Succinct Data Structures for Flexible Text Retrieval Systems[1]

Kunihiko Sadakane

Department of Computer Science and Communication Engineering, Kyushu University, Fukuoka, Japan.

Abstract

We propose succinct data structures for text retrieval systems supporting document listing queries and ranking queries based on the *tf*idf* (term frequency times inverse document frequency) scores of documents. Traditional data structures for these problems support queries only for some predetermined keywords. Recently Muthukrishnan proposed a data structure for document listing queries for arbitrary patterns at the cost of data structure size. For computing the *tf*idf* scores there has been no efficient data structures for arbitrary patterns.

Our new data structures support these queries using small space. The space is only $2/\epsilon$ times the size of compressed documents plus $10n$ bits for a document collection of length n, for any $0 < \epsilon \leq 1$. This is much smaller than the previous $\mathrm{O}(n \log n)$ bit data structures. Query time is $\mathrm{O}(m + q \log^{\epsilon} n)$ for listing and computing *tf*idf* scores for all q documents containing a given pattern of length m. Our data structures are flexible in a sense that they support queries for arbitrary patterns.

Key words: Information Retrieval, Tf*idf, Succinct data structures, Suffix trees, Inverted files

Preprint submitted to Journal of Discrete Algorithms *8 January 2006*

1 Introduction

Text retrieval systems are now indispensable to search for important documents from a large collection of text documents such as Web, genome sequence, etc. A text retrieval system stores a set of documents, and if a keyword is given by a user, it will return a set of documents each of which contains the keyword. This is a basic function of text retrieval systems and formulated as follows:

Problem 1 (Document Listing Problem [1]) *We are given a set of k text documents $d_1, d_2, \ldots, d_k$ with total length n, which may be preprocessed. The document listing query for a pattern p is to return the set of all documents in which p is present. That is, the output is $\{j | d_j[i..i+m-1] = p \text{ for some } i\}$ where m is the length of p.*

Though this problem is very basic, there was no efficient data structures eligible for arbitrary patterns before [1]. Traditional algorithms use the inverted file [2] for preprocessing. It partitions documents into words and creates an index for efficient search. As a result, it does not support the listing queries for arbitrary patterns, which causes a loss of accuracy of search for languages without word boundaries such as Japanese or Chinese. Muthukrishnan [1] proposed a data structure for the document listing problem for arbitrary patterns. It can perform a query in $\mathrm{O}(|p| + q)$ time after $\mathrm{O}(n)$ time preprocessing where $|p|$ denotes the length of the pattern p, q is the number of documents containing p, and n is the summation of the lengths of all documents.

Email address: `sada@csce.kyushu-u.ac.jp` (Kunihiko Sadakane).

[1] A preliminary version of this paper appeared in the proceedings of ISAAC, LNCS 2518, pp. 14–24, 2002.

A drawback of Muthukrishnan's data structure is its size. The size is $\mathrm{O}(n \log n)$ bits and in practice more than $20n$ bytes. On the other hand, the size of the documents is n bytes if alphabet size is 256, and the size of the inverted file for them is less than n bytes. Therefore the size of the data structure is not practical. The first contribution of this paper is to develop an alternate data structure to Muthukrishnan's one whose size is close to the document size.

Theorem 1 *After* $\mathrm{O}(n)$ *time preprocessing, the document listing problem is solved in* $\mathrm{O}(\mathit{Search}(p)+q\cdot \mathit{Lookup}(n))$ *time on word RAM using a data structure of size* $|CSA| + 4n + \mathrm{o}(n) + \mathrm{O}(k \log \frac{n}{k})$ *bits.*

Note that $\mathit{Search}(p)$ is the time to find a pattern p in the text collection of total length n, $\mathit{Lookup}(n)$ is the time to compute an entry of the suffix array and its inverse array, and $|CSA|$ is the total size of compressed texts, which will be described in Section 2.3. These values depend on the implementation of the compressed suffix arrays [3–7]. Normally $|CSA|$ is smaller than n bytes, the text size. That is, the size of the new data structure is almost the same as the text size. If we use an implementation of the compressed suffix arrays [5], we have the following result:

Corollary 2 *After* $\mathrm{O}(n)$ *time preprocessing, the document listing problem is solved in* $\mathrm{O}(|p| + q \log^{\epsilon} n)$ *time on word RAM using a data structure of size* $\mathrm{O}(\frac{1}{\epsilon} n (H_0 + 1)) + \mathrm{O}(k \log \frac{n}{k})$ *bits for any* $0 < \epsilon \leq 1$ *if the alphabet size is* $\sigma = \mathrm{polylog}(n)$ *where* H_0 *is the order-*0 *entropy of the texts.*

The document listing query is not enough for standard text retrieval systems because the answer contains a lot of documents from which users have to find important documents. The most common definition of importance of documents involves the *tf*idf* scores [8]. For a query for a set of patterns

$p_1, p_2, \ldots, p_\ell$, the *tf*idf* score of a document d is defined as $\sum_{i=1}^{\ell} tf(p_i, d) \cdot idf(p_i)$ where $tf(p_i, d)$ is the number of occurrences of p_i in a document d, $idf(p_i)$ is defined as $\log \frac{k}{df(p_i)}$, k is the number of documents in the database, and $df(p_i)$ is the number of documents in the database containing p_i. The larger the score is, the more important the document is. To compute this, text retrieval systems use again the inverted file. As a result, scores can be computed for only predetermined words. The second contribution of this paper is to develop a succinct data structure for computing the *tf*idf* scores for arbitrary patterns. The new data structure solves the following problem:

Problem 2 (TF*IDF Problem) *We are given a set of k text documents $d_1, d_2, \ldots, d_k$ with total length n, which may be preprocessed. The TF*IDF query $tf * idf(p)$ is to compute $q = df(p)$, and $tf(p, d)$ for all documents d which contain p.*

There has been no efficient data structures solving this problem for any pattern even if $O(n \log n)$-bit space is used. Our new data structure is not only applicable to any pattern, but also space efficient.

