Compressed Suffix Trees with Full Functionality

Kunihiko Sadakane
Department of Computer Science and Communication Engineering,
Kyushu University
Hakozaki 6-10-1, Higashi-ku, Fukuoka 812-8581, Japan
sada@csce.kyushu-u.ac.jp

Abstract

We introduce new data structures for *compressed suffix trees* whose size are linear in the text size. The size is measured in *bits*; thus they occupy only $O(n \log |\mathcal{A}|)$ bits for a text of length n on an alphabet $\mathcal{A}$. This is a remarkable improvement on current suffix trees which require $O(n \log n)$ bits. Though some components of suffix trees have been compressed, there is no linear-size data structure for suffix trees with full functionality such as computing suffix links, string-depths and lowest common ancestors.

The data structure proposed in this paper is the first one that has linear size and supports all operations efficiently. Any algorithm running on a suffix tree can also be executed on our compressed suffix trees with a slight slowdown of a factor of polylog(n).

1 Introduction

Suffix trees are basic data structures for string algorithms [13]. A pattern can be found in time proportional to the pattern length from a text by constructing the suffix tree of the text in advance. The suffix tree can also be used for more complicated problems, for example finding the longest repeated substring in linear time. Many efficient string algorithms are based on the use of suffix trees because this does not increase the asymptotic time complexity. A suffix tree of a string can be constructed in linear time in the string length [28, 21, 27, 5]. Therefore it is natural to use the suffix tree.

However, concerning the space complexity, the suffix tree is worse than the string. Let $\mathcal{A}$ be the alphabet and T be a string of length n on $\mathcal{A}$. Then the suffix tree is represented by $O(n)$ number of pointers together with the string itself. Because we need $\lg n$ bits[1] to encode a pointer, the suffix tree occupies $O(n \lg n)$ bits. On the other hand, the string occupies $n \lg |\mathcal{A}|$ bits. The alphabet size $|\mathcal{A}|$ is usually much smaller than n, for example, for the whole genome sequence of human, $|\mathcal{A}| = 4$ ($\mathcal{A} = \{a, c, g, t\}$) and $n > 2^{31}$ (2.8 Giga). Even with a space-efficient implementation, the suffix tree is 40 Gigabytes in size [18], whereas the string is only 700 Megabytes.

[1] Let $\lg n$ denote $\log_2 n$.

Therefore it is better to represent the suffix tree in size proportional in the string size, that is, in $O(n \lg |\mathcal{A}|)$ bits.

In this paper, we consider the following computation model. We assume a word-RAM [1, 14] with word size $O(\lg U)$ bits, where $n \leq U$, in which standard arithmetic and bitwise boolean operations on word-sized operands can be performed in constant time. We also have $O(U)$ memory cells, each of which has $O(\lg U)$ bits and is read/written in constant time. Therefore, by using pointers of $O(\lg n)$ bits, the suffix tree is represented in $O(n)$ cells and can be constructed in $O(n)$ time.

We measure the space complexity not by the number of cells but the number of bits. Then the current suffix trees occupy $O(n \lg n)$ bits. This is not practical because in most computers the memory size is measured not by words but bytes. For example, a 32-bit computer can handle not 2^{32} =4G words, but 4G bytes. For a 64-bit computer a word consists of eight bytes. We can assume that each byte consists of constant number of bits. Then, in general, a $\lg U$-bit computer will have U bytes, or $U/\lg U$ words. Then we can construct suffix trees only for strings of length $O(U/\lg U)$. On the other hand, we can store the string of length $O(U/\lg |\mathcal{A}|)$ because each character occupies $\lg |\mathcal{A}|$ bits. Therefore our aim is to construct suffix trees whose size are linear in the string, that is, of $O(n \lg |\mathcal{A}|)$ bits. This is important both theoretically and practically.

We propose $O(n \lg |\mathcal{A}|)$-bit data structures for suffix trees which have the full functionality of the current suffix trees. Though some data structures for suffix trees have been proposed [3, 24], the following are missed: (1) the suffix link of an internal node, (2) the depth of an internal node, and (3) the lowest common ancestor (*lca*) between any two nodes. The suffix link is necessary to use the suffix tree as an automaton recognizing all substrings of the text, which can be used to compute the longest common substring of two strings, matching statistics, etc. The node depth is necessary to implicitly enumerate all maximal repeated substrings of the text in linear time, which can be used for text data mining [26]. The *lca* is necessary to compute the longest common extension of two suffixes in constant time, which can be used in approximate string matching problems. The above elements are also frequently used to solve other problems. In this paper, we propose linear-size data structures for them. The data structures have size $|CSA| + 6n + o(n)$ bits where $|CSA|$ denotes the size of the compressed suffix array [11] of the text, which is also linear. As for the time complexity, our data structures support efficient operations on suffix trees. Any operation on a suffix tree is supported with a slowdown of a factor of polylog(n) in the time complexity.

The rest of the paper is organized as follows. Section 2 reviews the suffix trees, and Section 3 describes space-efficient data structures which are used in our compressed suffix trees. Section 4 proposes new compact data structures for storing longest common prefix information and range mininum queries. Section 5 states the main results: new data structures for compressed suffix trees. Section 6 shows some concluding remarks.

2 Suffix Trees

In this section we review suffix trees. Let $T[1..n] = T[1]T[2]\cdots T[n]$ be a text of length n on an alphabet $\mathcal{A}$ with $|\mathcal{A}| \leq n$. We assume that $T[n] = \$$ is a unique terminator which alphabetically precedes all other symbols. The j-th suffix of T is defined as $T[j..n] = T[j]T[j+1]\ldots T[n]$ and expressed by T_j. A substring $T[1..j]$ is

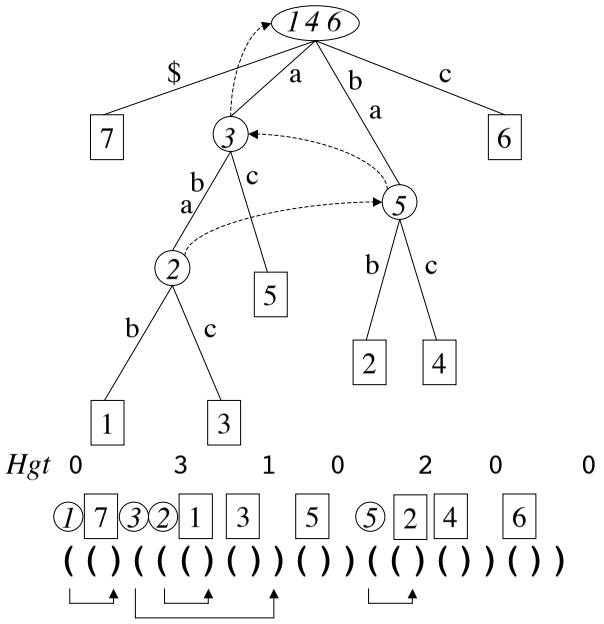

Figure 1: The suffix tree for "ababac$," and its balanced parentheses representation. Suffix links are shown in dotted line.

called a prefix of T.

