad00] has shown that compressed suffix arrays can be used for self-indexing texts. He bounds the space taken by compressed suffix arrays in terms of the order-0 entropy. He also uses Lemma 2 in Section 3.1 to show how to store the skip values of the suffix tree in $O(n)$ bits [Sad02]. These interesting results represent a recent new trend in text indexing, making space efficiency no longer a major obstacle to the large-scale application of index data structures [ZSNBY00]. Ideally we'd like to find an 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 (plus the output-sensitive cost).

1.3 Outline of the paper

In Section 2 we describe the ideas behind our new data structure for compressed suffix arrays. Details of our compressed suffix array construction are given in Section 3. In Section 4 we show how to use compressed suffix arrays to construct compressed suffix trees and a general space-efficient indexing mechanism to speed up text search. We give final comments 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.

2 Compressed Suffix Arrays

The compression of suffix arrays falls into the general framework presented by Jacobson [Jac89b] 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 less space efficient) 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 $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, and it is terminated in the nth position by a special end-of-string symbol `#`, such that $\mathtt{a} < \# < \mathtt{b}$.[2] We will discuss the case of alphabets of size $|\Sigma| > 2$ at the end of the section.

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

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 $n-1$. That is, not all the $n!$ permutations are necessarily suffix arrays.

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 $SA[1]$, $SA[2]$, $\ldots$, $SA[x-1]$, and it must have the symbol `b` in each of the positions pointed to by $SA[x+1]$, $SA[x+2]$, $\ldots$, $SA[n]$. 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 `b`s 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 plus `#`, namely, string T itself. The drawback, however, is the unaffordable cost of *lookup*. It takes $\Omega(n)$ time to decompress a single suffix 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.

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

In this section we describe an efficient method to represent suffix arrays in $O(n)$ bits with fast *lookup* operations. Our idea is to distinguish among the permutations of $\{1, 2, \ldots, n\}$ 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 permutation for an arbitrary (e.g., random) string T recursively in terms of shorter permutations. For some examples of divide-and-conquer construction of suffix arrays and suffix trees, see [AIL$^+$88, FC97, FCFM00, FCM96, MM93, SV94]. We reverse the construction process to discover a recursive structure of the permutations that makes their compression possible.

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 $\{1, 2, \ldots, n_k\}$. 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 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 $1 \leq i \leq n_k$, 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 $\mathbf{1}$s for each prefix of B_k. We use function $rank_k$ for this purpose; that is, $rank_k(j)$ counts how many $\mathbf{1}$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 following example illustrates the effect of a single application of Steps 1–4. Here, $\Psi_0(25) = 16$ as $SA_0[25] = 29$ and $SA_0[16] = 30$. The new suffix array SA_1 explicitly stores the suffix pointers (divided by 2) for the suffixes that start at even positions in the original text T. For example, $SA_1[3] = 5$ means that the third lexicographically smallest suffix that starts at an even position in T is the one starting at position $2 \times 5 = 10$, namely, `abbabaa...#`.

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*	*26*	*27*	*28*	*29*	*30*	*31*	*32*
T:	a	b	b	a	b	b	a	b	b	a	b	b	a	b	a	a	a	b	a	b	a	b	b	a	b	b	b	a	b	b	a	#
SA_0:	15	16	31	13	17	19	28	10	7	4	1	21	24	32	14	30	12	18	27	9	6	3	20	23	29	11	26	8	5	2	22	25
B_0:	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**0**
$rank_0$:	0	1	1	1	1	1	2	3	3	4	4	4	5	6	7	8	9	10	10	10	11	11	12	12	12	12	13	14	14	15	16	16
Ψ_0:	2	2	14	15	18	23	7	8	28	10	30	31	13	14	15	16	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

procedure *rlookup*(i, k):

  if $k = \ell$ **then**

    return $SA_\ell[i]$

  else

    return $2 \times rlookup\big(rank_k(\Psi_k(i)),\, k+1\big) + (B_k[i] - 1)$.

Figure 1: Recursive lookup of entry $SA_k[i]$ in a compressed suffix array.

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 $1 \leq i \leq n_k$,*

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

Proof: Suppose $B_k[i] = \mathbf{1}$. By Step 3, there are $rank_k(i)$ $\mathbf{1}$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 that $B_k[i] = \mathbf{0}$ 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 $B_k[i] - 1 = -1$. $\square$

In the previous example, $SA_0[25] = 2 \cdot SA_1[rank_0(16)] - 1 = 2 \cdot 15 - 1 = 29$. 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 3. As a result, we have $\ell + 1$ levels of information, numbered 0, 1, $\ldots$, ℓ, which form the *compressed representation* of suffix array SA:

- Level k, for each $0 \leq k < \ell$, 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 described in Section 3.
- 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 1. 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 of $2 \cdot SA_{k+1}\big[rank_k\big(\Psi_k(i)\big)\big] + (B_k[i] - 1)$ 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 3. Our main theorem below gives the resulting time and space complexity that we are able to achieve.

Theorem 1 (Binary alphabets) *Consider the suffix array SA built upon a binary string of length* $n-1$.

i. *We can implement compress in* $\frac{1}{2}n\log\log n + 6n + O(n/\log\log n)$ *bits and* $O(n)$ *preprocessing time, so that each call lookup*(i) *takes* $O(\log\log n)$ *time.*

ii. *We can implement compress in* $(1+\epsilon^{-1})\,n + O(n/\log\log n)$ *bits and* $O(n)$ *preprocessing time, so that each call lookup*(i) *takes* $O(\log^{\epsilon} n)$ *time, for any fixed value of* $0 < \epsilon \leq 1$.

The coefficients on the second-order terms can be tweaked theoretically by a more elaborate encoding. We also state the above results in terms of alphabets with $|\Sigma| > 2$.

Theorem 2 (General alphabets) *Consider the suffix array SA built upon a string of length* $n-1$ *over the alphabet* Σ *with size* $|\Sigma| > 2$.

i. *We can implement compress in* $(1 + \frac{1}{2}\log\log_{|\Sigma|} n)\,n\log|\Sigma| + 5n + O(n/\log\log n) = (1 + \frac{1}{2}\log\log_{|\Sigma|} n)\,n\log|\Sigma| + O(n)$ *bits and* $O(n\log|\Sigma|)$ *preprocessing time, so that each call lookup*(i) *takes* $O(\log\log_{|\Sigma|} n)$ *time.*

ii. *We can implement compress in* $(1+\epsilon^{-1})\,n\log|\Sigma| + 2n + O(n/\log\log n) = (1+\epsilon^{-1})\,n\log|\Sigma| + o(n\log|\Sigma|)$ *bits and* $O(n\log|\Sigma|)$ *preprocessing time, so that each call lookup*(i) *takes* $O(\log^{\epsilon}_{|\Sigma|} n)$ *time, for any fixed value of* $0 < \epsilon \leq 1$. *For* $|\Sigma| = O(1)$, *the space bound reduces to* $(1+\epsilon^{-1})\,n\log|\Sigma| + O(n/\log\log n) = (1+\epsilon^{-1})\,n\log|\Sigma| + o(n)$ *bits.*

Sadakane [Sad00] has shown that the space complexity in Theorem 1.*ii* and Theorem 2.*ii* can be restated in terms of the order-0 entropy $H_0 \leq \log|\Sigma|$ of the string, giving as a result $\epsilon^{-1}H_0\,n + O(n)$ bits.

The lookup process can be sped up when we need to report several contiguous entries, as in enumerative string matching queries. Let $lcp(i,j)$ denote the length of the longest common prefix between the suffixes pointed to by $SA[i]$ and $SA[j]$, with the convention that $lcp(i,j) = -\infty$ when $i < 1$ or $j > n$. We say that a sequence $i, i+1, \ldots, j$ of indices in SA is *maximal* if both $lcp(i-1,j)$ and $lcp(i,j+1)$ are strictly smaller than $lcp(i,j)$, as in enumerative queries.

Theorem 3 (Batch of lookups) *In each of the cases stated in Theorem 1 and Theorem 2, we can use additional space of* $O(n\log|\Sigma|)$ *bits and batch together* $j-i+1$ *procedure calls lookup*(i), *lookup*$(i+1)$, $\ldots$, *lookup*(j), *for a maximal sequence* $i, i+1, \ldots, j$, *so that the total cost is*

- $O\big(j-i+(\log n)^{1+\epsilon}(\log|\Sigma| + \log\log n)\big)$ *time when* $lcp(i,j) = \Omega(\log^{1+\epsilon} n)$, *namely, the suffixes pointed to by* $SA[i]$ *and* $SA[j]$ *have the same first* $\Omega(\log^{1+\epsilon} n)$ *symbols in common, or*
- $O(j-i+n^{\alpha})$ *time, for any constant* $0 < \alpha < 1$, *when* $lcp(i,j) = \Omega(\log n)$, *namely, the suffixes pointed to by* $SA[i]$ *and* $SA[j]$ *have the same first* $\Omega(\log n)$ *symbols.*

3 Algorithms for Compressed Suffix Arrays

In this section we constructively prove Theorems 1–3 by showing two ways to implement the recursive decomposition of suffix arrays discussed in Lemma 1 of Section 2. In particular, in Section 3.1 we address Theorem 1.i, and in Section 3.2 we prove Theorem 1.ii. Section 3.3 shows how to extend Theorem 1 to deal with alphabets of size $|\Sigma| > 2$, thus proving Theorem 2. In Section 3.4 we prove Theorem 3 on how to batch together the lookup of several contiguous entries in suffix arrays, which arises in enumerative string matching queries.

