Ultra-succinct Representation of Ordered Trees

Jesper Jansson*  Kunihiko Sadakane†  Wing-Kin Sung‡

April 8, 2006

Abstract

There exist two well-known succinct representations of ordered trees: BP (balanced parenthesis) [Jacobson FOCS89] and DFUDS (depth first unary degree sequence) [Benoit et al. Algorithmica 2005]. Both have size $2n + o(n)$ bits for $n$-node trees, which matches the information-theoretic lower bound. Many fundamental operations on trees are done in constant time on word RAM, for example finding parent, the first child, next sibling, the number of descendants, etc. However there has been no 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 existing operation 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 whose any 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 different 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 $(2n+1)/(2n+1) \approx 4^n$ such trees. Typical succinct data structures are the ones for storing ordered sets [23, 25, 24, 13], ordered trees [14, 19, 7, 8, 2, 21, 22, 3, 29], strings [9, 11, 10, 26, 31, 30], functions [22], cardinal trees [5], etc. The size of a succinct data structure storing an object from the universe is $(1 + o(1)) \log L$ bits$^2$. Many basic operations on the object can be done in constant time on the word RAM model with word-length $\Theta(\log n)$, for example, finding the parent of a node in a tree.

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

---

*Department of Computer Science and Communication Engineering, Kyushu University. Supported by JSPS (Japan Society for the Promotion of Science). jj@tcslab.csce.kyushu-u.ac.jp
†Department of Computer Science and Communication Engineering, Kyushu University. Hakozaki 6-10-1, Higashi-ku, Fukuoka 812-8581, Japan. sada@csce.kyushu-u.ac.jp 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.
$^2$Some papers use a weaker definition of succinctness that allows $O(\log L)$ bits.
1. No single succinct data structure supports all basic operations in constant time; the balanced parenthesis representation \([19, 7]\) (BP) does not support \(i\)-th child, while the depth-first unary degree sequence representation \([2, 8]\) (DFUDS) does not support lowest common ancestor (lca).

2. Though the space is optimal in the worst case, it is not optimal for trees belonging to a subset. For example, any \(n\)-node tree whose internal node has exactly two children can be encoded in \(n\) bits if we do not consider efficient queries, whereas both the BP and the DFUDS require \(2n\) 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 trees \([32, 16]\) or compressed suffix trees \([28]\). 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 \(4n + o(n)\) bits because the tree has \(2n - 1\) nodes. On the other hand, if we use the Patricia tree \([17]\), \(2n\) bits are enough to encode the suffix tree, though we do not know how to support efficient queries. Therefore we pay extra \(2n\) bits for supporting efficient queries. This is enormous for huge collection of documents.

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 basic operations on ordered trees in constant time.

2. Its size surpasses the information-theoretic lower bound and achieves the entropy of the tree defined below.

We define the tree shape entropy of an ordered tree \(T\) with \(n\) nodes as \(H^d(T) = \sum_i \frac{n_i}{n} \log \frac{n_i}{n}\) where \(n_i\) is the number of nodes having \(i\) children. 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

\[
\frac{1}{n} \binom{n}{n_0, n_1, \ldots, n_{n-1}},
\]

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 \(\log L \approx nH^d(T)\).

### 1.1 Our contribution

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

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

This theorem 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 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 \( o(n) \)-bit auxiliary data structures for computing \( \text{lca} \), \( \text{depth} \) and \( \text{level-ancestor} \) on the original DFUDS. Though the data structure of [8] supports \( \text{depth} \) and \( \text{level-ancestor} \), it does not support \( \text{lca} \). Moreover, 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 for 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 \( O(n(\log \log n)^2/\log n) \)-bit auxiliary data structures.

We also show applications of our succinct representation of ordered trees. The first one is space reduction of the compressed suffix trees [28] 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 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 shape entropy, while preserving the same query time complexities.

