Ultra-succinct Representation of Ordered Trees

Jesper Jansson* Kunihiko Sadakane† Wing-Kin Sung‡

Abstract

There exist two well-known succinct representations of ordered trees: BP (balanced parenthesis) [Munro, Raman 2001] and DFUDS (depth first unary degree sequence) [Benoit et al. 2005]. Both have size $2n + o(n)$ bits for n-node trees, which asymptotically matches the information-theoretic lower bound. Many fundamental operations on trees can be done in constant time on word RAM, for example finding the parent, the first child, the next sibling, the number of descendants, etc. However there has been no single representation supporting every existing operation in constant time; BP does not support i-th child, while DFUDS does not support *lca* (lowest common ancestor).

In this paper, we give the first succinct tree representation supporting every one of the fundamental operations previously proposed for BP or DFUDS along with some new operations in constant time. Moreover, its size surpasses the information-theoretic lower bound and matches the entropy of the tree based on the distribution of node degrees. We call this an ultra-succinct data structure. As a consequence, a tree in which every internal node has exactly two children can be represented in $n + o(n)$ bits. We also show applications for ultra-succinct compressed suffix trees and labeled trees.

1 Introduction

A *succinct data structure* is a data structure which stores an object using space close to the information-theoretic lower bound, while simultaneously supporting a number of primitive operations to be performed on the object in constant time. Here the information-theoretic lower bound for storing an object from a fixed universe with cardinality L is $\log L$ bits[1] because in the worst case this number of bits is necessary to distinguish two distinct objects. For example, that for a subset of the ordered set $\{1, 2, \ldots, n\}$ is n because there are 2^n different subsets, and that for an ordered tree with n nodes is $2n - \Theta(\log n)$ because there exist $\binom{2n-1}{n-1}/(2n-1) = 2^{2n}/\Theta(n^{\frac{3}{2}})$ such trees [19]. Typical succinct data structures are the ones for storing ordered sets [23, 25, 24, 13], ordered trees [14, 19, 8, 9, 2, 22, 21, 3, 28], strings [10, 11, 6, 26, 31, 29], functions [21], cardinal trees [2, 5], etc. The size of a succinct data structure storing an object from the universe is typically $(1 + o(1)) \log L$ bits[2]. Many fundamental operations on the object can be done in constant time on the word RAM model with word-length $\Theta(\log n)$, for example, counting the number of elements in a set which are smaller than a given value, finding the parent of a node in a tree, etc.

This paper considers succinct data structures for ordered trees. Though there exist many such data structures in the literature, they have the following disadvantages.

1. No single succinct data structure supports all fundamental operations in constant time; the balanced parenthesis representation [19, 8] (BP) does not support i-th child, while the depth-first unary degree sequence representation [2, 9] (DFUDS) does not support lowest common ancestor (*lca*).

2. Though the space is asymptotically optimal in the worst case, it is not optimal for certain classes of trees. For example, any n-node tree whose internal nodes have exactly two children can be encoded in n bits by writing `1` for internal nodes and `0` for leaves during the depth-first traversal of the tree, whereas both the BP and the DFUDS use $2n$ bits.

These drawbacks cause severe problems for document processing. Now many huge collections of documents are available, for example Web pages and genome sequences. To search such documents we use suffix

*Department of Computer Science and Communication Engineering, Kyushu University. Motooka 744, Nishi-ku, Fukuoka 819-0395, Japan. `jj@tcslab.csce.kyushu-u.ac.jp` Supported by JSPS (Japan Society for the Promotion of Science).

†Department of Computer Science and Communication Engineering, Kyushu University. Motooka 744, Nishi-ku, Fukuoka 819-0395, Japan. `sada@csce.kyushu-u.ac.jp` Work supported in part by the Grant-in-Aid of the Ministry of Education, Science, Sports and Culture of Japan.

‡School of Computing, National University of Singapore. `ksung@comp.nus.edu.sg`

[1] The base of logarithm is 2 unless specified. We define $0 \log 0 = 0$.

[2] Some papers use a weaker definition of succinctness that allows $O(\log L)$ bits.

trees [16] or compressed suffix trees [30] because they support efficient queries. The compressed suffix tree uses the BP (and some auxiliary information) to encode the tree because *lca* is crucial. Then the size of the BP is $4n + \mathrm{o}(n)$ bits because the tree has $2n-1$ nodes in the worst case. On the other hand, if we use the Patricia tree [17] to represent the suffix tree, its topology can be encoded in $2n$ bits, though we do not know how to support efficient queries. Therefore we pay $2n$ bits extra for supporting efficient queries. This cost is enormous for huge collections of documents.

Note that there exists no data structure for storing *any* n-node tree using less than cn bits for $c < 2$ because it surpasses the information-theoretic lower bound. However, if we consider only *typical* objects we can expect to reduce the size. This is the concept of data compression. We say a data structure storing an object is *ultra-succinct* if its size varies according to the object and the size achieves some entropy bound of the object. In the literature, there exist such data structures for strings [6, 10, 31] and ordered sets [24], but no such data structures for ordered trees.

1.1 Our contributions In this paper we solve the above problems by providing an ultra-succinct representation of ordered trees with the following properties:

1. It supports all previously defined fundamental operations on ordered trees listed in Section 2.2.1 in constant time.
2. Its size surpasses the information-theoretic lower bound and achieves the entropy of the tree defined below.

We introduce the following definition for the *tree degree entropy* of an ordered tree:

Definition 1. (Tree degree entropy) *For an ordered tree T with n nodes, having n_i nodes with i children, the tree degree entropy $H^*(T)$ of T is defined as*

$$H^*(T) = \sum_i \frac{n_i}{n} \log \frac{n}{n_i}.$$

This definition is natural because it matches the information-theoretic lower bound for ordered trees with a given degree distribution:

Lemma 1.1. (Rote [27]) *The number of ordered trees with n nodes, having n_i nodes with i children, for $i = 0, 1, \ldots,$ is*

$$\frac{1}{n}\binom{n}{n_0 \; n_1 \; \cdots \; n_{n-1}},$$

if $\sum_{i \geq 0} n_i(i-1) = -1$. If this equation does not hold, there are no such trees.

Let L denote this number. Then $\log L \approx nH^*(T)$.