3.1 Compressed Suffix Arrays in $\frac{1}{2}n \log\log n + O(n)$ Bits and $O(\log\log n)$ Access Time

In this section we describe the method referenced in Theorem 1.i for binary strings and show that it achieves $O(\log\log n)$ lookup time with a total space usage of $O(n \log\log n)$ bits. Before giving the algorithmic details of the method, let's continue the recursive decomposition of Steps 1–4 described in Section 2, for $0 \le k \le \ell - 1$, where $\ell = \lceil \log\log n \rceil$. The decomposition below shows the result on the example of Section 2:

	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 at most 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, for $0 \le k \le \ell-1$, is to investigate how to represent $rank_k$ and Ψ_k in $O(n)$ bits and support constant-time access. Given the bit vector B_k of $n_k = n/2^k$ bits, Jacobson [Jac89a] shows how to support constant-time access to $rank_k$ using only $O\big(n_k(\log\log n_k)/\log n_k\big)$ extra bits, with preprocessing time $O(n_k)$.

We show next how to represent Ψ_k implicitly. First we explain the representation by an example and then we describe it formally. In Lemma 3 we show that the space used to represent Ψ_k is $n(1/2 + 3/2^{k+1}) + O(n/2^k \log\log n)$ bits.

For each $1 \le i \le n_k/2$, let j be the index of the ith $\mathbf{1}$ in B_k. Consider the 2^k symbols in positions $2^k \cdot (SA_k[j]-1), \ldots, 2^k \cdot SA_k[j] - 1$ of T; these 2^k symbols immediately precede the $\big(2^k \cdot SA_k[j]\big)$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 $j \in [1, n/2^k]$ 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:

$$\begin{array}{lll} \texttt{a list:} & \langle 2,14,15,18,23,28,30,31\rangle, & |\texttt{a list}| = 8 \\ \texttt{b list:} & \langle 7,8,10,13,16,17,21,27\rangle, & |\texttt{b list}| = 8 \end{array}$$

Level 1:

$$\begin{array}{lll} \texttt{aa list:} & \emptyset, & |\texttt{aa list}| = 0 \\ \texttt{ab list:} & \langle 9\rangle, & |\texttt{ab list}| = 1 \\ \texttt{ba list:} & \langle 1,6,12,14\rangle, & |\texttt{ba list}| = 4 \\ \texttt{bb list:} & \langle 2,4,5\rangle, & |\texttt{bb list}| = 3 \end{array}$$

Level 2:

`aaaa` list:	$\emptyset$,	$\lvert$`aaaa` list$\rvert = 0$	`baaa` list:	$\emptyset$,	$\lvert$`baaa` list$\rvert = 0$
`aaab` list:	$\emptyset$,	$\lvert$`aaab` list$\rvert = 0$	`baab` list:	$\emptyset$,	$\lvert$`baab` list$\rvert = 0$
`aaba` list:	$\emptyset$,	$\lvert$`aaba` list$\rvert = 0$	`baba` list:	$\langle 1 \rangle$,	$\lvert$`baba` list$\rvert = 1$
`aabb` list:	$\emptyset$,	$\lvert$`aabb` list$\rvert = 0$	`babb` list:	$\langle 4 \rangle$,	$\lvert$`babb` list$\rvert = 1$
`abaa` list:	$\emptyset$,	$\lvert$`abaa` list$\rvert = 0$	`bbaa` list:	$\emptyset$,	$\lvert$`bbaa` list$\rvert = 0$
`abab` list:	$\emptyset$,	$\lvert$`abab` list$\rvert = 0$	`bbab` list:	$\emptyset$,	$\lvert$`bbab` list$\rvert = 0$
`abba` list:	$\langle 5, 8 \rangle$,	$\lvert$`abba` list$\rvert = 2$	`bbba` list:	$\emptyset$,	$\lvert$`bbba` list$\rvert = 0$
`abbb` list:	$\emptyset$,	$\lvert$`abbb` 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 $\mathbf{0}$s in B_k up to index i by computing $i - rank_k(i)$, i.e., by subtracting the number of $\mathbf{1}$s in the first i bits of B_k. Consider the 2^{2^k} lists concatenated together in lexicographic order of the 2^k-bit prefixes. We denote by L_k the resulting concatenated list, which has $|L_k| = n_k/2 = n/2^{k+1}$ total items. What we need to find now is the hth entry in L_k. For example, to determine $\Psi_0(25)$ in the example above, we find that there are $h = 13$ $\mathbf{0}$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 L_0 is the fifth entry 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 $h = 3$ $\mathbf{0}$s in the first eight entries of B_1. The third entry in the concatenated list L_1 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.

We now describe formally how to preprocess the input text T in order to form the concatenated list L_k on level k used for Ψ_k with the desired space and constant-time query performance. We first consider a variant of the "inventories" introduced by Elias [Eli74] to get average bit efficiency in storing sorted multisets. We show how to get worst-case efficiency.

Lemma 2 (Constant-time access to compressed sorted integers) *Given s integers in sorted order, each containing w bits, where $s < 2^w$, we can store them with at most $s(2 + w - \lfloor \log s \rfloor) + O(s/\log\log s)$ bits, so that retrieving the hth integer takes constant time.*

Proof: We take the first $z = \lfloor \log s \rfloor$ bits of each integer in the sorted sequence. Let $q_1, \ldots, q_s$ be the integers so obtained, called *quotients*, where $0 \le q_h \le q_{h+1} < s$ for $1 \le h < s$. (Note that multiple values are allowed.) Let $r_1, \ldots, r_s$ be the *remainders*, obtained by deleting the first z bits from each integer in the sorted sequence.

We store $q_1, \ldots, q_s$ in a table Q described below, requiring $2s + O(s/\log\log s)$ bits. We store $r_1, \ldots, r_s$ in a table R taking $s(w - z)$ bits. Table R is the simple concatenation of the bits representing $r_1, \ldots, r_s$.

As for Q, we use the unary representation $\mathbf{0}^i\mathbf{1}$ (i.e., i copies of $\mathbf{0}$ followed by $\mathbf{1}$) to represent integer $i \ge 0$. Then we take the concatenation of the unary representation of $q_1, q_2 - q_1, \ldots, q_s - q_{s-1}$. In other words, we take the first entry encoded in unary, and then the unary difference between the other consecutive entries, which are in nondecreasing order. Table Q is made up of the binary string obtained by the above concatenation S, augmented with the auxiliary data structure supporting *select* operations to locate the position of the hth $\mathbf{1}$ in constant time [Cla96, Jac89a, Mun96].

Since S requires $s + 2^z \leq 2s$ bits, the total space required by Q is $2s + O(s/\log\log s)$ bits; the big-oh term is due to the auxiliary data structure that implements *select*. In order to retrieve q_h, we find the position j of the hth **1** in S by calling *select*(h), and then compute the number of **0**s in the first j bits of S by returning $j - h$. As we can see, this number of **0**s gives q_h. The time complexity is constant.

In order to obtain the hth integer in the original sorted sequence, we find q_h by querying Q as described above, and we find r_i by looking up the hth entry in R. We then output $q_h \cdot 2^{w-z} + r_h$ as the requested integer, by simply returning the concatenation of the bit representations of q_h and r_h. □

We now proceed to the implementation of Ψ_k.

Lemma 3 *We can store the concatenated list L_k used for Ψ_k in $n(1/2+3/2^{k+1})+O(n/2^k \log\log n)$ bits, so that accessing the hth entry in L_k takes constant time. Preprocessing time is $O(n/2^k + 2^{2^k})$.*

Proof: There are $d = 2^{2^k}$ lists, some of which may be empty. We number the lists composing L_k from 0 to $2^{2^k} - 1$. Each integer x in list i, where $1 \leq x \leq n_k$, is transformed into an integer x' of $w = 2^k + \log n_k$ bits, by prepending the binary representation of i to that of $x - 1$. Given any such x', we can obtain the corresponding x in constant time. As a result, L_k contains $s = n_k/2 = n/2^{k+1}$ integers in increasing order, each integer of w bits. By Lemma 2, we can store L_k in $s(2 + w - \log s) + O(s/\log\log s)$ bits, so that retrieving the hth integer takes constant time. Substituting the values for s and w, we get the space bound $(n_k/2)\big(2 + 2^k + \log n_k - \log(n_k/2)\big) + O(n_k/\log\log n_k) = (n/2^{k+1})(2^k + 3) + O(n/2^k \log\log n) = n(1/2 + 3/2^{k+1}) + O(n/2^k \log\log n)$. □

A good way to appreciate the utility of the data structure for Ψ_k is to consider the naive alternative. Imagine that the information is stored naively in the form of an unsorted array of $s = n_k/2$ entries, where each entry specifies the particular list that the entry belongs to. Since there are $d = 2^{2^k}$ lists, the total number of bits to store the array in this naive manner is $s \log d = (n_k/2)2^k = n/2$, which is efficient in terms of space. Let us define the natural ordering $\prec$ on the array entries, in which we say that $i \prec j$ either if $i < j$ or if $i = j$ and the position of i in the array precedes the position of j. The naive representation does not allow us to efficiently lookup the hth $\prec$-ordered entry in the array, which is equivalent to finding the hth entry in the concatenated list L_k. It also doesn't allow us to search quickly for the gth occurrence of the entry i, which is equivalent to finding the gth item in list i. In contrast, the data structure described in Lemma 3 supports both of these query operations in linear space and constant time:

Corollary 1 *Given an unsorted array of s entries, each in the range $[0, d-1]$, we can represent the array in a total of $s \log d + O(s)$ bits so that, given h, we can find the hth entry (in $\prec$ order) in the array in constant time. We can also represent the array in $O(s \log d)$ bits so that, given g and i, we can find the gth occurrence of i in the array in constant time. The latter operation can be viewed as a generalization of the select operation to arbitrary input patterns.*

Proof: The first type of query is identical to finding the hth item in the concatenated list L_k, and the bound on space follows from the construction in Lemma 3. The corresponding values of s and w in the proof of Lemma 3 are s and $\log d + \log s$, respectively.

