SIAM J. COMPUT.
Vol. 35, No. 2, pp. 378–407

COMPRESSED SUFFIX ARRAYS AND SUFFIX TREES WITH APPLICATIONS TO TEXT INDEXING AND STRING MATCHING*

ROBERTO GROSSI† AND JEFFREY SCOTT VITTER‡

Abstract. The proliferation of online text, such as found on the World Wide Web and in online databases, motivates the need for space-efficient text indexing methods that support fast string searching. We model this scenario as follows: Consider a text T consisting of n symbols drawn from a fixed alphabet Σ. The text T can be represented in $n \lg |\Sigma|$ bits by encoding each symbol with $\lg |\Sigma|$ bits. The goal is to support fast online queries for searching any string pattern P of m symbols, with T being fully scanned only once, namely, when the index is created at preprocessing time.

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

We present a new text index that is based upon compressed representations of suffix arrays and suffix trees. It achieves a fast $O(m/\lg_{|\Sigma|} n + \lg^{\epsilon}_{|\Sigma|} n)$ search time in the worst case, for any constant $0 < \epsilon \leq 1$, using at most $\left(\epsilon^{-1} + O(1)\right) n \lg |\Sigma|$ bits of storage. Our result thus presents for the first time an efficient index whose size is provably linear in the size of the text in the worst case, and for many scenarios, the space is actually sublinear in practice. As a concrete example, the compressed suffix array for a typical 100 MB ASCII file can require 30–40 MB or less, while the raw suffix array requires 500 MB. Our theoretical 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 $O(occ \lg^{\epsilon}_{|\Sigma|} n)$ time as a result. When the patterns are sufficiently long, we can use auxiliary data structures in $O(n \lg |\Sigma|)$ bits to obtain a total search bound of $O(m/\lg_{|\Sigma|} n + occ)$ time, which is optimal.

Key words. compression, text indexing, text retrieval, compressed data structures, suffix arrays, suffix trees, string searching, pattern matching

AMS subject classifications. 68W05, 68Q25, 68P05, 68P10, 68P30

DOI. 10.1137/S0097539702402354

1. Introduction.

A great deal of textual information is available in electronic form in online databases and on the World Wide Web, and therefore devising efficient text indexing methods to support fast string searching is an important topic for investigation. A typical search scenario involves string matching in a text string T of length n [49]: given an input pattern string P of length m, the goal is to find occurrences of P in T. Each symbol in P and T belongs to a fixed alphabet Σ of size $|\Sigma| \leq n$. An occurrence of the pattern at position i means that the substring $T[i, i + m - 1]$ is equal to P, where $T[i, j]$ denotes the concatenation of the symbols

*Received by the editors February 9, 2002; accepted for publication (in revised form) May 2, 2005; published electronically October 17, 2005. A preliminary version of these results appears in [37].

http://www.siam.org/journals/sicomp/35-2/40235.html

†Dipartimento di Informatica, Università di Pisa, Largo B. Pontecorvo 3, 56127 Pisa, Italy (grossi@di.unipi.it). This author's work was supported in part by the United Nations Educational, Scientific and Cultural Organization (UNESCO) and by the Italian Ministry of Research and Education (MIUR).

‡Department of Computer Sciences, Purdue University, West Lafayette, IN 47907–2066 (jsv@purdue.edu). Part of this work was done while the author was at Duke University and on sabbatical at INRIA in Sophia Antipolis, France. It was supported in part by Army Research Office MURI grants DAAH04–96–1–0013, DAAD19–01–1–0725, and DAAD19–03–1–0321 and by National Science Foundation research grants CCR–9877133 and IIS–0415097.

in T at positions $i, i+1, \ldots, j$.

In this paper, we consider three types of string matching queries: existential, counting, and enumerative. An existential query returns a Boolean value that indicates whether 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 offline string matching algorithms, such as that of Knuth, Morris, and Pratt [49], can answer each individual query in $O(m+n)$ time via an efficient text scan.

The large mass of existing online 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, at preprocessing time when the indexes are created. After that, searching is output-sensitive, that is, the time complexity of each online query is proportional to either $O(m \lg |\Sigma| + occ)$ or $O(m + \lg n + occ)$, which is much less than $\Theta(m+n)$ when n is sufficiently large.

The most popular indexes currently in use are inverted lists and signature files [48]. Inverted lists are theoretically and practically superior to signature files [72]. 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 words) in which T is partitioned into nonoverlapping 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 list (or position list) $\{i_k\}$ of the occurrences of that term in T. As a result, in order to search efficiently, search queries must be limited to terms or prefixes of them; it does not allow for efficient searching of 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. 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 as many as $\binom{n}{2} = \Theta(n^2)$ distinct substrings in the worst case, while the number of distinct terms is at most n (considered as nonoverlapping substrings). Suffix arrays [55, 35], suffix trees [57, 68], and similar tries or automata [20] are among the prominent data structures 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 n leaves represent the n text suffixes $T[1, n]$, $T[2, n]$, $\ldots$,$T[n, n]$. By "compact" we mean that each internal node has at least two children. Each edge in the tree is labeled with one or more symbols for purposes of search navigation. The leaf with value ℓ represents the suffix $T[\ell, n]$. The leaf values in an in-order traversal of the tree represent the n suffixes of T in lexicographic order. An example suffix tree appears in Figure 1.

A *suffix array* $SA = SA[1, n]$ for text $T = T[1, n]$ consists of the values of the leaves of the suffix tree in in-order, but without the tree structure information. In other words, $SA[i] = \ell$ means that $T[\ell, n]$ is the ith smallest suffix of T in lexicographic order. The suffix array corresponding to the suffix tree of Figure 1 appears in Figure 2.

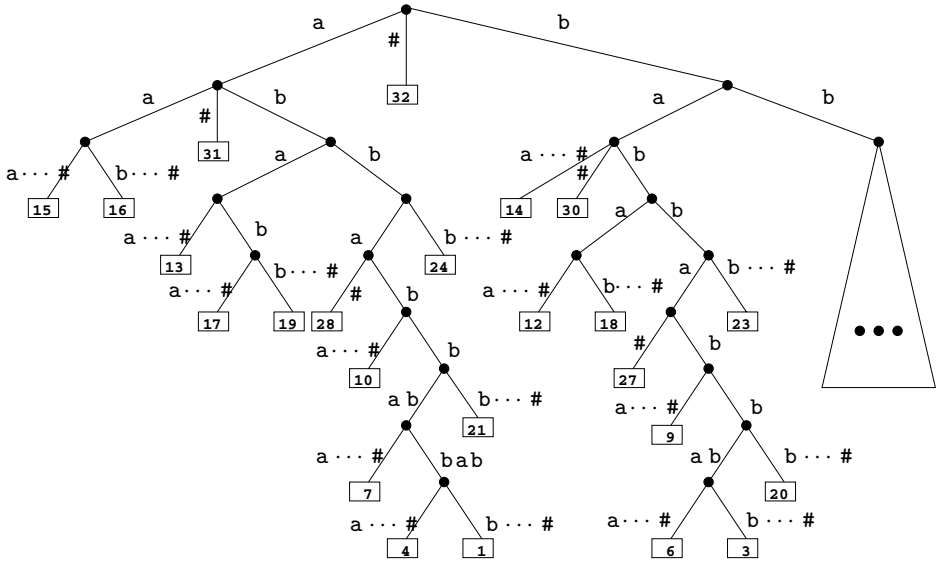

FIG. 1. *Suffix tree built on text* $T =$ `abbabbabbabbabaaababababbabbbabba#` *of length* $n = 32$*, where the last character is an end-of-string symbol* `#`*. The rightmost subtree (the triangle representing the suffixes of the form* `bb···#`*) is not expanded in the figure. The edge label* `a···#` *or* `b···#` *on the edge leading to the leaf with value* ℓ *denotes the remaining characters of the suffix* $T[\ell, n]$ *that have not already been traversed. For example, the first suffix in lexicographic format is the suffix* $T[15, n]$*, namely,* `aaababababbbabba#`*, and the last edge represents the* 16*-symbol substring that follows the prefix* `aa`*.*

To speed up searches, a separate array is often maintained, which contains auxiliary information such as the lengths of the longest common prefixes of a subset of the suffixes [55].

Suffix trees and suffix arrays organize the suffixes so as to support the efficient search of their prefixes. Given a search pattern P, in order to find an occurrence $T[i, i + m - 1] = P$, we can exploit the property that P must be the prefix of suffix $T[i, n]$. In general, existential and counting queries take $O(m \lg |\Sigma|)$ time using automata or suffix trees and their variations, and they take $O(m + \lg n)$ 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, $1, 2, \ldots, n$, but without the longest common prefix information mentioned above. Full-text indexes such as suffix arrays are more powerful than term-level inverted lists, since full-text indexes can also implement inverted lists efficiently by storing the suffixes $T[i_k, n]$ that correspond to the occurrences of the terms.