Our main contribution of this paper is an ultra-succinct data structure for ordered trees whose size asymptotically matches the tree degree entropy. Not only is it smaller than the existing data structures, but it supports any operation in the same time complexity as the existing data structures. Our proof is not by showing each operation separately; instead we prove a more general result on the instant decodability of the DFUDS:

Theorem 1.1. *For any rooted ordered tree T with n nodes, there exists a data structure using $nH^*(T) + \mathrm{O}(n(\log\log n)^2/\log n)$ bits such that any consecutive $\log n$ bits of DFUDS of T can be computed in constant time on word RAM.*

Note that $nH^*(T) \leq 2n$ for any tree, implying that the size of our data structure is never more than BP nor DFUDS. Theorem 1.1 also implies that we can assume we had the DFUDS in the original form. Then it is obvious that any operation can be done on our ultra-succinct data structure in the same time complexity as the original DFUDS. Even if a new operation on the DFUDS is proposed, it also works on our representation in the same time complexity.

Another contribution of this paper is to give $\mathrm{o}(n)$-bit auxiliary data structures for computing *lca*, *depth*, and *level-ancestor* on the original DFUDS. Though the data structure of [9] supports *depth* and *level-ancestor*, it does not support *lca* and it modifies the original DFUDS. As a result it is not guaranteed that any algorithm on the original DFUDS also works on the modified DFUDS. Moreover, we cannot apply our compression technique to the modified DFUDS. Therefore it is important to support these operations on the original DFUDS. We show the following:

Theorem 1.2. *The lowest common ancestor between any two given nodes, the depth and the level-ancestor of a given node can be computed in constant time on the DFUDS using $\mathrm{O}(n(\log\log n)^2/\log n)$-bit auxiliary data structures.*

Our new auxiliary data structures have another benefit. Their size is smaller than the existing ones [8, 9] which use $\mathrm{O}(n \log\log n/\sqrt{\log n})$ bits.

We also show applications of our succinct representation of ordered trees. The first one is space reduction of the compressed suffix trees [30] which uses the BP. We give operations on the DFUDS which are equivalent to the ones on BP. As a result we can perform any operation for the suffix tree on a more compact data structure in the same time complexity as the original compressed suffix trees. The next one is space reduction of the succinct representation of labeled trees [5].

We can further compress a labeled tree into the tree degree entropy, while preserving the same query time complexities.

1.2 Organization of paper The rest of the paper is organized as follows. In Section 2 we review existing succinct data structures. In Section 3 we propose simple and space-efficient auxiliary data structures for *lca*, *depth* and *level-ancestor* on DFUDS, which is summarized as Theorem 1.2. Section 4 gives the data structure to compress the DFUDS into the tree degree entropy and thus proves Theorem 1.1. In Section 5 we show applications of our new representation of trees for reducing the size of labeled trees and compressed suffix trees.

2 Preliminaries

First we explain some basic data structures used in this paper. For the computation model, we use the word RAM with word-length $\Theta(\log n)$ where any arithmetic operation for $\Theta(\log n)$-bit numbers and $\Theta(\log n)$-bit memory I/Os are done in constant time.

2.1 Succinct data structures for rank/select Consider a string $S[1..n]$ on an alphabet $\mathcal{A}$ with alphabet size σ. We define *rank* and *select* for S as follows. $rank_c(S,i)$ is the number of occurrences c in $S[1..i]$, and $select_c(S,i)$ is the position of the i-th occurrence of c in S. Note that $rank_c(S, select_c(S,i)) = i$. We may omit S if it is clear from the context.

There exist many succinct data structures for rank/select [23, 25, 14, 18]. A basic one uses $n + \mathrm{o}(n)$ bits for $\sigma = 2$ [18] and supports rank/select in constant time on word RAM with word length $\mathrm{O}(\log n)$. The space can be reduced if the number of 1's is small. For a string with m 1's, there exists a data structure for rank/select using $\log \binom{n}{m} + \mathrm{O}(n \log\log n/\log n) = m \log \frac{n}{m} + \Theta(m) + \mathrm{O}(n\log\log n/\log n)$ bits [25]. This data structure is called *fully indexable dictionary* or FID. If $m = \mathrm{O}(n/\log n)$, the space is $\mathrm{O}(n\log\log n/\log n)$. We extensively use FID in this paper to compress pointers. For general alphabets, there exists a data structure for constant time rank/select queries using $n\log\sigma + \mathrm{o}(n\log\sigma)$ bits [7], though we do not use it in this paper.

The rank/select functions are extended for counting occurrences of multiple characters [20]. For a pattern P on the alphabet, $rank_P(S,i)$ is the number of occurrences of the pattern P whose starting positions are in $S[1..i]$, and $select_P(S,i)$ is the starting position of the i-th occurrence of P. Note that occurrences of P may overlap in S. Both functions take constant time and the size of the data structure is the same as that of FID if the pattern length is constant.

For a dense subset such that $n = m(\log m)^{\mathrm{O}(1)}$, $rank_1(S,i)$ and $select_1(S,i)$ are computed in constant time using $\log\binom{n}{m} + \mathrm{O}(m(\log\log m)^2/\log m)$ bits [23].

A crucial technique for succinct data structures is *table lookup*. For small-size problems we construct a table which stores answers for all possible queries. For example, for *rank* and *select*, we use a table storing all answers for all 0,1 patterns of length $\frac{1}{2}\log n$. Because there exist only $2^{\frac{1}{2}\log n} = \sqrt{n}$ different patterns, we can store all answers in a table using $\sqrt{n}\cdot \mathrm{polylog}(n) = \mathrm{o}(n)$ bits, which can be accessed in constant time on word RAM.

2.2 Succinct data structures for trees We consider the set of all rooted ordered trees with n nodes. There exist $\binom{2n-1}{n-1}/(2n-1)$ such trees [19]. Therefore the information-theoretic lower bound of succinct data structures is $2n - \Theta(\log n)$ bits. Many data structures achieving a matching upper bound asymptotically have been proposed [14, 19, 8, 9, 2, 22, 21, 3, 28].

2.2.1 Balanced parenthesis encoding (BP) The most well-known representation of ordered trees is the balanced parenthesis representation [19], which we call BP in this paper. A tree is represented by a string P of balanced parentheses of length $2n$. A node is represented by a pair of matching parentheses $(\ldots)$ and all subtrees rooted at the node are encoded in order between the matching parentheses (see Figure 1 for an example). To allow tree navigational operations, the following operations are supported in constant time on the word RAM [19]:

- *findclose*(P,x), *findopen*(P,x): find the index of the closing (opening) parenthesis that matches a given opening (closing) parenthesis $P[x]$,
- *enclose*(P,x): find the index of the opening parenthesis of the pair that most tightly encloses $P[x]$.