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 \( \text{lca} \), \( \text{depth} \) and \( \text{level-ancestor} \) on DFUDS, which is summarized as Theorem 1.2. In Section 4 we give the data structure to compress the DFUDS into the tree shape entropy to prove Theorem 1.1. In Section 5 we show applications of our new representation of trees for reducing the size of compressed suffix trees and labeled 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 \( \sigma \). We define \( \text{rank} \) and \( \text{select} \) for \( S \) as follows. \( \text{rank}_c(S, i) \) is the number of \( c \) in \( S[1..i] \), and \( \text{select}_c(S, i) \) is the position of the \( i \)-th occurrence of \( c \) in \( S \). Note that \( \text{rank}_c(S, \text{select}_c(S, i)) = i \). We may omit \( S \) if it is clear from context.

There exist many succinct data structures for rank/select [23, 25, 14, 18]. A basic one uses \( n + o(n) \) bits for \( \sigma = 2 \) [18] and supports rank/select in constant time on word RAM with word length \( O(\log n) \). 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 \( \log \binom{n}{m} + O(n \log \log n / \log n) = m \log \frac{n}{m} + \Theta(m) + O(n \log \log n / \log n) \) bits [25]. This data structure is called **fully indexable dictionary** or FID. If \( m = O(n / \log n) \), the space is \( O(n \log \log n / \log n) \). 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 \( n \log \sigma + o(n \log \sigma) \) bits [6], 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, \( \text{rank}_P(S, i) \) is the number of occurrences of the pattern \( P \) whose starting positions are in \( S[1..i] \), and \( \text{select}_P(S, i) \) is the starting position of the \( i \)-th occurrence of \( P \). Note that occurrences of \( P \) may overlap in \( S \).

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 \( 1/2 \log n \). Because there exist only \( 2^{\frac{1}{2} \log n} = \sqrt{n} \) different
patterns, we can store all answers in a table using $\sqrt{n} \cdot \text{polylog}(n) = o(n)$ 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 $\frac{(2n+1)}{2n+1}$ such trees. 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, 7, 8, 2, 21, 22, 3, 29]. Among them, the most famous one 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 (()) and all subtrees rooted at the node is encoded in order between the matching parentheses.

To support tree navigational operations, the following operations are supported in constant time on the word RAM:

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

By using these operations, the following are supported:

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

These operations are done in constant time using $o(n)$-bit auxiliary data structures except for $\text{child}$ and $\text{childrank}$.

The DFUDS (depth-first unary degree sequence) representation [2, 8] of an ordered tree is defined as follows. A tree with only one leaf is represented as $()$, which is the same as the BP. If a tree $T$ rooted at a node $r$ has $k$ subtrees $T_1, \ldots, T_k$, the DFUDS of $T$ is the concatenation of $k + 1$ $\text{[}$ for a $\text{]}$ and DFUDS's of $T_1, \ldots, T_k$, with the first $\text{[}$ for each $T_1, \ldots, T_k$ being omitted. Then the resulting DFUDS also forms a balanced parenthesis sequence. A node with degree $k$ is encoded by $k$ $\text{[}$'s, followed by a $\text{]}$. We use the position of the leftmost parenthesis of the encoding of a node as its representative. The parenthesis is $\text{[}$ for internal nodes, and $\text{]}$ for leaves. The leftmost $\text{[}$ of the DFUDS of any tree is considered as a dummy parenthesis. 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 any operations for BP in constant time, except for $\text{lca}$, $\text{depth}$, and $\text{LA}$. Moreover, $\text{depth}$ and $\text{LA}$ can be supported if we modify DFUDS [8]. An example of BP and DFUDS is shown in Figure 1.

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 $\sigma$ is compressed into $nH_k(S) + O(n(k \log \sigma + \log \log n) / \log \sigma n)$ bits so that any substring of $S$ of length $O(\log \sigma n)$ (i.e., $O(\log n)$ bits)
Figure 1: Succinct representations of trees.

is decoded 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 $\text{FID}$ for a set $\hat{S} \subset \{1,2,\ldots,n\}$ is compressed into $nH_k(S) + O(n \log \log n / \log n)$ bits for any $0 \leq k \leq O(\log \log n)$ where $S$ is a 0,1-string such that $S[i] = 1 \iff i \in \hat{S}$, while the time complexities for $\text{rank}$ and $\text{select}$ remain unchanged.