1.1. Issues on space efficiency. Suffix arrays and suffix trees are data structures with increasing importance because of the growing list of their applications. Besides string searching, they also have significant use in molecular biology, data compression, data mining, and text retrieval, to name but a few applications [7, 38, 55]. However, the sizes of the data sets in these applications can become extremely large, and space occupancy is often a critical issue. A major disadvantage that limits the applicability of text indexes based upon suffix arrays and suffix trees is that they occupy significantly more space than do inverted lists.

1	15	aaababababbbbabba#
2	16	aababababbbbabba#
3	31	a#
4	13	abaaababababbbbabba#
5	17	abababbabbbabba#
6	19	ababbabbbabba#
7	28	abba#
8	10	abbabaaababababbbbabba#
9	7	abbabbabaaababababbbbabba#
10	4	abbabbabbabaaababababbbbabba#
11	1	abbabbabbabbabaaababababbbbabba#
12	21	abbabbbabba#
13	24	abbbabba#
14	32	#
15	14	baaababababbbbabba#
16	30	ba#
17	12	babaaababababbbbabba#
18	18	bababbabbbabba#
19	27	babba#
20	9	babbabaaababababbbbabba#
21	6	babbabbabaaababababbbbabba#
22	3	babbabbabbabaaababababbbbabba#
23	20	babbabbbabba#
24	23	babbbabba#
⋮	⋮	⋮
32	25	bbbabba#

FIG. 2. *Suffix array for the text T shown in Figure* 1*, where* `a` $<$ `#` $<$ `b`. *Note that the array values correspond to the leaf values in the suffix tree in Figure* 1 *traversed in in-order.*

We can illustrate this point by a more careful accounting of the space requirements in the unit cost RAM model. We assume that each symbol in the text T is encoded by $\lg|\Sigma|$ bits, for a total of $n\lg|\Sigma|$ bits.[1] In suffix arrays, the positions of the n suffixes of T are stored as a permutation of $1, 2, \ldots, n$, using $n\lg n$ bits (kept in an array consisting of n words, each of $\lg n$ bits). Suffix trees require considerably more space: between $4n\lg n$ and $5n\lg n$ bits (stored in $4n$–$5n$ words) [55]. In contrast, inverted lists require only approximately 10% of the text size [58], and thus suffix arrays and suffix trees require significantly more bits. From a theoretical point of view, if the alphabet is very large, namely, if $\lg|\Sigma| = \Theta(\lg n)$, then suffix arrays require roughly the same number of bits as the text. However, in practice, the alphabet size $|\Sigma|$ is typically a fixed constant, such as $|\Sigma| = 256$ in electronic documents in ASCII or larger in UNICODE format, and $\Sigma = 4$ in DNA sequences. In such cases in practice, suffix arrays and suffix trees are larger than the text by a significant multiplicative factor of $\Theta(\lg_{|\Sigma|} n) = \Theta(\lg n)$. 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 do not support a general search on unstructured sequences.

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

In [62], Munro, Raman, and Rao solve the open question raised by Muthukrishnan by showing how to represent suffix trees in $n \lg n + O(n)$ bits while allowing $O(m)$-time search of binary pattern strings of length m. This result highlights the conceptual barrier of $n \lg n$ bits of storage needed for text indexing. In this paper, we go one step further and investigate whether it is possible to design a full-text index in $o(n \lg n)$ bits, while still supporting efficient search.

The question of space usage is important in both theory and practice. Prior to our work, the state of the art has taken for granted that at least $n \lg n$ bits are needed to represent the permutation of the text positions for any efficient full-text index. On the other hand, if we note that each text of n symbols is in one-to-one correspondence with a suffix array, then we can easily see by a simple information-theoretic argument that $\Omega(n \lg |\Sigma|)$ bits are required to represent the permutation. The argument is based upon the fact that there are $|\Sigma|^n$ different text strings of length n over the alphabet Σ; hence, there are that many different suffix arrays, and we need $\Omega(n \lg |\Sigma|)$ bits to distinguish them from one another. It is therefore an interesting problem to close this gap in order to see if there is an efficient representation of suffix arrays that use nearly $n \lg |\Sigma| + O(n)$ bits in the worst case, even for random strings.

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 $\lg n$ bits together into a word (in effect, constructing a large alphabet) and create a text of length $n/\lg n$ and a pattern of length $m/\lg n$. The suffix array on the new text requires $O((n/\lg n) \lg n) = O(n)$ bits. Searching for a pattern of length m must also consider situations when the pattern is not aligned at the precise word boundaries. What is the searching cost? It appears that we have to handle $\lg n$ situations, with a slowdown factor of $\lg n$ 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 from scratch, such as in [49]. 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 with the boundary by padding k bits to the left of the pattern. Since we do not know the correct k bits to prepend to the pattern, we must try all 2^k possible settings of the k bits. When $k \approx \lg n$, we have to query the index $2^k = \Omega(n)$ times in the worst case. (See the sparse suffix trees [47] cited in section 1.3 to partially alleviate 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 1.3, 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 in the literature to date use $\Theta(n \lg |\Sigma|) + o(n \lg n)$ bits and support fast queries in $o(m \lg |\Sigma|)$ or $o(m + \lg n)$ worst-case time. Our goal in this paper is to simultaneously reduce *both* the space bound *and* the query time bound.

1.2. Our results. In this paper, we begin the study of the compressibility of suffix arrays and related full-text indexes. 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, as shown in Figure 2. We represent SA in the form of a permutation of the starting positions, $1, 2, \ldots, n$, of the

suffixes in T. For all $1 \le i < j \le n$, we have $T\big[SA[i], n\big] < T\big[SA[j], n\big]$ in lexicographic order. We call each entry in SA a *suffix pointer*.

Given a text T and its suffix array SA, we consider the problem of obtaining a *compressed suffix array* from both T and SA so as to support the following two basic operations:

1. *compress*(T, SA): Compress SA to obtain its succinct representation. After that, text T is retained while SA can be discarded.

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

(More functionalities are introduced in [64].) 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 and space taken by *compress*.

In this paper, we exploit the "implicit structure" underlying the permutation of the suffix pointers stored in SA, which takes advantage of the fact that not all permutations are valid suffix arrays. For any fixed value of $0 < \epsilon \le 1$, we show how to implement operation *compress* in $(1 + \epsilon^{-1})\, n \lg |\Sigma| + o(n \lg |\Sigma|)$ bits so that each call to *lookup* takes sublogarithmic worst-case time, that is, $O(\lg^{\epsilon}_{|\Sigma|} n)$ time. We can also achieve $(1 + \frac{1}{2} \lg \lg_{|\Sigma|} n)\, n \lg |\Sigma| + O(n)$ bits so that calls to *lookup* can be done in $O(\lg \lg_{|\Sigma|} n)$ time. The preprocessing time is $O(n \lg |\Sigma|)$. Note that the auxiliary space during preprocessing is larger, i.e., $O(n \lg n)$ bits, since our preprocessing requires the suffix array in uncompressed form to output it in compressed form. Our findings have several implications as follows:

1. When $|\Sigma| = O\big(2^{o(\lg n)}\big)$, we break the space barrier of $\Omega(n \lg n)$ bits for a suffix array while retaining $o(\lg n)$ lookup time in the worst case. We refer the reader to the literature described in section 1.3.

2. We can implement a form of compressed suffix trees in $2n \lg |\Sigma| + O(n)$ bits by using compressed suffix arrays (with $\epsilon = 1$) and the techniques for compact representation of Patricia tries presented in [62]. They occupy asymptotically up to a small constant factor the *same* space as that of the text string being indexed.

3. 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.

4. We can build a hybrid full-text index on T in at most $\big(\epsilon^{-1} + O(1)\big)\, n \lg |\Sigma|$ bits by a suitable combination of our compressed suffix trees and previous techniques [17, 45, 62, 59]. We can answer existential and counting queries of any pattern string of length m in $O(m/\lg_{|\Sigma|} n + \lg^{\epsilon}_{|\Sigma|} n)$ search time in the worst case, which is $o\big(\min\{m \lg |\Sigma|, m + \lg n\}\big)$, smaller than previous search bounds. For enumerative queries, we introduce a sublogarithmic slowdown factor in the output-sensitive cost, giving $O(occ \lg^{\epsilon}_{|\Sigma|} n)$ time as a result. When the patterns are sufficiently long, namely, for $m = \Omega\big((\lg^{2+\epsilon} n)(\lg_{|\Sigma|} \lg n)\big)$, we can use auxiliary data structures in $O(n \lg |\Sigma|)$ bits to obtain a total search bound of $O(m/\lg_{|\Sigma|} n + occ)$ time, which is optimal.