The second type of query is identical to finding the gth entry in list i. It can be turned into the first type of query if we can compute the value of h that corresponds to g and i; that is, we need to find the global position h (with respect to $\prec$) of the gth entry in list i. If $d \leq s$, then we can explicitly store a table that gives for each $0 \le i < d$ the first location h' in the concatenated list L_k that corresponds to an entry in list i. We then set $h = h' + g - 1$ and do the first type of query. (If list i has fewer than g entries, which can be detected after the query is done, the value returned by the first query must be nullified.) The total space used is $d \log s$, which by the assumption $d \le s$ is at most $s \log d$. If instead $d > s$, then we can use the same approach as above, except that we substitute a perfect hash function to compute the value h'. The space for the hash table is $O(s \log s) = O(s \log d)$. □

Putting it all together. At this point, we have all the pieces needed to finish the proof of Theorem 1.i. Given text T and its suffix array, we proceed in $\ell = \lceil \log \log n \rceil$ levels of decomposition as discussed in procedure *compress* in Section 2. The last level ℓ stores explicitly a reduced suffix array in $(n/2^\ell) \log n \le n$ bits. The other levels $0 \le k \le \ell - 1$ store three data structures each, with constant time access:

- Bit vector B_k of size $n_k = n/2^k$, with $O(n_k)$ preprocessing time.
- Function $rank_k$ in $O\big(n_k (\log \log n_k)/ \log n_k\big)$ bits, with $O(n_k)$ preprocessing time.
- Function Ψ_k in $n(1/2 + 3/2^{k+1}) + O(n/2^k \log \log n)$ bits, with $O(n_k + 2^{2^k})$ preprocessing time (see Lemma 3).

By summing over the levels, substituting the values $\ell = \lceil \log \log n \rceil$ and $n_k = n/2^k$, we get the following bound on the total space:

$$\begin{aligned}
&\frac{n \log n}{2^\ell} + \sum_{k=0}^{\ell-1} n \left(\frac{1}{2^k} + O\left(\frac{1}{2^k} \frac{\log \log (n/2^k)}{\log (n/2^k)} \right) + \frac{1}{2} + \frac{3}{2^{k+1}} + O\left(\frac{1}{2^k \log \log n} \right) \right) \\
&< \frac{n \log n}{2^\ell} + n \left(2 + O\left(\frac{\log \log n}{\log n} \right) + \frac{1}{2}\ell + 3 + O\left(\frac{1}{\log \log n} \right) \right) \\
&= \frac{n \log n}{2^\ell} + \frac{1}{2}\ell n + 5n + O\left(\frac{n}{\log \log n} \right). \qquad (1)
\end{aligned}$$

It's easy to show that $(n \log n)/2^\ell + \frac{1}{2}\ell n \le \frac{1}{2} n \log \log n + n$, which combined with (1) gives us the desired space bound $\frac{1}{2} n \log \log n + 6n + O(n/ \log \log n)$ in Theorem 1.i.

The total preprocessing time of *compress* is $\sum_{k=0}^{\ell-1} O(n_k + 2^{2^k}) = O(n)$. A call to *lookup* goes through the $\ell + 1$ levels, in constant time per level, with a total cost of $O(\log \log n)$. This completes the proof of Theorem 1.i.

3.2 Compressed Suffix Arrays in $\epsilon^{-1} n + O(n)$ Bits and $O(\log^\epsilon n)$ Access Time

In this section we give the proof of Theorem 1.ii. Each of the $\lceil \log \log n \rceil$ levels of the data structure discussed in the previous Section 3.1 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. For example, we can 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 store the $n_{\ell'}$ indices from SA_0 that correspond to the entries of $SA_{\ell'}$ in a new dictionary D_0, and similarly we store the n_ℓ indices from $SA_{\ell'}$ that correspond to the entries of SA_ℓ in a new dictionary $D_{\ell'}$. By using the efficient static dictionary representation in [BM99, Pag01], we need less than $O\big(\log \binom{n}{n_{\ell'}}\big) = O(n_{\ell'} \ell')$ bits for D_0 and $O\big(\log \binom{n_{\ell'}}{n_\ell}\big) = O(n_\ell \ell)$ bits for $D_{\ell'}$. A dictionary lookup requires constant time, as does a rank query to know how many smaller or equal indices are stored in the dictionary [Pag01].

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**. We denote by Φ_k the resulting composition of Ψ_k and Ψ'_k, for $1 \leq i \leq n_k$:

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

We implement Φ_k by merging the concatenated lists L_k of Ψ_k with the concatenated lists L'_k of Ψ'_k. For example, in level $k = 0$ shown in Section 3.1, we merge the `a` list of L_k with the `a` list of L'_k, and so on (we need also the singleton list for `#`). This is better than storing L_k and L'_k separately. Computing $\Phi_k(i)$ amounts to taking the ith entry in its concatenated list, and we do not need anymore the bitvector B_k.

Lemma 4 *We can store the concatenated lists used for Φ_k in $n + O(n/\log\log n)$ bits for $k = 0$, and $n(1 + 1/2^{k-1}) + O(n/2^k \log\log n)$ bits for $k > 0$, so that accessing the hth entry takes constant time. Preprocessing time is $O(n/2^k + 2^{2^k})$.*

Proof: For $k > 0$, the proof is identical to that of Lemma 3, except that $s = n_k$ instead of $s = n_k/2$. For $k = 0$, we have only the `a` list and the `b` list to store, with the singleton `#` list handled a bit differently. Specifically, we encode `a` and `#` by **0** and `b` by **1**. Then, we create a bitvector of n bits, where the bit in position f is **0** if the list for Φ_0 contains either `a` or `#` in position f, and it is **1** if it contains `b` in position f. We use auxiliary information to access the ith **1** of the bitvector in constant time by using $select(i)$ or the ith **0** by using $select_0(i)$. We also keep a counter c_0 to know the total number of **0**'s in the bitvector (note that the single occurrence of **0** corresponding to `#` in the bitvector is the c_0th **0** in the bitvector as we assumed `a` < `#` < `b`; it is not difficult to treat the more common case `#` < `a` < `b`). The additional space is $O(n/\log\log n)$ due to the implementation of $select$ and $select_0$. Suppose now that we want to recover the hth entry in the list for Φ_0. If $h = c_0$, then we must recover the position of `#` by returning $select_0(c_0)$. If $h < c_0$, then we must recover the hth **0** (i.e., `a`) in the bitvector, by returning $select_0(h)$. Otherwise, we call $select(h - c_0)$ to get the position in the bitvector of the $(h - c_0)$th **1** (i.e., `b`). In any case, with $n + O(n/\log\log n)$ bits to implement Φ_0, we can execute $\Phi_0(h)$ in constant time. $\square$

In order to determine $SA[i] = SA_0[i]$, we use function Φ_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 an index stored in dictionary D_0. Let s be the number of steps in the walk, and r be the rank of the index thus found in D_0. We switch to level ℓ', and reconstruct the rth entry at level ℓ' from the explicit representation of SA_ℓ at level ℓ by a similar walk until we find an index stored in $D_{\ell'}$. Let s' be the number of steps in the latter walk, and r' be the rank of the index thus found in $D_{\ell'}$. We return $(SA_\ell[r'] \cdot 2^\ell + s' \cdot 2^{\ell'} + s \cdot 2^0)$ as this is the value of $SA_0[i]$. We defer details for reasons of brevity. The maximum length of each walk is $\max\{s, s'\} \leq 2^{\ell'} < 2\sqrt{\log n}$, and thus the lookup procedure requires $O(\sqrt{\log n}\,)$ time.

To get the more general result stated in Theorem 1.ii, we need to keep a total of $\epsilon^{-1} + 1$ levels, for constant $0 < \epsilon \leq 1$. More formally, let us assume that $\epsilon\ell$ is an integer. We maintain the $\epsilon^{-1} + 1$ levels 0, $\epsilon\ell$, $2\epsilon\ell$, ..., ℓ. The maximum length of each walk is $2^{\epsilon\ell} < 2\log^\epsilon n$, and thus the lookup procedure requires $O(\log^\epsilon n)$ time.