Theorem 3 *After $O(n)$ time preprocessing, the TF*IDF problem is solved in $O(Search(p) + q \cdot (Lookup(n) + \log\log q))$ time on word RAM using a data structure of size $2|CSA| + 10n + o(n) + O(k \log \frac{n}{k})$ bits. If only $df(p)$ is necessary, it can be computed in $O(Search(p))$ time.*

The query time is also expressed as $O(|p| + q \log^{\epsilon} n)$, as in Corollary 2 using an appropriate compressed suffix array.

The rest of the paper is organized as follows. In Section 2 we describe some previous data structures used in our data structures. In Section 3 we propose a succinct data structure for range minimum queries, which is of independent interest and used to solve the above problems. In Section 4 we propose a succinct data structure for the document listing problem. In Section 5 we propose a succinct data structure to compute *tf*idf* scores for arbitrary patterns. Section 6 summarizes the results.

2 Preliminaries

2.1 Computation Models

We consider the word RAM as the computation model. The CPU has pointers of $\mathrm{O}(\lg n)$ bits[2] and can perform logical and arithmetic operations on two $\mathrm{O}(\lg n)$-bit integers in constant time. The CPU also can read/write $\mathrm{O}(\lg n)$ bits of memory in constant time.

We measure the size of data structures by the number of bits used. For example, a length-n array of integers in range $[1, n]$ is of size $n \lg n$ bits. On the other hand, a length-n text on alphabet $\mathcal{A}$ has size $n \lg \sigma$ bits where σ is the alphabet size. We assume that σ is a power of two and $\sigma = \mathrm{o}(n)$. If $\sigma = 256$, the text size is $8n$ bits. Normally we need $\mathrm{O}(n \lg n)$ bits for data structures to search the text because we use $\mathrm{O}(n)$ number of pointers, each of which occupies $\mathrm{O}(\lg n)$ bits. This is much larger than the text size. Therefore we want to reduce the data structure size to $\mathrm{O}(n \lg \sigma)$ or less, which is the main topic of this paper.

[2] Let lg denote the logarithm of base two.

Let $T[1..n] = T[1]T[2] \cdots T[n]$ be a text of length n on an alphabet $\mathcal{A}$. The j-th suffix of T is defined as $T[j..n] = T[j]T[j+1] \ldots T[n]$ and denoted by T_j. A substring $T[1..l]$ is called a prefix of T. The suffix array [9] of T is an array $SA[1..n]$ of integers j that represent suffixes T_j. The integers are sorted in lexicographic order of the corresponding suffixes. The suffix tree [10] of T is a compressed trie built on all suffixes of T. Each edge has a label which is a substring of T. The suffix tree has n leaves, and the concatenation of edge labels on the path from the root to each leaf is coincident with a suffix T_j. Edge labels between internal edges are represented by pointers to substrings of T, and labels for leaves are represented by pointers to suffixes. Therefore leaves are identical to the suffix array of T. Let *leaf*(i) denote the leaf that corresponds to the i-th suffix in lexicographic order, which is $T_{SA[i]}$.

Any pattern p in T is represented uniquely by a prefix of a path from the root node to a node v of the suffix tree of T. Therefore the existence of p in T is determined in $\mathrm{O}(|p|)$ time. On the other hand, this is solved in $\mathrm{O}(|p| \lg n)$ time using the suffix array. The size of the suffix tree is $\mathrm{O}(n \lg n)$ bits, and that of the suffix array is exactly $n \lg n$ bits. Both are not linear to the text size $n \log \sigma$. They can be constructed in $\mathrm{O}(n)$ time [11].

If we are given a set of k text documents $d_1, d_2, \ldots, d_k$, we concatenate them into a text T and construct the *generalized suffix tree* [12] for T, denoted by GST. The GST is the compressed trie of all suffixes of the k documents. To make any leaf have a unique label, we append a unique terminator for each documents, that is, we let $T = d_1\$_1 d_2\$_2 \cdots d_k\$_k$. We assume that $\$_1 < \$_2 <$

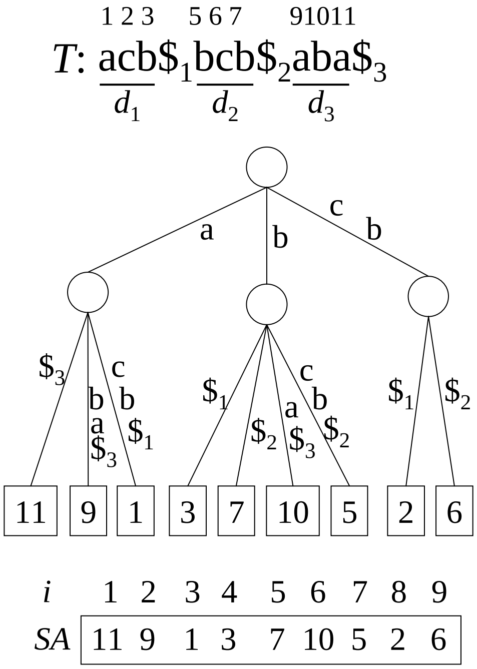


Fig. 1. The generalized suffix tree and the suffix array for "acb$_1$bcb$_2$aba$_3$"

$\cdots < \$_k$, and $\$_k$ is smaller than any character in $\mathcal{A}$. Figure 1 shows an example of the generalized suffix tree and the suffix array of the concatenated text.