The bounds claimed in point 4 need further elaboration. Specifically, searching takes $O(1)$ time for $m = o(\lg n)$, and $O(m/\lg_{|\Sigma|} n + \lg^{\epsilon}_{|\Sigma|} n) = o(m \lg |\Sigma|)$ time otherwise. That is, we achieve optimal $O(m/\lg_{|\Sigma|} n)$ search time for sufficiently large $m = \Omega(\lg^{1+\epsilon}_{|\Sigma|} n)$. For enumerative queries, retrieving all occ occurrences has cost $O(m/\lg_{|\Sigma|} n + occ \lg^{\epsilon}_{|\Sigma|} n)$ when both conditions $m \in \big[\epsilon \lg n, o(\lg^{1+\epsilon} n)\big]$ and $occ = o(n^{\epsilon})$ hold, and cost $O\big(m/\lg_{|\Sigma|} n + occ + (\lg^{1+\epsilon} n)(\lg |\Sigma| + \lg \lg n)\big)$ otherwise.

The results described in this paper are theoretical, but they also have substantial practical value. The ideas described here, with the extensions described by Sadakane [64], have been tested experimentally in further work discussed in section 1.3. Central to our algorithms is the definition of function Φ and its implications described in section 2. This function is at the heart of many subsequent papers on compressed text indexing (such as suffix links in suffix trees, binary search trees in sorted dictionaries, and join operators in relational databases), thus is the major by-product of the findings presented in this paper. Ultimately Φ is related to sorting-based compression because it represents the inverse of the last-to-first mapping for the Burrows–Wheeler transform [14]. We refer the interested reader to section 1.3 for the state of the art in compressed text indexing and to section 2 for a discussion of the function Φ.

1.3. Related work. The seminal paper by Knuth, Morris, and Pratt [49] provides the first string matching solution taking $O(m+n)$ time and $O(m)$ words to scan the text. The space requirement was remarkably lowered to $O(1)$ words in [33, 19]. The new paradigm of compressed pattern matching was introduced in [2] and explored for efficiently scanning compressed texts in [3, 24]. When many queries are to be performed on the same text, it is better to resort to text indexing. A relevant paper [68] 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 $O(m \lg |\Sigma|)$ search time plus the output-sensitive cost $O(occ)$ to report the occurrences, where $occ \leq n$. Since then, a plethora of papers have studied the text indexing problem in several contexts, sometimes using different terminology [10, 11, 18, 28, 50, 41, 57, 55, 67]; for more references see [7, 20, 38]. Although very efficient, the resulting index data structures are greedy in terms of space, using at least n words or $\Omega(n \lg n)$ 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 terms, or at achieving tradeoff between space requirements and search time complexity. Some authors improved the multiplicative constants in the $O(n \lg n)$-bit practical implementations. For the analysis of constants, we refer the reader to [6, 15, 34, 44, 53, 54, 55]. Other authors devised several variations of sparse suffix trees to store a subset of the suffixes [5, 35, 47, 46, 56, 59]. 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, it is as expensive as scanning the whole text in $O(m+n)$ time. Another interesting index, the Lempel-Ziv index [45], occupies $O(n)$ bits and takes $O(m)$ time to search patterns shorter than $\lg n$ with an output-sensitive cost for reporting the occurrences; for longer patterns, it may occupy $\Omega(n \lg n)$ bits. An efficient and practical compressed index is discussed in [21], but its searches are at word level and are not full text (i.e., with arbitrary substrings).

An alternative line of research has been built upon succinct representation of trees in $2n$ bits, with navigational operations [42]. That representation was extended in [16] to represent a suffix tree in $n \lg n$ bits plus an extra $O(n \lg \lg n)$ expected number of bits. A solution requiring $n \lg n + O(n)$ bits and $O(m + \lg \lg n)$ search time was described in [17]. Munro, Raman, and Rao [62] used it along with an improved succinct representation of balanced parentheses [61] in order to get $O(m \lg |\Sigma|)$ search time with only $n \lg n + o(n)$ bits. They also show in [62] how to get $O(m)$ time and

$O(n \lg n / \lg \lg n)$ bits for existential queries in binary patterns.

The preliminary version of our results presented in [37] stimulated much further work; according to a search on http.//www.scholar.google.com, as of May 2005 more than 80 interesting results have appeared citing this work. A first question raised concerns lower bounds. Assuming that the text is read-only and using a stronger version of the bit-probe model, Demaine and López-Ortiz [22] have shown in the worst case 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 extra storage space. (See also Gál and Miltersen [32] for a general class of lower bounds.) Thus, our index is space-optimal in this sense. A second concerns compressible text. Ferragina and Manzini [29, 30] have devised the Fast Minute index (FM-index), based upon the Burrows–Wheeler transform [14], that asymptotically achieves the order-k empirical entropy of the text and allows them to obtain self-indexing texts (i.e., the compressed text and its index are the same sequence of bits). Sadakane [64] has shown that compressed suffix arrays can be used for self-indexing texts, with space bound by the order-0 entropy. (He also uses our Lemma 2 in section 3.1 to show how to store the skip values of the suffix tree in $O(n)$ bits [65].) The space of compressed suffix arrays has been further reduced to the order-k entropy (with a multiplicative constant of 1) by Grossi, Gupta, and Vitter [36] using a novel analysis based on a finite set model. Both the compressed suffix array and the FM-index require $O(n \lg n)$ auxiliary bits of space during preprocessing, so a third question arises concerning a space-efficient construction. Hon, Sadakane, and Sung [40] have shown how to build both the compressed suffix array and the FM-index with $O(n \lg |\Sigma|)$ bits of auxiliary space by using the text alone and small bookkeeping data structures. Numerous other papers have appeared as well, representing a recent new trend in text indexing, causing space efficiency to no longer be a major obstacle to the large-scale application of index data structures [71]. Ideally we'd like to find an index that uses as few as bits as possible and supports enumerative queries for each query pattern in sublinear time in the worst case (in addition to the output-sensitive cost).

1.4. Outline of the paper. In section 2 we describe the ideas behind our new data structure for compressed suffix arrays, including function Φ. 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 with which we compare. We use standard arithmetic and Boolean operations on words of $O(\lg n)$ bits. Each operation takes constant time; each word is read or written in constant time.

2. Compressed suffix arrays. The compression of suffix arrays falls into the general framework presented by Jacobson [43] 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 information-theoretic argument implies that each such data structure can be canonically identified by $\lg |C_n|$ bits. We try to give a succinct implementation of the same data structure in $O\big(\lg |C_n|\big)$ bits, while supporting the operations within time complexity comparable with that of the natural implementation. However, the information-theoretic argument alone 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. The example below shows the suffix arrays corresponding to the 16 binary strings of length 4:

aaaa#	aaab#	aaba#	aabb#	abaa#	abab#	abba#	abbb#
12345	12354	14253	12543	34152	13524	41532	15432
baaa#	baab#	baba#	babb#	bbaa#	bbab#	bbba#	bbbb#
23451	23514	42531	25143	34521	35241	45321	54321

The natural explicit implementation of suffix arrays requires $O(n \lg n)$ 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 bs in positions 3, 2, 1. The resulting string T, therefore, must be bbba#.

The abstract optimization does not say anything regarding the efficiency of the supported operations. By the correspondence above, we can define a trivial *compress* operation that transforms SA into a sequence of $n-1$ bits 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.

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 "deconstruction" 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

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

construction of suffix arrays and suffix trees, see [8, 25, 26, 27, 55, 66]. We reverse the construction process to discover a recursive structure of the permutations that makes their compression possible. We describe the decomposition scheme in section 2.1, giving some intuition on the compression in section 2.2. We summarize the results thus obtained in section 2.3.

2.1. Decomposition scheme. 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 $\{1, 2, \ldots, n_k\}$. (Intuitively, this permutation is that resulting from sorting the suffixes of T whose suffix pointers are multiples of 2^k.) We run four main steps as follows 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 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 in 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 $2 \times rlookup\big(rank_k(\Psi_k(i)),\, k+1\big) + (B_k[i] - 1)$.

FIG. 3. *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* $1 \le i \le n_k$*:*

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

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

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 $\ell = \Theta(\lg \lg n)$ will be determined in section 3. As a result, we have $\ell + 1$ levels of information, numbered $0, 1, \ldots, \ell$, which form the *compressed representation* of suffix array SA as follows:

1. Level k, for each $0 \le k < \ell$, stores B_k, Ψ_k, and $rank_k$. We do not store SA_k, but we refer to it for the sake of discussion. The arrays Ψ_k and $rank_k$ are not stored explicitly, but are stored in a specially compressed form described in section 3.