By using these operations, the following are supported [19, 21, 28, 3]:

- *parent*(x), *firstchild*(x), *sibling*(x): the parent, the first child, the next sibling node of node x, respectively,
- *depth*(x): the depth of x,
- *desc*(x): the number of descendants of x,
- *rank*(x): the preorder of x,
- *select*(i): the node with preorder i,
- *LA*(x,d): the ancestor of x with depth d (*level-ancestor*),
- *lca*(x,y): the lowest common ancestor of nodes x and y,
- *degree*(x): the number of children of x,
- *child*(x,i): the i-th child of x,
- *childrank*(x): return i such that x is the i-th child of its parent.

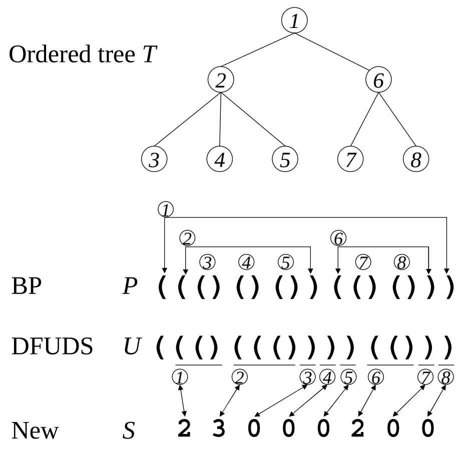
Ordered tree T
BP P ((( ) ( ) ( )) (( ) ( )))
DFUDS U ((( ) ((( )))) (( )))
New S 2 3 0 0 0 2 0 0

Figure 1: Succinct representations of trees.

These operations are done in constant time using o(n)-bit auxiliary data structures, except for *child* and *childrank*, which take O(i) time.

2.2.2 Depth-first unary degree sequence (DFUDS) The DFUDS (depth-first unary degree sequence) representation [2, 9] of an ordered tree is defined as follows. A tree with only one leaf is represented as $\boxed{()}$, which is the same as the BP. If a tree T has k subtrees $T_1, \ldots, T_k$, the DFUDS of T is the concatenation of $k+1$ $\boxed{(}$, a $\boxed{)}$, and DFUDS's of $T_1, \ldots, T_k$, with the first $\boxed{(}$ for each $T_1, \ldots, T_k$ being omitted. Then the resulting DFUDS also forms a balanced parenthesis sequence (see Figure 1 for an example). The leftmost $\boxed{(}$ of the DFUDS of any tree is considered as an imaginary superroot. Ignoring the imaginary superroot, the DFUDS can be interpreted as a preorder listing of the nodes where each node with degree k is encoded by k $\boxed{(}$'s, followed by a $\boxed{)}$. We use the position of the leftmost parenthesis of the encoding of a node as its representative. Such parenthesis is $\boxed{(}$ for internal nodes, and $\boxed{)}$ for leaves. We assume the position of parentheses begins with 0. Therefore the position of the root node is 1. The DFUDS [2] uses the same space as BP, and supports all of the operations listed above in constant time, except for *lca*, *depth*, and *LA*. However, *depth* and *LA* can be supported in a modified variant of DFUDS [9].

2.3 Compressing succinct data structures Sadakane and Grossi [31] proposed a general compression algorithm for strings.

THEOREM 2.1. ([31]) *A string S of length n with alphabet size σ can be compressed into at most $nH_k(S) + \mathrm{O}(n((k+1)\log\sigma + \log\log n)/\log_\sigma n)$ bits for any $k \geq 0$ so that any substring of S of length $\mathrm{O}(\log_\sigma n)$ (i.e., $\mathrm{O}(\log n)$ bits) is decodable in constant time on word RAM.*

Here $H_k(S)$ is the k-th order empirical entropy of S [15] and $H_k(S) \leq H_{k-1}(S) \leq \cdots \leq H_0(S) \leq \log\sigma$. This theorem implies that we can regard the data structure as the uncompressed string. Any algorithm on the uncompressed string which does not change the string also works on the compressed one in the same time complexity. For example, the size of FID for a set $\tilde{S} \subset \{1, 2, \ldots, n\}$ is reduced to $nH_k(S) + \mathrm{O}(n\log\log n/\log n)$ bits for any $0 \leq k \leq \mathrm{O}(\log\log n)$ where S is a 0,1-string such that $S[i] = 1 \iff i \in \tilde{S}$, while the time complexities for *rank* and *select* remain unchanged. Note that Theorem 2.1 holds for any $k \geq 0$ simultaneously. The condition $k \leq \mathrm{O}(\log\log n)$ is necessary to bound the lower-order term by $\mathrm{O}(n\log\log n/\log n)$.

In their data structure, the string S is parsed into distinct substrings called *phrases* which are stored in a tree with at most $c = n/\log_\sigma n$ nodes. Therefore to decode a substring of S, which consists of a number of phrases, we extract paths in the tree that represent the phrases. Decoding a phrase (or its substring of length $w = \frac{1}{2}\log_\sigma n$) in constant time is done as follows. In a subtree with at most $w/2$ nodes, we can obtain any path in constant time by table lookups. Therefore we remove all such subtrees from the tree. Then the remaining tree has at most $2c/w$ leaves and $2c/w - 1$ branching nodes. For each branching node in the remaining tree, we store the path of length w from the node to the root, which enables us to obtain the path in constant time. The space to store these paths is $\mathrm{O}(c) = \mathrm{o}(n\log\sigma)$ bits. Other paths which do not include any branching node can be obtained in constant time because they consist of unary nodes and they are stored in consecutive regions of memory. If a substring of S of length w contains many short phrases, it is stored without compression. In this paper, we slightly change this data structure in Section 4.2.

2.4 Data structures for level-ancestors Bender and Farach-Colton [1] proposed a simple O(n)-word (O($n\log n$)-bit) data structure for constant time level-ancestor queries. In their data-structure, the tree T is broken into disjoint paths as follows. First, a longest root-leaf path in T is found and removed from T. This breaks the tree into disjoint subtrees. Recursively, every subtree is partitioned into disjoint paths. Then for each resulting path we extend it toward the root so that the length of the path is doubled. More precisely, let $v_1, v_2, \ldots, v_h, v_{h+1}, \ldots, v_d$ be the path from a leaf v_1 to the root v_d. If the disjoint path is $v_1, v_2, \ldots, v_h$, we extend it to $v_1, v_2, \ldots, v_{2h}$. We call it a *doubled long-path ladder.* Each doubled long-path ladder is represented by an array of nodes; therefore the level-ancestors on each ladder are easily found. The total number of nodes on all ladders is at most $2n$. We use another data structure called jump-pointers, which are pointers to nodes $LA(v, \ell)$ from some selected node v of T called *macro nodes* for $\ell = d-1, d-2, d-4, d-8, \ldots$ where $d = depth(v)$.