2.1 Definitions

The suffix tree of a text $T[1..n]$ is a compressed trie built on all suffixes of T. It has n leaves, each of which corresponds to a suffix of T. Each label is labeled by a string, called *edge-label*. The concatenation of labels on a path from the root to a node is called the *path-label* of the node. The path-label of each leaf coincides with a suffix. For each internal node, its children are sorted in the alphabetic order of the first characters of edge-labels. Then the leaves are sorted in the lexicographic order of suffixes. We store the indices j of suffixes T_j in this order in an integer array $SA[1..n]$. This is called the *suffix array* [20]. Figure 1 shows the suffix tree for a text "ababac$." Leaf nodes are shown by boxes, and numbers in the boxes represent the elements of the suffix array. Internal nodes are shown by circles, and numbers in them represent their *inorder* ranks, which are defined later.

The string-depth of a node is the length of the string represented by the node, while the node-depth of a node is the number of nodes on the path from the root node to the node, including itself and excluding the root node.

Suffix links are defined for all internal nodes, except the root node, of a suffix tree as follows.

Definition 1 *The suffix link $sl(v)$ of an internal node v with path-label $x\alpha$, where x denotes a single character and α denotes a possibly empty substring, is the node with path-label α.*

Suffix links are not only necessary to construct a suffix tree in linear time but also

solve problems efficiently, for example finding the longest common substring of two strings in linear time.

We consider the following operations on suffix trees, where v and w denote nodes in the suffix tree, and c denotes a character in $\mathcal{A}$:

Definition 2 *A suffix tree for a text supports the following operations:*

1. *root(): returns the root node.*
2. *isleaf(v): returns Yes if v is a leaf, and No otherwise.*
3. *child(v, c): returns the node w that is a child of v and the edge (v, w) begins with character c, or returns 0 if no such child.*
4. *sibling(v): returns the next sibling of node v.*
5. *parent(v): returns the parent node of v.*
6. *edge(v, d): returns the d-th character of the edge-label of an edge pointing to v.*
7. *depth(v): returns the string-depth of node v.*
8. *lca(v, w): returns the lowest common ancestor between nodes v and w.*
9. *sl(v): returns the node w that is pointed to by the suffix link from v.*

2.2 Data Structures

A suffix tree is decomposed into five components: text, tree topology, node-depths, edge-labels, and suffix links. A basic representation is the following. Text is the string T, which is encoded in $n \lg |\mathcal{A}|$ bits. Tree topology represents parent-child relationships of nodes and consists of $O(n)$ pointers, which occupies $O(n \lg n)$ bits. Node-depths store the string-depth for each internal node in $n \lg n$ bits. An edge-label between internal nodes is represented by a pointer to the text in $n \lg n$ bits. Note that the length of the edge is computed from node-depths of both endpoints of the edge. An edge-label between an internal node and a leaf is represented by the suffix array of T, which occupies $n \lg n$ bits. Suffix links are pointers between internal nodes and represented in $n \lg n$ bits.

3 Succinct Data Structures

In this section we review succinct data structures which are used in our compressed suffix trees.

3.1 Compressed Suffix Arrays

Compressed suffix arrays, proposed by Grossi and Vitter [11], are data structures which reduce the size of suffix arrays from $n \lg n$ bits to $O(n \lg |\mathcal{A}|)$ bits at the cost of increasing access time from constant time to $O(\lg^{\epsilon} n)$ time where ϵ is any constant with $0 < \epsilon \leq 1$.

Table 1: The size and query time of compressed suffix arrays.

size (bits)	t_{SA}	t_Ψ	references
$O(n \lg \lvert\mathcal{A}\rvert)$	$O(\lg^\epsilon n)$	$O(1)$	[11, 12]
$O(nH_0 + n \lg\lg \lvert\mathcal{A}\rvert)$	$O(\lg^\epsilon n)$	$O(1)$	[25] ($\lvert\mathcal{A}\rvert = \text{polylog}(n)$)
$nH_h + O\left(\frac{n \lg\lg n}{\lg_{\lvert\mathcal{A}\rvert} n}\right)$	$O(\lg^2 n / \lg\lg n)$	$O(\lg \lvert\mathcal{A}\rvert)$	[10, Theorem 4.2]

Many variations of the compressed suffix array have been proposed [6, 7, 10, 8, 25]. Ferragina and Manzini [6, 7] proposed the FM-index, a kind of compressed suffix array of size $5nH_k + O(\frac{n}{\lg n}(\lvert\mathcal{A}\rvert + \lg\lg n) + n^\epsilon \lvert\mathcal{A}\rvert 2^{\lvert\mathcal{A}\rvert \lg \lvert\mathcal{A}\rvert}))$ bits where H_k is the order-k entropy of the text. This holds for any integer $k > 0$. They also proposed an algorithm to search for a pattern of length m in $O(m + \lg^{1+\epsilon} n)$ time without using the text T. We call it a *self-indexing* data structure. Ferragina and Manzini [8] also proposed another type of compressed suffix array in $O(nH_k \lg^\epsilon n) + o(n)$ bits which supports $O(m + occ)$ time enumerating query where *occ* is the number of occurrences of the pattern.

Sadakane [25] modified the original compressed suffix array so that it acts as a self-indexing data structure. He also reduced the size from $O(n \lg \lvert\mathcal{A}\rvert)$ bits to $O(n \lg H_0)$ bits. Grossi et al. [10] further reduced its size to $nH_h + o(n)$ bits for any $h \leq \alpha \lg_{\lvert\mathcal{A}\rvert} n$ with $0 < \alpha < 1$. A pattern can be found in $O(m \lg \lvert\mathcal{A}\rvert + \text{polylog}(n))$ time.