2. 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 $rlookup(i, k)$ is the procedure described recursively for level k in Figure 3.

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 $SA_k[i]$.

2.2. Compressibility. As previously mentioned, B_k, $rank_k$, and Ψ_k are the key ingredients for representing a compressed suffix array. Storing B_k and $rank_k$ succinctly, with constant-time access, can be done using previous work (see, e.g., [42]). Hence, we focus on function Ψ_k, which is at the heart of the compressed suffix array since its compression is challenging.

Before giving some intuition on the compressibility, a few comments are in order. When $\Psi_0(i) \neq i$, we observe that Ψ_0 is the analogue of the suffix links in McCreight's

suffix tree construction [57]. (We recall that a suffix link for a node storing a nonempty string $c\alpha$, where $c \in \Sigma$, points to the node storing α.) This is clear when we consider its extension, Φ_0, defined in section 3.2 as follows:

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

Indeed, if i is the position in SA for suffix $T\big[SA[i], n\big]$, then $\Phi_0(i)$ returns j, which is the position in SA for suffix $T\big[SA[i]+1, n\big]$ (when $\Phi_0(i)$ is seen as a suffix link, $c = T\big[SA[i]\big]$ and $\alpha = T\big[SA[i]+1, n\big]$). Analogously, we can see Ψ_k and Φ_k as the suffix links for the positions in SA_k, for any $k \geq 0$.

Functions Ψ_k and Φ_k can be seen as by-products of the suffix array construction. Let the inverse suffix array, SA^{-1}, be the array satisfying $SA^{-1}\big[SA[i]\big] = SA\big[SA^{-1}[i]\big] = i$. Note that SA^{-1} is well defined since SA stores a permutation of $1, 2, \ldots, n$. When $\Phi_0(i) \neq 1$, we have $\Phi_0(i) = SA^{-1}\big[SA[i]+1\big]$. An analogous argument holds for any $k \geq 0$.

In order to see why Ψ_0 and Φ_0 are compressible, we focus on Ψ_0. If we consider the values of Ψ_0 in the example of section 2.1, we do not see any particular order. However, if we restrict our focus to the positions i ($1 \leq i \leq n$) having (a) value of $\mathbf{0}$ in $B_0[i]$, and (b) the *same* leading character in the corresponding suffix, $T\big[SA[i], n\big]$, we observe that the values of Ψ_0 in those positions yield an increasing sequence. For example, choosing `a` as the leading character in condition (b), we find that the positions satisfying also condition (a) are $i = 1, 3, 4, 5, 6, 9, 11, 12$. Their corresponding values are $\Psi_0(i) = 2, 14, 15, 18, 23, 28, 30, 31$, respectively. The latter values form a sorted sequence (called `a` list) that can be implicitly represented in several ways. We clearly have to represent lists for all distinct characters that appear in the text (`a` list, `b`$list, \ldots$).

In this paper, we represent the lists by relating them to the positions of the companion $\mathbf{1}$'s in B_0. Let the preceding character for position i in SA be $T\big[SA[i]-1\big]$ for $SA[i] \neq 1$; otherwise, let it be $T[n]$. We implicitly associate the preceding character for position i with each entry of B_0 containing $\mathbf{1}$. For example, in the case of the `a` list, the positions corresponding to the $\mathbf{1}$'s in B_0 and with preceding character `a` are $2, 14, 15, 18, 23, 28, 30, 31$, which are exactly the items in the `a` list itself! (The motivation for this nice property is that the suffixes remain sorted in relative order, even if interspersed with other suffixes, when we remove their leading character `a`.) By exploiting this relation, we can implement constant-time access to Ψ_0's values without needing to store them explicitly. Further details on how to represent $rank_k$, Ψ_k, and Φ_k in compressed form and how to implement *compress* and *lookup*(i) will be given in section 3.

2.3. Results. 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* $\frac{1}{2}n \lg\lg n + 6n + O(n/\lg\lg n)$ *bits and* $O(n)$ *preprocessing time, so that each call lookup*(i) *takes* $O(\lg\lg n)$ *time.*

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

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

We remark that Sadakane [64] has shown that the space complexity in Theorem 1(ii) and Theorem 2(ii) can be restated in terms of the order-0 entropy $H_0 \le \lg|\Sigma|$ of the string, giving as a result $\epsilon^{-1}H_0\, n + O(n)$ bits. Grossi, Gupta, and Vitter [36] have shown how to attain order-h entropy, namely, $H_h\, n + O(n\lg\lg n/\lg_{|\Sigma|} n)$ bits, where $H_h \le H_0$.

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. (Intuitively, a maximal sequence in SA corresponds to all the occurrences of a pattern in T.)

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

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

(ii) *$O(j-i+n^{\alpha})$ time, for any constant $0 < \alpha < 1$, when $lcp(i,j) = \Omega(\lg n)$, namely, the suffixes pointed to by $SA[i]$ and $SA[j]$ have the same first $\Omega(\lg n)$ 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 section 2.1. 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, showing 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 $\frac{1}{2}n\lg\lg n + O(n)$ bits and $O(\lg\lg n)$ access time. In this section we describe the method referenced in Theorem 1(i) for binary strings and show that it achieves $O(\lg\lg n)$ lookup time with a total space usage of $O(n\lg\lg n)$ bits. Before giving the algorithmic details of the method, let's continue the recursive decomposition of Steps 1–4 described in section 2.1, for $0 \le k \le \ell - 1$, where $\ell = \lceil \lg\lg n \rceil$. The decomposition below shows the result on the example of section 2.1:

	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 $n/\lg n$ 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 Ψ_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 $O(\lg \lg n)$ 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 [42] shows how to support constant-time access to $rank_k$ using only $O\big(n_k(\lg \lg n_k)/\lg n_k\big)$ 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 $n(1/2 + 3/2^{k+1}) + O(n/2^k \lg \lg n)$ bits.

For each $1 \le i \le n_k/2$, let j be the index of the ith **1** in B_k. Consider the 2^k symbols 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:

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

Level 1:

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

Level 2:

`aaaa` list:	$\emptyset$,	$\lvert$`aaaa` list$\rvert = 0$	`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 call to *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 [23] 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 $s(2 + w - \lfloor \lg s \rfloor) + O(s/\lg\lg s)$ bits, so that retrieving the hth integer takes constant time.*

Proof. We take the first $z = \lfloor \lg s \rfloor$ 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 $2s + O(s/\lg\lg s)$ 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 [15, 42, 60].

Since S requires $s + 2^z \le 2s$ bits, the total space required by Q is $2s + O(s/\lg\lg s)$ 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 $\mathbf{1}$ in S by calling $select(h)$, and then compute the number of $\mathbf{0}$'s in the first j bits of S by returning $j - h$. As we can see, this number of $\mathbf{0}$'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 $n(1/2+3/2^{k+1})+O(n/2^k \lg\lg n)$ 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 $1 \le x \le n_k$, is transformed into an integer x' of $w = 2^k + \lg n_k$ 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 $s(2 + w - \lg s) + O(s/\lg\lg s)$ bits, so that retrieving the hth integer takes constant time. Substituting the values for s and w, we get the space bound $(n_k/2)\big(2 + 2^k + \lg n_k - \lg(n_k/2)\big) + O(n_k/\lg\lg n_k) = (n/2^{k+1})(2^k + 3) + O(n/2^k \lg\lg n) = n(1/2 + 3/2^{k+1}) + O(n/2^k \lg\lg n)$. □

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 to which the entry belongs. Since there are $d = 2^{2^k}$ lists, the total number of bits needed to store the array in this naive manner is $s \lg d = (n_k/2)2^k = n/2$, 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 look up 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 $s \lg d + O(s)$ 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 $O(s \lg d)$ 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 $\lg d + \lg s$, 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 $d \le s$, 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 $d \lg s$, which by the assumption $d \le s$ is at most $s \lg d$. 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 $O(s \lg s) = O(s \lg d)$. □

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 $\ell = \lceil \lg \lg n \rceil$ levels of decomposition as discussed in procedure *compress* in section 2. The last level ℓ stores explicitly a reduced suffix array in $(n/2^\ell) \lg n \le n$ bits. The other levels $0 \le k \le \ell - 1$ store three data structures each, with constant time access as follows:

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

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

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

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

3.2. Compressed suffix arrays in $\epsilon^{-1}n + O(n)$ bits and $O(\lg^\epsilon n)$ access time. In this section we give the proof of Theorem 1(ii). Each of the $\lceil \lg \lg n \rceil$ 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 $\ell' = \lceil \frac{1}{2} \lg \lg n \rceil$ and, as before, $\ell = \lceil \lg \lg n \rceil$. 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 [13, 63], we need less than $O\big(\lg \binom{n}{n_{\ell'}}\big) = O(n_{\ell'} \ell')$ bits for D_0 and $O\big(\lg \binom{n_{\ell'}}{n_\ell}\big) = O(n_\ell \ell)$ 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 [63].