In their data structure the string $S$ is parsed into distinct substrings called phrases and they are stored in a tree. To guarantee to decode any $1/2 \log_\sigma n$ characters ($1/2 \log n$ bits) of the string at the same time, some paths of the tree and some short phrases are stored explicitly. 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. 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 the ladder is easily found. The total number of nodes on all ladders is at most $2n$. We use another data structure called jump-pointers. For some nodes $v$ of $T$ called macro nodes, we store $\text{LA}(v, \ell)$ for $\ell = d-1, d-2, d-4, d-8, \ldots$ where $d = \text{depth}(v)$.

The query $\text{LA}(v, \ell)$ is solved as follows. First we 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 $\ell$ by following a jump-pointer. Then we reach a doubled long-path ladder including $\text{LA}(v, \ell)$. Therefore it is obvious to obtain it.
3 New Operations on DFUDS

In this section we propose simple algorithms and data structures for supporting \textit{lea}, \textit{depth}, \textit{level-ancestor} and \textit{childrank} on the original DFUDS. The algorithm for \textit{lea} is completely new. For operations \textit{depth} and \textit{level-ancestor}, Geary et al. [8] 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 \textit{depth} and \textit{level-ancestor} on the original DFUDS. These data structures are much simpler than those of [8]. More importantly, we improve the lower order term of the size for \textit{level-ancestor}. The previous ones use $O(n \log \log n / \sqrt{\log n})$ bits [8, 22], while our new data structure uses $O(n(\log \log n)^2 / \log n)$ bits. The algorithm for \textit{childrank} is also proposed in [8] for the modified DFUDS. We show a simple algorithm for the original DFUDS.

3.1 LCA Let $U$ be the DFUDS of a tree $T$. The excess sequence $L$ of $U$ is defined so that $L[i] = (\text{number of } \mathbb{K} \text{ in } U[0..i]) - (\text{number of } \mathbb{K} \text{ in } U[0..i])$. Note that the excess values represent node depths if we regard $U$ as a BP sequence, but they differ from node depths of $T$.

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 the subtree of $T$ rooted at $v$. We also assume that $U[l_i..r_i]$ stores the DFUDS for $T_i$ for $1 \leq i \leq k$. Note that $l_i = r_{i-1} + 1$. Let $d = L[r_0]$. Then we have the following property of the excess values.

\textbf{Lemma 3.1.}

$\begin{align*}
L[r_i] &= L[r_{i-1}] - 1 = d - i \\ L[j] &> L[r_i] \quad (l_i \leq j < r_i)
\end{align*}$

\hfill \square

\textit{Proof.} By the construction of DFUDS, if we add a $\mathbb{K}$ 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 $L[r_i] = L[r_{i-1}] - 1$ for $1 \leq i \leq k$. Because $L[r_0] = d$, we have $L[r_i] = d - i$. The second property $L[j] > L[r_i]$ ($l_i \leq j < r_i$) is obvious because the sequence is balanced if an open parenthesis is added at the beginning. \hfill \square

\textbf{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.

\hfill \square

\textit{Proof.} The \textit{lea}(\textit{x}, \textit{y}) ($\textit{x} < \textit{y}$) is computed as follows.

\begin{align*}
  w &= \text{RMQ}_L(x, y - 1) + 1 \\
  z &= \text{parent}(w)
\end{align*}