2.3 Succinct Data Structures

We use several basic data structures to reduce the size of the data structure for the document problems. A basic one is the succinct representation of trees [13]. An n-node tree is represented by a nested parenthesis sequence P of length $2n$. The sequence is defined from the tree as follows. We traverse the tree from the root in a depth-first manner. When we go down an edge we put an open parenthesis '`(`,' and when we go up an edge we put a close parenthesis '`)`' to P. Then a traversal on the tree can be simulated by a traversal on the sequence. An example of the sequence is depicted in Fig. 3.

To make the traversal quick, we use auxiliary data structures that support the following functions. The function $rank_p(P, i)$ returns the number of occurrences of pattern p up to the position i in a string P, where p is for example '`()`.' The function $select_p(P, i)$ returns the position of i-th occurrence of pattern p. Both functions take constant time using auxiliary data structures of size o(n) bits [14]. By using these data structures, tree traversal operations such as finding the parent, the first child, the next sibling, and computing the number of leaves below a node, are done in constant time on word RAM.

The (generalized) suffix tree and the suffix array can be compressed. We use the compressed suffix array [3] and its variations. The suffix array is compressed from $n \lg n$ bits to $\mathrm{O}(n \log \sigma)$ bits. Therefore the size of the compressed suffix array is proportional to the text size. Each element $SA[i]$ is decompressed in polylog(n) time. There are several different implementations of the compressed suffix arrays, but they support the following operations:

- Given i, compute $SA[i]$ and $SA^{-1}[i]$ in $Lookup(n)$ time,
- Given a pattern p, compute the interval $[l, r]$ of the suffix array in which prefixes of all suffixes in the interval match with the pattern, that is, $T[SA[i]..SA[i]+|p|-1] = p$ for any $i \in [l, r]$, in $Search(p)$ time.

Note that by using compressed suffix arrays we can extract any portion of the text. This means that we need not to store the text itself. Table 1 summarizes variations of compressed suffix arrays.

We also use a succinct data structure for computing *lowest common ancestor* (*lca*) between two nodes of a tree [5], which is based on the algorithm of Bender and Farach-Colton [16]. Let $lca(v, w)$ be the lowest common ancestor of nodes v and w. After O(n) time preprocessing to an n-node tree, $lca(v, w)$ is computed in constant time for any nodes v and w using a data structure

Table 1

The size and query time of compressed suffix arrays. ϵ is an arbitrary constant such that $0 < \epsilon \leq 1$. σ is the alphabet size of the text. H_k is the order-k entropy of the text.

size (bits)	*Lookup*(n)	*Search*(p)	references
$\mathrm{O}(\frac{1}{\epsilon} n \lg \sigma)$	$\mathrm{O}(\log_\sigma^\epsilon n)$	$\mathrm{O}(\lvert p\rvert / \log_\sigma n + \log_\sigma^\epsilon n)$	[3]
$\mathrm{O}(\frac{1}{\epsilon} n(H_0 + 1))$	$\mathrm{O}(\lg^\epsilon n)$	$\mathrm{O}(\lvert p\rvert)$	[5] $(\sigma = \mathrm{polylog}(n))$
$nH_k + \mathrm{O}(n/\lg^\epsilon n)$	$\mathrm{O}(\lg^{1+\epsilon} n)$	$\mathrm{O}(\lvert p\rvert)$	[15] $(\sigma = \mathrm{polylog}(n))$
$\mathrm{O}(nH_k \lg^\gamma n + n/\lg^\epsilon n)$	$\mathrm{O}(1)$	$\mathrm{O}(\lvert p\rvert)$	[7] $(\sigma = \mathrm{polylog}(n), \gamma > 0)$

of size $2n + \mathrm{o}(n)$ bits. In this paper we propose a data structure for range minimum queries on arbitrary arrays using the data structure for *lca* queries.

3 Succinct Data Structure for Range Minimum Query

In this section we propose a succinct data structure for range minimum queries on arbitrary arrays, which will be used in the proposed algorithms. First we define the problem.

Problem 3 (Range Minimum Query) *Given indices* l *and* r *of an array* $C[1, n]$*, the range minimum query* $\mathrm{RMQ}_C(l, r)$ *returns the index of the smallest element in the subarray* $C[l..r]$*. If there is a tie-breaking we choose the leftmost one.*

It is known that a query can be done in constant time using $\mathrm{O}(n \log n)$-bit space after $\mathrm{O}(n)$ time preprocessing [16]. Here we propose a succinct data structure, which is summarized as follows:

Theorem 4 *For an array C of n elements, a range minimum query is done in constant time using a data structure of size $4n + \mathrm{o}(n)$ bits after $\mathrm{O}(n)$ time preprocessing.*

Note that we do not store the array C itself. Therefore we can find only the index i of the minimum element $C[i]$.

The range minimum query is reduced to finding the *lca* between two nodes in a tree [16]. Consider an imaginary binary tree storing pairs $(i, C[i])$ for all $1 \leq i \leq n$, which will be converted into another data structure without the array C. The root node of the tree stores $(x, C[x])$ where $C[x]$ is the minimum in $C[1..n]$. If there are more than one minimum values, we determine the order by their indices. Therefore there always exists a unique minimum. The left subtree stores $(i, C[i])$ for $1 \leq i \leq x - 1$ and the right subtree stores those for $x + 1 \leq i \leq n$ recursively. This tree is called a Cartesian tree and is constructed in $\mathrm{O}(n)$ time. Then $\mathrm{RMQ}_C(l, r)$ is equal to the index stored in the *lca* between two nodes storing $C[l]$ and $C[r]$. Figure 2 shows an example of the Cartesian tree for array C. Each pair $(i, C[i])$ is stored in a node where i is put in the upper half of the node, and $C[i]$ is in the lower.