The query $LA(v, \ell)$ is solved as follows. First find v's nearest ancestor w which is a macro node by table lookups (for details refer to [1]). Then find the farthest ancestor of w whose depth is no less than ℓ by following a jump-pointer. Then we reach a doubled long-path ladder including $LA(v, \ell)$. Therefore it is obvious how to obtain $LA(v, \ell)$.

3 New Operations on DFUDS

In this section we propose simple algorithms and data structures for supporting *lca*, *depth*, *level-ancestor* and *childrank* on the original DFUDS. The algorithm for *lca* is completely new. For operations *depth* and *level-ancestor*, Geary et al. [9] showed that the operations can be implemented on a modified DFUDS. However, the data structure is complicated and is difficult to compress. On the other hand, we propose the first data structures for *depth* and *level-ancestor* on the original DFUDS. These data structures are much simpler than those of [9]. More importantly, we improve the lower order term of the size for *level-ancestor.* The previous ones use $O(n \log\log n/\sqrt{\log n})$ bits [9, 21], while our new data structure uses $O(n(\log\log n)^2/\log n)$ bits. An algorithm for *childrank* is also proposed in [9] for the modified DFUDS. We provide a simpler algorithm for the original DFUDS.

From here on, we identify the node with preorder x with its starting position in DFUDS, which is computed in constant time by $(select_{)}(x-1)) + 1$.

3.1 LCA

Let U be the DFUDS of a tree T. The *excess sequence* E of U is defined so that $E[i]$ = (number of (in $U[0..i]$) − (number of) in $U[0..i]$). Note that for a BP sequence, the excess values correspond to node depths, but that they have a slightly different interpretation for a DFUDS sequence.

Consider an internal node v of T, which has k subtrees $T_1, \ldots, T_k$ as its children. Suppose that $U[l_0..r_0]$ stores the DFUDS for v. We also assume that $U[l_i..r_i]$ stores the DFUDS for T_i for $1 \le i \le k$. Note that $l_i = r_{i-1} + 1$. Let $d = E[r_0]$. Then we have the following property of the excess values.

Lemma 3.1.

$$\begin{aligned} E[r_i] &= E[r_{i-1}] - 1 = d - i \quad (1 \le i \le k) \\ E[j] &> E[r_i] \quad (l_i \le j < r_i) \end{aligned}$$

Proof. By the construction of DFUDS, if we add a (at the beginning of the parenthesis sequence $U[l_i..r_i]$ for a subtree T_i, it becomes balanced. In a balanced parenthesis sequence the number of open and close parentheses are the same. Therefore in $U[l_i..r_i]$ the number of close parentheses is one more than that of open parentheses, and we have $E[r_i] = E[r_{i-1}] - 1$ for $1 \le i \le k$. Because $E[r_0] = d$, we have $E[r_i] = d - i$. The second property $E[j] > E[r_i]$ $(l_i \le j < r_i)$ is obvious because the sequence is balanced if an open parenthesis is added at the beginning. □

Lemma 3.2. *The lowest common ancestor of nodes x and y in an ordered tree can be computed in constant time from their preorders using the DFUDS of the tree and an* $O(n(\log\log n)^2/\log)$*-bit auxiliary data structure.*

Proof. The $lca(x, y)$ $(x < y)$ between nodes x and y is computed by $w = RMQ_E(x, y-1)$, and then $z = parent(w+1)$, where $RMQ_E(x, y-1)$ is called *range minimum query* and returns the position of the smallest element in $E[x..y-1]$. If there is a tie, RMQ returns the leftmost position. RMQ can be computed in constant time using an $O(n(\log\log n)^2/\log n)$-bit auxiliary data structure [28].

We now prove the correctness of this method. Let v be the true $lca(x, y)$, $T_1, \ldots, T_k$ be the subtrees of v, and $U[l_i..r_i]$ be the DFUDS for T_i $(1 \le i \le k)$. Then x and y are in some subtrees T_α and T_β $(\alpha < \beta)$, respectively. Assume that $E[r_\beta] = d$. Then from Lemma 3.1 $E[r_{\beta-1}] = d+1$ and $E[i] > d+1$ for $l_1 \le i < r_{\beta-1}$ and $l_\beta \le i \le r_\beta - 2$. There are two cases: Case (1) If $y < r_\beta$ (i.e., if y is not the rightmost leaf of T_β), $E[y-1] > d+1$, and by the range minimum query we obtain $w = r_{\beta-1}$. Case (2) If $y = r_\beta$, $E[y-1] = d+1$, and therefore there are two minimum values $d+1$ in $E[x..y-1]$. By the range minimum query we can find the left one, which is $r_{\beta-1}$. In either case, we have $w + 1 = l_\beta$, which is the position of a subtree of v. By computing $z = parent(w+1)$, we obtain $lca(x, y)$. □

3.2 Depth

We use two-level data structures, but we first explain the general data structure for both levels. We partition the DFUDS U of a tree T into blocks of size B. For a fixed subset M of the nodes of T, the data structure for each level stores the following information.

For a node v, we denote by $f(v)$ its farthest ancestor that belongs to the same block as v. For every node

$v \in M$, we need the relative pointer from v to $f(v)$. We also need the difference of depths between them. We call this information I_1.

Let $p(v)$ denote the parent of node v, and $B(v)$ denote the block that contains v. We call an edge $(v, p(v))$ of T a *far edge* if $B(v) \neq B(p(v))$. We call nodes $p(v)$ of far edges *far nodes*. If there exist one or more far edges $(v_i, p(v_i))$ ($1 \leq i \leq k$, $v_i \in M$, $v_1 > v_2 > \cdots > v_k$) such that $B(v_1) = \cdots = B(v_k)$ and $B(p(v_1)) = \cdots = B(p(v_k))$, we say that $p(v_1), \ldots, p(v_k)$ form a group and call $p(v_1)$ the *pioneer* of the group. Note that $p(v_1) \leq p(v_2) \leq \cdots \leq p(v_k)$ because the edges $(v_i, p_i(v))$ are nested. We need a relative pointer from each far node $p(v_i)$ to its pioneer and the difference of depths. We call this information I_2.