where $x$ and $y$ are the preorders of the nodes and $\text{RMQ}_L(x, y - 1)$ is called \textit{range minimum query} and returns the position of the smallest element in $L[x, y - 1]$. If there is a tie $\text{RMQ}$ returns the leftmost position. $\text{RMQ}$ can be computed in constant time using an $O(n(\log \log n)^2 / \log n)$-bit auxiliary data structure [29]. Let $v$ be the true $\text{lea}(x, y)$, $T_1, \ldots, T_k$ be the subtrees of $v$, and $U[l_i..r_i]$ be the DFUDS for $T_i$ ($1 \leq i \leq k$). Then $x$ and $y$ are in some subtrees $T_\alpha$ and $T_\beta$ ($\alpha < \beta$), respectively.
Assume that $L[\beta] = d$. Then from Lemma 3.1 $L[\beta - 1] = d + 1$ and $L[i] > d + 1$ for $l_1 \leq i < r_{\beta - 1}$ and $l_\beta \leq i \leq r_\beta - 2$. If $y < r_\beta$, $L[y - 1] > d + 1$, and by the range minimum query we obtain $w = RMQ(x, y - 1) + 1 = r_{\beta - 1} + 1 = l_\beta$. If $y = r_\beta$, $L[y - 1] = d + 1$. Therefore there are two minimum values $d + 1$ in $L[x..y - 1]$. By the range minimum query we can find the left one, which is $r_{\beta - 1}$. In either case, we have $w = l_\beta$, which is the position of a subtree of $v$. By computing $z = parent(w)$, we obtain $z = v = lca(x, y)$.

3.2 Depth We use two-level data structures, but we first explain the general data structure for each level. 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 store 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 $(v_1, p(v_1))$ (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, 7]. 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. An example is shown in Figure 2. 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 $B_L = 1/2 \log n$ and $M_L$ is the set of all nodes of $T$. We call the blocks small blocks. We call pioneers for small block 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 $select_I(J_s, rank_I(J_s, v))$ 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 $B_L = 1/2 \log n$, the difference of depths between them can be computed in constant time by table lookups. We define $M_U$ for the upper level as the set of all lower level pioneers.

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. For information $I_1$, we store for each node $v \in M_U$ the relative pointer from $v$ to $f(v)$ and the difference of their depth explicitly. Because both $v$ and $f(v)$ 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 $O(n \log n / \log n)$ 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 $f(v)$ in the small block of $v$ by table lookups. Then we compute the parent $w = p(f(v))$. 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 $z$, $f(z)$ is stored as a relative pointer from $z$. If $p(f(z))$ 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 similar data structure as Bender and Farach-Colton [1], but we change it to a two-level data structure to reduce the space. The lower level of the data structure is for computing $LA$ inside a large block of size $\log^4 n$. 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 $1/2 \log n$. 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 large 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 $d = \text{depth}(v)$. For each lower level pioneer we need $O((\log \log n)^2)$ bits and there are $O(n/\log n)$ pioneers. Therefore we need $O(n (\log \log n)^2 / \log n)$ 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 the upper level pioneer for $v_i$, which we call $w_i$. For the ladder for a lower level pioneer $v_1$, we use two bit-vectors $D_i[1..\log^4 n]$ and $M_i[1..\log^4 n]$. If a pioneer $v$ is on the ladder, $M_i[v - q_i] = 1$, and $D_i[\text{depth}(v) - \text{depth}(q_i)] = 1$. Let $n_i$ be the number of lower level pioneers on the $i$-th ladder. Then the total space for storing all ladders by FID is

$$\sum_i O \left( \log \left( \frac{\log^4 n}{n_i} \right) \right) = O \left( \log \left( \frac{\log n}{\sum_i n_i} \frac{\log^4 n}{\log n} \right) \right) = O \left( \frac{n \log \log n}{\log n} \right).$$

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

8
If the level-ancestor belongs to a different subtree with \( v \), we use a data structure for the upper level that is similar to the lower level. Because there are \( O(n/\log^3 n) \) upper level pioneers for large blocks the data structure is stored in \( O(n/\log n) \) bits.

### 3.4 Childrank

To compute \( \text{childrank}(v) \), 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 \( \text{select}_v(0) = 0 \), 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 parentheses 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:

\[
\text{select}_v(\text{rank}_v(\text{findopen}(v - 1)) + 1) - \text{findopen}(v - 1)
\]

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

### 4 Compressed DFUDS

We consider how to compress the DFUDS \( U \) of a tree \( T \) with \( n \) nodes. Let \( \sigma \) be the maximum degree of nodes in \( T \). The basic idea is to convert the unary degree encoding of DFUDS into 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. It is obvious how to convert between \( S \) and \( U \) in \( O(n) \) time. We show how to compress \( S \) into \( nH^d(T) + o(n) \) 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 general alphabet.