We consider succinct representations for this tree. We can encode it in $2n$ bits by a parenthesis sequence P. See also Figure 2 for an example. Sadakane [5] showed that for an n-node tree encoded in the parenthesis sequence, *lca* is computed in constant time on word RAM using an auxiliary data structure of $\mathrm{o}(n)$ bits provided that the *preorders* of the nodes are given. Here the preorder of a node is defined as the number of nodes visited before arriving the node in the preorder traversal of the tree. However, this data structure is not directly applied for succinct data structures for range minimum queries because we

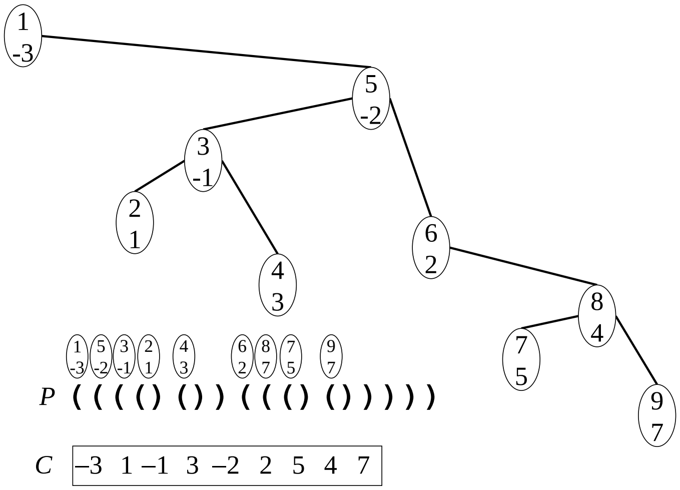


Fig. 2. The Cartesian tree and its balanced parenthesis representation of array C

cannot store the *preorders* for all nodes which require $O(n \lg n)$ bits. Instead we use a different tree so that the *preorder* of the node for $C[i]$ is computed from i in constant time.

The reason that the *preorder* of a node cannot be computed in constant time in the original Cartesian tree is that a node of the tree may not have the left or the right child. If we store $C[i]$ in internal nodes and encode the tree into the parenthesis sequence, we cannot distinguish between a node having only the left child and that having only the right child. For example, the node with index 6 in Figure 2 has only the right child. However in the parenthesis sequence we do not know whether the node has the right child or the left child. To solve the problem Munro and Raman [13] use an isomorphism between a binary tree and an ordered tree. Although their method can encode the tree in $2n + o(n)$ bits, it is not applicable to our problem because we cannot compute the *lca* of nodes.

In this paper we propose another representation of a binary tree. We change the tree into ternary by adding a new leaf node to each node. For an internal

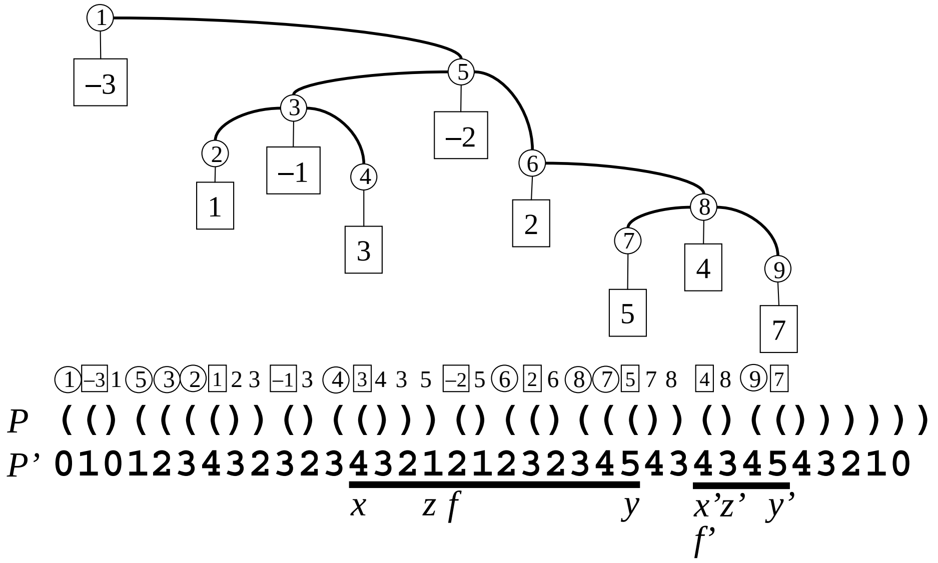

1
−3
5
3
−2
2
−1
4
1
3
6
2
8
7
4
9
5
7
①−3 1 ⑤③② 1 2 3 −1 3 ④ 3 4 3 5 −2 5 ⑥ 2 6 ⑧⑦ 5 7 8 4 8 ⑨ 7
P (() (((()) () (())) () (() ((()) () (())))))
P’ 010123432323432121232345434345432 10
x z f y x’z’ y’
f’

Fig. 3. A new representation of array C and its parenthesis encoding.

node of the original tree, the new leaf is added between the left and the right children. Then we can distinguish them and compute *lca* between nodes. Furthermore, we can compute the *inorder* of each leaf from the sequence because for each leaf its *preorder* and *inorder* coincide. We will describe the details.

We temporarily construct the Cartesian tree for C and add new nodes to it. We add a new leaf to each internal node as a middle child. Let us denote the tree by M. Each internal node stores the index i, and its middle child stores $C[i]$. Then the tree M is represented in $4n$ bits because it has n internal nodes and n leaves. Each node is represented by the position of an open parenthesis in P. Figure 3 shows an example.