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 $1 \le i \le n_k$:

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

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 no longer need the bit vector B_k.

LEMMA 4. *We can store the concatenated lists used for Φ_k in $n + O(n/\lg\lg n)$ bits for $k = 0$, and $n(1 + 1/2^{k-1}) + O(n/2^k \lg\lg n)$ 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 bit vector 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 bit vector 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 bit vector (note that the single occurrence of **0** corresponding to # in the bit vector is the c_0th **0** in the bit vector as we assumed a < # < b; it is not difficult to treat the more common case # < a < b). The additional space is $O(n/\lg\lg n)$ 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 return the position of # by invoking $select_0(c_0)$. If $h < c_0$, then we return the hth **0** (i.e., a) in the bit vector by invoking $select_0(h)$. Otherwise, we invoke $select(h - c_0)$ to get the position in the bit vector of the $(h - c_0)$th **1** (i.e., b). In this way, we simulate the concatenation of lists needed for L_0. With $n + O(n/\lg\lg n)$ bits to implement Φ_0, we can execute $\Phi_0(h)$ in constant time. ∎

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 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 $\max\{s, s'\} \le 2^{\ell'} < 2\sqrt{\lg n}$, and thus the lookup procedure requires $O(\sqrt{\lg n})$ 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 $0 < \epsilon \le 1$. More formally, let us assume that $\epsilon\ell$ is an integer. We maintain the $\epsilon^{-1}+1$ levels $0, \epsilon\ell, 2\epsilon\ell, \ldots, \ell$. The maximum length of each walk is $2^{\epsilon\ell} < 2\lg^\epsilon n$, and thus the lookup procedure requires $O(\lg^\epsilon n)$ time.

By an analysis similar to the one we used at the end of section 3.1, the total space bound is given by $(n/2^\ell)\lg n \le n$ 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:

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

We have to add the contribution of the space $\sum_k |D_k| = O(n_{\epsilon\ell}\,\ell) = O(n(\lg\lg n)/\lg^{\epsilon} n)$ taken by the dictionaries at the ϵ^{-1} levels, but this bound is hidden by the $O(n/\lg\lg n)$ term in the above formula. The final bound is $(1+\epsilon^{-1})\,n + O(n/\lg\lg n)$, 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 $\lg|\Sigma|$ bits, so that the text T can be seen as an array of n entries, each of $\lg|\Sigma|$ bits, or equivalently as a binary string that occupies $n\lg|\Sigma|$ bits. We describe how to extend the ideas presented in sections 3.1–3.2. We redefine ℓ to be $\lceil \lg\lg_{|\Sigma|} n \rceil$. 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 $n\big((1/2)\lg|\Sigma| + 3/2^{k+1}\big) + O(n/2^k \lg\lg n)$ 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 $d = |\Sigma|^{2^k} = 2^{2^k \lg|\Sigma|}$ 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 nonempty lists forms L_k. We number these lists from 0 to $2^{2^k \lg|\Sigma|} - 1$. 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 $w = 2^k \lg|\Sigma| + \lg n_k$ bits, by prepending the binary representation of i to that of $x-1$. By Lemma 2, we can store L_k in $s(2 + w - \lg s) + O(s/\lg\lg s)$ bits, so that retrieving the hth integer takes constant time. Substituting the values for s and w, we get the space bound $(n_k/2)\big(2 + 2^k \lg|\Sigma| + \lg n_k - \lg(n_k/2)\big) + O(n_k/\lg\lg n_k) = n\big((1/2)\lg|\Sigma| + 3/2^{k+1}\big) + O(n/2^k \lg\lg n)$ ∎

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

$$\frac{n\lg n}{2^\ell} + \sum_{k=0}^{\ell-1} n\left(\frac{1}{2^k} + O\left(\frac{1}{2^k}\frac{\lg\lg(n/2^k)}{\lg(n/2^k)}\right) + \frac{\lg|\Sigma|}{2} + \frac{3}{2^{k+1}} + O\left(\frac{1}{2^k\lg\lg n}\right)\right)$$
$$< \left(1 + \frac{1}{2}\lg\lg_{|\Sigma|} n\right) n\lg|\Sigma| + 5n + O\left(\frac{n}{\lg\lg n}\right),$$

as $(n\lg n)/2^\ell + \frac{1}{2}\ell n \le (1 + \frac{1}{2}\lg\lg_{|\Sigma|} n)\, n\lg|\Sigma|$, 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 $n(\lg|\Sigma| + 1/2^{k-1}) + O(n/2^k \lg\lg n)$ 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 $n\lg|\Sigma| + O(|\Sigma| n/\lg\lg n) = n\lg|\Sigma| + o(n)$ 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 $k = 0$ as in the proof of Lemma 4. We associate $\lg|\Sigma|$ bits with each character in Σ according to its lexicographic order. Then we use a bit vector of $n\lg|\Sigma|$ bits to represent Φ_0, in which the fth chunk of $\lg|\Sigma|$ bits encoding a character $c \in \Sigma$ represents the fact that the

c list for Φ_0 contains 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 bit vector. To this end, it treats each occurrence of the $\lg|\Sigma|$ bits for c in the bit vector as a single **1** and the occurrences of the rest of the characters as single **0**'s. It should be clear that the implementation of each version of *select* can be done in $O(n/\lg\lg n)$ 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

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

When $|\Sigma| = O(1)$, we can split the above sum for $k = 0$ and apply Lemma 6 to get $(1+\epsilon^{-1})\, n\lg|\Sigma| + O(n/\lg\lg n)$ 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 $SA_0[i], \ldots, SA_0[j]$ of entries from the compressed suffix array under the hypothesis that the sequence i, $i+1, \ldots, j$ 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 $O(n\lg|\Sigma|)$ 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 [46]. Let D be a two-dimensional orthogonal range query data structure on q points in the grid space $[1\ldots U] \times [1\ldots U]$, where $1 \le q \le U$. Let $P(q)$ be its preprocessing time, $S(q)$ the number of occupied words of $O(\lg U)$ bits each, and $T(q)+O(k)$ 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 $S(n') = O(n/\lg n)$. If such an n' exists, we can report $SA[i], \ldots, SA[j]$ in $O\big(\lg^{1+\epsilon}_{|\Sigma|} n + (n/n')(T(n') + \lg|\Sigma|) + j - i\big)$ time when the sequence i, $i+1, \ldots, j$ is maximal and the suffixes pointed to by $SA[i], \ldots, SA[j]$ have the same first $\Omega(n/n')$ symbols in common. Preprocessing time is $P(n') + O(n\lg|\Sigma|)$ and space is $O(n\lg|\Sigma|)$ bits in addition to that of the compressed version of SA.*

Proof. Suppose by hypothesis that the suffixes pointed to by $SA[i], \ldots, SA[j]$ 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 $b_0, b_1, \ldots, b_{l-1}$, 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, $2l$, $3l, \ldots$). Specifically, the point corresponding to pointer $p = SA[s]$, where $1 \le s \le n$ 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 $S(n') \cdot O(\lg n) + (\epsilon^{-1} + O(1))\, n \lg |\Sigma| = O(n \lg |\Sigma|)$ bits. Preprocessing time is $P(n') + O(n \lg |\Sigma|)$.

We now describe how to query D and output $SA[i], \ldots, SA[j]$ in l stages, with one range query per stage. In stage 0, we perform a range query for the points in $[i \ldots j] \times [1 \ldots n]$. 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 $\lg n$ bits at a time. As a result, we determine two positions g and h of SA^R in $O(l \lg |\Sigma| + \lg^{1+\epsilon}_{|\Sigma|} n)$ time such that the sequence $g, g+1, \ldots, h$ is maximal for SA^R and the suffixes of T^R pointed to by $SA^R[g], \ldots, SA^R[h]$ start with $b_{l-1} \cdots b_1 b_0$.