#### 4.1 Trees with constant maximum degrees

First we consider how to encode a tree with constant maximum degree. Thus the alphabet size of \( S \) is a constant. We apply the method from Section 2.3 to compress \( S \) into \( nH_k(S) + O(n \log \log n / \log n) \) 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 show that for any given \( i \), we can decode \( U[i..i + w - 1] \) where \( w = 1/2 \log n \) in constant time from \( S \). We define a mapping \( f \) from \( U \) to \( S \) so that if \( S[i] \) is encoded in \( U[l_i..r_i] \), \( f(j) = i \) for \( l_i \leq j \leq r_i \). For each \( U[jw] \) \((j = 1, 2, \ldots, n/w) \) we mark the position \( f(jw) \) of \( S \). We can use \textsc{find} to mark them using \( O(n \log \log n / \log n) \) bits. We also store for each \( U[jw] \) the offset of the position \( jw \) from the starting position of the unary code for \( S[m_j] \). (By a unary code, we mean the sequence of consecutive open parentheses \( [ \) followed by a close parenthesis \( ] \) in \( S \) that encode a particular node.) That is, if \( d = S[m_j] \) is encoded in \( U[l..l + d] \) \((l \leq jw \leq l + d) \), the offset is \( jw - l \). It is stored in \( O(\log \log n) \) 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 \( S[m_j] \) 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 \( O(\log n) \) 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 \( nH_k(S) + O(n(k + \log \log n) / \log n) \) bits for any \( 0 \leq k \leq O(\log \log n) \). Because \( H_0(S) = H^d(T) \), the size is \( nH^d(T) + O(n \log \log n / \log n) \). 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 \( A \) of \( S \) into two sets; \( A_1 \) for values larger than or equal to \( \log n \), and \( A_2 \) for the rest. We define a string \( S_2 \) that consists of all the
characters of $S$ in $A_2$. Then the alphabet size of $S'$ is $\sigma = |A_2| = \log n$ and each value is encoded in $\log \log n$ bits.

We use a bit-vector $D_1$ of length $n$ to distinguish them. If $D_1[i] = 1$, $S[i] \in A_1$, that is, the node of $T$ with preorder $i$ has a degree at least $\log n$. Because there are at most $n/\log n$ such nodes, we can encode $D_1$ in $O(n \log \log n/\log n)$ bits by using FID. We also use two bit-vectors $D_2$ and $D_3$ which represents the starting and ending positions of unary codes for the degrees. Namely, if a node with degree $k$, which is represented by $k$ open parentheses followed by a close parenthesis, is encoded in $U[i..i+k]$, then we set $D_2[i] = 1$ and $D_3[i+k] = 1$. These vectors can be also stored in $O(n \log \log n/\log n)$ bits.

The auxiliary data structure to decode $U$ from $S$ 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 $O(\log n)$ bits of $U$ will be $O(\log \log n)$ if we temporarily decode $\log n$ characters in uncompressed form.

We solve the above problem by using another intermediate encoding of $S$ when we decode it from the compressed representation of $\mathcal{S}$. Recall that $\mathcal{S}$ is parsed into phrases and they are stored as paths in some tree. We change the internal data structure of [31] that stores some substrings of $S$. Originally the substrings are stored by fixed-width encoding, that is, each character is encoded in $\log \sigma = \log \log n$ bits. We change this so that an integer $i$ is encoded by a variable-length encoding in $O(\log i)$ bits by using for example the gamma code or the delta code [4]. We also change the substrings so that each substring contains the maximum number of consecutive characters in $S$ that are encoded in $1/2 \log n$ bits by the variable-length codes. We can guarantee that any sequence of numbers encoded in $1/2 \log n$ bits in DFUDS is also encoded in $O(\log n)$ bits by the variable-length encoding. Therefore we can obtain the $\log n$ bits of $U$ in constant time from $S'$ by table lookups. We can also obtain the $\log n$ bits of $S$ by merging the two strings for $A_1$ and $A_2$ in constant time by table lookups.