To solve a range minimum query $\mathrm{RMQ}_C(l, r)$ it is necessary to convert an index i to the element $C[i]$ into the position e of the open parenthesis in P of the leaf whose parent has label i. Because each leaf has no child, it is represented by `()` in P. Moreover, because leaves appear in P in the order of depth-first search, the order of the leaves in P is determined by the parents' labels. Therefore e and i are converted to each other as follows:

$$\begin{aligned} e &= \mathit{select}_{()}(P, i) \\ i &= \mathit{rank}_{()}(P, e). \end{aligned}$$

To find the *lca* between two nodes in a parenthesis sequence, we consider an imaginary integer array P'. We define $P'[i] = \mathit{rank}_{(}(P, i) - \mathit{rank}_{)}(P, i) - 1$. In other words, $P'[i]$ is the depth of a node with *preorder i*. We do not store P' explicitly because each value of P' is computed in constant time from P and an auxiliary data structure of size $\mathrm{o}(n)$ bits [13]. Then an *lca* query is reduced to range minimum query on P' where the difference between two adjacent elements is always 1 or -1. We call this query $\mathrm{RMQ}^{\pm}_{P'}$. Now we reduced the range minimum query on C with n elements into $\mathrm{RMQ}^{\pm}_{P'}$ with $2n$ elements, which is solved in constant time using $4n + \mathrm{o}(n)$ bits [5].

The index i of the minimum element $C[i]$ in $C[l..r]$ can be found in constant time as follows:

(1) $x = \mathit{select}_{()}(P, l)$, $y = \mathit{select}_{()}(P, r)$

(2) $z = \mathrm{RMQ}^{\pm}_{P'}(x, y)$

(3) if $P[z + 2] =$ ')' then $f = z + 1$ else $f = z - 1$

(4) $i = \mathit{rank}_{()}(P, f)$.

Step 3 finds the position f of the open parenthesis that represents the middle leaf. $P[z + 1]$ is the open parenthesis of a child. If it is a leaf $P[z + 2]$ is the close parenthesis, otherwise $P[z + 2]$ is the open parenthesis of its child. In the latter case $P[z]$ is the close parenthesis and $P[z - 1]$ is the open parenthesis of the leaf.

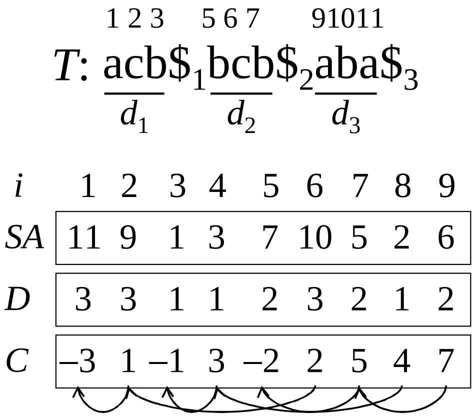
For example, to find the minimum in $C[4..7]$ we first compute positions x and y of '`()`' in P corresponding to $C[4]$ and $C[7]$ (see Fig. 3). Then we compute the position z of the minimum element in $P'[x..y]$. In this case $P[z]$ represents the close parenthesis of the left child of $C[5]$ and $P[z+1]$ represents the open parenthesis of the middle child. Therefore the index i of the minimum element $C[i]$ is equal to the number of '`()`' in $P[1..z+1]$. In another example, to find the minimum in $C[8..9]$, we find the position z' of the minimum element in $P'[x'..y']$ where x' and y' correspond to $C[8]$ and $C[9]$, respectively. Then $P[z']$ is the close parenthesis of the middle child of $C[8]$.

4 Succinct Data Structure for the Document Listing Problem

We propose a succinct data structure for the Problem 1 (Document Listing Problem), whose properties are summarized in Theorem 1. Our data structure is based on Muthukrishnan's original one. Therefore we first describe it, then we give our new data structure and query algorithms.

4.1 Original Algorithm for Document Listing Problem

The original algorithm and data structure for the document listing problem [1] is as follows. Two integer arrays $C[1..n]$ and $D[1..n]$ are defined as follows. Let us define $D[i] = c$ if the suffix $T_{SA[i]}$ is contained in document d_c, and define $C[i] = j$ where j is the largest index such that $j < i$ and $D[j] = D[i]$. If such j does not exist $C[i] = -1$. To store the set of documents, the generalized suffix tree *GST* is used. An example of the arrays is shown in Figure 4. The *GST* for this example is shown in Figure 1. These data structures are constructed



i	1	2	3	4	5	6	7	8	9
SA	11	9	1	3	7	10	5	2	6
D	3	3	1	1	2	3	2	1	2
C	−3	1	−1	3	−2	2	5	4	7

Fig. 4. Original data structure for document listing problem.

in O(n) time.

For a query for a pattern p we first find the interval $[l, r]$ of the lexicographic order of suffixes which match with p by using GST. Then we call $DLP(l, l, r)$ shown in Fig. 5. By using this algorithm, we can output document ID's without duplication in O(q) time where q is the size of output, that is, the number of documents containing p. The correctness of the algorithm is explained as follows. The array C is regarded as a set of linked lists each of which corresponds to a document. Therefore we enumerate the first element of each list which appears in $C[l..r]$. To do so, we find the minimum $C[x]$ in $C[l..r]$ in constant time. If $C[x] \geq l$, $C[x]$ is not the first element of a list and thus the algorithm terminates, otherwise outputs $D[x]$ and continues to find the mini-

```
Procedure $DLP(s, l, r)$
$x :=$ RMQ$_C(l, r)$
if $C[x] < s$ then
    output $D[x]$
    $DLP(s, l, x - 1)$
    $DLP(s, x + 1, r)$
```

Fig. 5. Pseudo code for original document listing algorithm.

mum in $C[l..x-1]$ and $C[x+1..r]$ recursively. Because the number of times that range minimum query is performed is at most $2q$, the algorithm runs in $\mathrm{O}(|p|+q)$ time. The space complexity is $\mathrm{O}(n \lg n)$ bits.