For our compressed suffix trees, compressed suffix arrays should support the following operations:

Definition 3 *A compressed suffix array for a text T is a data structure supporting the following operations:*

- *lookup(i): returns $SA[i]$ in time t_{SA},*
- *inverse(i): returns $j = SA^{-1}[i]$, defined such that $SA[j] = i$, in time t_{SA},*
- *$\Psi[i]$: returns $SA^{-1}[SA[i] + 1]$ in time t_Ψ, and*
- *substring(i, l): returns $T[SA[i]..SA[i] + l - 1]$ in $O(l \cdot t_\Psi)$ time.*

The function $\Psi[i]$ in the compressed suffix array is defined as follows:

Definition 4

$$\Psi[i] \equiv \begin{cases} i' \text{ such that } SA[i'] = SA[i] + 1 & (\text{if } SA[i] < n) \\ 0 & (\text{if } SA[i] = n) \end{cases}$$

Table 1 summarizes variations of compressed suffix arrays. We do not include the FM-index because it does not have Ψ.

Any character $T[j]$ in a text can be extracted in constant time by *substring*$(i, 1)$ if the lexicographic order i of the suffix $T_j = T[j..n]$ is given. Furthermore, even if we do not know the lexicographic order we can compute it by using *inverse*(j). Therefore we need not to store the text T explicitly.

The Ψ function is basically used in the other three operations in Definition 3. Therefore it seems that we can remove it from the definition of the compressed suffix

array. However, it is very impotant for compressed suffix trees because it is used to compute a suffix link in constant time. Though the Ψ function can be computed by using *lookup* and *inverse* as in the definition, it cannot be done in constant time.

3.2 Balanced parentheses representations of trees

We use a balanced parentheses encoding of a tree [23, 24]. An m-node rooted ordered tree can be encoded in $2m + o(m)$ bits with various constant time navigational operations. The tree is encoded into m nested open and close parentheses as follows. During a preorder traversal of the tree, write an open parenthesis when a node is visited, then traverse all subtrees of the node, and write a close parenthesis. An example is shown in Figure 1. Any node in the tree is represented by a pair of open and close parentheses '$(\ldots)$.' However, because we can compute the position of the close parenthesis from that of the open parenthesis in constant time (see [23]), we represent a node by the position of the open parenthesis in the sequence. Since there are $2n - 1$ nodes in a suffix tree for a text of length n, exactly n leaves and at most $n - 1$ internal nodes, the suffix tree of the text can be encoded in at most $4n + o(n)$ bits.

Navigational operations on the tree are defined by $rank_p$, $select_p$, *findclose*, *enclose*, etc. The function $rank_p(i)$ returns the number of occurrences of pattern p up to the position i, where p is for example '`()`.' The function $select_p(i)$ returns the position of i-th occurrence of pattern p. The function *findclose*(i) returns the position of the close parenthesis that matches the open parenthesis in position i in the sequence of parentheses. The function *enclose*(i) finds the closest enclosing matching parenthesis pair of a parenthesis pair whose open parenthesis is in position i, which corresponds to compute the parent of a node. All functions take constant time.

By using the above functions, we can perform various operations on a tree. Among them, the following are important for compressed suffix trees:

- *leftrank*(v): returns the number of leaves to the left of node v in preorder,
- *leftmost*(v): returns the leftmost leaf in the subtree rooted at node v, and
- *rightmost*(v): returns the rightmost leaf in the subtree rooted at node v.

These are computed in constant time [24]. Actually they have proposed a compressed suffix tree using the suffix array and the parentheses encoding of Pat tree [9] in $n \lg n + 4n + o(n)$ bits. Our compressed suffix tree is similar to this; however we also support *depth*(v), *lca*(v, w) and *sl*(v).

3.3 Simulating suffix tree traversal by suffix array and height array

Kasai et al. [17] showed that a bottom-up traversal of a suffix tree can be simulated by using only the suffix array and an array storing a set of the lengths of the longest common prefixes between two suffixes, called *Hgt* array. The array $Hgt[1..n]$ is defined as follows.

Definition 5

$$Hgt[i] \equiv \begin{cases} lcp(T_{SA[i]}, T_{SA[i+1]}) & (1 \leq i \leq n-1) \\ 0 & (i = n) \end{cases}$$

Though the *Hgt* array stores the lengths of the longest common prefixes only between adjacent suffixes in a suffix array, it is enough for bottom-up traversal of the suffix tree. Many problems are solved by a bottom-up traversal of the tree.

3.4 Related work

Though space efficient suffix trees have been proposed [3, 24], their size are still not linear in n even for a constant size alphabet, and these are focused on only finding the number of occurrences and positions of a pattern. They do not store suffix links and node depths. Therefore they do not have the full functionality of suffix trees.

For node depths, Clark and Munro [3] used a $\lg\lg\lg n$ bits field to store the length of an edge instead of storing the depth of the corresponding node although the size was still not linear. Moreover the method cannot be used to answer the depth of a node quickly. Munro et al. [24] proposed an algorithm for searching for patterns without storing edge lengths. The algorithm calculates them online whenever needed. Though their algorithm has the same time complexity to find a pattern as the algorithm that requires edge lengths, it may not be suitable for traversing the nodes of the suffix tree. In this case their algorithm takes $O(n^2)$ time because the sum of all edge-lengths in the suffix tree is $O(n^2)$.

4 New data structures for *lcp* information

In this section we propose space-efficient data structures for storing *lcp* (longest common prefix) information between suffixes. First we show a data structure to represent the *Hgt* array, which is also used to encode string-depths in a suffix tree. Then we show a data structure for Range Minimum Query, which is used to compute *lowest common ancestors* in a suffix tree.

We first propose succinct representations of the *Hgt* array. Though values of the *Hgt* array for a text T are usually small, they may reach $n-1$. Therefore it is necessary to use an array of integers each with $\lg n$-bits width if we use fixed-width integers. However we can efficiently store the values by using the properties of *Hgt* array, which can be efficiently extracted by using the information of the suffix array:

Theorem 1 *Given i and $SA[i]$, the value $Hgt[i]$ can be computed in constant time using a data structure of size $2n + o(n)$ bits.*

Proof: See Section 4.1. □

From this theorem we can store data structures that can be used to simulate an in-order or a bottom-up traversal of a suffix tree in $|SA| + 2n + o(n)$ bits where $|SA|$ is the size of the suffix array or the compressed suffix array. The time complexity for the traversal is $O(n \cdot t_{SA})$ where t_{SA} is the time to compute an element of the suffix array, that is, $O(1)$ time for suffix array, or $O(\lg^{\epsilon} n)$ time for the compressed suffix array.