The space for storing the compressed $U$ is as follows. For characters in $A_1$, we can store them using $D_1$, $D_2$ and $D_3$ in $O(n \log \log n/\log n)$ bits. For characters in $A_2$, the string $S'$ is encoded in $nH_0(S') + o(n)$ bits. Let $n_i$ be the number of characters $i$ in $S$. Then

$$nH_0(S') = \sum_{i \in A_2} n_i \log \frac{n'}{n_i} \leq \sum_{i \in A} n_i \log \frac{n}{n_i} = nH^d(T)$$

where $n' = \sum_{i \in A_2} n_i$. Therefore the total space is $nH^d(T) + O(n \log \log n/\log n) = nH^d(T) + O(n(\log \log n)^2/\log n)$ bits. Note that the compressed size may be much smaller than $nH^d(T)$ because we can achieve $nH_k(S) \leq nH_0(S) = nH^d(T)$.

5 Applications

5.1 Ultra-succinct compressed suffix trees

Suffix trees [32, 16] are data structures for string matching. A string $S$ of length $s$ on an alphabet $A$ is preprocessed so that for any pattern $P$ its occurrence in $S$ is 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. An example of the suffix tree is given in Figure 3.

A drawback of the suffix tree is that it requires huge space. The size of the data structure is $O(s \log s)$ bits, which is not practical for large collections of documents. To reduce the size, compressed suffix trees have been proposed [28]. The compressed suffix tree for a string $S$ consists of three components: the
Figure 3: The suffix tree for the string “ababac$,” and its DFUDS representation. Suffix links are shown in dotted lines.

tree structure, the string depths, and the compressed suffix array [11] of $S$. Each occupies $4s + o(s)$ bits, $2s + o(s)$ bits, and $|CSA(S)|$ bits, respectively, where $|CSA(S)|$ denotes the size of the compressed suffix array of $S$. In total, the compressed suffix tree for a string $S$ has size $|CSA(S)| + 6s + o(s)$ bits [28].

For the size of the compressed suffix array, one implementation achieves the asymptotically optimal size $sH_k(S) + o(s)$ bits [9]. Below, we show how to further reduce the size of the other two components: the tree structure and the string depths.

5.1.1 Compressing tree structures In the original compressed suffix tree for a string of length $s$ [28], the tree structure is encoded by the BP in $2(s + t)$ bits where $t \leq s - 1$ is the number of internal nodes. The main reason why the BP was used is that we needed to be able to compute the lca efficiently. We change it to use the DFUDS to compress the data structure into the tree shape entropy. (Recall that according to Section 3.1, we can now compute the lca efficiently for DFUDS.) Next we have to show that all required functions for suffix trees can be also implemented on the DFUDS efficiently. The following additional operations need to be supported for a suffix tree $T$:

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

We give the definition of an inorder of an internal node.

**Definition 1.** The inorder rank of an internal node $v$ is defined as the number of visited internal nodes, including $v$, in the depth-first traversal, when $v$ is visited from a child of it and another child of it will be visited next.

Note that there are no unary nodes in the suffix tree. Therefore each internal node has at least one inorder number. A node with $d$ children has $d - 1$ different inorder numbers. Note that the preordering
includes the leaves whereas the inordering does not. We need the following in addition to \( \text{lca} \) \cite{28}:

- \textit{leaf\_rank}(v): Return the number of leaves before or equal to the node \( v \) in the preordering of the tree.
- \textit{leaf\_select}(i): Return the \( i \)th leaf in the preordering of the tree.
- \textit{preorder\_rank}(v): Return the preorder number of the node \( v \).
- \textit{preorder\_select}(i): Return the node whose preorder number is \( i \).
- \textit{inorder\_rank}(v): Return the smallest inorder number of the internal node \( v \).
- \textit{inorder\_select}(i): Return the node whose inorder number is \( i \).
- \textit{leftmost\_leaf}(v), \textit{rightmost\_leaf}(v): Return the leftmost (rightmost) leaf in the subtree rooted at \( v \).