By an analysis similar to the one we used at the end of Section 3.1, the total space bound is given by $(n/2^\ell)\log n \leq n$ plus a sum over the ϵ^{-1} indices $k \in \{0, \epsilon\ell, 2\epsilon\ell, 3\epsilon\ell, \ldots, (1-\epsilon)\ell\}$. We split the sum into two parts, one for $k = 0$ and the other for the remaining $\epsilon^{-1} - 1$ values of $k > 0$, and apply Lemma 4:

$$
\begin{aligned}
& \frac{n \log n}{2^\ell} + n + O\left(\frac{n}{\log\log n}\right) + \sum_{\substack{k=i\epsilon\ell \\ 1 \le i < \epsilon^{-1}}} n\left(1 + \frac{1}{2^{k-1}} + O\left(\frac{1}{2^k \log\log n}\right)\right) \\
\le \;& (1+\epsilon^{-1})\, n + O\left(\frac{n}{\log\log n}\right) + O\left(\frac{n}{\log^\epsilon n}\right) \\
= \;& (1+\epsilon^{-1})\, n + O\left(\frac{n}{\log\log n}\right). \qquad (2)
\end{aligned}
$$

We have to add the contribution of the space $\sum_k |D_k| = O(n_{\epsilon\ell}\, \ell) = O(n(\log\log n)/\log^\epsilon n)$ taken by the dictionaries at the ϵ^{-1} levels, but this bound is hidden by the $O(n/\log\log n)$ term in the above formula. The final bound is $(1+\epsilon^{-1})\, n + O(n/\log\log n)$, as stated in Theorem 1.ii.

3.3 Extension to Alphabets of Size $|\Sigma| > 2$

We now discuss the case of alphabets with more than two symbols. In this case, we encode each symbol by $\log|\Sigma|$ bits, so that the text T can be seen as an array of n entries, each of $\log|\Sigma|$ bits, or equivalently as a binary string that occupies $n \log|\Sigma|$ bits. We describe how to extend the ideas presented in Sections 3.1–3.2. We redefine ℓ to be $\lceil \log\log_{|\Sigma|} n \rceil$. The definitions of suffix arrays SA and SA_k, bit vector B_k, and functions $rank_k$ and Ψ_k are the same as before. Their representation does not change, with the notable exception of Ψ_k, as noted below in Lemma 5 (the analogue of Lemma 3):

Lemma 5 *When $|\Sigma| > 2$, we can store the concatenated list L_k used for Ψ_k in $n\big((1/2)\log|\Sigma| + 3/2^{k+1}\big) + O(n/2^k \log\log n)$ bits, so that accessing the hth entry in L_k takes constant time. Preprocessing time is $O(n/2^k + 2^{2^k})$.*

Proof: The extension of L_k with $|\Sigma| > 2$ is straightforward. For each of $d = |\Sigma|^{2^k} = 2^{2^k \log|\Sigma|}$ patterns of 2^k symbols preceding the $\big(2^k \cdot SA_k[j]\big)$th suffix in T, we keep an ordered list like the a list and b list described in Section 3.1. Some of these lists may be empty and the concatenation of non-empty lists forms L_k. We number these lists from 0 to $2^{2^k \log|\Sigma|} - 1$. Note that the number of entries in L_k remains unchanged, namely, $s = n_k/2 = n/2^{k+1}$. Each integer x in list i, where $1 \le x \le n_k$, is transformed into an integer x' of $w = 2^k \log|\Sigma| + \log n_k$ bits, by prepending the binary representation of i to that of $x-1$. By Lemma 2, we can store L_k in $s(2+w-\log s) + O(s/\log\log s)$ bits, so that retrieving the hth integer takes constant time. Substituting the values for s and w, we get the space bound $(n_k/2)\big(2+2^k\log|\Sigma| + \log n_k - \log(n_k/2)\big) + O(n_k/\log\log n_k) = n\big((1/2)\log|\Sigma| + 3/2^{k+1}\big) + O(n/2^k \log\log n)$. ☐

By replacing the space complexity of Ψ_k in formula (1) at the end of Section 3.1, we obtain

$$
\begin{aligned}
& \frac{n \log n}{2^\ell} + \sum_{k=0}^{\ell-1} n\left(\frac{1}{2^k} + O\left(\frac{1}{2^k}\frac{\log\log(n/2^k)}{\log(n/2^k)}\right) + \frac{\log|\Sigma|}{2} + \frac{3}{2^{k+1}} + O\left(\frac{1}{2^k \log\log n}\right)\right) \\
< \;& \left(1 + \frac{1}{2}\log\log_{|\Sigma|} n\right) n \log|\Sigma| + 5n + O\left(\frac{n}{\log\log n}\right),
\end{aligned}
$$

as $(n\log n)/2^\ell + \frac{1}{2}\ell n \le (1 + \frac{1}{2}\log\log_{|\Sigma|} n)\, n \log|\Sigma|$, thus proving Theorem 2.i.

To prove Theorem 2.ii, we follow the approach of Section 3.2. We need dictionaries D_k and functions Φ_k, for $k \in \{0, \epsilon\ell, 2\epsilon\ell, 3\epsilon\ell, \ldots, (1-\epsilon)\ell\}$. Their definitions and representations do not change, except for the representation of Φ_k, for which we need Lemma 6 (the analogue of Lemma 4):

Lemma 6 *We can store the concatenated lists used for* Φ_k *in* $n(\log|\Sigma|+1/2^{k-1})+O(n/2^k \log\log n)$ *bits, so that accessing the hth entry takes constant time. Preprocessing time is* $O(n/2^k+2^{2^k})$*. When* $|\Sigma| = O(1)$ *and* $k = 0$*, we can store* Φ_k *in* $n\log|\Sigma| + O(|\Sigma|n/\log\log n) = n\log|\Sigma| + o(n)$ *bits.*

Proof: The proof is identical to that of Lemma 5, except that $s = n_k$ instead of $s = n_k/2$. When $|\Sigma| = O(1)$, we can use a better approach for $k = 0$ like in the proof of Lemma 4. We associate $\log|\Sigma|$ bits to each character in Σ according to its lexicographic order. Then we use a bitvector of $n\log|\Sigma|$ bits to represent Φ_0, in which the fth group of $\log|\Sigma|$ bits encoding a character $c \in \Sigma$ represents the fact that the list for Φ_0 has c in position f. We then implement $|\Sigma| = O(1)$ versions of *select*, one version per character of Σ. The version for $c \in \Sigma$ is in charge of selecting the ith occurrence of c encoded in binary in the bitvector. To this end, it treats each occurrence of the $\log|\Sigma|$ bits for c in the bitvector as a single **1** and the occurrences of the rest of the characters as single **0**'s. As it should be clear, the implementation of each version of *select* can be done in $O(n/\log\log n)$ bits. To execute $\Phi_0(h)$ in constant time, we proceed as in Lemma 4, generalized to more than two characters. □

By an analysis similar to the one we used in formula (2) at the end of Section 3.2, we obtain

$$\begin{aligned}
&\frac{n\log n}{2^\ell} + \sum_{\substack{k=i\epsilon\ell\\ 0\le i<\epsilon^{-1}}} n\left(\log|\Sigma| + \frac{1}{2^{k-1}} + O\left(\frac{1}{2^k\log\log n}\right)\right)\\
&\le \quad (1+\epsilon^{-1})\, n\log|\Sigma| + 2n + O\left(\frac{n}{\log\log n}\right).
\end{aligned}$$

When $|\Sigma| = O(1)$, we can split the above sum for $k = 0$ and apply Lemma 6 to get $(1+\epsilon^{-1})\, n\log|\Sigma| + O(n/\log\log n)$ bits, thus proving Theorem 2.ii.

3.4 Output-Sensitive Reporting of Multiple Occurrences

In this section we prove Theorem 3 by showing how to output a contiguous set $SA_0[i]$, ..., $SA_0[j]$ of entries from the compressed suffix array under the hypothesis that the sequence i, $i+1$, ..., j is maximal (according to the definition given before Theorem 3) and the corresponding suffixes share at least a certain number of initial symbols. This requires adding further $O(n\log|\Sigma|)$ bits of space to the compressed suffix array. One way to output the $j-i+1$ entries is via a reduction to two-dimensional orthogonal range search [KU96a]. Let D be a two-dimensional orthogonal range query data structure on q points in the grid space $[1\dots U]\times[1\dots U]$, where $1 \le q \le U$. Let $P(q)$ be its preprocessing time, $S(q)$ the number of occupied words of $O(\log U)$ bits each, and $T(q)+O(k)$ be the cost of searching and retrieving the k points satisfying a given range query in D.