Before proceeding with 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 $i_0, i_1, \ldots, i_{l-1}$, 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 $j_0 = j, j_1, \ldots, j_{l-1}$. In the same way, we use the function Φ_0 in the compressed version of SA^R to obtain $g_0 = g, g_1, \ldots, g_{l-1}$ and $h_0 = h, h_1, \ldots, h_{l-1}$. We then run the tth stage, for $1 \leq t \leq l-1$, in which we perform a range query for the points in $[i_t \ldots j_t] \times [g_{l-t} \ldots h_{l-t}]$. 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 $SA[i], \ldots, SA[j]$. 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 $i, i+1, \ldots, j$ and $g, g+1, \ldots, h$ are maximal sequences in their respective suffix arrays implies that no other suffix pointers besides those in $SA[i], \ldots, SA[j]$ 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 $O((n/n') \lg |\Sigma| + \lg^{1+\epsilon} n)$ 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 $O\big((n/n')(T(n') + \lg |\Sigma|) + \lg^{1+\epsilon} n + j - i\big)$, 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 [1] takes $P(q) = O(q \lg q)$ preprocessing time by using the perfect hash in [39], which has constant lookup time and takes $O(q \lg q)$ construction time. Space is $S(q) = O(q \lg^{\epsilon} q)$ words and query time is $T(q) = O(\lg \lg q)$. Plugging these bounds into Lemma 7 gives $n' = \Theta(n / \lg^{1+\epsilon} n)$, and hence $O((\lg^{1+\epsilon} n)(\lg |\Sigma| + \lg \lg n) + j - i)$ retrieval time for suffix pointers sharing $\Omega(\lg^{1+\epsilon} n)$ symbols. Preprocessing time is $O(n \lg |\Sigma|)$ and additional space is $O(n \lg |\Sigma|)$ bits.

The second data structure in [9, 69] has preprocessing time $P(q) = O(q \lg q)$, 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 $n' = \Theta(n/\lg n)$ and $O(n^\beta \lg n + j - i) = O(n^\alpha + j - i)$ retrieval time for suffix pointers sharing at least $\Omega(\lg n)$ symbols (by choosing $\alpha > \beta$). Preprocessing time is $O(n \lg |\Sigma|)$ and additional space is $O(n \lg |\Sigma|)$ 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 a *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 [72] 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. Let us take a De Bruijn sequence S of length n, in which each substring of $\lg n$ 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 $k = 2 \lg n$. 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 $\lg \binom{2^k}{s} = \Omega(n)$ 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 generalizations of inverted lists. ∎

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 $\lg |\Sigma|$ bits, which is the case with ASCII and UNICODE text files when 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 $O(n \lg |\Sigma|)$ time such that the index occupies $\big(\epsilon^{-1} + O(1)\big)\, n \lg |\Sigma|$ bits, for any fixed value of $0 < \epsilon \leq 1$, and supports the following queries on any pattern string P of m symbols packed into $O(m/\lg_{|\Sigma|} n)$ words:*

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

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

As a result, an enumerative query can be done in optimal $\Theta(m/\lg_{|\Sigma|} n + occ)$ time for sufficiently large patterns or number of occurrences, namely, when $m = \Omega\big((\lg^{2+\epsilon} n) \lg_{|\Sigma|} \lg n)\big)$ 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 [45, 59, 62] and that are needed later on.

The first data structure is the Lempel-Ziv (LZ) index [45]. It is a powerful tool

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

The second data structure is the Patricia trie [59], 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 $|\Sigma| \geq 2$ by using a $|\Sigma|$-way tree. Each internal node also keeps an integer (called a skip value) to locate the position of the branching character while descending toward 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 [31, 39]) on keys from Σ, which provides constant lookup time and uses $O(t)$ words of space and $O(t \lg t)$ construction time, in the worst case.

Suffix trees are often implemented by building a Patricia trie on the suffixes of T as follows [35]: First, text T is encoded as a binary sequence of $n \lg |\Sigma|$ bits, and its n suffixes are encoded analogously. Second, a Patricia trie is built upon these suffixes; the resulting suffix tree still has n leaves (not $n \lg |\Sigma|$). 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 [62], which builds upon previous work to succinctly represent binary trees and Patricia tries [16, 42, 60, 61]. When employed to store s out of the n suffixes of T, the regular Patricia trie [59] occupies $O(s \lg n)$ bits. This amount of space usage is the result of three separate factors [15, 16], 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 [62] 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 $O(m \lg |\Sigma|)$ time and accesses $O(\min\{m \lg |\Sigma|, s\}) = O(s)$ suffix pointers in the worst case. For each traversed node, its corresponding skip value is computed in time $O(skip\_value)$ by accessing the suffix pointers in its leftmost and rightmost descendant leaves. 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 $\lg n$ bits of the pattern at a time (maybe less if the pattern length is not a multiple of $\lg n$).

Let us first consider the $|\Sigma|$-way Patricia trie PT outlined in section 4 for storing s binary strings, each of length at least $\lg n$. (For example, they could be some suffixes of the text.) To handle border situations, we assume that these strings are (implicitly) padded with $\lg_{|\Sigma|} n$ symbols #. We will show how to reduce the search time for an m-symbol pattern in PT from $O(m \lg |\Sigma|)$ to $O(m/\lg_{|\Sigma|} n + \lg^{\epsilon}_{|\Sigma|} n)$. Without loss of

generality, it suffices to show how to achieve $O(m/\lg_{|\Sigma|} n + \sqrt{\lg_{|\Sigma|} n})$ 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 $\lg n$ bits (or equivalently, $\lg_{|\Sigma|} n$ symbols) in constant time, independently of the number of traversed nodes. For that purpose, we use a perfect hash function h (e.g., see [31]) on keys each of length at most $2 \lg n$ bits. In particular, we use the perfect hash function in [39], which has constant lookup time and takes $O(k)$ words of space and $O(k \lg k)$ 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 $\lg_{|\Sigma|} n$ 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 $2 \lg n$ bits. Formally, the relation between these parameters must satisfy the following two conditions in the case of a successful search of x from node i:

1. Node j is the node identified by starting out from node i and traversing downward toward the nodes of 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 $\lg_{|\Sigma|} n$ (or $\sqrt{\lg_{|\Sigma|} n}$) 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 $\lg_{|\Sigma|} n$ (or $\sqrt{\lg_{|\Sigma|} n}$) 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 $b = l - \lg_{|\Sigma|} n$ (or $b = l - \sqrt{\lg_{|\Sigma|} n}$) 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 $|x| = \lg_{|\Sigma|} n$ 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 $\lg_{|\Sigma|} n$.

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 $|x| = \sqrt{\lg_{|\Sigma|} n}$ 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 of a double traversal of PT followed by the construction of H_1 and H_2, in $O(s \lg s + n)$ 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 $\lg_{|\Sigma|} n$ 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 $\lg_{|\Sigma|} n$ symbols in the pattern (read in $O(1)$ time). Instead, if the hash lookup fails (i.e., no pair is found or fewer than $\lg_{|\Sigma|} n$ symbols are left in the pattern), we switch to H_2 and take only the next $\sqrt{\lg_{|\Sigma|} n}$ symbols in the pattern to branch further in PT. Here the scheme is the same as that of H_1, except that we compare $\sqrt{\lg_{|\Sigma|} n}$ symbols at a time. Finally, when we fail branching again, we have to match no more than $\sqrt{\lg_{|\Sigma|} n}$ 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 $\lg_{|\Sigma|} n$ symbols each over the alphabet Σ, we can preprocess PT in $O(s \lg s + n)$ time so that searching a pattern of length m requires $O(m/\lg_{|\Sigma|} n + \lg^{\epsilon}_{|\Sigma|} n)$ 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 [62]. The speedup we need while searching is easier to obtain. We need not skip nodes, but need only compare $\Theta(\lg n)$ bits at a time in constant time by precomputing a suitable table. The search cost is therefore $O(m/\lg_{|\Sigma|} n)$ 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 a hybrid index data structure, called the *compressed suffix tree*, which follows the multilevel scheme adopted in [17, 62]. Because of the LZ index, it suffices to describe how to support searching of patterns of length $m > \epsilon \lg n$. 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 $O(n \lg |\Sigma|)$ 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 as follows:

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 $s_1 = n/\lg_{|\Sigma|} n$ suffixes pointed to by $SA[1]$, $SA[1 + \lg_{|\Sigma|} n]$, $SA[1 + 2\lg_{|\Sigma|} n], \ldots$. This implicitly splits SA into s_1 subarrays of size $\lg_{|\Sigma|} n$, except the last one (which can be smaller).

Complexity. The size of PT^1 is $O(s_1 \lg n) = O(n \lg |\Sigma|)$ bits. It can be built in $O(n \lg |\Sigma|)$ time by a variation of the standard suffix tree construction [51, 52] and the preprocessing described in Lemma 9.

2. At the second level, we process the s_1 subarrays from the first level, and create

s_1 space-efficient Patricia tries [62], denoted PT_1^2, $PT_2^2, \ldots, PT_{s_1}^2$. We associate the ith Patricia PT_i^2 with the ith subarray. Assume without loss of generality that the subarray consists of $SA[h+1]$, $SA[h+2], \ldots, SA[h+\lg_{|\Sigma|} n]$ for a value of $0 \leq h \leq n - \lg_{|\Sigma|} n$. We build PT_i^2 upon the $s_2 = \lg_{|\Sigma|}^{\epsilon/2} n$ suffixes pointed to by $SA[h+1]$, $SA[h+1+\lg_{|\Sigma|}^{1-\epsilon/2} n]$, $SA[h+1+2\lg_{|\Sigma|}^{1-\epsilon/2} n], \ldots$. This process splits each subarray into smaller subarrays, where each subarray is of size $\lg_{|\Sigma|}^{1-\epsilon/2} n$.