This result can be extended to compute *lcp* between two arbitrary suffixes:

Theorem 2 *Given i and j, the length of the longest common prefix between suffixes $T_{SA[i]}$ and $T_{SA[j]}$ can be computed in $O(t_{SA})$ time using a data structure of size $|SA| + 6n + o(n)$ bits.*

Proof: See Section 4.1. □

4.1 Data structures for *Hgt* array

Here we show the data structure of Theorem 1. To achieve the space complexity, we use a space efficient data structure for storing sorted integers [4] and the select function [22] as it is used in the compressed suffix array [11].

Lemma 1 [12] *Given s integers in sorted order, each containing w bits, where $s < 2^w$, we can store them in at most $s(2 + w - \lfloor \lg s \rfloor) + O(s/\lg\lg s)$ bits, so that retrieving the h-th integer takes constant time.*

To encode the *Hgt* array, the above data structure cannot be directly used because the numbers are not sorted. However we can convert them into sorted ones by using the following lemma.

Lemma 2 $Hgt[\Psi[i]] \geq Hgt[i] - 1$

Proof: Let $p = SA[i]$, $q = SA[i+1]$ and $l = Hgt[i] = lcp(T_p, T_q)$. If $T[p] \neq T[q]$, then $Hgt[i] = 0$ and the inequality holds because $Hgt[\Psi[i]] \geq 0$. If $T[p] = T[q]$, consider suffixes T_{p+1} and T_{q+1}. From the definition of Ψ, $SA[\Psi[i]] = p + 1$ and $SA[\Psi[i+1]] = q + 1$. The suffix T_{q+1} is lexicographically larger than the suffix T_{p+1} from the definition of lexicographic order. That is, $\Psi[i] < \Psi[i+1]$. Therefore an integer i' such that $\Psi[i] + 1 = i' \leq \Psi[i+1]$ exists. The suffix $T_{SA[i']}$ has a prefix of length $l-1$ that matches with both prefixes of T_{p+1} and T_{q+1} because of the definition of lexicographic order. This completes the proof. □

Note that this lemma is identical to that in Kasai et al. [17]. From this lemma, we have the following relation for $p = SA^{-1}[1]$:

$$\begin{aligned} Hgt[p] &\leq Hgt[\Psi[p]] + 1 \\ &\leq Hgt[\Psi^k[p]] + k \\ &\leq Hgt[\Psi^{n-1}[p]] + n - 1 \\ &= n - 1 \end{aligned}$$

where the equality comes from the fact that $SA[\Psi^{n-1}[p]] = n$ and the *Hgt* value for the suffix T_n is 0 because $T[n] = \$$ is a unique terminator.

Now we have a sequence of n sorted numbers $Hgt[\Psi^k[p]] + k$ for $k = 0, 1, \ldots, n-1$ in the range $[0, n-1]$ which can be stored using $2n + o(n)$ bits and can be accessed in constant time. To obtain $Hgt[i]$, the remaining task is to compute k such that $i = \Psi^k[p]$, which is easily done as follows. From the definition of Ψ,

$$SA[\Psi^k[i]] = SA[i] + k$$

for any i. Therefore

$$SA[i] = SA[\Psi^k[p]] = SA[p] + k = k + 1,$$

that is, $k = SA[i] - 1$.

Figure 2 shows an example of creating a sorted sequence of numbers. The last row shows $SA[i] + Hgt[i]$ for $i = 1, 2, \ldots, n$, which is the summation of the second and the third rows. If we sort the numbers in order of $SA[i]$ values, we obtain a sorted sequence "4 4 4 4 5 6 7." Then these are encoded in a 0,1 sequence "00001 1 1 1 01 01 01" whose length is at most $2n$ bits. The blanks in the sequence are only for explanation. We can uniquely decode the numbers from the sequence.

The algorithm to calculate $Hgt[i]$ becomes as follows:

i	1	2	3	4	5	6	7
Hgt	0	3	1	0	2	0	0
SA	7	1	3	5	2	4	6
$SA + Hgt$	7	4	4	5	4	4	6

Figure 2: How to create a sorted sequence from Hgt.

1. Extract the k-th entry v ($k \geq 0$) of the sorted numbers where $k = SA[i] - 1$.
2. Subtract k from v.

A problem with our encodings of the Hgt array is that a value $Hgt[i]$ is stored in the bit-vector H in the order of not i but $SA[i]$. Therefore access to H becomes random if we retrieve the suffix array lexicographically. Another problem is that both i and $SA[i]$ are necessary to compute $Hgt[i]$. If we use the compressed suffix array, retrieving $SA[i]$ takes $O(t_{SA})$ time.

4.2 Computing *lcp* between arbitrary suffixes

We describe the data structure of Theorem 2. It consists of two components: one to store the Hgt array of Theorem 1, and one to perform RMQ (range minimum query) in Hgt array. The former occupies $2n + o(n)$ bits and the latter $4n + o(n)$ bits.

Because the suffixes are lexicographically sorted in the suffix array, computing *lcp* between two suffixes $T_{SA[l]}$ and $T_{SA[r]}$ ($l < r$) is equivalent to computing the minimum of $Hgt[l], Hgt[l+1], \ldots, Hgt[r-1]$. Let $Hgt[m]$ be the minimum value among them. If there are two or more minimum values, we can choose one arbitrary. To compute the minimum value, we use the algorithm for computing *lca* nodes [2] because of the following property:

Proposition 1 *Let v be the node $lca(leaf(l), leaf(r))$ in the suffix tree and m be the index defined above. Then $v = lca(leaf(m), leaf(m+1))$ and v has string-depth $Hgt[m]$.*

Proof: Because of the construction of the suffix tree, the node $v' \equiv lca(leaf(m), leaf(m+1))$ has string-depth $lcp(T_{SA[m]}, T_{SA[m+1]}) = Hgt[m]$. The node v also has string-depth $Hgt[m]$. Because $T_{SA[m]}$ locates lexicographically between $T_{SA[r]}$ and $T_{SA[r]}$ and these three suffixes have a common prefix of length $Hgt[m]$, the nodes v and v' represent the same string. This means $v = v'$. □

Therefore we first compute the node v, secondly compute the index m, then compute $Hgt[m]$ by using the data structure in Section 4.1.

