アルゴリズム 86－2
（2002．9．19）

圧縮接尾辞木のためのデータ構造

定兼 邦彦

東北大学大学院情報科学研究科システム情報科学専攻
〒 980-8579 仙台市青葉区荒巻字青葉 09
sada@dais.is.tohoku.ac.jp

接尾辞木は文字列検索のためのデータ構造であり，多くのアルゴリズムで使われている．しかし接尾辞木はサイズが大きいため，長い文字列に対して適用することが難しい．本論文では，接尾辞木のサイズを小さくした圧縮接尾辞木を提案する．サイズは長さ n の文字列に対して $O(n \log |\mathcal{A}|)$ ビット ($\mathcal{A}$ はアルファベット) となり，文字列サイズに比例する．一方，接尾辞木のサイズは $O(n \log n)$ ビットであるため，線形ではなかった．本論文の圧縮接尾辞木では，接尾辞木上の全ての演算を高速に行うことが出来る．既存手法との違いは，接尾辞リンクや最近共通祖先ノードの計算ができる点である．

Data Structures for Compressed Suffix Trees

Kunihiko Sadakane

Graduate School of Information Sciences, Tohoku University
Aramaki Aza Aoba09, Aoba-ku, Sendai 980-8579, Japan
sada@dais.is.tohoku.ac.jp

Suffix trees are data structures for string matching. Though they are used in many algorithms, it is difficult to apply them to long strings because of the size of the suffix trees. We propose *compressed suffix trees* which are space-efficient alternatives to the suffix trees. The size of the compressed suffix tree for a string of length n on an alphabet $\mathcal{A}$ is $O(n \log |\mathcal{A}|)$ bits. The most important thing is that its size is proportional to the string size. The size of conventional suffix trees is $O(n \log n)$ bits, which is not linear. Our compressed suffix trees perform all operations on suffix trees quickly. The improvement from the previous works is that we can compute suffix links and lowest common ancestors in suffix trees.

1 Introduction

Backgrounds: As the size of textual data grows, it becomes difficult to find a text we really want, and the importance of more complex queries than simple pattern matching increases. We usually use keyword search to find texts. However it is difficult to find a set of really valuable text because the result of a search will contain many texts. Thus we should consider the relation of keywords to increase the accuracy of search. We also consider more complex queries: *text data mining* [18]. Unlike keyword search, we want to find rules of keywords without specifying keywords. In these cases it is necessary to store plenty information of a text for efficient queries.

Suffix trees are popular data structures for not only pattern matching but also more complicated queries [8]. 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 be also used for more difficult problems, for example finding the longest repeated substring in linear time. To solve various problems efficiently, a suffix tree should calculate the following in constant time: (1) the suffix link of an internal node (2) the depth of an internal node (3) the lowest common ancestor (*lca*) of 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 [18]. The *lca* is necessary to compute the longest common extension of two suffixes in constant time, which can be used approximate string matching problems. The above elements are also frequently used to solve other problems.

The size of the suffix tree is $O(n)$ words for the text of length n. Since it is too large for practical use, space reduction of suffix trees [11] or space-economical alternatives, like suffix arrays [12], have been proposed in the literature. Though the suffix tree has linear size in *words*, the size is not linear to the text size, $n \log |\Sigma|$ bits, because each word occupies $\log n$ bits[1]. A lower bound on the size of data structures for string matching in linear time is proportional to the text size [3]. Therefore we want to construct suffix trees whose sizes are proportional to the text size, that is, $O(n \log |\Sigma|)$ bits for a text of length n on an alphabet Σ. It was believed that suffix arrays were difficult to compress. Therefore researches on compressing suffix trees are focused on compact representations of tree topologies of the suffix tree [2, 14]. Very recently it was proved that suffix arrays can be compressed in size linear to the text size at the cost of increasing access time from constant time to $O(\log^{\epsilon} n)$ time [7, 5, 16]. Though recently space efficient suffix trees were proposed [2, 14], their size are still not linear to 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 these do not have full functionalities of suffix trees.

Clark and Munro [2] decomposed a suffix tree into three components: tree topology, edge lengths and leaf indices (suffix arrays). Many compact representations have been proposed for the first one [9, 13, 1] and the third one [7, 5]. However there is no data structure that has good worst-case space bound for the second one. Furthermore they did not store suffix links.

In this paper we decompose the suffix tree into five components: text, tree topology, node depths, suffix array, and suffix links. The first one can be eliminated by using self-indexing data structures [5, 16]. For node depths, Clark and Munro [2] used a $\log \log \log n$ 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. [14] 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 total of edge lengths in the suffix tree is $O(n^2)$.