We show that the number of pioneers is at most $2n/B - 3$ as in [19, 8]. Consider a graph $G = (V, E)$ whose nodes correspond to all the blocks. For each pair $(p(f(v)), f(v))$ such that $v \in M$ and $p(f(v))$ is a pioneer, we create an edge of G between nodes for $B(p(f(v)))$ and $B(f(v))$. Then the graph is outer-planar, and there are no multiple edges. Therefore the number of edges is at most $2n/B - 3$, which is an upper-bound of the number of pioneers. Figure 2 shows the pioneers for the example tree in Figure 1. The bold arcs show the edges of the graph.

Now we explain the two-level data structure. For the lower level, we use block size $B_L = \frac{1}{2}\log n$ and M_L is the set of all nodes of T. We call the blocks *small blocks*. We call pioneers for small blocks *lower level pioneers*. The information I_1 is computed in constant time using o(n)-bit tables. For I_2, we store the positions of lower level pioneers by FID. Let J_s be a bit-vector of length $2n$ such that $J_s[i] = 1$ if $U[i]$ is a lower level pioneer. Because the number of lower level pioneers is $O(n/B_L) = O(n/\log n)$, J_s is stored in $O(n \log\log n/\log n)$ bits. The lower level pioneer of a node v is computed by $select_1(J_s, rank_1(J_s, v))$ because the tree nodes are encoded in depth-first order. Because each far node and its lower level pioneer belongs to the same small block of size $B_L = \frac{1}{2}\log n$, the difference of depths between them can be computed in constant time by table lookups.

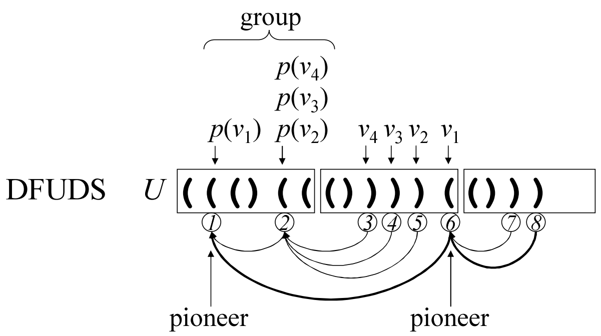

Figure 2: Pioneers for blocks of size $B = 6$ in the DFUDS sequence for the tree shown in Figure 1.

For the upper level, we use block size $B_U = \log^2 n$ and M_U is the set of lower level pioneers defined above. We call the blocks *large blocks* and pioneers for large blocks *upper level pioneers*. Let $f_U(v)$ denote the farthest ancestor of v inside the large block of v. For information I_1, we store for each node $v \in M_U$ the relative pointer from v to $f_U(v)$ and the difference of their depth explicitly. Because both v and $f_U(v)$ belong to the same large block of size $B_U = \log^2 n$, the information can be stored in $O(\log B_U)$ bits. Because there are $|M_U| = O(n/\log n)$ nodes we can store I_1 in $O(|M_U| \log B_U) = O(n \log\log n/\log n)$ bits.

For information I_2, we explicitly store the relative pointers and the differences of depths between far nodes and their pioneers. This information can be also stored in $O(n \log\log n/\log n)$ bits because each pair of far node and its pioneer belong to the same large block. For upper level pioneers we store their depths explicitly using $\log n$ bits. Because the number of upper level pioneers is at most $2n/B_U - 3 = O(n/\log^2 n)$, we need only $O(n/\log n)$ bits.

The query for a node v is done as follows. First we find $f_L(v)$ which is the farthest ancestor of v in the small block of v by table lookups. Then we compute the parent $w = p(f_L(v))$. We can determine if it is a lower level pioneer by using FID. If it is not, its lower level pioneer must be the closest one on U to the left, because the graph is planar. Therefore we can find the lower level pioneer z by *rank* and *select*. We can compute the relative depth of w from z by table lookups. Next we use data structures for large blocks. For the node z, $f_U(z)$ is stored as a relative pointer from z. If $p(f_U(z))$ is not an upper level pioneer, we move to the upper level pioneer by using the pointer stored for the node. Then we can obtain the depth of the upper level pioneer because it is explicitly stored.

3.3 Level-ancestor We consider the DFUDS sequence U for a tree T, which is partitioned into blocks of several sizes. We use a data structure similar to the one by Bender and Farach-Colton [1] outlined in Section 2.4, but we change it to a two-level data structure to reduce the space to $O(n(\log\log n)^2/\log n)$. The lower level of the data structure is for computing LA inside a block of size $\log^4 n$ (called huge block). The upper level is for the whole tree.

Consider computing $w = LA(v, d)$. Let z be the lower level pioneer of $p(f(v))$ where $f(v)$ is the farthest ancestor of v that belongs to the same small block of size $\frac{1}{2}\log n$. If z is an ancestor of w, we can find w by table lookups. Therefore it is enough to consider level-ancestors only for the lower level pioneers for the small blocks. Let M be the set of these pioneers.

The lower level data structure is to compute the level-ancestor $w = LA(v, d)$ if it belongs to the same huge block as v. For each node $v \in M$, we store its jump-pointers; we store level-ancestors with depths $d-1, d-2, d-4, \ldots, d-\log^4 n$ where $d = \mathit{depth}(v)$. For each lower level pioneer we need $\mathrm{O}((\log\log n)^2)$ bits and there are $\mathrm{O}(n/\log n)$ pioneers. Therefore we need $\mathrm{O}(n(\log\log n)^2/\log n)$ bits for all lower level pioneers.

For a doubled long-path ladders for a lower level pioneer v_i, we store a part of the ladder between v_i and $f(v_i)$, the farthest ancestor of v_i that is in the same huge block. For the ladder for a lower level pioneer v_i, we use two bit-vectors $D_i[0..\log^4 n]$ and $M_i[0..\log^4 n]$ where M_i is to indicate lower level pioneers on the ladder, and D_i is to encode the depths of the pioneers. If a lower level pioneer v is on the ladder, $M_i[v - f(v_i)] = 1$, and $D_i[\mathit{depth}(v) - \mathit{depth}(f(v_i))] = 1$. Let n_i be the number of lower level pioneers on the i-th ladder. By concatenating the vectors D_i and M_i for all i and compressing them by a data structure for rank/select for dense sets [23] (see Section 2.1), the total space for storing all ladders is

$$\log\binom{\frac{n}{\log n}\cdot\log^4 n}{\sum_i n_i} + \mathrm{o}\left(\sum_i n_i\right) = \mathrm{O}\left(\frac{n\log\log n}{\log n}\right).$$

Note that if the number of the pioneers is small, we can add dummy elements to make the set dense.