We define the following Range Minimum Query (RMQ) problem:

Problem 1 (Range Minimum Query) *For indices l and r, between 1 and n, of an array A, the range minimum query $\mathrm{RMQ}_A(l, r)$ returns the index of the smallest element in the subarray $A[l..r]$. If there is a tie-breaking we choose the leftmost one.*

The algorithm of Bender and Farach-Colton [2] reduces the problem of computing *lca* to the Range Minimum Query as follows. We traverse the nodes of the tree (in our case, the suffix tree) in depth-first manner and store their node-depths in an array L. The node-depth of a node is the number of nodes on the path from the root to the

node and is usually different from its string-depth. An example is shown in Figure 1. We also store the position in L for each node of the tree that corresponds to the first visit to the node in the depth-first traversal. Then the *lca* between two nodes can be represented as the minimum value in the subarray of L, where the boundaries of the subarray correspond to the nodes.

In the original algorithm, they store L as an integer array of size $2n-1$ for an n-node tree. Therefore L occupies $O(n \lg n)$ bits. They also store the positions of L representing nodes in another integer array, which occupies $O(n \lg n)$ bits. On the other hand, we store these information in $4n + o(n)$ bits as follows. Because the difference between two adjacent elements of L is 1 or -1, we encode the differences by a sequence P of open and close parentheses (see Figure 1). Actually, the parentheses sequence P is exactly the same as the encoding for a tree [23]. The tree is encoded into $2n$ nested open and close parentheses as follows. During a pre-order traversal of the tree, write an open parenthesis when a node is visited, then traverse all subtrees of the node, and write a close parenthesis.

To compute the elements of L in constant time, we explicitly store them for every $\lg^2 n$ elements, and for every $\lg n$ elements we store the difference from the nearest explicitly-stored element. The former occupies $\frac{n}{\lg^2 n} \lg n = o(n)$ bits, and the latter occupies $\frac{n}{\lg n} \lg(\lg^2 n) = o(n)$ bits. We need to compute the position in L corresponding to each leaf of the tree.

Lemma 3 *The position x of a pair of parentheses '()' in P that represents leaf(i) can be computed by*

$$x = \mathit{select}_{()}(P, i).$$

Proof: Because any occurrence of '()' in P corresponds to a leaf of the tree and the leaves appear in the lexicographic order of the suffixes, the lemma holds. □

Because the data structure for *select* function occupies $o(n)$ bits in addition to the parentheses sequence P, the total size is $4n + o(n)$ bits. We can run the algorithm for the Range Minimum Query on P instead of L.

We can compute the index m from the result of the Range Minimum Query for computing $\mathit{lca}(\mathit{leaf}(l), \mathit{leaf}(r))$.

Lemma 4 *Let x and y be the positions of '()' in P that represent leaves $T_{SA[l]}$ and $T_{SA[r]}$ respectively. Then the index m of $Hgt[m]$ that attains the minimum value among $Hgt[l], Hgt[l+1], \ldots, Hgt[r-1]$ can be computed by*

$$m = \mathit{rank}_{()}(P, \mathrm{RMQ}_L(x, y)).$$

Proof: $\mathrm{RMQ}_L(x, y)$ returns the position p of a close parenthesis ')' in the parentheses sequence P corresponding to the node $v = \mathit{lca}(\mathit{leaf}(l), \mathit{leaf}(r))$. The node is equal to $\mathit{lca}(\mathit{leaf}(m), \mathit{leaf}(m+1))$. Because P represents a depth-first traversal of the tree, especially in the lexicographic order of leaves, leaves $\mathit{leaf}(i)$ for $i = 1, 2, \ldots, m$ appear to the left of p in P, and other leaves appear to the right of p. Therefore m is equal to the number of leaves encoded to the left of p in P, that is, $m = \mathit{rank}_{()}(P, p)$. □

Now we have the algorithm to compute $\mathit{lcp}(T_{SA[l]}, T_{SA[r]})$.

1. Compute $x = \mathit{select}_{()}(P, l)$ and $y = \mathit{select}_{()}(P, r)$.
2. Compute $m = \mathit{rank}_{()}(P, \mathrm{RMQ}_L(x, y))$.
3. Compute $Hgt[m]$.

4.3 Data structures for Range Minimum Query

The data structure of Bender and Farach-Colton [2] for the Range Minimum Query occupies $O(n)$ words, in other words, $O(n \lg n)$ bits. We propose a modified data structure which occupies only $2n + o(n)$ bits.

The original algorithm: To compute the index of the minimum value in a subarray in constant time, we use precomputed tables of size $o(n)$ bits. We divide the whole array L into blocks of size $\frac{\lg n}{2}$, and define an array $L'[0, \ldots, 2n/\lg n]$ such that $L'[i]$ stores the minimum value in the i-th block. To compute $\mathrm{RMQ}_L(x, y)$, we first compute the indices x' and y' of blocks containing x and y. Then the minimum value of $L[x..y]$ is equal to the minimum of

1. the minimum of $L[x..e]$ where e is the last element in the same block as x,
2. the minimum of $L'[(x'+1)..(y\prime-1)]$, and
3. the minimum of $L[s..y]$ where s is the first element in the same block as y.

The minimum value and its index for the first and the third ones can be computed in constant time using tables of size $o(n)$ bits. For the second one, we construct another two-dimensional table $M[i, k]$. For each i and k ($i = 0, 1, \ldots, 2n/\lg n$, $k = 0, 1, \ldots, \lfloor \lg n \rfloor$), $M[i, k]$ stores the index of the minimum value in $L'[i..i + 2^k - 1]$. Then the index of the minimum value between $L'[x']$ and $L'[y']$ can be computed in constant time by $\min\{M[x', k], M[y' - 2^k + 1, k]\}$ where $k = \lfloor \lg(y' - x') \rfloor$. We also use a table of size $o(n)$ bits to compute $\lfloor \lg x \rfloor$ in constant time. This data structure occupies $O(n)$ words because its size is $(2n/\lg n) \cdot \lg n$.

Our algorithm: We divide the array L into blocks of size $\lg^3 n$. Then the array $M[i, k]$ occupies only $O(\frac{n}{\lg^3 n} \cdot \lg n \cdot \lg n) = O(\frac{n}{\lg n}) = o(n)$ bits. To compute the minimum element in the j-th block ($j = 0, 1, \ldots, n/\lg^3 n$), we further divide the block into subblocks of size $\frac{\lg n}{2}$. We create a two-dimensional table $M_j[i, k]$ where $i = 0, 1, \ldots, 2\lg^2 n - 1$ and $k = 0, 1, \ldots, \lg(2\lg^2 n)$ for each block. These tables occupy $O(\frac{n}{\lg n} \cdot \lg(\lg^2 n) \cdot \lg(\lg^2 n)) = o(n)$ bits in total. We also use a table for subblocks of length $\frac{\lg n}{2}$. It also occupies $o(n)$ bits.