Complexity. The size of each PT_i^2 is $O(s_2)$ bits without accounting for the suffix array, and its construction takes $O(s_2)$ time [62]. Hence, the total size is $O(s_1 s_2) = O(n/\lg_{|\Sigma|}^{1-\epsilon} n)$ bits and the total processing time is $O(n \lg |\Sigma|)$.

3. In the remaining $2\epsilon^{-1} - 2$ intermediate levels, we proceed as in the second level. Each new level splits every subarray into $s_2 = \lg_{|\Sigma|}^{\epsilon/2} n$ 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 $O(n/\lg_{|\Sigma|}^{\epsilon} n)$ bits and the total processing time is $O(n \lg |\Sigma|)$.

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 $\big(\epsilon^{-1} + O(1)\big) n \lg |\Sigma|$ bits; accessing a pointer through a call to *lookup* takes $O(\lg_{|\Sigma|}^{\epsilon/2} n)$ time; the cost of *compress* is $O(n \lg |\Sigma|)$ 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 $O(n \lg |\Sigma|)$ bits and $O(n \lg |\Sigma|)$ construction time. Temporary storage is $O(n \lg n)$ bits.

4.3. Search algorithm. We now have to show that searching for an arbitrary pattern P in the text T costs $O(m/\lg_{|\Sigma|} n + \lg_{|\Sigma|}^{\epsilon} n)$ 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 $SA[i]$, $SA[i+1], \ldots, SA[j]$ 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*$(i+1), \ldots,$ *lookup*(j) to list all the occurrences.

We restrict our discussion to finding 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 $PT_{i_1}^2$ 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 $O(m/\lg_{|\Sigma|} n + \lg_{|\Sigma|}^{\epsilon} n)$ time at the first level by Lemma 9. The intermediate levels cost $O(m/\lg_{|\Sigma|} n + s_2)$ time each, giving a total of $O(m/\lg_{|\Sigma|} n + \lg_{|\Sigma|}^{\epsilon} n)$. 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 $O(\lg_{|\Sigma|}^{\epsilon/2} n)$ cost of *lookup* as given in Theorem 1 (using $\epsilon/2$ in place of ϵ), we obtain $O(\lg_{|\Sigma|}^{\epsilon} n)$ time in addition to $O(m/\lg_{|\Sigma|} n + \lg_{|\Sigma|}^{\epsilon} n)$. 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 rightmost sharing $m = \Omega(\lg n)$ symbols. Combining this cost with the $O(\lg_{|\Sigma|}^{\epsilon} n)$ cost for retrieving any single pointer in Theorem 1, we obtain $O(m/\lg_{|\Sigma|} n + occ \lg_{|\Sigma|}^{\epsilon} n)$ time when both conditions $m \in \big[\epsilon \lg n, o(\lg^{1+\epsilon} n)\big]$ and $occ = o(n^{\epsilon})$ hold, and in $O\big(m/\lg_{|\Sigma|} n + occ + (\lg^{1+\epsilon} n)(\lg|\Sigma| + \lg\lg n)\big)$ 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, efficient lookup time and linear space. For many scenarios, the space requirement is actually sublinear in practice. Specifically, our algorithm uses $o\big(\min\{m\lg|\Sigma|, m+\lg n\}\big)$ search time and $\big(\epsilon^{-1}+O(1)\big)\, n\lg|\Sigma|$ bits of space (where T requires $n\lg|\Sigma|$ bits). Our method is based upon notions of compressed suffix arrays and suffix trees. Given any pattern P of m symbols encoded in $m\lg|\Sigma|$ bits, we can count the number of occurrences of P in T in $o\big(\min\{m\lg|\Sigma|, m+\lg n\}\big)$ time. Namely, searching takes $O(1)$ time when $m = o(\lg n)$, and $O(m/\lg_{|\Sigma|} n + \lg_{|\Sigma|}^{\epsilon} n)$ time otherwise. We achieve optimal $O(m/\lg_{|\Sigma|} n)$ search time for sufficiently large $m = \Omega(\lg_{|\Sigma|}^{1+\epsilon} n)$. For an enumerative query retrieving all *occ* occurrences with sufficiently long patterns, namely, $m = \Omega\big((\lg^{2+\epsilon} n)\lg_{|\Sigma|}\lg n\big)$, we obtain a total search bound of $O(m/\lg_{|\Sigma|} n + occ)$, which is optimal. Namely, searching takes $O(m/\lg_{|\Sigma|} n + occ\lg_{|\Sigma|}^{\epsilon} n)$ time when both conditions $m \in \big[\epsilon\lg n, o(\lg^{1+\epsilon} n)\big]$ and $occ = o(n^{\epsilon})$ hold, and $O\big(m/\lg_{|\Sigma|} n + occ + (\lg^{1+\epsilon} n)(\lg|\Sigma| + \lg\lg n)\big)$ time otherwise. Crucial to our results are functions Ψ_k and Φ_k (see section 2), which are the building blocks of many other results in compressed text indexing.

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 $m \in \big[\epsilon\lg n, o(\lg^{1+\epsilon} n)\big]$ 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 Theorems 3 and 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 [62]. 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 [4, 12, 35, 70].

REFERENCES

[1] S. Alstrup, G. S. Brodal, and T. Rauhe, *New data structures for orthogonal range searching*, in Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science, 2000, pp. 198–207.

[2] A. Amir and G. Benson, *Efficient two-dimensional compressed matching*, in Proceedings of the IEEE Data Compression Conference, J. A. Storer and M. Cohn, eds., IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 279–288.
[3] A. Amir, G. Benson, and M. Farach, *Let sleeping files LIE: Pattern matching in Z-compressed files*, J. Comput. System Sci., 52 (1996), pp. 299–307.
[4] A. Amir, D. Keselman, G. M. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh, *Text indexing and dictionary matching with one error*, J. Algorithms, 37 (2000), pp. 309–325.
[5] A. Andersson, N. J. Larsson, and K. Swanson, *Suffix trees on words*, Algorithmica, 23 (1999), pp. 246–260.
[6] A. Andersson and S. Nilsson, *Efficient implementation of suffix trees*, Software Practice and Experience, 25 (1995), pp. 129–141.
[7] A. Apostolico, *The myriad virtues of suffix trees*, in Combinatorial Algorithms on Words, A. Apostolico and Z. Galil, eds., NATO Advanced Science Institutes Series F: Computer and System Sciences 12, Springer-Verlag, Berlin, 1985, pp. 85–96.
[8] A. Apostolico, C. Iliopoulos, G. M. Landau, B. Schieber, and U. Vishkin, *Parallel construction of a suffix tree with applications.*, Algorithmica, 3 (1988), pp. 347–365.
[9] J. L. Bentley and H. A. Maurer, *Efficient worst-case data structures for range searching*, Acta Inform., 13 (1980), pp. 155–168.
[10] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas, *The smallest automation recognizing the subwords of a text*, Theoret. Comput. Sci., 40 (1985), pp. 31–55.
[11] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht, *Complete inverted files for efficient text retrieval and analysis*, J. ACM, 34 (1987), pp. 578–595.
[12] G. S. Brodal and L. Gąsieniec, *Approximate dictionary queries*, in Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching, D. S. Hirschberg and E. W. Myers, eds., Lecture Notes in Comput. Sci. 1075, Springer-Verlag, Berlin, New York, 1996, pp. 65–74.
[13] A. Brodnik and J. I. Munro, *Membership in constant time and almost-minimum space*, SIAM J. Comput., 28 (1999), pp. 1627–1640.
[14] M. Burrows and D. J. Wheeler, *A Block Sorting Data Compression Algorithm*, Tech. report, Digital Systems Research Center, Palo Alto, CA, 1994.
[15] D. Clark, *Compact Pat Trees*, Ph.D. thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1996.
[16] D. R. Clark and J. I. Munro, *Efficient suffix trees on secondary storage (extended abstract)*, in Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 1996, pp. 383–391.
[17] L. Colussi and A. De Col, *A time and space efficient data structure for string searching on large texts*, Inform. Process. Lett., 58 (1996), pp. 217–222.
[18] M. Crochemore, *Transducers and repetitions*, Theoret. Comput. Sci., 45 (1986), pp. 63–86.
[19] M. Crochemore and D. Perrin, *Two–way string matching*, J. ACM, 38 (1991), pp. 651–675.
[20] M. Crochemore and W. Rytter, *Text Algorithms*, Oxford University Press, Oxford, UK, 1994.
[21] E. S. de Moura, G. Navarro, and N. Ziviani, *Indexing compressed text*, in Proceedings of the South American Workshop on String Processing, Carleton University Press, Ottawa, Ontario, Canada, 1997, pp. 95–111.
[22] E. D. Demaine and A. López-Ortiz, *A linear lower bound on index size for text retrieval*, in Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 2001, pp. 289–294.
[23] P. Elias, *Efficient storage and retrieval by content and address of static files*, J. ACM, 21 (1974), pp. 246–260.
[24] M. Farach and M. Thorup, *String matching in Lempel-Ziv compressed strings*, Algorithmica, 20 (1998), pp. 388–404.
[25] M. Farach-Colton, *Optimal suffix tree construction with large alphabets*, in Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, Miami Beach, FL, 1997, pp. 137–143.
[26] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan, *On the sorting-complexity of suffix tree construction*, J. ACM, 47 (2000), pp. 987–1011.
[27] M. Farach-Colton and S. Muthukrishnan, *Optimal logarithmic time randomized suffix tree construction*, in Automata, Languages, and Programming, 23rd International Colloquium, F. Meyer auf der Heide and B. Monien, eds., Lecture Notes in Comput. Sci. 1099, Springer-Verlag, Berlin, New York, 1996, pp. 550–561.