To find $LA(v_i, \ell)$ for a lower level pioneer v_i which is on the i-th ladder, we find its farthest lower level pioneer ancestor w with depth no less than ℓ by $w = \mathit{select}_1(M_i, \mathit{rank}_1(D_i, \ell))$. Then we obtain $LA(v_i, \ell)$ by table lookups if it belongs to the same huge block as v_i.

If the level-ancestor is not in the same huge block as v, we use a data structure for the upper level that is similar to the lower level. Because there are $\mathrm{O}(n/\log^4 n)$ upper level pioneers for huge blocks, the data structure is stored in $\mathrm{O}(n/\log n)$ bits.

3.4 Childrank

To compute *childrank*(v), i.e., the i such that v is the i-th child of its parent, proceed as follows. First determine if v is the root, e.g., by checking if $\mathit{select}_{\boxed{)}}(v) = 0$, and if so, return 0. If v is not the root, count the number of left siblings of v by finding the opening parenthesis in the description of the parent of v which matches the closing parenthesis immediately before the current node, and then counting how many opening parenthesis there are between this position and the end of the description of the parent node. More precisely, when v is not the root of the tree, the childrank of v is given by the expression:

$$\mathit{select}_{\boxed{)}}(\mathit{rank}_{\boxed{)}}(\mathit{findopen}(v-1))+1) - \mathit{findopen}(v-1)$$

Each of the involved operations takes $O(1)$ time, so the running time for *childrank*(v) is $O(1)$, and no additional space is needed.

4 Compressed DFUDS

We now consider how to compress the DFUDS U of a tree T with n nodes. Let σ be the maximum degree of nodes in T. The basic idea is to convert the unary degree encoding of DFUDS into a binary encoding. Let $S[1..n]$ be an integer array storing the degrees of nodes of T in preorder. Each element of S is encoded in $\log\sigma$ bits. Note that S is equivalent to U if we replace every $S[i]$ by a sequence of $S[i]$ open parentheses $\boxed{(}$ followed by a close parenthesis $\boxed{)}$. Hence, it is obvious how to convert between S and U in $\mathrm{O}(n)$ time (see Figure 1). We show how to compress S into $nH^*(T) + \mathrm{o}(n)$ bits, and how to retrieve any consecutive $\log n$ bits of U from the compressed representation of S in constant time.

A similar approach is used in the original paper for DFUDS [2] to encode cardinal trees. They use prefix codes to encode node degrees for a special case $\sigma = 4$. Below, we propose data structures for ordered trees which achieve the entropy bound for a general alphabet.

4.1 Trees with constant maximum degrees

First we consider how to encode a tree with constant maximum degree, that is, the alphabet size of S is a constant. We apply the method from Section 2.3 to compress S into $nH_k(S) + \mathrm{O}(n\log\log n/\log n)$ bits so that we can obtain any consecutive $\log n$ numbers of S in constant time. Therefore we can assume that we had S as in the uncompressed form.

We give a data structure such that for any i, we can decode $U[i..i+w-1]$, where $w = \frac{1}{2}\log n$, in constant time from S. Define a mapping f from U to S such that if the unary code of $S[i]$ is encoded in $U[l_i..r_i]$, then $f(j) = i$ for $l_i \le j \le r_i$. (By a *unary code*, we mean the sequence of consecutive open parentheses $\boxed{(}$ followed by a close parenthesis $\boxed{)}$ in S that encode a particular node.) For each $U[jw]$ $(j = 1, 2, \ldots, n/w)$ we mark the position $m_j = f(jw)$ of S. We can use FID to mark them using $\mathrm{O}(n\log\log n/\log n)$ bits. We also store for each $U[jw]$ the offset of the position jw from the starting position of the unary code for $S[m_j]$. That is, if $d = S[m_j]$ is encoded in $U[l..l+d]$ $(l \le jw \le l+d)$, the offset is $jw - l$. It is stored in $\mathrm{O}(\log w) = \mathrm{O}(\log\log n)$ bits.

Without loss of generality we can assume i is a multiple of w. To decode $U[i..i+w-1]$, we first find the position of the element $S[m_j]$ ($j = i/w$) that is encoded around $U[i]$. The number of elements of S corresponding to the substring of U of length w is at most w, and the elements are stored in $\mathrm{O}(\log n)$ bits. Therefore we can convert them into a sequence of unary codes in constant time using table lookups.

The size of the data structure is $nH_k(S) + \mathrm{O}(n(k + \log\log n)/\log n)$ bits for any $0 \le k \le \mathrm{O}(\log\log n)$. Because $H_0(S) = H^*(T)$, the size is $nH^*(T) + \mathrm{O}(n \log\log n/\log n)$. Recall that any operation on the original DFUDS can be done in the same time complexity.

4.2 Trees with unbounded degrees First we divide the alphabet $\mathcal{A}$ of S into two sets; $\mathcal{A}_1$ for values larger than or equal to $\log n$, and $\mathcal{A}_2$ for the rest. We then define strings S_1 and S_2 which are the restrictions of S to $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. The alphabet size of S_2 is $\sigma = |\mathcal{A}_2| \le \log n$ and each value is encoded in $\log\log n$ bits. We compress S_1 and S_2 in different ways. To obtain a substring of U, we first extract substrings of U which are from S_1 and S_2, and then merge them.

We first describe how to compress S_1. We use a bit-vector D_1 of length n to indicate if $S[i] \in \mathcal{A}_1$ by setting $D_1[i] = 1$. Because there are at most $n/\log n$ values in $\mathcal{A}_1$, we can encode D_1 in $\mathrm{O}(n \log\log n/\log n)$ bits by using FID. We also use two bit-vectors D_2 and D_3 which represent the starting and ending positions of unary codes for the values in S_1. Namely, if an integer k is encoded in $U[i..i+k]$, then we set $D_2[i] = 1$ and $D_3[i+k] = 1$. These vectors are also stored in $\mathrm{O}(n \log\log n/\log n)$ bits. From these vectors, we can obtain a substring of U with length w which is from S_1 in constant time.