Lemma 7 *Fix* $U = n$ *in the range query data structure* D*, and let* $n' \ge 1$ *be the largest integer such that* $S(n') = O(n/\log n)$*. If such* n' *exists, we can report* $SA[i]$*, ...,* $SA[j]$ *in* $O\big(\log_{|\Sigma|}^{1+\epsilon} n + (n/n')(T(n') + \log|\Sigma|) + j - i\big)$ *time when the sequence* i*,* $i+1$*, ...,* j *is maximal and the suffixes pointed to by* $SA[i]$*, ...* $SA[j]$ *have the same first* $\Omega(n/n')$ *symbols in common. Preprocessing time is* $P(n') + O(n\log|\Sigma|)$ *and space is* $O(n\log|\Sigma|)$ *bits in addition to that of the compressed version of SA.*

Proof: Suppose by hypothesis that the suffixes pointed to by $SA[i]$, ..., $SA[j]$ have in common at least $l = \lceil n/n' \rceil$ symbols. (This requirement can be further reduced to $l = \Theta(n/n')$.) We denote these symbols by b_0, b_1, ..., b_{l-1}, from left to right.

In order to define the two-dimensional points in D, we need to build the compressed version of the suffix array SA^R for the reversal of the text, denoted T^R. Then we obtain the points to keep in D by processing the suffix pointers in SA that are multiples of l (i.e., they refer to the suffixes in T starting at positions l, $2l$, $3l$, ...). Specifically, the point corresponding to pointer $p = SA[s]$, where $1 \leq s \leq n$ and p is a multiple of l, has first coordinate s. Its second coordinate is given by the position r of $(T[1, p-1])^R$ in the sorted order induced by SA^R. In other words, s is the rank of $T[p, n]$ among the suffixes of T in lexicographic order, and r is the rank of $(T[1, p-1])^R$ among the suffixes of T^R (or, equivalently, the reversed prefixes of T). Point $\langle s, r \rangle$ corresponding to p has label p to keep track of this correspondence.

Since there are $q \leq n'$ such points stored in D and we build the compressed suffix array of T^R according to Theorem 2.ii, space is $S(n') \cdot O(\log n) + (\epsilon^{-1} + O(1))\, n \log |\Sigma| = O(n \log |\Sigma|)$ bits. Preprocessing time is $P(n') + O(n \log |\Sigma|)$.

We now describe how to query D and output $SA[i]$, ..., $SA[j]$ in l stages, with one range query per stage. In stage 0, we perform a range query for the points in $[i \dots j] \times [1 \dots n]$. For these points, we output the suffix pointers labeling them. Then we locate the leftmost suffix and the rightmost suffix in SA^R starting with $b_{l-1} \cdots b_1 b_0$. For this purpose, we run a simple binary search in the compressed version of SA^R, comparing at most $\log n$ bits at a time. As a result, we determine two positions g and h of SA^R in $O(l \log |\Sigma| + \log_{|\Sigma|}^{1+\epsilon} n)$ time, such that the sequence g, $g+1$, ..., h is maximal for SA^R and the suffixes of T^R pointed to by $SA^R[g]$, ..., $SA^R[h]$ start with $b_{l-1} \cdots b_1 b_0$.

Before going on in the next stages, we precompute some sequences of indices starting from i, j, g and h, respectively, as done in Section 3.2. We use the function Φ_0 in the compressed version of $SA = SA_0$ to walk along indices i_0, i_1, ..., i_{l-1}, such that $i_0 = i$, $SA_0[i_0] + 1 = SA_0[i_1]$, $SA_0[i_1] + 1 = SA_0[i_2]$, and so on. An analogous walk applies to $j_0 = j$, j_1, ..., j_{l-1}. In the same way, we use the function Φ_0 in the compressed version of SA^R to obtain $g_0 = g$, g_1, ..., g_{l-1} and $h_0 = h$, h_1, ..., h_{l-1}. We then run the tth stage, for $1 \leq t \leq l-1$, in which we perform a range query for the points in $[i_t \dots j_t] \times [g_{l-t} \dots h_{l-t}]$. For each of these points, we retrieve its label p and output $p - t$.

In order to see why the above method works, let us consider an arbitrary suffix pointer in $SA[i]$, ..., $SA[j]$. By the definition of the points kept in D, this suffix pointer can be written as $p - t$, where p is the nearest multiple of l and $0 \leq t \leq l-1$. We show that we output $p - t$ correctly in stage t. Let $\langle s, r \rangle$ be the point with label p in D. We have to show that $i_t \leq s \leq j_t$ and $g_{l-t} \leq r \leq h_{l-t}$ (setting border values $g_l = 1$ and $h_l = n$). Recall that the suffixes pointed to by $SA[i]$, $p - t$ and $SA[j]$ are in lexicographic order by definition of the (compressed) suffix array, and moreover they share the first l symbols. If we remove the first $t < l$ symbols from each of them, the lexicographic order must be preserved because these symbols are equal. Consequently, $SA[i] - t$, p, and $SA[j] - t$ are still in lexicographic order, and their ranks are i_h, s, and j_h, respectively. This implies $i_t \leq s \leq j_t$. A similar property holds for $g_{l-t} \leq r \leq h_{l-t}$, and we can conclude that p is retrieved in stage t giving $p - t$ as output. Finally, the fact that both i, $i+1$, ..., j and g, $g+1$, ..., h are maximal sequences in their respective suffix arrays, implies that no other suffix pointers besides those in $SA[i]$, ..., $SA[j]$ are reported.

The cost of each stage is $T(n')$ plus the output-sensitive cost of the reported suffix pointers. Stage 0 requires an additional cost of $O((n/n') \log |\Sigma| + \log^{1+\epsilon} n)$ to compute g and h, and a cost of $O(n/n')$ to precompute the four sequences of indices, because the length of the walks is l. The total time complexity is therefore $O\big((n/n')(T(n') + \log |\Sigma|) + \log^{1+\epsilon} n + j - i\big)$, where $O(j - i + 1)$ is the sum of the output-sensitive costs for reporting all the suffix pointers. □

We use Lemma 7 to prove Theorem 3. We employ two range query data structures for D. The first one in [ABR00] takes $P(q) = O(q \log q)$ preprocessing time by using the perfect hash in [HMP00], which has constant lookup time and takes $O(q \log q)$ construction time. Space is $S(q) = O(q \log^\epsilon q)$ words and query time is $T(q) = O(\log \log q)$. Plugging these bounds into Lemma 7 gives $n' = \Theta(n/\log^{1+\epsilon} n)$ and hence $O((\log^{1+\epsilon} n)(\log|\Sigma| + \log\log n) + j - i)$ retrieval time for suffix pointers sharing $\Omega(\log^{1+\epsilon} n)$ symbols. Preprocessing time is $O(n \log|\Sigma|)$ and additional space is $O(n \log|\Sigma|)$ bits.

The second data structure in [BM80, Wil86] has preprocessing time $P(q) = O(q \log q)$, space $S(q) = O(q)$, and query time $T(q) = O(q^\beta)$ for any fixed value of $0 < \beta < 1$. Consequently, Lemma 7 gives $n' = \Theta(n/\log n)$ and $O(n^\beta \log n + j - i) = O(n^\alpha + j - i)$ retrieval time, for suffix pointers sharing at least $\Omega(\log n)$ symbols (by choosing $\alpha > \beta$). Preprocessing time is $O(n \log|\Sigma|)$ and additional space is $O(n \log|\Sigma|)$ bits.

4 Text Indexing, String Searching, and Compressed Suffix Trees

We now describe how to apply our compressed suffix array to obtain a text index, called *compressed suffix tree*, which is very efficient in time and space complexity. We first show that, despite their extra functionality, compressed suffix trees (and compressed suffix arrays) require the same asymptotic space of $\Theta(n)$ bits as inverted lists in the worst case. Nevertheless, inverted lists are space efficient in practice [ZMR98] and can be easily maintained in a dynamic setting.

Lemma 8 *In the worst case, inverted lists require $\Theta(n)$ bits for a binary text of length n.*

Proof Sketch: Let us take a De Bruijn sequence S of length n, in which each substring of $\log n$ bits is different from the others. Now let the terms in the inverted lists be those obtained by partitioning S into $s = n/k$ 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 terms, and so it requires at least $\log \binom{2^k}{s} = \Omega(n)$ bits by a simple information-theoretic argument. The upper bound $O(n)$ follows from Theorem 1 and Theorem 4 below, since we can see compressed suffix arrays and suffix trees as a generalization of inverted lists. $\square$

We now describe our main result on text indexing for constant size alphabets. Here, we are given a pattern string P of m symbols over the alphabet Σ, and we are interested in its occurrences (perhaps overlapping) in a text string T of n symbols (where # is the nth symbol). We assume that each symbol in Σ is encoded by $\log|\Sigma|$ bits, which is the case with ASCII and UNICODE text files where two or more symbols are packed in each word.