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

[29] P. Ferragina and G. Manzini, *Opportunistic data structures with applications*, in Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science, 2000, pp. 390–398.

[30] P. Ferragina and G. Manzini, *An experimental study of an opportunistic index*, in Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 2001, pp. 269–278.

[31] M. L. Fredman, J. Komlós, and E. Szemerédi, *Storing a sparse table with $O(1)$ worst case access time*, J. ACM, 31 (1984), pp. 538–544.

[32] A. Gál and P. B. Miltersen, *Automata, Languages and Programming*, in Proceedings of the 30th International Colloquium, Lecture Notes in Comput. Sci. 2719, Springer, Berlin, New York, 2003, pp. 332–344.

[33] Z. Galil and J. Seiferas, *Time-space-optimal string matching*, J. Comput. System Sci., 26 (1983), pp. 280–294.

[34] R. Giegerich, S. Kurtz, and J. Stoye, *Efficient implementation of lazy suffix trees*, in Proceedings of the 3rd Workshop on Algorithm Engineering, J. S. Vitter and C. D. Zaroliagis, eds., Lecture Notes in Comput. Sci. 1668, Springer-Verlag, Berlin, 1999, pp. 30–42.

[35] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider, *New indices for text: PAT trees and PAT arrays*, in Information Retrieval: Data Structures And Algorithms, Prentice–Hall, Englewood Cliffs, NJ, 1992, pp. 66–82.

[36] R. Grossi, A. Gupta, and J. S. Vitter, *High-order entropy-compressed text indexes*, in Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 2003, pp. 841–850.

[37] R. Grossi and J. S. Vitter, *Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract)*, in Proceedings of the 32nd Annual ACM Symposium on the Theory of Computing, Portland, OR, 2000, pp. 397–406.

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

[39] T. Hagerup, P. B. Miltersen, and R. Pagh, *Deterministic dictionaries*, J. Algorithms, 41 (2001), pp. 69–85.

[40] W.-K. Hon, K. Sadakane, and W.-K. Sung, *Breaking a time-and-space barrier in constructing full-text indices*, in Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS), 2003, pp. 251–260.

[41] R. W. Irving, *Suffix Binary Search Trees*, Tech. report TR-1995-7, Computing Science Department, University of Glasgow, Glasgow, UK, 1995.

[42] G. Jacobson, *Space-efficient static trees and graphs*, in Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science, 1989, pp. 549–554.

[43] G. Jacobson, *Succinct Static Data Structures*, Tech. report CMU-CS-89-112, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1989.

[44] J. Kärkkäinen, *Suffix cactus: A cross between suffix tree and suffix array*, in Combinatorial Pattern Matching, Lecture Notes in Comput. Sci. 937, Springer, Berlin, New York, 1995, pp. 191–204.

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

[46] J. Kärkkäinen and E. Ukkonen, *Lempel-Ziv parsing and sublinear-size index structures for string matching*, in Proceedings of the 3rd South American Workshop on String Processing, N. Ziviani, R. Baeza-Yates, and K. Guimarães, eds., Carleton University Press, Ottawa, Ontario, Canada, 1996, pp. 141–155.

[47] J. Kärkkäinen and E. Ukkonen, *Sparse suffix trees*, in Computing and Combinatorics, Lecture Notes in Comput. Sci. 1090, Springer, Berlin, 1996, pp. 219–230.

[48] D. E. Knuth, *Sorting and Searching*, 2nd ed., The Art of Computer Programming 3, Addison-Wesley, Reading, MA, 1998.

[49] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, *Fast pattern matching in strings*, SIAM J. Comput., 6 (1977), pp. 323–350.

[50] S. R. Kosaraju, *Real-time pattern matching and quasi-real-time construction of suffix tree*, in Proceedings of the 26th ACM Symposium on the Theory of Computing, ACM, New York, pp. 310–316.

[51] S. R. Kosaraju and A. 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, Las Vegas, NV, 1995, pp. 169–177.

[52] S. R. Kosaraju and A. L. Delcher, *Correction: Large-scale assembly of DNA strings and space-efficient construction of suffix trees*, in Proceedings of the 28th Annual ACM Symposium on the Theory of Computing, Philadelphia, PA, 1996, p. 659.
[53] S. Kurtz, *Reducing the Space Requirement of Suffix Trees*, Software Practice and Experience, 29 (1999), pp. 1149–1171.
[54] V. Mäkinen, *Compact suffix array*, in Combinatorial Pattern Matching, Lecture Notes in Comput. Sci. 1848, Springer, Berlin, New York, 2000, pp. 305–319.
[55] U. Manber and G. Myers, *Suffix arrays: A new method for on-line string searches*, SIAM J. Comput., 22 (1993), pp. 935–948.
[56] U. Manber and S. Wu, *GLIMPSE: A tool to search through entire file systems*, in Proceedings of the USENIX Winter 1994 Technical Conference, 1994, pp. 23–32.
[57] E. M. McCreight, *A space-economical suffix tree construction algorithm*, J. ACM, 23 (1976), pp. 262–272.
[58] A. Moffat and J. Zobel, *Self-indexing inverted files for fast text retrieval*, ACM Trans. Inform. Systems, 14 (1996), pp. 349–379.
[59] D. R. Morrison, *PATRICIA—Practical algorithm to retrieve information coded In alphanumeric*, J. ACM, 15 (1968), pp. 514–534.
[60] J. I. Munro, *Tables*, in Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Comput. Sci. 1180, Springer, Berlin, pp. 37–42.
[61] J. I. Munro and V. Raman, *Succinct representation of balanced parentheses, static trees and planar graphs*, in Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, 1997, pp. 118–126.
[62] J. I. Munro, V. Raman, and S. S. Rao, *Space efficient suffix trees*, J. Algorithms, 39 (2001), pp. 205–222.
[63] R. Pagh, *Low redundancy in static dictionaries with constant query time*, SIAM J. Comput., 31 (2001), pp. 353–363.
[64] K. Sadakane, *Compressed text databases with efficient query algorithms based on the compressed suffix array*, in Proceedings of ISAAC '00, Lecture Notes in Comput. Sci. 1969, Springer, Berlin, New York, 2000, pp. 410–421.
[65] K. Sadakane, *Succinct representations of lcp information and improvements in the compressed suffix arrays*, in Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 2002, pp. 225–232.
[66] S. C. Sahinalp and U. Vishkin, *Symmetry breaking for suffix tree construction*, in Proceedings of the 26th Annual Symposium on the Theory of Computing, ACM, 1994, pp. 300–309.
[67] E. Ukkonen, *On-line construction of suffix trees*, Algorithmica, 14 (1995), pp. 249–260.
[68] P. Weiner, *Linear pattern matching algorithm*, in Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory, 1973, pp. 1–11.
[69] D. E. Willard, *On the application of sheared retrieval to orthogonal range queries*, in Proceedings of the Second Annual Symposium on Computational Geometry, ACM, New York, 1986, pp. 80–89.
[70] A. C. Yao and F. F. Yao, *Dictionary look-up with one error*, J. Algorithms, 25 (1997), pp. 194–202.
[71] N. Ziviani, E. S. de Moura, G. Navarro, and R. Baeza-Yates, *Compression: A key for next-generation text retrieval systems*, IEEE Comput., 33 (2000), pp. 37–44.
[72] J. Zobel, A. Moffat, and K. Ramamohanarao, *Inverted files versus signature files for text indexing*, ACM Trans. Database Systems, 23 (1998), pp. 453–490.