Next, we describe how to compress S_2. The auxiliary data structure used to decode U from S_2 is the same as the one for constant maximum degrees. However, in the worst case, we need $\log n \log\log n$ bits of S to decode $\log n$ bits of U because each character of S consists of $\log\log n$ bits. Therefore the time complexity to decode $\mathrm{O}(\log n)$ bits of U would be $\mathrm{O}(\log\log n)$ if we temporarily decode $\log n$ characters in uncompressed form. We avoid this problem by changing the internal encoding of the data structure of [31]. Recall from Section 2.3 that using this scheme, S_2 is parsed into phrases and they are stored as paths in some tree. To decode any path of $\log n$ bits in constant time, paths of length w are stored explicitly for branching nodes. Originally the paths are stored by fixed-length encoding, that is, each character is encoded in $\log\sigma = \log\log n$ bits. See [31] for details on how extract the paths. We change this so that any integer i (i.e., a symbol from $\mathcal{A}_2$) is encoded by a variable-length encoding in $\mathrm{O}(\log(i+2))$ bits by using for example the gamma code or the delta code [4]. We then extend the paths towards the root so that each path contains the maximum number of characters on the path towards the root that can be encoded in $\frac{1}{2}\log n$ bits by the variable-length codes. The paths on unary nodes are also stored by the variable-length codes. For each substring of S containing many phrases, which is stored explicitly, we also extend it so that it stores the maximum number of characters that can be stored in $\frac{1}{2}\log n$ bits. We can read any integer stored by a variable-length code in constant time because we can store pointers to the codes in $\mathrm{O}(n \log\log n/\log n)$ bits by FID. It is obvious that any sequence of numbers encoded in $\frac{1}{2}\log n$ bits in DFUDS is also encoded in $\mathrm{O}(\log n)$ bits by the variable-length encoding. Therefore we can obtain any $\log n$-bit sequence of unary codes of U from the variable-length encoding in constant time by table lookups. The details will be given in the full paper.

Finally, we merge the sequences of unary codes from S_1 and S_2 as follows. To decode $U[jw..(j+1)w-1]$, first obtain positions of characters in S which is the first and the last ones in the substring. Then we check if there is a character in $\mathcal{A}_1$ between them. If so, compute its position by using D_2 and D_3 and decode the unary code. Because each integer in S_1 is encoded in at least $\log n$ bits in U, any $\log n$-bit substring of U overlaps with at most two integers in S_1. Therefore it is easy to concatenate the unary codes from S_1 and S_2 in constant time.

The space for storing the compressed U is as follows. For the string S_1, we can store it using D_1, D_2 and D_3 in $\mathrm{O}(n \log\log n/\log n)$ bits. The string S_2 is encoded in $n'H_0(S_2) + \mathrm{O}(n \log\log n/\log_\sigma n)$ bits by Theorem 2.1 where n' is the length of S_2. Let n_i be the number of occurrences of integer i in S. Then

$$n'H_0(S_2) = \sum_{i \in \mathcal{A}_2} n_i \log \frac{n'}{n_i} \le \sum_{i \in \mathcal{A}} n_i \log \frac{n}{n_i} = nH^*(T).$$

Therefore the total space is $nH^*(T) + \mathrm{O}(\frac{n \log\log n}{\log_\sigma n}) = nH^*(T) + \mathrm{O}(\frac{n(\log\log n)^2}{\log n})$ bits. Note that the compressed size may be much smaller than $nH^*(T)$ because we can achieve $nH_k(S) \le nH_0(S) = nH^*(T)$.

Actually our compression technique works not only for the degree sequence for a tree, but also for any sequence of unary codes.

COROLLARY 4.1. *For a sequence of n integers encoded by unary codes, let n_i denote the number of occurrences of integer $i \ge 0$. If the summation of all the integers* *is* $\mathrm{O}(n)$*, the sequence is compressed in* $\sum_i n_i \log \frac{n}{n_i} + \mathrm{O}(n(\log\log n)^2/\log n)$ *bits so that any* $\log n$ *bits of the sequence is retrieved in constant time on word RAM.*

5 Applications

We describe some applications of our ultra-succinct representation of ordered trees. The proofs will be given in the full paper.

5.1 Labeled tree encoding Ferragina et al. [5] proposed **xbw**, a transformation between a rooted, ordered, edge-labeled tree T and two strings S_α and S_{last}. Each label is in the alphabet $\mathcal{A}$ with alphabet size σ. Let n be the number of nodes in T. The string S_α is a permutation of edge labels of T and the string S_{last} is a 0,1-string of length $2n$ representing the topology of T. They showed that tree navigational operations can be done on the strings. The size of the strings is $n \log \sigma + 2n$ bits, which matches the information-theoretic lower bound. They defined the k-th order entropy of the labels $H_k(T)$ and showed the string S_α is compressed into that entropy:

Theorem 5.1. ([5]) *Let* $\mathcal{C}$ *be a compressor that compresses any string* w *into* $|w|H_0(w) + \mu|w|$ *bits. The string* $\mathbf{xbw}(T)$ *can be compressed in* $nH_k(T) + n(\mu + 2) + \mathrm{o}(n) + g_k$ *bits, where* g_k *is a parameter that depends on* k *and on the alphabet size (but not on* $|w|$*).*

In the above theorem only the string S_α is compressed. In this paper we consider to compress the other string: S_{last}. It encodes the degrees of the nodes of T by unary codes after the stable sort. Therefore by using Corollary 4.1, we can compress it into the tree degree entropy $H^*(T)$. We obtain the following theorem:

Theorem 5.2. *The string* $\mathbf{xbw}(T)$ *of a labeled tree* T *can be compressed in* $nH_k(T) + nH^*(T) + \mathrm{o}(n \log \sigma) + g_k$ *bits, and any consecutive* $\mathrm{O}(\log n)$ *bits of* **xbw** *can be decoded in constant time on word RAM.*

5.2 Ultra-succinct compressed suffix trees *Suffix trees* [16] are useful data structures for string matching. Here, a given string S of length s is preprocessed in $\mathrm{O}(s)$ time to build its suffix tree so that for any pattern P its occurrences in S can be determined quickly. Many problems on string matching are solved efficiently using the suffix tree [12], for example finding the longest common substring of any two strings in linear time, finding the length of the longest common prefix of two suffixes in constant time, etc. For this kind of problems, the rich structure of the suffix tree is important.