New results: The purpose of this paper is to construct a suffix tree whose size is linear the text length, that is, $O(n \log |\Sigma|)$ bits.

We also show how to represent the fifth component, suffix links in a suffix tree. As a byproduct of the compressed suffix array [7] and tree encoding of Munro and Raman [13], suffix links can be represented in $o(n)$ bits so that each suffix link can be computed in constant time.

[1] The base of logarithm is two throughout this paper.

The size of our compressed suffix tree is not only linear to the text size but also smaller than the text size if $|\Sigma|$ is not too small. The text and the suffix array can be compressed in $\frac{1}{\epsilon}nH_1 + O(n)$ bits where H_1 is the order-0 entropy of the text. Indices for fast queries have size $O(n) + o(nH_1)$ bits. Because $H_1 \leq \log|\Sigma|$ and other components of the suffix tree have $O(n)$ bits. the compressed suffix tree will become smaller than the original text if H_1 is small enough. Note that Grossi and Vitter [7] have already proposed compressed suffix trees. Though they can be used to find any pattern of length m from a text in $o(m)$ time, they are for only binary alphabets and their size is larger than n bits, the text size. Moreover, they can be used for only simple pattern matching. Our compressed suffix tree occupies only $\frac{1}{\epsilon}nH_1 + O(n) + o(nH_1)$ bits, and supports $O(m)$ query for a pattern of length m, $O(m\log^{\epsilon} n)$ time query for the longest common substring between the pattern and the text, and $O(n\log^{\epsilon} n)$ time traversal of all nodes of the suffix tree, where $0 < \epsilon < 1$ is any fixed constant. We use $\epsilon = 1/2, 1/4$, etc. If $\frac{1}{\epsilon}H_1 < \log|\Sigma|$, the size may be smaller than $n\log|\Sigma|$ bits, the text size.

Note that a traversal of a suffix tree using our encoding has no locality of reference. This causes a slowdown if the data structure is stored in external memory. Fortunately nowadays computers will have several gigabytes internal memory. Therefore we can construct *on-memory text databases* for huge texts. String algorithms using on-memory suffix trees will be faster than those using external memory suffix trees [4, 2]. Kurtz [11] conjectured that his space efficient suffix tree for human genome sequence of length $3 \cdot 10^9$ will require 45 gigabytes of internal memory. On the other hand, our suffix tree will occupy less than 6 gigabytes with reasonable query time. By using the proposed data structures we can construct the suffix tree for a large collection of texts, which can be used to solve many problems we are facing to.

2 Preliminaries

In this section we review suffix arrays, suffix trees and their succinct representations. Let $T[1..n] = T[1]T[2]\cdots T[n]$ be a text of length n on an alphabet Σ. We assume that $T[n] = \$$ is a unique terminator which is smaller than any other symbol. 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 called a prefix of T. The suffix array $SA[1..n]$ of T is an array of integers j that represent suffixes T_j. The integers are sorted in lexicographic order of the corresponding suffixes.

As described above, a suffix tree consists of five components: text, tree topology, edge lengths, suffix array and suffix links. The first component occupies $n\log|\Sigma|$ bits where $|\Sigma|$ is the alphabet size. The second component is efficiently stored by encoding it as balanced parentheses [9, 13, 1]. The fourth component is also efficiently stored in $O(n)$ bits [7, 5, 16]. The first component can be represented in $O(nH_k)$ bits [5] or $O(nH_1)$ bits [16], where H_k and H_1 are the order-k and the order-0 entropy of the text, respectively.

2.1 Suffix trees

The suffix tree of a text $T[1..n]$ is a compressed trie built on all suffixes of T. It has n leaves, each leaf L_i corresponds to a suffix $T_{SA[i]}$. A string is labeled to each edge. Each label is represented by a pair of start and end indices to the text. Concatenation of labels on a path from the root to a node is called path-label of the node. The path-label of a node becomes the longest common prefix of all suffixes represented by leaves under the node. The path-label of a leaf L_i corresponds to the suffix $T_{SA[i]}$. 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. The label on a edge is represented by starting and ending indices of the text T to achieve $O(n)$ words space because the total of edge lengths is $O(n^2)$. A pattern $P[1..m]$ can be found in $O(m)$ time from $T[1..n]$ by using the suffix tree of T if any child of a node in the suffix tree can be found in constant time. 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.