4.2 New Algorithm

We store the arrays succinctly. The value $D[i]$ is calculated in constant time from the suffix array $SA[i]$ using additional $\mathrm{O}(k \log \frac{n}{k})$ bit space as follows. We consider the ordered set D' of positions of the first character of each document d_i $(i = 1, 2, \ldots, k)$. Then $D[i]$ is equal to the number of elements in D' which are no greater than $SA[i]$, that is, $D[i] = |\{x \in D' | x < SA[i]\}|$. Therefore computing $D[i]$ is a kind of rank query and done in constant time using a data structure of size $\mathrm{O}(k \log \frac{n}{k})$ bits [17] if we know $SA[i]$. Let $D'(SA[i])$ denote $D[i]$. The computation of $SA[i]$ takes $\mathit{Lookup}(n)$ time by using the compressed suffix array. For range minimum queries in the array C we use the data structure in Section 3 which has size $4n + \mathrm{o}(n)$ bits. We sightly change the definition of the array C. We define $C[i] = -D[i]$ if $D[i]$ is the leftmost one among the same numbers. The new definition is just for convenience of making the values unique.

Our algorithm for the document listing query list(p) is similar to the original [1]. Instead of using the GST, we use the compressed suffix array to compute the interval $[l, r]$ such that suffixes $T_{SA[l]}, T_{SA[l+1]}, \ldots, T_{SA[r]}$ have p as their prefixes, which is done in $\mathit{Search}(p)$ time.

In the original algorithm the array C is used to avoid outputting a duplicate document ID. However we cannot use the same algorithm because the values

```
**Procedure** $CDLP(l, r)$
**if** $l > r$ **return**
$x :=$ RMQ$_C(l, r)$
$d := D'(SA[x])$
**if** $V[d] = 0$ **then**
    **output** $d$
    $V[d] := 1$
    $CDLP(l, x - 1)$
    $CDLP(x + 1, r)$
```

Fig. 6. Pseudo code for new document listing algorithm.

of C are not available. Instead we use a simple marking algorithm. To check the duplication of the output, we use a bit-vector $V[1..k]$. In the preprocess we set $V[i] = 0$ for $i = 1, 2, \ldots, k$, which takes $\mathrm{O}(k) = \mathrm{O}(n)$ time. In a query list(p) we check whether $V[x] = 1$ or 0 before outputting $D[x]$. If $V[x] = 0$, we output $D[x]$ and set $V[x] = 1$. After outputting all document ID's, we set $V[x] = 0$ for each x which was output. Therefore the query is done in $\mathrm{O}(\mathit{Search}(p) + q \cdot \mathit{Lookup}(n))$ time where q is the output size. The new algorithm is described in Fig. 6. Note that this algorithm requires $q \log k$ bits of temporary space to store the set of document ID's which are output.

Let us compute the size of the data structure. The array D' is encoded in $\mathrm{O}(k \log \frac{n}{k})$ bits. The data structure for range minimum queries on the array C is represented in $4n + \mathrm{o}(n)$ bits. The vector V has size k bits. Therefore the total is $|CSA| + 4n + \mathrm{o}(n) + O(k \log \frac{n}{k})$ bits. Note that $|CSA| = \frac{1}{\epsilon} n H_0 + \mathrm{O}(n)$ bits to obtain $\mathrm{O}(|p| + q \log^{\epsilon} n)$ time queries (see Section 2.3).

5 Succinct Data Structure for TF*IDF Problem

In this section we propose a succinct data structure for the Problem 2 (TF*IDF Problem). Inverted files can solve the problem for only predetermined patterns, whereas ours can solve for any pattern.

5.1 Data Structure for computing $tf(p, d)$

We construct the compressed suffix array CSA_d for each document d to which a terminator $\$_d$ is appended. Then the term frequency $tf(p, d)$ is obviously computed in $O(Search(p))$ time by using CSA_d as follows. Let $[l, r]$ be the interval that corresponds to p in SA_d. Then $tf(p, d) = r - l + 1$.

A naive algorithm to compute $tf(p, d)$ for all documents d containing p will be as follows. We first enumerate all d by using the document listing query, and compute $tf(p, d)$ for each d. However, this is not efficient because it takes $O(q \cdot Search(p))$ time where q is the number of documents containing p. We give an algorithm to compute all the scores in $O(Search(p) + q \cdot (Lookup(n) + \log\log q))$ time.

In addition to the compressed suffix array for each document, we use the one for the concatenation of all the documents, denoted by CSA, which is used for the document listing problem. We first find the interval $[l, r]$ in CSA in $O(m)$ time. Then for each distinct $d \in D[l, r]$, we compute the leftmost and the rightmost indices $i, j \in [l, r]$ such that $D[i] = D[j] = d$ as follows. The leftmost indices of all distinct values are computed by the same algorithm as the document listing problem. We next find the rightmost indices j. We use another data structure which is similar to C. The array C is regarded as a set of linked lists for each document. We define another array C' that represents linked lists in the opposite direction. We define $C'[i] = j$ where j is the smallest index such that $i < j$ and $D[j] = D[i]$. We define $C'[i] = n + D[i]$ if such j does not exist. Then we can enumerate the rightmost indices j of all distinct values in $D[l, r]$ by using range maximum queries to C' which are solved by using range minimum queries after negating all the values in C'.