A drawback of the suffix tree is that it requires huge space. The size of the data structure is $\mathrm{O}(s \log s)$ bits, which is not practical for large collections of documents. To reduce the size, *compressed suffix trees* have been proposed [30]. The compressed suffix tree for a string S consists of three components: the tree topology, the string depths, and the compressed suffix array [11] of S. Each occupies $2(s+t) + \mathrm{o}(s)$ bits, $2s + \mathrm{o}(s)$ bits, and $|CSA(S)|$ bits, respectively, where $t \le s-1$ is the number of internal nodes, and $|CSA(S)|$ denotes the size of the compressed suffix array of S. In total, the compressed suffix tree for a string S has size $|CSA(S)| + 4s + 2t + \mathrm{o}(s)$ bits [30].

For the size of the compressed suffix array, one implementation achieves the asymptotically optimal size $sH_k(S) + \mathrm{o}(s)$ bits [10]. Below, we show how to further reduce the size of the tree topology.

In the original compressed suffix tree for a string of length s [30], the tree topology is encoded by the BP in $2(s+t)$ bits. We change it to use the DFUDS to compress the data structure into the tree degree entropy. The following additional operations need to be supported for a suffix tree T:

- *string_depth*(v): the length of the path-label of a node v (the characters on the path from the root to v),
- *sl*(v): the node with path-label α if an internal node v has path-label $c\alpha$ for some character c (the suffix link of v).

To support the above functions, we need the following in addition to *lca* [30], which can be also computed on the DFUDS:

- $leaf\_rank(v) = rank_{\boxed{))}}(v)$
- $leaf\_select(i) = select_{\boxed{))}}(i)$
- $preorder\_rank(v) = (rank_{\boxed{)}}(v - 1)) + 1$
- $preorder\_select(i) = (select_{\boxed{)}}(i - 1)) + 1$
- $inorder\_rank(v) = rank_{\boxed{))}}(second\_child(v) - 1)$
- $inorder\_select(i) = parent((select_{))}(i)) + 2)$
- $leftmost\_leaf(v) = leaf\_select(leaf\_rank(v-1)+1)$
- $rightmost\_leaf(v) = findclose(enclose(v))$

The DFUDS sequence for the suffix tree can be compressed into the tree degree entropy. Furthermore, if the alphabet is binary, the tree topology is encoded in $2s + \mathrm{o}(s)$ bits:

Theorem 5.3. *The tree topology of the suffix tree* T *with* s *leaves and* t *internal nodes can be encoded in* $s \log \frac{s+t}{s} + t \log \frac{s+t}{t} + 2t + \mathrm{o}(s)$ *bits. Any operation on the compressed suffix tree is done in the same complexity as the one using the BP representation. Especially, if* $\sigma = 2$*, the tree topology can be encoded in* $2s + \mathrm{o}(s)$ *bits.*

Note that $s \log \frac{s+t}{s} + t \log \frac{s+t}{t} + 2t < 2(s+t)$ for any $0 < t < s$. Therefore our representation is always smaller than the BP.

6 Concluding Remarks

In this paper we have given a natural definition of the entropy of tree topology and proposed a succinct data structure for storing a tree whose size matches this entropy. Each fundamental operation on succinct representation of ordered trees [19, 2] is done in constant time. We also showed applications to reduce the size of the compressed suffix trees [30] and labeled trees [5] further. An open problem is to support complex queries on DFUDS such as finding the nearest node to the right having the same depth as the query node, which can be computed on BP in constant time [21].

References

[1] M. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.

[2] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing Trees of Higher Degree. *Algorithmica*, 43(4):275–292, 2005.

[3] Y.-T. Chiang, C.-C. Lin, and H.-I. Lu. Orderly Spanning Trees with Applications. *SIAM Journal on Computing*, 34(4):924–945, 2005.

[4] P. Elias. Universal codeword sets and representation of the integers. *IEEE Trans. Inform. Theory*, IT-21(2):194–203, March 1975.

[5] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. IEEE FOCS*, pages 184–196, 2005.

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

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

[8] R. F. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. In *CPM*, pages 159–172, 2004.

[9] R. F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. In *Proc. ACM-SIAM SODA*, pages 1–10, 2004.

[10] R. Grossi, A. Gupta, and J. S. Vitter. High-Order Entropy-Compressed Text Indexes. In *Proc. ACM-SIAM SODA*, pages 841–850, 2003.

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

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

[13] W. K. Hon, K. Sadakane, and W. K. Sung. Succinct Data Structures for Searchable Partial Sums. In *Proc. of ISAAC*, pages 505–516. LNCS 2906, 2003.

[14] G. Jacobson. Space-efficient Static Trees and Graphs. In *Proc. IEEE FOCS*, pages 549–554, 1989.

[15] R. Kosaraju and G. Manzini. Compression of low entropy strings with lempel-ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.

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

[17] D. R. Morrison. Patricia - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[18] 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.

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

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

[21] J. I. Munro and S. S. Rao. Succinct Representations of Functions. In *Proceedings of ICALP*, LNCS 3142, pages 1006–1015, 2004.

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

[23] R. Pagh. Low Redundancy in Static Dictionaries with Constant Query Time. *SIAM Journal on Computing*, 31(2):353–363, 2001.

[24] C. K. Poon and W. K. Yiu. Opportunistic Data Structures for Range Queries. In *Proceedings of COCOON*, LNCS 3595, pages 560–569, 2005.

[25] R. Raman, V. Raman, and S. S. Rao. Succinct Indexable Dictionaries with Applications to Encoding k-aray Trees and Multisets. In *Proc. ACM-SIAM SODA*, pages 233–242, 2002.

[26] S. Srinivasa Rao. Time-space trade-offs for compressed suffix arrays. *Information Processing Letters*, 82(6):307–311, 2002.

[27] G. Rote. Binary trees having a given number of nodes with 0, 1, and 2 children, 1996. Semin. Lothar. Comb. 38, B38b, 6 pp. `http://www.emis.de/journals/SLC/wpapers/s38proding.html`.

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

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

[30] K. Sadakane. Compressed Suffix Trees with Full Functionality. *Theory of Computing Systems*, 2005. In press.

[31] K. Sadakane and R. Grossi. Squeezing Succinct Data Structures into Entropy Bounds. In *Proc. ACM-SIAM SODA*, pages 1230–1239, 2006.