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 necessary to not only construct a suffix tree in linear time but also solve problems efficiently, for example finding the longest

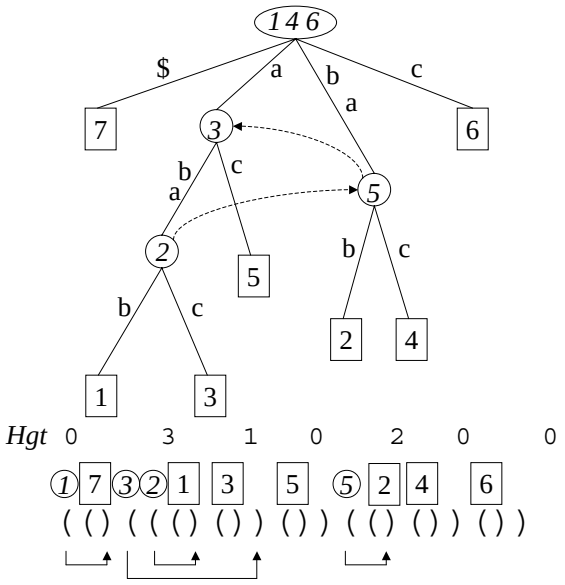

Figure 1: The suffix tree for "ababac$," and its balanced parentheses representation. Suffix links are shown in dotted line.

common substring of two strings in linear time. Note that Weiner's suffix tree construction algorithm [19] uses suffix links in the opposite direction; from $sl(v)$ with path-label α to the node v with path-label $x\alpha$. Thus each internal node may have $|\Sigma|$ suffix links. We call this type of suffix links as Weiner-type suffix links in this paper.

2.2 Balanced parentheses representations of trees

An n-node rooted ordered tree can be encoded in $2n+o(n)$ bits with various constant time navigational operations [9, 13, 1]. The tree is encoded into n 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.

Navigational operations on the tree are defined by $rank_p$, $select_p$, *findclose*, *enclose*, *double_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. The function *double_enclose*(x, y) finds a parenthesis pair that most tightly encloses parenthesis pairs whose open parentheses appear in position x and y. All functions take constant time. Supported navigational operations are *left_child* and *right_child* of any node in a binary tree, *parent*, *lca*, etc. The *parent* operation is defined by the *enclose* function. The *lca* operation returns the lowest common ancestor of two nodes in a tree and it is defined by the *double_enclose* function. These operations take constant time.

Since the number of nodes in a suffix tree is at most $2n-1$ for a text of length n, the suffix tree of the text can be encoded in at most $4n + o(n)$ bits.

2.3 Leaf indices

If leaves of a suffix tree are numbered by a preorder traversal of the tree, indices to the text stored in the leaves coincide with the suffix array [12] of the text. Since it is an array of the indices, it occupies $n \log n$ bits. Any pattern of length m can be found in $O(m \log n)$ time by a binary search using the suffix array and the text, or in $O(m + \log n)$ time using *lcp* information of suffixes.

The compressed suffix array of Grossi and Vitter [7] reduces the size of the suffix array for binary alphabets from $n \log n$ bits to $O(n)$ bits. Instead access time for an entry changes from constant time to $O(\log^{\epsilon} n)$ time where ϵ is any constant in $0 < \epsilon < 1$. Sadakane [16] showed that the size of the compressed suffix array is $O(nH_1)$ bits for arbitrary alphabets where H_1 is the order-0 entropy of the text. The compressed suffix array stores a function Ψ instead of SA. The function $\Psi[i]$ in the compressed suffix array is defined as follows:

Definition 2

$$\Psi[i] \equiv \begin{cases} i' \text{ s. t. } SA[i'] = SA[i] + 1 & (SA[i] < n) \\ i' \text{ s. t. } SA[i'] = 1 & (SA[i] = n) \end{cases}$$

In other words, $\Psi[i] = SA^{-1}[SA[i] + 1]$ unless $SA[i] = n$. Each value of Ψ can be computed in constant time.

The FM-index [5] is a *self-indexing* data structure, that is, it can be used to find a pattern without using the text itself. This means that we can find a pattern from a compressed text because the size of the FM-index will be smaller than text size $n \log |\Sigma|$. The number of occurrences of a pattern of length m can be found in

$O(m)$ time. Computing the position of each occurrence in the text takes additional $O(\log^{\epsilon} n)$ time.

Sadakane [16] showed that the compressed suffix array can be modified to form a self-indexing data structure. Its size is expressed by the order-0 entropy of the text. Therefore it will be smaller than the original text. A character $T[j]$ in a text can be extracted in constant time if the lexicographic order i of the suffix $T_j = T[j..n]$ is given. The algorithm becomes as follows. We use a bit-vector $D[1..n]$ so that $D[i] = 1$ if $T[SA[i]] \neq T[SA[i-1]]$ or $i = 1$. Since the suffixes are lexicographically sorted, their first characters are also sorted alphabetically. Thus the number of 1's in $D[1..i]$ represents the number of distinct characters in the text which are smaller than $T[SA[i]]$. It is computed in constant time using the rank function. This number can be easily converted to character codes. The bit-vector occupies $n + o(n)$ bits or about $|\Sigma| \log \frac{n}{|\Sigma|} + o(n)$ bits [15].