The value of $\mathit{tf}(p, d)$ is equal to the number of d in $D[l, r]$. Instead of counting it directly, we use the compressed suffix array CSA_d for the document d. Suffixes of document d which match with p are in a consecutive region of the suffix array SA_d, and they appear in the suffix array SA for T in the same relative order in SA_d. Therefore the leftmost and the rightmost suffixes for p in SA_d are identical with those in SA. From this observation, we can compute $\mathit{tf}(p, d)$ as follows.

Let i and j be the leftmost and the rightmost index of d in $D[l..r]$ computed by using C and C', respectively. We compute $x = SA[i]$ and $y = SA[j]$ using CSA. Then we convert them into those in document d, say x' and y' as follows. We compute the predecessor z of x in D' in constant time [17], which is the position of the first character of d in T. Then $x' = x - z + 1$ and $y' = y - z + 1$ hold. Next we compute $i' = SA_d^{-1}[x']$ and $j' = SA_d^{-1}[y']$ in $O(\mathit{Lookup}(n_d))$ time by using CSA_d where $n_d < n$ is the length of the document. Finally we have $\mathit{tf}(p, d) = j' - i' + 1$.

The size of the data structure becomes as follows. The size of the compressed suffix array for T is denoted by $|CSA|$. The total of the size of the compressed suffix array for each document is roughly equal to $|CSA|$. The sizes of array~~e~~

C and C' are $4n + \mathrm{o}(n)$ bits and $4(n+k) + \mathrm{o}(n)$ bits, respectively. The size of D' is $\mathrm{O}(k \log \frac{n}{k})$. Therefore the total is $2|CSA| + 8n + \mathrm{o}(n) + \mathrm{O}(k \log \frac{n}{k})$ bits.

Note that the order of the output of the range minimum query and that of the range *maximum* query will be different. Therefore we need to sort the document ID's. We use an $\mathrm{O}(q \log\log q)$ time sorting algorithm of Andersson et al. [18]. Therefore we have the following:

Lemma 5 *The term frequency $tf(p,d)$ is computed in $\mathrm{O}(Search(p))$ time, and term frequencies for all q documents containing a pattern p are computed in $\mathrm{O}(Search(p) + q \cdot (Lookup(n) + \log\log q))$ time using a data structure of size $2|CSA| + 8n + \mathrm{o}(n) + \mathrm{O}(k \log \frac{n}{k})$ bits.*

5.2 Data Structure for computing df(p)

The document frequency $df(p)$ is computed in $\mathrm{O}(Search(p) + q \cdot Lookup(n))$ time by using the data structure for the document listing problem. However this is too slow if we want only the value of $df(p)$. Here we propose an algorithm to compute it in $\mathrm{O}(Search(p))$ time using CSA and a data structure of size at most $2n + \mathrm{o}(n)$ bits. We use Hui's algorithm [19] and modify its data structure to reduce the size.

Hui's algorithm works as follows. In each internal node v of GST the original algorithm stores a number $u(v)$ which represents how many "duplicate" suffixes from the same document occur in v's subtree. More precisely, let $n_d(v)$ be the number of leaves from document d in the subtree rooted at v. We have $u(v) = \sum_{d:n_d(v)>0}(n_d(v) - 1)$. Let l and r be the indices of the leftmost and the rightmost leaves in the subtree rooted at v. Then $(r - l + 1) - u(v)$ is the number of distinct document ID's in the subtree rooted at v, which is $df(p)$ if v corresponds to p.

Let $L_1^d, L_2^d, \ldots, L_m^d$ be the leaves of the subtree rooted at v which are from document d and sorted in lexicographic order. Then the value $n_d(v)-1$ is equal to the number of times that $lca(L_i^d, L_{i+1}^d)$ $(1 \leq i \leq m-1)$ is in the subtree. Therefore for each node w of GST we compute the summation $h(w)$ of the number of times that $lca(L_i^d, L_{i+1}^d) = w$ for all d. This is done in $\mathrm{O}(n)$ time by using constant time lca queries. Then $u(v) = \sum_{d:n_d(v)>0}(n_d(v)-1) = \sum[h(w) :$ w is in the subtree of $v]$. Then the values of $u(v)$ for all nodes are computed in linear time by a bottom-up traversal of GST. Finally we have $df(p) = (r-l+1)-u(v)$ and $idf(p) = \log \frac{k}{df(p)}$, where v is the node corresponding to p.

The above data structure has size $\mathrm{O}(n \log n)$ bits. We reduce the size to $2n + \mathrm{o}(n)$ bits. We temporarily construct a GST' of T in which all internal nodes have two children, that is, any internal node of GST which has $c > 2$ children is divided into $c-1$ nodes each of which has two children (compare Fig. 7 with Fig. 1). Then we compute $h(w)$ for each node in GST' in linear time.

Instead of storing $u(v)$ we store $h(v)$ for all internal nodes in an array $H[1..n-1]$ in which the values are arranged in *inorder* of nodes. Let i be the *inorder* of a node v. Then $v = lca(leaf(i), leaf(i+1))$ holds and we store $h(v)$ in $H[i]$.