To compute inorders we use this lemma:

**Lemma 5.1.** \cite{28} There is a one-to-one correspondence between the leaves and the inorder numbers for the internal nodes, and an inorder rank of \( v \) is equal to the number of leaves that have smaller preorder ranks than \( v \).

\( \square \)

**Proof.** Inorder numbers are assigned during a depth-first traversal, which is divided into upgoing and downgoing paths. An inorder number is assigned to an internal node \( v \) if the node is between consecutive upgoing and downgoing paths, and each upgoing path starts from a leaf.

\( \square \)

We can implement each one of these operations to run in constant time and using no extra space as follows:

- \textit{leaf\_rank}(v) = rank_\[\,\](v)
  [Each leaf is represented by an occurrence of the pattern \[\], and the leaves appear in the sequence from left to right according to their preorder, so we count the number of occurrences of leaves to the left or equal to \( v \).]
- \textit{leaf\_select}(i) = select_\[\,\](i)
  [Find the \( i \)-th leaf according to the above.]
- \textit{preorder\_rank}(v) = (rank_\[\,\](v - 1)) + 1
  [The description of each node consists of a consecutive sequence of \[\] symbols followed by a single \[\], and the nodes appear in the sequence according to their preorder. Therefore \( \text{rank}_\[\,\](v - 1) \) gives the preorder of the preceding node of \( v \). By adding 1 we obtain the answer. ]
- \textit{preorder\_select}(i) = (select_\[\,\](i - 1)) + 1
  [Find the end of the description of the \( (i - 1) \)th node according to preorder, and go to the position immediately after. This is the inverse of \textit{preorder\_rank}.]
- \textit{inorder\_rank}(v) = \textit{leaf\_rank}(\text{child}(v, 2) - 1)
  [According to Lemma 5.1, count the number of leaves to the left of the second child of \( v \).]
- \textit{inorder\_select}(i) = \text{parent}((\text{leaf\_select}(i)) + 2)
  [Let \( v \) denote the desired node. First find the \( i \)-th occurrence of a leaf in the sequence. By adding 2 to the position we obtain the start position of the description of the first node reached by a downgoing edge from the node with inorder \( i \). By computing its parent we obtain the answer.]
- \textit{leftmost\_leaf}(v) = \text{leaf\_select}(\text{leaf\_rank}(v - 1) + 1)
  [We obtain the number of leaves appearing before \( v \) in preorder by \textit{leaf\_rank}(v - 1). By finding the next leaf in preorder we obtain the answer.]

12
The rightmost leaf is encoded in the last position of the encoding for the subtree rooted at \( v \). If it were encoded by the BP, we can find it by \( \text{findclose}(v) \). However in the DFUDS the leftmost open parenthesis is omitted. The omitted parenthesis is moved to another position to enclose \( v \).

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

**Theorem 5.1.** The tree shape of the suffix tree \( T \) with \( s \) leaves and \( t \) internal nodes can be encoded in \( s \log \frac{s+t}{s} + t \log \frac{s+t}{t} + 2t + o(s) \) 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 shape can be encoded in \( 2s + o(s) \) bits.

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

**Proof.** We already showed above that each necessary operation on the BP is also supported on the compressed DFUDS in the same time complexity.

We compute an upper bound of the tree shape entropy. We consider a two-level code which first encodes whether a node is leaf or not, then if it is not a leaf, encodes the degree of the node. The number of occurrences of value \( i \) is \( \sum n_i \) and \( \sum (i + 1) n_i \leq t + s < 2s \) therefore there exist at most \( 2s / \log s \) numbers which are greater than \( \log s \). We can use the same technique as Section 4.2 to compress this sequence. The compressed size is expressed by the entropy based on the frequency of numbers. Roughly speaking, the numbers represent edge lengths of the suffix tree. Therefore the frequency of 1 is high in practice. Though we cannot give any upper bound better than \( 2s \), we can expect good compression in practice.