Theorem 4 *Given a text string T of length n over an alphabet Σ of constant size, we can build a full text index on T in $O(n \log|\Sigma|)$ time such that the index occupies roughly $\big(\epsilon^{-1} + O(1)\big)\, n \log|\Sigma|$ bits, for any fixed value of $0 < \epsilon \le 1$, and supports the following queries on any pattern string P of m symbols packed into $O(m/\log_{|\Sigma|} n)$ words:*

i. Existential and counting queries can be done in $o\big(\min\{m \log|\Sigma|, m + \log n\}\big)$ time; in particular, they take $O(1)$ time for $m = o(\log n)$, and $O(m/\log_{|\Sigma|} n + \log^\epsilon_{|\Sigma|} n)$ time otherwise.

ii. An enumerative query listing the occ occurrences of P in T can be done in $O(m/\log_{|\Sigma|} n + occ \log^\epsilon_{|\Sigma|} n)$ time. We can use auxiliary data structures in $O(n \log|\Sigma|)$ bits to reduce the search bound to $O\big(m/\log_{|\Sigma|} n + occ + (\log^{1+\epsilon} n)(\log|\Sigma| + \log\log n)\big)$ time, when either $m = \Omega(\log^{1+\epsilon} n)$ or $occ = \Omega(n^\epsilon)$.

As a result, an enumerative query can be done in optimal $\Theta(m/\log_{|\Sigma|} n+occ)$ time for sufficiently large patterns or number of occurrences, namely, when $m = \Omega\big((\log^{2+\epsilon} n)\log_{|\Sigma|}\log n)\big)$ or $occ = \Omega(n^{\epsilon})$.

In order to prove Theorem 4, we first show how to speed up the search on compacted tries in Section 4.1. Then we present the index construction in Section 4.2. Finally, we give the description of the search algorithm in Section 4.3. Let's briefly review three important data structures presented in [KS98, Mor68, MRS01] and needed later on.

The first data structure is the Lempel-Ziv (LZ) index [KS98]. It 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 $m \le \epsilon \log n$ 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 $\Omega(n \log n)$ bits.) The LZ index allows us to concentrate on patterns of length $m > \epsilon \log n$.

The second data structure is the Patricia trie [Mor68], another powerful tool in text indexing. It is a binary tree that stores a set of distinct binary strings, in which each internal node has two children and each leaf stores a string. For our purposes, we can generalize it to handle alphabets of size $|\Sigma| \ge 2$ by using a $|\Sigma|$-way tree. Each internal node also keeps an integer (called skip value) to locate the position of the branching character while descending towards a leaf. Each child arc is implicitly labeled with one symbol of the alphabet. For space efficiency, when there are $t > 2$ child arcs, we can represent the child arcs by a hash table of $O(t)$ entries. In particular, we use a perfect hash function (e.g., see [FKS84, HMP00]) on keys from Σ, which provides constant lookup time and uses $O(t)$ words of space and $O(t \log t)$ construction time, in the worst case.

Suffix trees are often implemented by building a Patricia trie on the suffixes of T as follows [GBS92]: First, text T is encoded as a binary sequence of $n \log |\Sigma|$ bits, and its n suffixes are encoded analogously. Second, a Patricia trie is built upon these suffixes; the resulting suffix tree has still n leaves (not $n \log |\Sigma|$). Third, searching for P takes $O(m)$ time and retrieves only the suffix pointer in at most two leaves (i.e., the leaf reached by branching with the skip values, and the leaf corresponding to an occurrence). According to our terminology, it requires only $O(1)$ calls to the *lookup* operation in the worst case.

The third data structure is the space-efficient incarnation of binary Patricia tries in [MRS01], which builds upon previous work to succinctly represent binary trees and Patricia tries [CM96, Jac89a, Mun96, MR97]. When employed to store s out of the n suffixes of T, the regular Patricia trie [Mor68] occupies $O(s \log n)$ bits. This amount of space usage is the result of three separate factors [Cla96, CM96], namely, the Patricia trie topology, the skip values, and the string pointers. 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 [MRS01] 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 $O(m \log |\Sigma|)$ time and accesses $O(\min\{m \log |\Sigma|, s\}) = O(s)$ suffix pointers in the worst case. For each traversed node, its corresponding skip value is computed in time $O(\mathit{skip\_value})$ by accessing the suffix pointers in its leftmost and rightmost descendant leaves. In our terminology, 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 (maybe less if the pattern length is not a multiple of $\log n$).

Let us first consider the $|\Sigma|$-way Patricia trie PT outlined in Section 4 for storing s binary strings, each of length at least $\log n$. (For example, they could be some suffixes of the text.) To handle border situations, we assume that these strings are (implicitly) padded with $\log_{|\Sigma|} n$ symbols #. We will show how to reduce the search time for an m-symbol pattern in PT from $O(m \log |\Sigma|)$ to $O(m/\log_{|\Sigma|} n + \log^{\epsilon}_{|\Sigma|} n)$. Without loss of generality, it suffices to show how to achieve $O(m/\log_{|\Sigma|} n + \sqrt{\log_{|\Sigma|} n}\,)$ time, since this bound extends from $1/2$ to any exponent $\epsilon > 0$. The point is that, in the worst case, we may have to traverse $\Theta(m)$ nodes, so we need a tool to skip most of these nodes. Ideally, we would like to branch downward matching $\log n$ bits (or equivalently, $\log_{|\Sigma|} n$ symbols) in constant time, independently of the number of traversed nodes. For that purpose, we use a perfect hash function h (e.g., see [FKS84]) on keys each of length at most $2 \log n$ bits. In particular, we use the perfect hash function in [HMP00], which has constant lookup time and takes $O(k)$ words of space and $O(k \log k)$ construction time on k keys, in the worst case.

First of all, we enumerate the nodes of PT in preorder starting from the root, with number 1. Second, we build hash tables to mimic a downward traversal from a given node i, which is the starting point for searching strings x of length less than or equal to $\log_{|\Sigma|} n$ symbols. Suppose that, in this traversal, we successfully match all the symbols in x and we reach node j (a descendent of i). In general, there can be further symbols to be added to equal the skip value in j; let $b \geq 0$ be this number of symbols. We represent the successful traversal in a single entry of the hash table. Namely, we store pair $\langle j, b\rangle$ at position $h(i, x)$, where the two arguments i and x can be seen as a single key of at most $2 \log n$ bits. Formally, the relation between these parameters must satisfy two conditions in case of successful search of x from node i:

1. Node j is the node identified by starting out from node i and traversing downward PT according to the symbols in x;
2. b is the unique nonnegative 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 proper ancestor of j.

The rationale behind conditions 1–2 is that of defining shortcut links from certain nodes i to their descendents j, so that each successful branching takes constant time, matches $|x|$ symbols (with b further symbols to check) and skips no more than $|x|$ nodes downward. If the search is unsuccessful, we do not hash any pair.

The key mechanism that makes the above scheme efficient is that we adaptively follow the trie topology of Patricia, so that the strings that we hash are not all possible substrings of $\log_{|\Sigma|} n$ (or $\sqrt{\log_{|\Sigma|} n}\,$) symbols, but only a subset of those that start at the distinct nodes in the Patricia trie. Using an uncompacted trie would make this method inefficient. To see why, let us examine a Patricia edge corresponding to a substring of length l. We hash only its first $\log_{|\Sigma|} n$ (or $\sqrt{\log_{|\Sigma|} n}\,$) symbols because the rest of the symbols are uniquely identified (and we can skip them). Using an uncompacted trie would force us to traverse further $b = l - \log_{|\Sigma|} n$ (or $b = l - \sqrt{\log_{|\Sigma|} n}\,$) nodes.

In order to keep small the number of shortcut links, we set up two hash tables H_1 and H_2. The first table stores entries

$$H_1\big[h(i, x)\big] = \langle j, b\rangle$$

such that all strings x consist of $|x| = \log_{|\Sigma|} n$ symbols, and the shortcut links stored in H_1 are selected adaptively by a top-down traversal of PT. Namely, we create all possible shortcut links from the root. This step links the root to a set of descendents. We recursively link each of these nodes to its descendents in the same fashion. Note that PT is partitioned into subtries of depth at most $\log_{|\Sigma|} n$.

We set up the second table H_2 analogously. We examine each individual subtrie, and start from the root of the subtrie by using strings of length $|x| = \sqrt{\log_{|\Sigma|} n}$ symbols. Note that the total number of entries in H_1 and H_2 is bounded by the number of nodes in PT, namely, $O(s)$.

In summary, the preprocessing consists in a double traversal of PT followed by the construction of H_1 and H_2, in $O(s \log s + n)$ worst-case time and $O(s)$ words of space. In the general case, we go on recursively and build ϵ^{-1} hash tables whose total number of entries is still $O(s)$. The preprocessing time does not change asymptotically.

We are now ready to describe the search of a pattern (encoded in binary) in the Patricia trie PT thus augmented. It suffices to show how to match its longest prefix. We compute hash function $h(i, x)$ with i being the root of PT and x being the concatenation of the first $\log_{|\Sigma|} n$ symbols in the pattern. Then we branch quickly from the root by using $H_1[h(i,x)]$. If the hash lookup in H_1 succeeds and gives pair $\langle j, b\rangle$, we skip the next b symbols in the pattern and recursively search in node j with the next $\log_{|\Sigma|} n$ symbols in the pattern. Instead, if the hash lookup fails (i.e., no pair found or fewer than $\log_{|\Sigma|} n$ symbols left in the pattern), we switch to H_2 and take only the next $\sqrt{\log_{|\Sigma|} n}$ symbols in the pattern to branch further in PT. Here the scheme is the same as that of H_1, except that we compare $\sqrt{\log_{|\Sigma|} n}$ symbols at a time. Finally, when we fail branching again, we have to match no more than $\sqrt{\log_{|\Sigma|} n}$ symbols remaining in the pattern. We complete this task by branching in the standard way, one symbol a time. The rest of the search is identical to the standard procedure of Patricia tries. This completes the description of the search in PT.