The value $u(v)$ is equal to the summation of $h(w)$ for all descendants of v. Because $h(w)$'s are stored in *inorder*, $u(v)$ is equal to the summation of $H[l..r-1]$ where l and r are the lexicographic orders of suffixes stored in the leftmost and the rightmost leaves of v, respectively. If v is the node corresponding to a pattern p, the indices l and r are calculated in $Search(p)$ time by using the

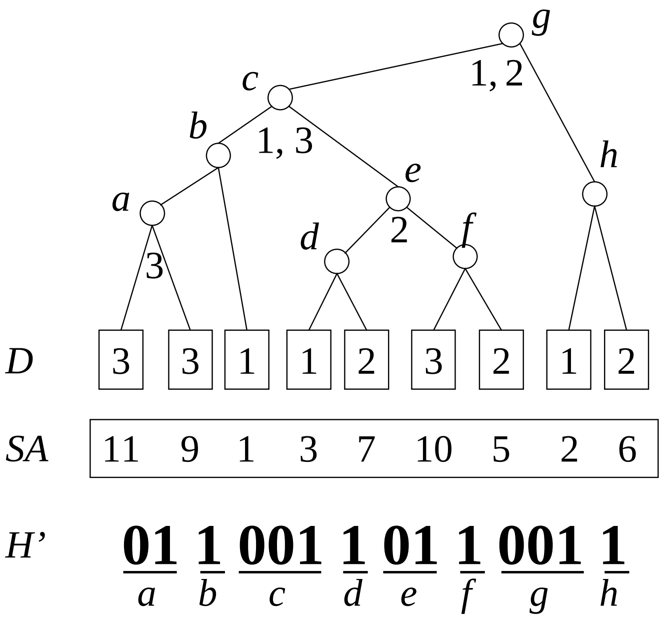


Fig. 7. Data structure for computing $df(p)$. Numbers below an internal node w show the values of d such that $lca(L_i^d, L_{i+1}^d) = w$ for some i.

compressed suffix array of T.

The value $h(v)$ is encoded as unary code in a sequence H', that is, encoded as $h(v)$ zeroes followed by a one, for example 0 is encoded as `1` and 2 as `001`. Then $u(v)$ is computed as follows:

$$\begin{aligned} x &= select_1(H', l-1) + 1 \\ y &= select_1(H', r) \\ u(v) &= rank_0(H', y) - rank_0(H', x) \end{aligned}$$

where l and r are indices defined above. Therefore $u(v)$ is computed in constant time. The size of the bit-vector is at most $2n - d$ bits because there are n ones and at most $n - d$ zeroes. Thus we have the following.

Lemma 6 *Given the interval of the suffix array of T that corresponds to a pattern p, the document frequency $df(p)$ can be computed in constant time using a data structure of size $2n + o(n)$ bits.*

For the TF*IDF problem, we use both data structures in Sections 5.1 and 5.2. Therefore the space is $2|CSA| + 10n + \mathrm{o}(n) + \mathrm{O}(k \log \frac{n}{k})$ bits, which proves the Theorem 3.

6 Concluding Remarks

We have extended the data structure for the document listing problem so that it can be used to compute *tf*idf* scores. The size of the data structure is proportional to the text size, which is an improvement from the previous algorithm using $\mathrm{O}(n \log n)$ bit space. The size of the data structure is $2|CSA| + 10n + \mathrm{o}(n) + \mathrm{O}(k \log \frac{n}{k})$ bits where $|CSA|$ is the size of the compressed suffix array for a collection of documents, and it can be smaller than the size of the documents if the parameter is set to be $\epsilon = 1$. Therefore the size of the whole data structure is about three times larger than inverted files. Though the time complexities are at most $\mathrm{O}(\log n)$ times larger than that by using inverted files, our data structures support queries for any pattern. The query time is further improved by increasing the size in constant factor.

Acknowledgments

The author would like to thank Prof. Takeshi Tokuyama of Tohoku University and Dr. Jesper Jansson of Kyushu University, and anonymous referees for their valuable comments. The work of the author was supported in part by the Grant-in-Aid of the Ministry of Education, Science, Sports and Culture of Japan.

References

[1] S. Muthukrishnan, Efficient Algorithms for Document Retrieval Problems, in: Proc. ACM-SIAM SODA, 2002, pp. 657–666.

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

[3] R. Grossi, J. S. Vitter, Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching, SIAM Journal on Computing 35 (2) (2005) 378–407.

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

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

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

[7] P. Ferragina, G. Manzini, Indexing compressed texts, Journal of the ACM 52 (4) (2005) 552–581.

[8] G. Salton, A. Wong, C. S. Yang, A Vector Space Model for Automatic Indexing, Communications of the ACM 18 (11) (1975) 613–620.

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

[10] P. Weiner, Linear Pattern Matching Algorihms, in: Proceedings of the 14th IEEE Symposium on Switching and Automata Theory, 1973, pp. 1–11.

[11] M. Farach, Optimal Suffix Tree Construction with Large Alphabets, in: 38th IEEE Symp. on Foundations of Computer Science, 1997, pp. 137–143.

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

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

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

[15] P. Ferragina, G. Manzini, V. Mäkinen, G. Navarro, Succinct Representation of Sequences, Technical Report TR/DCC-2004-5, Dept. of Computer Science, Univ. of Chile, `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/sequences.ps.gz` (Aug. 2004).

[16] M. Bender, M. Farach-Colton, The LCA Problem Revisited, in: Proceedings of LATIN, LNCS 1776, 2000, pp. 88–94.

[17] R. Raman, V. Raman, S. S. Rao, Succinct Indexable Dictionaries with Applications to Encoding k-aray Trees and Multisets, in: Proc. ACM-SIAM SODA, 2002, pp. 233–242.

[18] A. Andersson, T. Hagerup, S. Nilsson, R. Raman, Sorting in Linear Time?, in: ACM Symposium on Theory of Computing, 1995, pp. 427–436.

[19] L. Hui, Color Set Size Problem with Applications to String Matching, in: Proc. of the 3rd Annual Symposium on Combinatorial Pattern Matching (CPM'92), LNCS 644, 1992, pp. 227–240.