By using the above data structure, the position of the minimum value among $L[x], L[x + 1], \ldots, L[y]$ can be computed in constant time.

5 New Compressed Suffix Trees

In this section we show algorithms for navigating compressed suffix trees for small alphabets using the data structures proposed in Section 4.

The notation is as follows. Nodes of suffix trees are represented by integers u, v, w, etc., instead of using pointers, and the corresponding nodes are denoted by $\bar{u}, \bar{v}, \bar{w}$, etc. Nodes are also represented by their ranks: *preorder* or *inorder* ranks and denoted by i, j, etc.

The main result of this paper is stated as follows:

Theorem 3 *A compressed suffix tree for a text of length n, supporting the operations $child(v, c)$ in $O(\min\{|\mathcal{A}|, \lg n\} t_{SA})$ time, $depth(v)$ and $edge(v, d)$ in $O(t_{SA})$ time, $sl(v)$ in $O(t_\Psi)$ time, and other operations in constant time, can be represented in $|CSA| + 6n + o(n)$ bits where $|CSA|$ denotes the size of the compressed suffix array.*

Proof: As for the time complexity, please refer to Sections 5.1 to 5.6. Concerning the space complexity, we use the compressed suffix array, the compressed representation of *Hgt* array in $2n + o(n)$ bits, and the parentheses encoding of the suffix tree in $4n + o(n)$ bits. □

Table 2: The size and query time of compressed suffix trees

size (bits)	$child(v, c)$	$depth(v)$, $edge(v, d)$	$sl(v)$
$\|CSA\| + 6n + o(n)$	$O(\min\{\|\mathcal{A}\|, \lg n\} t_{SA})$	$O(t_{SA})$	$O(t_\Psi)$
$nH_h + 6n + O\left(\frac{n \lg\lg n}{\lg_{\|\mathcal{A}\|} n}\right)$	$O(\min\{\|\mathcal{A}\|, \lg n\} \lg^2 n / \lg\lg n)$	$O(\lg^2 n / \lg\lg n)$	$O(\lg \|\mathcal{A}\|)$
$O(\frac{1}{\epsilon} n \lg \|\mathcal{A}\|)$	$O(\lg^\epsilon n)$	$O(\lg^\epsilon n)$	$O(1)$

5.1 How to represent a node

It was shown by Munro and Raman [23] that a node of a tree can be represented by a pair of parentheses '(...)' in a nested parentheses sequence. Therefore we represent a node $\bar{v}$ of a suffix tree by an integer $v \in [1, 2n]$ which is the position of the open paranthesis representing the node in the parentheses sequence P. To perform some operations on the suffix tree, we also need to represent a node by its *preorder* and *inorder*. We first show how to convert these values.

The parentheses sequence P is produced during a preorder traversal of the tree. Therefore the *preorder* i of a node $\bar{v}$ and the position v in P can be easily converted from each other in constant time:

$$\begin{aligned} i &= rank_{(}(v) \\ v &= select_{(}(i). \end{aligned}$$

We also need the *inorder* of an internal node. First we give its definition.

Definition 6 *The inorder rank of an internal node v is defined as the number of visited internal nodes, including v, in the alphabetic depth-first traversal, when v is visited from a child of it and another child of it will be visited next.*

Note that only internal nodes have inorder ranks and an internal node may have two or more ranks if it has more than two children. Figure 1 shows an example of inorder ranks. Each number in a circle represents an irorder rank of an internal node. The root node has three inorder ranks 1, 4 and 6.

The following lemma gives an algorithm to compute the *inorder* of an internal node.

Lemma 5 *Let $\bar{v}$ be an internal node, and let i be the inorder of the node. Then v and i can be converted from each other in constant time by*

$$\begin{aligned} i &= rank_{()}(findclose(v+1)) \\ v &= parent(select_{)(}(i)+1) \end{aligned}$$

Proof: Given an internal node $\bar{v}$, $v+1$ represents the position of an open parenthesis representing the leftmost child w of v. Thus the position $u = findclose(v+1)$ is the corresponding close parenthesis and the subtree rooted at w is expressed between v and u in the balanced parentheses sequence. Since each internal node has at most two children, an inorder rank of v is defined by the number of leaves that have smaller preorder ranks than v, and it is calculated by $rank_{()}(u)$.

From the definition of *inorder*, the *inorder* i of a node v is the number of times that during the preorder traversal from the root to v we climb up an edge and immediately go down another edge. This movement is represented by '`)(`' in the parentheses sequence. Therefore we first compute the position $x = select_{)(}(P, i)$. Then the open parenthesis in position $x+1$ represents a child of the node v. Because we want to know which parenthesis corresponds to v, we compute $parent(P, x+1)$, which returns the position of open parenthesis of v. □

Note that if a node has two or more inorders, the algorithm returns the smallest one, and the *parent* operation is described in Section 5.4.

The balanced parentheses representations of the example suffix tree are shown at the bottom of Figure 1. An internal node is represented by an open parenthesis followed by another open parenthesis and it is arranged in order of its preorder rank. Its inorder rank becomes the number of '`()`' up to the position indicated by the arrow from the open parenthesis.

5.2 How to associate information to nodes

We can associate additional information with nodes of a suffix tree. This is necessary to use a suffix tree as a tool to solve another problem. It is enough to assign consecutive numbers to the nodes. Let m be the number of internal nodes in a suffix tree with n leaves. If we want to store some information in each node, we store them in an array and use the *preorder* of nodes as indices to the array. The preorders have values 1 to $m+n$, that is, there is a one-to-one mapping from the *preorders* and $[1, m+n]$.

If we want to store information in only internal nodes, we can use the *inorder* of nodes as indices. If a node has two or more *inorders* we use the smallest one. The *inorders* have m values in $[1, n]$. Therefore this method is slightly redundant.

We can also use another order of internal nodes. Let v be an internal node. Then its index is computed by $rank_{(}(P, v) - rank_{()}(P, v)$, that is, the *preorder* assigned only internal nodes.

If we want to store information in leaves only, we can use the lexicographic orders of nodes as indices. Obviously, there exists a one-to-one mapping between a leaf representing a suffix and the lexicographic order of the suffix.