Lemma 9 *Given a Patricia trie PT storing s strings of at least $\log_{|\Sigma|} n$ symbols each over the alphabet Σ, we can preprocess PT in $O(s \log s + n)$ time, so that searching a pattern of length m requires $O(m/\log_{|\Sigma|} n + \log^{\epsilon}_{|\Sigma|} n)$ time.*

Note that a better search bound in Lemma 9 does not improve the final search time obtained in Theorem 4.

Finally, let us consider a space-efficient Patricia trie [MRS01]. The speedup we need while searching is easier to obtain. 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 $O(m/\log_{|\Sigma|} n)$ plus a linear cost proportional to the number of traversed nodes.

A general property of our speedup of Patricia tries is that we do not increase the original number of *lookup* calls originating from the data structures.

4.2 Index Construction

We blend the tools mentioned so far with our compressed suffix arrays of Section 3 to design an index data structure, called the *compressed suffix tree*, which follows the multilevel scheme adopted in [CD96, MRS01]. Because of the LZ index, it suffices to describe how to support searching of patterns of length $m > \epsilon \log n$. We assume that $0 < \epsilon \leq 1/2$ as the case $1/2 < \epsilon \leq 1$ requires minor modifications.

Given text T in input, we build its suffix array SA in a temporary area, in $O(n \log |\Sigma|)$ time via the suffix tree of T. At this point, we start building the $O(\epsilon^{-1})$ levels of the compressed suffix tree in top-down order, after which we remove SA:

1. At the first level, we build a regular Patricia trie PT^1 augmented with the shortcut links as mentioned in Lemma 9. The leaves of PT^1 store the $s_1 = n/\log_{|\Sigma|} n$ suffixes pointed to by $SA[1]$, $SA[1 + \log_{|\Sigma|} n]$, $SA[1 + 2\log_{|\Sigma|} n]$, ... This implicitly splits SA into s_1 subarrays of size $\log_{|\Sigma|} n$, except the last one (which can be smaller).

   Complexity: The size of PT^1 is $O(s_1 \log n) = O(n \log |\Sigma|)$ bits. It can be built in $O(n \log |\Sigma|)$ time by a variation of the standard suffix tree construction [KD95, KD96] and the preprocessing described in Lemma 9.

2. At the second level, we process the s_1 subarrays at the first level, and create s_1 space-efficient Patricia tries [MRS01], denoted $PT^2_1, PT^2_2, \ldots, PT^2_{s_1}$. We associate the ith Patricia PT^2_i with the ith subarray. Assume without loss of generality that the subarray consists of $SA[h+1]$, $SA[h+2]$, ..., $SA[h + \log_{|\Sigma|} n]$ for a value of $0 \leq h \leq n - \log_{|\Sigma|} n$. We build PT^2_i upon the $s_2 = \log^{\epsilon/2}_{|\Sigma|} n$ suffixes pointed to by $SA[h+1]$, $SA[h+1+\log^{1-\epsilon/2}_{|\Sigma|} n]$, $SA[h+1+2\log^{1-\epsilon/2}_{|\Sigma|} n]$, ... This process splits each subarray into smaller subarrays, each of size $\log^{1-\epsilon/2}_{|\Sigma|} n$.

   Complexity: The size of each PT^2_i is $O(s_2)$ bits without accounting for the suffix array, and its construction takes $O(s_2)$ time [MRS01]. Hence, the total size is $O(s_1 s_2) = O(n/\log^{1-\epsilon}_{|\Sigma|} n)$ bits and the total processing time is $O(n \log |\Sigma|)$.

3. In the remaining $2\epsilon^{-1} - 2$ intermediate levels, we go on like the second level. Each new level splits every subarray into $s_2 = \log^{\epsilon/2}_{|\Sigma|} n$ smaller subarrays and creates a set of space efficient Patricia tries of size $O(s_2)$ each. We stop when we are left with small subarrays of size at most s_2. We build space efficient Patricia tries on all the remaining entries of these small subarrays.

   Complexity: For each new level thus created, the total size is $O(n/\log^{\epsilon}_{|\Sigma|} n)$ bits and the total processing time is $O(n \log |\Sigma|)$.

4. At the last level, we execute *compress* on the suffix array SA, store its compressed version in the level, and delete SA from the temporary area.

   Complexity: By Theorem 2, the total size is $\big(\epsilon^{-1} + O(1)\big) n \log |\Sigma|$ bits; accessing a pointer through a call to *lookup* takes $O(\log^{\epsilon/2}_{|\Sigma|} n)$ time; the cost of *compress* is $O(n \log |\Sigma|)$ time. (Note that we can fix the value of ϵ arbitrarily when executing *compress*.)

By summing over the levels, we obtain that the compressed suffix tree of T takes $O(n \log |\Sigma|)$ bits and $O(n \log |\Sigma|)$ construction time.

4.3 Search Algorithm

We now have to show that searching for an arbitrary pattern P in the text T costs $O(m/\log_{|\Sigma|} n + \log^{\epsilon}_{|\Sigma|} n)$ time. The search locates the leftmost occurrence and the rightmost occurrence of P as a prefix of the suffixes represented in SA, without having SA stored explicitly. Consequently, a successful search determines two positions $i \leq j$, such that the sequence $i, i+1, \ldots, j$ is maximal (according to the definition given before Theorem 3) and $SA[i]$, $SA[i+1]$, ..., $SA[j]$ contain 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*$(i+1)$, ..., *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 at each level of the compressed suffix tree in Section 4.2. We examine the levels in a top-down manner. While searching in the levels, we execute *lookup*(i) whenever we need the ith pointer of the compressed SA. We begin by searching P at the first level. We perform the search on PT^1 in the bounds stated in Lemma 9. As a result of the first search, we locate a subarray at the second level, say, the i_1th subarray. We go on and search in $PT^2_{i_1}$ according to the method for space-efficient Patricia tries described at the end of Section 4.1. We repeat the latter search for all the intermediate levels. We eventually identify a position at the last level, namely, the level which contains the compressed suffix array. This position corresponds to the leftmost occurrence of P in SA.

The complexity of the search procedure is $O(m/\log_{|\Sigma|} n + \log^{\epsilon}_{|\Sigma|} n)$ time at the first level by Lemma 9. The intermediate levels cost $O(m/\log_{|\Sigma|} n + s_2)$ time each, giving a total of $O(m/\log_{|\Sigma|} n + \log^{\epsilon}_{|\Sigma|} n)$. We have to account for the cost of the *lookup* operations. These calls originated from the several levels. In the first level, we call *lookup* $O(1)$ times; in the $2\epsilon^{-1} - 1$ intermediate levels we call *lookup* $O(s_2)$ times each. Multiplying these calls by the $O(\log^{\epsilon/2}_{|\Sigma|} n)$ cost of *lookup* as given in Theorem 1 (using $\epsilon/2$ in place of ϵ), we obtain $O(\log^{\epsilon}_{|\Sigma|} n)$ time in addition to $O(m/\log_{|\Sigma|} n + \log^{\epsilon}_{|\Sigma|} n)$. Finally, the cost of retrieving all the occurrences is the one stated in Theorem 3, whose hypothesis is satisfied because the suffixes pointed to by $SA[i]$ and $SA[j]$ are respectively the leftmost and the rightmost sharing $m = \Omega(\log n)$ symbols. Combining this cost with the $O(\log^{\epsilon}_{|\Sigma|} n)$ cost for retrieving any single pointer in Theorem 1, we obtain $O(m/\log_{|\Sigma|} n + occ \log^{\epsilon}_{|\Sigma|} n)$ time when both conditions $m \in \left[\epsilon \log n, o(\log^{1+\epsilon} n)\right]$ and $occ = o(n^{\epsilon})$ hold, and in $O\big(m/\log_{|\Sigma|} n + occ + (\log^{1+\epsilon} n)(\log|\Sigma| + \log\log n)\big)$ time otherwise. This argument completes the proof of Theorem 4 on the complexity of our text index.