Sadakane also showed that the inverse array of the suffix array can be expressed by additional $n + o(n)$ bits data structure to the original compressed suffix array with $O(\log^{\epsilon} n)$ access time. The inverse suffix array is important to traverse a suffix tree quickly without using the original text. Our algorithms use the lexicographic orders of suffixes to represent them. Therefore to obtain the k-th character on a edge of the suffix tree it is necessary to compute $SA^{-1}[SA[i] + k]$. On the other hand, if we do not have the inverse function, it takes $O(k)$ time.

2.4 Longest common prefix information

Longest common prefix (*lcp*) information is used to speed up the search using suffix arrays. It represents the length of the longest common prefix of two strings s, t and is expressed by $lcp(s, t)$. We store it in an array $Hgt[1..n]$ defined as follows:

Definition 3

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

This array has size $n \log_2 n$ bits. However it can be compressed into $2n + o(n)$ bits provided the suffix array or the compressed suffix array [17].

Kasai et al. [10] showed that a bottom-up traversal of a suffix tree can be simulated by using only the suffix array and *Hgt* array. 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.

The *Hgt* array is also used to represent the depths of nodes of a suffix tree. The depth of any node is computed by using the (compressed) suffix array, the *Hgt* array, and the balanced parentheses representation of the suffix tree.

3 Navigating Operations

In this section we show algorithms for navigating compressed suffix trees. It was shown [13] that a node of a tree is represented by a pair of parentheses '$(\ldots)$' in a nested parentheses sequence. We represent a node by its *preorder*. Therefore we first show how to convert the *preorder* of a node to the position of the open parenthesis that represents the node. Not only the *preorder* we also use the *inorder* of a node. We also show how to convert them.

Next we show an algorithm to compute the *lowest common ancestor* (*lca*) between two nodes in a tree. We also show an algorithm to compute the string-depth of a node in a suffix tree. Finally we show how to compute the suffix links in a suffix tree.

3.1 How to represent a node

As described above, a node of a suffix tree is represented by a pair of parentheses in a nested parentheses sequence [13]. The parentheses sequence P is produced during a preorder traversal of the tree. Therefore the *preorder* p of a node and the position i of the open parenthesis of the node in P can be easily converted from each other in constant time:

$$\begin{aligned} p &= rank_{(}(P, i) \\ i &= select_{(}(P, p). \end{aligned}$$

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

Definition 4 *The inorder rank of an internal node v is assigned as the number of visited internal nodes, including v, in the 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 1 *Let i be the position in a parentheses sequence representing an internal node, and let q be the inorder of the node. Then i and q can be converted from each other in constant time by*

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

Note that if a node has two or more inorders, the algorithm returns the smallest one.

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

From the definition of *inorder*, the *inorder* q of a node s is the number of times that during the preorder traversal from the root to s 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 $x = select_{)(}(P,q)$. Then the open parenthesis in position $x+1$ represents a child of the node s. Because we want to know the parenthesis corresponds to s, we compute $parent(P, x+1)$, which returns the position of open parenthesis of s. □

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.

3.2 How to associate information to nodes

We can associate additional information with nodes of a suffix tree. 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 between 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 a little bit 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 only leaves, we can use the lexicographic orders of nodes as indices. There exists obviously a one-to-one mapping.

3.3 How to find a child node

To traverse the path in the suffix tree that corresponds to a pattern P, we need to find a node w of an internal node v such that the edge-label from v to w begins by a certain character.

We construct space-efficient suffix trees which use the Pat tree [6]. The node corresponding to P can be found in $O(|P| \log |\mathcal{A}| \log^{\epsilon} n)$ time. The size of the tree is $|CSA| + n(6 + \log\log|\mathcal{A}| + o(1))$ bits.

We can also construct another type of suffix tree called Weiner-type suffix trees [19]. The size is $|CSA| + n(6 + o(1))$ bits and The node corresponding to P can be found in $O(|P| \log^{\epsilon} n)$ time. We omit the details.

3.4 How to compute string-depths of nodes

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

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

Recall that the label can be represented without using the text T.