5.3 How to represent edge-labels

The string-depth of an internal node v, and the edge-label between nodes v and $parent(v)$ is represented as follows. Let $i = inorder(v)$ be the *inorder* of the node

v. Then suffixes $T_{SA[i]}$ and $T_{SA[i+1]}$ share the prefix of length $Hgt[i]$. That is, the string-depth of v is equal to $Hgt[i]$. The edge-label between nodes v and $parent(v)$ is represented by $T[SA[i] + d_1..SA[i] + d_2 - 1]$ where

$$\begin{aligned} i &= inorder(v) \\ d_1 &= Hgt[inorder(parent(v))] \\ d_2 &= Hgt[i]. \end{aligned}$$

Therefore $edge(v, d) = T[SA[i] + d_1 + d - 1$ is computed in $O(t_{SA})$ time. Recall that the label can be represented without using the text T. It can be extracted in $O(t_{SA} + (d_2 - d_1)t_\Psi)$ time.

5.4 Navigating the suffix tree

Finding the root node The root node can be found easily because its *preorder* is 1 and it is represented by the first open parenthesis in P. That is, $root() \equiv 1$.

A node of the suffix tree is a leaf if and only if its parentheses representation is `()`. Therefore $isleaf(v)$ is computed in constant time.

Finding a child node and a sibling Finding a sibling of a node is necessary to traverse all the nodes. For efficient query we normally store for each node pointers to its first child and the next sibling, which can be represented by the parentheses sequence. We can compute the next sibling in constant time by $sibling(w) = findclose(P, w)+1$.

Finding a child is necessary to pattern queries. From the construction of the parentheses sequence, the first child of v is computed by $w = v + 1$. Then we iteratively compute the next sibling. In each step we recover the first character of the edge-label between v and w and compare with c in $O(t_{SA})$ time. Therefore the time for the operation $child(v, c)$ is $O(|\mathcal{A}|t_{SA})$.

We can also use a binary search on the (compressed) suffix array to find $child(v, c)$. We can find the leftmost and the rightmost descendants of v in constant time [24]. The range $[l, r]$ of the suffix array corresponding to v is then calculated in constant time. Let P be the string represented by the node v. Then we can find a sub-range of $[l, r]$ corresponding to a pattern Pc, which is a concatenation of P and c, by a simple binary search in $O(\lg n \cdot t_{SA})$ time. From the sub-range we can find $child(v, c)$ in constant time by using *lca* operation described below. To sum up, $child(v, c)$ can be found in $O(\min\{|\mathcal{A}|, \lg n\}t_{SA})$ time.

Finding the parent Given a node $\bar{v}$ in the suffix tree, finding the parent node $\bar{w}$ is just to compute $w = enclose(v)$. It takes constant time.

5.5 How to compute *lca*

We show an algorithm to compute *lca* between two nodes represented by a parentheses sequence in constant time.

Lemma 6 *The lowest common ancestor between two nodes in the compressed suffix tree can be computed in constant time using an auxiliary data structure of size $o(n)$ bits.*

Proof: Let v, w be the positions in a parentheses sequence P representing nodes $\bar{v}$ and $\bar{w}$ respectively. We can easily check whether $\bar{v}$ is an ancestor of $\bar{w}$ or vice versa. Assume that $v < w$. Then $\bar{v}$ is an ancestor of $\bar{w}$ if and only if $\mathit{findclose}(v) > \mathit{findclose}(w)$. In this case we return $\mathit{lca}(v, w) = v$.

Assume that v is not an ancestor of w, and vice versa. Then the position u of open parenthesis representing the node $\mathit{lca}(v, w)$ is computed in constant time by

$$u = \mathit{parent}(\mathrm{RMQ}_{P'}(v, w) + 1)$$

where P' is a sequence of integers such that $P'[i] = \mathit{rank}_{(}(i) - \mathit{rank}_{)}(i)$. Note that we need not to store P' explicitly. Each element of it can be computed in constant time using $o(n)$ bits indices.

The range minimum query returns the index m of the minimum element in $P'[i..j]$. Then $P[m] =$ ')' and $P[m+1] =$ '(' always hold because of the minimality. Therefore $P[m+1]$ is the open parenthesis of a child of $\mathit{lca}(v, w)$. Therefore the position of open parenthesis of the node $\mathit{lca}(v, w)$ is computed by $\mathit{parent}(m + 1)$. □

5.6 How to compute suffix links

Lemma 7 *Let v be the position in a parentheses sequence P representing a non-root node $\bar{v}$. Then the position w of open parenthesis representing the node $sl(v)$ is computed in $O(t_\Psi)$ time by*

$$\begin{aligned} x &= \mathit{rank}_{()}(v-1)+1 \\ y &= \mathit{rank}_{()}(\mathit{findclose}(v)) \\ x' &= \Psi[x] \\ y' &= \Psi[y] \\ w &= \mathit{lca}(\mathit{select}_{()}(x'), \mathit{select}_{()}(y')). \end{aligned}$$

Proof: The algorithm first computes the leftmost and the rightmost leaves that are descendants of v. Because the leftmost leaf that is a descendant of v is the first leaf appearing in the parentheses sequence after v, the lexicographic index of the leftmost leaf is computed by $x = \mathit{rank}_{()}(v-1)+1$. Concerning the index of the rightmost leaf, because all leaves below v are encoded in the parentheses sequence between the open and the close parentheses representing v, the lexicographic order of the rightmost leaf is computed by $y = \mathit{rank}_{()}(P, \mathit{findclose}(v))$. The leaves represent suffixes $T_{SA[x]}$ and $T_{SA[y]}$. From the definition of Ψ, $\mathit{leaf}(x')$ and $\mathit{leaf}(y')$ represent suffixes $T_{SA[x']} = T_{SA[x]+1}$ and $T_{SA[y']} = T_{SA[y]+1}$, respectively. Let $l = \mathit{lcp}(T_{SA[x]}, T_{SA[y]})$. Then l is equal to the string-depth of node v because $\mathit{leaf}(x)$ and $\mathit{leaf}(y)$ are the leftmost and the rightmost descendants of v. Obviously $l-1 = \mathit{lcp}(T_{SA[x']}, T_{SA[y']})$ holds. Then the node $\mathit{lca}(\mathit{select}_{()}(P, x'), \mathit{select}_{()}(P, y'))$ has string-depth $l-1$, which means it is $sl(v)$. □

5.7 Variations of compressed suffix trees

We can apply any compressed suffix arrays to our compressed suffix trees. We show two examples. One is the most space-efficient one.