5.1.2 Compressing string depths In the original compressed suffix tree [28], string depths of internal nodes are sorted by inorder ranks and conceptually stored in an array \( Hgt[1..t] \), which is actually represented by a sequence of \( t \) increasing numbers \( 0 \leq x_1 \leq x_2 \leq \cdots \leq x_t \leq s \), and they are encoded in a sequence of unary codes for \( x_i - x_{i-1} + 1 \) \( (i > 1) \). The length of the sequence is at most \( t + s < 2s \). We consider how to compress the sequence by converting it into a sequence of \( s \) integers. Let \( n_i \) be the number of occurrences of value \( i \). Then \( \sum n_i = s \) and \( \sum (i + 1) n_i \leq t + s < 2s \) therefore there exist at most \( 2s / \log s \) numbers which are greater than \( \log s \). We can use the same technique as Section 4.2 to compress this sequence. The compressed size is expressed by the entropy based on the frequency of numbers. Roughly speaking, the numbers represent edge lengths of the suffix tree. Therefore the frequency of 1 is high in practice. Though we cannot give any upper bound better than \( 2s \), we can expect good compression in practice.

5.1.3 Operations on suffix trees We state how to implement \( \text{string-depth}(v) \) and \( \text{sl}(v) \) for completeness. For details and the correctness, please refer to [28].

- \( \text{string-depth}(v) = Hgt[\text{inorder-\text{rank}}(v)] \)
- \( \text{sl}(v) = \lca(\text{leaf-\text{rank}}(\Psi[x]), \text{leaf-\text{rank}}(\Psi[y])) \) where \( x = \text{leaf-\text{rank}}(\text{leftmost-leaf}(v)) \) and \( y = \text{leaf-\text{rank}}(\text{rightmost-leaf}(v)) \).

5.2 Labeled tree encoding Ferragina et al. [5] proposed \( \text{xbw} \), a transformation between a rooted, ordered, edge-labeled tree \( T \) and two strings \( S_\alpha \) and \( S_{\text{last}} \). Each label is in the alphabet \( \mathcal{A} \) with alphabet size \( \sigma \). Let \( n \) be the number of nodes in \( T \). The string \( S_\alpha \) is a permutation of edge labels of \( T \) and the
string $S_{\text{last}}$ is a 0,1-string of length $2n$ representing the shape 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 $S_{\alpha}$ is compressed into that entropy:

**Theorem 5.2.** ([5]) Let $C$ be a compressor that compresses any string $w$ into $|w|H_0(w) + \mu|w|$ bits. The string $\text{xbw}(T)$ can be compressed in $nH_k(T) + n(\mu + 2) + o(n) + g_k$ 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 $S_{\alpha}$ is compressed. In this paper we consider to compress the other string: $S_{\text{last}}$. It encodes the degrees of the nodes of $T$ by unary codes. Therefore by using the same technique as for the DFUDS, we can compress it into the tree shape entropy $H^d(T)$. We obtain the following theorem:

**Theorem 5.3.** The string $\text{xbw}(T)$ of a labeled tree $T$ can be compressed in $nH_k(T) + nH^d(T) + o(n \log \sigma) + g_k$ bits, and any consecutive $O(\log n)$ bits of $\text{xbw}$ can be decoded in constant time on word RAM.

**Proof.** For the compressor $C$ of Theorem 5.2 we use the one in Theorem 2.1 for $k = 0$. Then $\mu = \log \sigma \log \log n / \log n$ and $S_{\alpha}$ is compressed into $nH_k(T) + o(n \log \sigma) + g_k$ bits. The string $S_{\text{last}}$ is compressed into $nH^d(T) + o(n)$ bits by Theorem 1.1. In both compression algorithms, any consecutive $O(\log n)$ bits can be decoded in constant time.

### 6 Concluding Remarks

In this paper we gave a natural definition of the entropy of tree shape and gave a succinct data structure for storing a tree whose size matches the entropy. Each existing operation on succinct representation of ordered trees [19, 2] is done in constant time. We also showed applications of the new succinct representation of trees. The size of the compressed suffix trees [28] and labeled trees [5] can be reduced further, while preserving the same query time complexities.

### References