5 Conclusions

We have presented the first indexing data structure for a text T of n symbols over alphabet Σ that achieves, in the worst case, $o\big(\min\{m \log|\Sigma|, m + \log n\}\big)$ search time and roughly $\big(\epsilon^{-1} + O(1)\big)\, n \log|\Sigma|$ bits of space. Our method is based upon notions of compressed suffix arrays and suffix trees and, for any fixed constant $0 < \epsilon \le 1$, uses about $\epsilon^{-1} n \log|\Sigma|$ bits to index text string T (which requires $n \log|\Sigma|$ bits). Given any pattern P of m symbols encoded in $m \log|\Sigma|$ bits, we can count the number of occurrences of P in T in $o\big(\min\{m \log|\Sigma|, m + \log n\}\big)$ time. Namely, searching takes $O(1)$ time when $m = o(\log n)$, and $O(m/\log_{|\Sigma|} n + \log^{\epsilon}_{|\Sigma|} n)$ time otherwise. We achieve optimal $O(m/\log_{|\Sigma|} n)$ search time for sufficiently large $m = \Omega(\log^{1+\epsilon}_{|\Sigma|} n)$. For an enumerative query retrieving all *occ* occurrences with sufficiently long patterns, namely, $m = \Omega\big((\log^{2+\epsilon} n) \log_{|\Sigma|} \log n\big)$, we obtain a total search bound of $O(m/\log_{|\Sigma|} n + occ)$, which is optimal. Namely, searching takes $O(m/\log_{|\Sigma|} n + occ \log^{\epsilon}_{|\Sigma|} n)$ time when both conditions $m \in \left[\epsilon \log n, o(\log^{1+\epsilon} n)\right]$ and $occ = o(n^{\epsilon})$ hold, and $O\big(m/\log_{|\Sigma|} n + occ + (\log^{1+\epsilon} n)(\log|\Sigma| + \log\log n)\big)$ time otherwise.

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 $O(occ)$ also when $m \in \left[\epsilon \log n, o(\log^{1+\epsilon} n)\right]$ and $occ = o(n^{\epsilon})$. Another possibility for that is to devise a range query data structure that improves the data structures at the end of Section 3.4. This, in turn, would improve Theorem 3 and Theorem 4. 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. Additional open problems are listed in [MRS01]. 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 [AKL+00, BG96, GBS92, YY97].

References

[ABR00] Stephen Alstrup, Gerth S. Brodal, and Theis Rauhe. New data structures for orthogonal range searching. In *Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science*, pages 198–207, 2000.

[AIL+88] Alberto Apostolico, Costas Iliopoulos, Gadi M. Landau, Baruch Schieber, and Uzi Vishkin. Parallel construction of a suffix tree with applications. *Algorithmica*, 3:347–365, 1988.

[AKL+00] Amihood Amir, Dmitry Keselman, Gadi M. Landau, Moshe Lewenstein, Noa Lewenstein, and Michael Rodeh. Text indexing and dictionary matching with one error. *J. Algorithms*, 37(2):309–325, 2000.

[ALS99] Arne Andersson, N. Jesper Larsson, and Kurt Swanson. Suffix trees on words. *Algorithmica*, 23(3):246–260, 1999.

[AN95] Arne Andersson and Stefan Nilsson. Efficient implementation of suffix trees. *Software Practice and Experience*, 25(2):129–141, February 1995.

[Apo85] Alberto 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.

[BBH+85] Anselm Blumer, Janet Blumer, David Haussler, Andrzej Ehrenfeucht, M.T. Chen, and Joel Seiferas. The smallest automation recognizing the subwords of a text. *Theoretical Computer Science*, 40(1):31–55, September 1985.

[BBH+87] Anselm Blumer, Janet Blumer, David Haussler, Ross McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, July 1987.

[BG96] Gerth S. Brodal and Leszek Gąsieniec. Approximate dictionary queries. In Daniel S. Hirschberg and Eugene 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.

[BM80] Jon L. Bentley and Hermann A. Maurer. Efficient worst-case data structures for range searching. *Acta Informatica*, 13:155–168, 1980.

[BM99] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM Journal on Computing*, 28(5):1627–1640, October 1999.

[CD96] Livio Colussi and Alessia De Col. A time and space efficient data structure for string searching on large texts. *Information Processing Letters*, 58(5):217–222, October 1996.

[Cla96] David Clark. Compact Pat trees. PhD Thesis, Department of Computer Science, University of Waterloo, 1996.

[CM96] David R. Clark and J. Ian 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 January 1996.

[CP91] Maxime Crochemore and Dominique Perrin. Two–way string matching. *Journal of the ACM*, 38:651–675, 1991.

[CR94] Maxime Crochemore and Wojciech Rytter. *Text Algorithms.* Oxford University Press, 1994.

[Cro86] Maxime Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45(1):63–86, 1986.

[DO01] Erik D. Demaine and Alejandro López Ortiz. A linear lower bound on index size for text retrieval. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 289–294. ACM/SIAM, 2001.

[Eli74] Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21(2):246–260, April 1974.

[FC97] Martin Farach-Colton. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science*, pages 137–143, Miami Beach, Florida, 20–22 October 1997.

[FCFM00] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000.

[FCM96] Martin Farach-Colton and S. Muthukrishnan. Optimal logarithmic time randomized suffix tree construction. In Friedhelm Meyer auf der Heide and Burkhard 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.

[FG99] Paolo Ferragina and Roberto 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, March 1999.

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.

[FM00] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science*, pages 390–398, 2000.

[FM01] Paolo Ferragina and Giovanni Manzini. An experimental study of an opportunistic index. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 269–278. ACM/SIAM, 2001.

[GBS92] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim 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.

[GKS99] Robert Giegerich, Stefan Kurtz, and Jens 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.

[GS83] Zvi Galil and Joel Seiferas. Time-space-optimal string matching. *Journal of Computer and System Sciences*, 26:280–294, 1983.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[GV00] Roberto Grossi and Jeffrey S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proceedings of the Thirdy-Second Annual ACM Symposium on the Theory of Computing*, pages 397–406, Portland, OR, 2000.

[HMP00] Torben Hagerup, Peter Bro Miltersen, and Rasmus Pagh. Deterministic dictionaries. Submitted manuscript (`http://www.daimi.au.dk/~bromille/Papers/det-jour.pdf`), 2000.

[Irv95] Robert W. Irving. Suffix binary search trees. Technical Report TR-1995-7, Computing Science Department, University of Glasgow, 1995.

[Jac89a] Guy Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 549–554, 1989.

[Jac89b] Guy Jacobson. Succinct static data structures. Technical Report CMU-CS-89-112, Dept. of Computer Science, Carnegie-Mellon University, January 1989.

[Kär95] Juha 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.

[KD95] S. Rao Kosaraju and Arthur L. Delcher. Large-scale assembly of DNA strings and space-efficient construction of suffix trees. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing*, pages 169–177, Las Vegas, Nevada, 29 May–1 June 1995.

[KD96] S. Rao Kosaraju and Arthur L. Delcher. Correction: Large-scale assembly of DNA strings and space-efficient construction of suffix trees. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*, page 659, Philadelphia, Pennsylvania, 22–24 May 1996.

[KMP77] Donald E. Knuth, J. H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.

[Knu98] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, second edition, 1998.

[KS98] Juha Kärkkäinen and Erkki Sutinen. Lempel-Ziv index for q-grams. *Algorithmica*, 21(1):137–154, 1998.

[KU96a] Juha Kärkkäinen and Esko 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.

[KU96b] Juha Kärkkäinen and Esko Ukkonen. Sparse suffix trees. *Lecture Notes in Computer Science*, 1090:219–230, 1996.

[Kur99] Stefan Kurtz. Reducing the Space Requirement of Suffix Trees. *Software – Practice and Experience*, 29(13):1149–1171, 1999.

[Mäk00] Veli Mäkinen. Compact suffix array. In *Combinatorial Pattern Matching*, volume 1848 of *Lecture Notes in Computer Science*, pages 305–319. Springer, 2000.

[McC76] Eduard M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[MM93] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[Mor68] Donald R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded In Alphanumeric. *Journal of the ACM*, 15(4):514–534, October 1968.

[MR97] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *38th Annual Symposium on Foundations of Computer Science*, pages 118–126, 1997.

[MRS01] J. Ian Munro, Venkatesh Raman, and S. Srinivasa Rao. Space efficient suffix trees. *Journal of Algorithms*, 39:205–222, 2001.

[Mun96] J. Ian Munro. Tables. *FSTTCS: Foundations of Software Technology and Theoretical Computer Science*, 16:37–42, 1996.

[MW94] Udi Manber and Sun Wu. GLIMPSE: A tool to search through entire file systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 23–32, 1994.

[MZ96] Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, October 1996.

[Pag01] Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31:353–363, 2001.

[Sad00] Kunihiko Sadakane. Compressed Text Databases with Efficient Query Algorithms based on the Compressed Suffix Array. In *Proceedings of ISAAC'00*, number 1969 in LNCS, pages 410–421, 2000.

[Sad02] Kunihiko Sadakane. Succinct representations of *lcp* information and improvements in the compressed suffix arrays. In *Proceedings of the Thitheenth Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM/SIAM, to appear, 2002.

[SNZ97] Edleno Silva de Moura, Gonzalo Navarro, and Nivio Ziviani. Indexing compressed text. In *Proc. of the South American Workshop on String Processing*, pages 95–111. Carleton University Press, 1997.

[SV94] Suleyman Cenk Sahinalp and Uzi 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.

[Ukk95] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, September 1995.

[Wei73] Peter Weiner. Linear pattern matching algorithm. *Proc. 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[Wil86] Dan 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.

[YY97] Andrew C. Yao and Frances F. Yao. Dictionary look-up with one error. *Journal of Algorithms*, 25(1):194–202, October 1997.

[ZMR98] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490, December 1998.

[ZSNBY00] Nivio Ziviani, Edleno Silva de Moura, Gonzalo Navarro, and Ricardo Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEEE Computer*, 33(11):37–44, November 2000.