Corollary 1 *A compressed suffix tree, supporting the operations child(v, c) in $O(\min\{|\mathcal{A}|, \lg n\} \lg^2 n / \lg\lg n)$ time, edge(v, d) and depth(v) in $O(\lg^2 n / \lg\lg n)$ time, sl(v) in $O(\lg |\mathcal{A}|)$ time, and other operations in constant time, can be represented in $nH_h + 6n + O(n \lg\lg n / \lg_{|\mathcal{A}|} n)$ bits for any $h \leq \alpha \lg_{|\mathcal{A}|} n$ with $0 < \alpha < 1$.*

Proof: Follows from Theorem 3 and Grossi et al. [10, Theorem 4.2]. □

The other is the most time-efficient one.

Corollary 2 *A compressed suffix tree, supporting the operations child(v, c), edge(v, d) and depth(v) in $O(\lg^\epsilon n)$ time, and other operations in constant time, can be represented in $O(\frac{1}{\epsilon} n \lg |\mathcal{A}|)$ bits for any $0 < \epsilon \leq n$.*

Proof: We use the compressed suffix tree of Grossi-Vitter [12, Theorem 3]. It uses the Patricia trie and perfect hash functions to find a child node. The space complexity is $O(\frac{1}{\epsilon} n \lg |\mathcal{A}|)$. Given a pattern of length m, we can compute the lexicographic order of a suffix whose prefix is the same as the pattern in $O(m \lg |\mathcal{A}| / \lg n + \lg^\epsilon)$ time. By using this data structure, the operation *child*(v, c) takes $O(\lg^\epsilon)$ time. The space complexity does not change asymptotically. □

6 Concluding Remarks

This paper has proposed linear-size data structures for compressed suffix trees which also support efficient string-depth, lowest common ancestor, and suffix link queries. The size of the data structure is only $6n + o(n)$ bits larger than that of the compressed suffix array. Any unit operation on an ordinary suffix tree can be performed in polylog(n) time. Actually we can implement a compressed suffix tree in $nH_h + 6n + o(n)$ bits, which may be smaller than the text consisting of $n \lg |\mathcal{A}|$ bits. Therefore we have solved the problem of the necessary space for suffix trees.

It is also important to mention the complexity of the working space to construct a compressed suffix tree. There are many algorithms for constructing compressed suffix arrays and trees using linear working space [19, 15, 16]. Therefore our compressed suffix trees can also be constructed using linear working space.

An open question is the following: Can we reduce the linear term $6n$ to $o(n)$?

Acknowledgments

The author would like to thank Takeshi Tokuyama, Roberto Grossi, Jesper Jansson, and Wing-Kin Sung for their valuable comments. This work is supported in part by the Grant-in-Aid of the Ministry of Education, Science, Sports and Culture of Japan.

References

[1] P. Beame and F. E. Fich. Optimal Bounds for the Predecessor Problem. In *Proc. ACM STOC*, pages 295–304, 1999.

[2] M. Bender and M. Farach-Colton. The LCA Problem Revisited. In *Proceedings of LATIN2000*, LNCS 1776, pages 88–94, 2000.

[3] D. R. Clark and J. I. Munro. Efficient Suffix Trees on Secondary Storage. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391, 1996.

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

[5] M. Farach. Optimal Suffix Tree Construction with Large Alphabets. In *38th IEEE Symp. on Foundations of Computer Science*, pages 137–143, 1997.

[6] P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *41st IEEE Symp. on Foundations of Computer Science*, pages 390–398, 2000.

[7] P. Ferragina and G. Manzini. An Experimental Study of an Opportunistic Index. In *Proc. ACM-SIAM SODA*, pages 269–278, 2001.

[8] P. Ferragina and G. Manzini. On Compressing and Indexing Data. Technical Report TR-02-01, Dipartimento di Informatica, Università di Pisa, 2002.

[9] G. H. Gonnet, R. Baeza-Yates, and T. Snider. New Indices for Text: PAT trees and PAT arrays. In W. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Algorithms and Data Structures*, chapter 5, pages 66–82. Prentice-Hall, 1992.

[10] R. Grossi, A. Gupta, and J. S. Vitter. Higher Order Entropy Analysis of Compressed Suffix Arrays. In *DIMACS Workshop on Data Compression in Networks and Applications*, pages 841–850, March 2003.

[11] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. In *32nd ACM Symposium on Theory of Computing*, pages 397–406, 2000.

[12] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching, 2001. submitted for publication.

[13] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

[14] T. Hagerup. Sorting and Searching on the Word RAM. In *Proc. STACS*, pages 366–398, 1998.

[15] W. K. Hon, T. W. Lam, K. Sadakane, and W. K. Sung. Constructing Compressed Suffix Arrays with Large Alphabets. In *Proc. ISAAC 2003*, 2003. to appear.

[16] W. K. Hon, K. Sadakane, and W. K. Sung. Breaking a Time-and-Space Barrier in Constructing Full-Text Indices. In *Proc. IEEE FOCS*, 2003. to appear.

[17] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In *Proc. the 12th Annual Symposium on Combinatorial Pattern Matching (CPM'01)*, LNCS 2089, pages 181–192, 2001.

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

[19] T. W. Lam, K. Sadakane, W. K Sung, and S. M Yiu. A Space and Time Efficient Algorithm for Constructing Compressed Suffix Arrays. In *Proc. COCOON*, pages 401–410. LNCS 2387, 2002.

[20] U. Manber and G. Myers. Suffix arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

[21] E. M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(12):262–272, 1976.

[22] J. I. Munro. Tables. In *Proceedings of the 16th Conference on Foundations of Software Technology and Computer Science (FSTTCS '96)*, LNCS 1180, pages 37–42, 1996.

[23] J. I. Munro and V. Raman. Succinct Representation of Balanced Parentheses and Static Trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[24] J. I. Munro, V. Raman, and S. Srinivasa Rao. Space Efficient Suffix Trees. *Journal of Algorithms*, 39(2):205–222, May 2001.

[25] K. Sadakane. New Text Indexing Functionalities of the Compressed Suffix Arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[26] S. Shimozono, H. Arimura, and S. Arikawa. Efficient Discovery of Optimal Word-Association Patterns in Large Text Databases. *New Generation Computing*, 18:49–60, 2000.

[27] E. Ukkonen. On-line Construction of Suffix Trees. *Algorithmica*, 14(3):249–260, September 1995.

[28] P. Weiner. Linear Pattern Matching Algorihms. In *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.