3.5 How to compute *lca*

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

Lemma 2 *Let i, j be the positions in a parentheses sequence P representing nodes v and w respectively. Assume that v is not an ancestor of w, and vice versa. Then the position l of open parenthesis representing the node $lca(v, w)$ is computed in constant time by*

$$l = parent(P, \mathrm{RMQ}_{P'}(i, j) + 1)$$

where P' is a sequence of integers such that $P'[i] = rank_{(}(P, i) - rank_{)}(P, i)$.

Note that we need not to store P' explicitly. Each element of it can be computed in constant time using $o(n)$ bits indices.

Proof: 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 $lca(v, w)$. Therefore the position of open parenthesis of the node $lca(v, w)$ is comuted by $parent(P, m+1)$. □

Note that we can easily check whether v is an ancestor of w or not. Assume that $i < j$. Then v is an ancestor of w if and only if $findclose(P, i) > findclose(P, j)$.

3.6 How to compute suffix links

Lemma 3 *Let i be the position in a parentheses sequence P representing a non-root node v. Then the position j of open parenthesis representing the node $sl(v)$ is computed in constant time by*

$$\begin{aligned} x &= rank_{(}(P, i-1) + 1 \\ y &= rank_{(}(P, findclose(P, i)) \\ x' &= \Psi[x] \\ y' &= \Psi[y] \\ j &= lca(select_{(}(P, x'), select_{(}(P, y')). \end{aligned}$$

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 $x = rank_{(}(P, i-1) + 1$. 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 $y = rank_{(}(P, findclose(P, i))$. The leaves represent suffixes $T_{SA[x]}$ and $T_{SA[y]}$. From the definition of Ψ $leaf(x')$ and $leaf(y')$ represent suffixes $T_{SA[x']} = T_{SA[x]+1}$ and $T_{SA[y']} = T_{SA[y]+y}$, respectively. Let $l = lcp(T_{SA[x]}, T_{SA[y]})$. Then l is equal to the string-depth of node v because $leaf(x)$ and $leaf(y)$ are the leftmost and the rightmost descendants of v. Obviously $l-1 = lcp(T_{SA[x']}, T_{SA[y']})$ holds. Then the node $lca(select_{(}(P, x'), select_{(}(P, y'))$ has string-depth $l-1$, which means it is $sl(v)$. □

We use the tree encoding of Munro and Raman [13] because it supports constant time *lca* (lowest common ancestor) queries.

3.7 Typical applications of suffix trees

A typical algorithm that cleverly uses suffix links is the algorithm to find the longest common substring of a pattern P of length m and a text T of length n. Because our compressed suffix tree has full functions of the suffix tree, the algorithm runs in $O(m \log |\Sigma| \log^{\epsilon} n)$ time using the compressed suffix tree of T.

4 Concluding remarks

We have proposed two types of compressed suffix tree, both will be smaller than the text size. They can be used without the original text and support full functions of suffix trees. Their size and time for typical operations are summarized in Table 1, where T_P is time to find the longest common substring between a pattern of length m and a text T of length n and T_T is time for bottom-up traversal of the suffix tree of T. These suffix trees are useful to perform complicated queries on a large collection of texts.

References

[1] D. Benoit, E. D. Demaine, J. I. Munro, and V. Raman. Representing Trees of Higher Degree. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures (WADS'99)*, LNCS 1663, pages 169–180, 1999.

[2] 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.

[3] 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*

Table 1: The size and query time of compressed suffix trees

size (bits)	T_P	T_T	type
$\|CSA\| + n(6 + \log\log\|\Sigma\| + o(1))$	$O(m \log \|\Sigma\| T_{SA})$	$O(nT_{SA})$	McCreight
$\|CSA\| + o(nH_1) + n(6 + o(1))$	$O(mT_{SA})$ if $\|\Sigma\| = \log^{O(1)} n$	$O(nT_{SA})$	Weiner

Symposium on Discrete Algorithms, pages 289–294, 2001.

[4] P. Ferragina and R. Grossi. The String B-Tree: a New Data Structure for String Search in External Memory and its Applications. *Journal of the ACM*, 46(2):236–280, 1999.

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

[6] 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.

[7] 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.

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

[9] G. Jacobson. Space-efficient Static Trees and Graphs. In *30th IEEE Symp. on Foundations of Computer Science*, pages 549–554, 1989.

[10] 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.

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

[12] 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.

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

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

[15] R. Pagh. Low redundancy in static dictionaries with $O(1)$ worst case lookup time. In *Proceedings of ICALP'99*, LNCS 1644, pages 595–604, 1999.

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

[17] K. Sadakane. Succinct Representations of *lcp* Information and Improvements in the Compressed Suffix Arrays. In *Proc. ACM-SIAM SODA 2002*, pages 225–232, 2002.

[18] 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.

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