Journal of Computer and System Sciences 78 (2012) 619–631

Contents lists available at SciVerse ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Ultra-succinct representation of ordered trees with applications ☆

Jesper Jansson [a,1], Kunihiko Sadakane [b,*,2], Wing-Kin Sung [c,3]

[a] Ochanomizu University, 2-1-1 Otsuka, Bunkyo-ku, Tokyo 112-8610, Japan
[b] National Institute of Informatics (NII), Hitotsubashi 2-1-2, Chiyoda-ku, Tokyo 101-8430, Japan
[c] Department of Computer Science, National University of Singapore, 3 Science Drive 2, Singapore 117543

ARTICLE INFO

Article history:
Received 18 February 2010
Received in revised form 29 August 2011
Accepted 8 September 2011
Available online 14 September 2011

Keywords:
Succinct data structure
Ordered tree
Tree degree entropy
Xbw
Compressed suffix tree

ABSTRACT

There exist two well-known succinct representations of ordered trees: BP (balanced parenthesis) (Munro and Raman, 2001) [20] and DFUDS (depth first unary degree sequence) (Benoit et al., 2005) [1]. Both have size $2n + o(n)$ bits for n-node trees, which asymptotically matches the information-theoretic lower bound. Importantly, many fundamental operations on trees can be done in constant time on the word RAM when using BP or DFUDS, for example finding the parent, the first child, the next sibling, the number of descendants, etc. Although the space needed to store the BP or DFUDS representation of an ordered tree matches the lower bound, this is not optimal when we consider encodings for certain special classes of trees such as trees in which every internal node has exactly two children. In this paper, we introduce a new, conditional entropy for trees called the *tree degree entropy*, and give a succinct tree representation with matching size. We call such a representation an *ultra-succinct data structure*. We show how to modify the DFUDS representation to obtain a "compressed DFUDS", and as a consequence, a tree in which every internal node has exactly two children can be represented in $n + o(n)$ bits. We also describe applications of the compressed DFUDS representation to 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 in constant time. Here the information-theoretic lower bound for storing an object from a fixed universe with cardinality L is $\log L$ bits[4] because in the worst case this number of bits is necessary to distinguish two distinct objects. For example, the lower bound for storing 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 [20]. Typical succinct data structures include the ones for storing ordered sets [13,23–25], ordered trees [1,2,8,9,14,20–22,29], strings [6,10,11,26,28,30], functions [22], cardinal trees [1,5], etc. The size of a succinct data structure storing an object from the universe is typically $(1 + o(1)) \log L$ bits.[5] Furthermore, many

☆ A preliminary version of this paper appeared in Proc. ACM-SIAM SODA 2007, pp. 575–584.

* Corresponding author.
E-mail addresses: Jesper.Jansson@ocha.ac.jp (J. Jansson), sada@nii.ac.jp (K. Sadakane), ksung@comp.nus.edu.sg (W.-K. Sung).

[1] Work supported in part by the Special Coordination Funds for Promoting Science and Technology and KAKENHI grant No. 23700011.
[2] Work supported in part by the Grant-in-Aid of the Ministry of Education, Science, Sports and Culture of Japan (KAKENHI 23240002).

[3] Also affiliated with Genome Institute of Singapore, 60 Biopolis Street, Genome, Singapore 138672.

[4] The base of all logarithms is 2 unless specified. We define $0 \log 0 = 0$.

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

0022-0000/$ – see front matter
doi:10.1016/j.jcss.2011.09.002

fundamental operations on those objects 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.[6] Though there exist many such data structures in the literature such as BP and DFUDS (reviewed in Section 2.2), they have the following disadvantages.

1. Though the space is asymptotically optimal in the worst case, it is not optimal for certain special classes of trees. For example, any n-node tree whose internal nodes have exactly two children can be encoded in n bits if we do not require efficient queries 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.
2. There exist representations of ordered trees supporting many operations [3,9]. However they are based on the tree cover technique and therefore auxiliary data structures for supported queries work only for the representations.

The first drawback causes severe problems for document processing. Nowadays, many huge collections of documents such as web pages and genome sequences are available. To search such documents we can use suffix trees [17,33] or compressed suffix trees [29] because they support efficient queries. The compressed suffix tree uses the BP (and some auxiliary information) to encode the tree because *lca* is crucial, and the size of the BP is $4n + o(n)$ bits because the tree has $2n - 1$ nodes in the worst case (see Section 2.4). On the other hand, if we use the Patricia tree [18] 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. For example, an implementation of compressed suffix tree for human genome [32] using the BP has size 8.5 GB in total, while the genome sequence itself is stored in 750 MB. Therefore it is desirable to store the suffix tree structure as compactly as possible.?

Note that no data structure can store *any* n-node tree using less than cn bits for $c < 2$ because it would surpass the information-theoretic lower bound. However, if we consider only *typical* objects we can expect to reduce the size. This is the main idea behind 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, such data structures exist for strings [6,10,30] and ordered sets [24], but not yet for ordered trees.

1.1. Our contributions

In this paper we solve the above problems by providing an ultra-succinct representation of ordered trees called *compressed DFUDS* 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.
3. The representation of the tree topology is compressed but can be regarded as the original DFUDS because any one word ($\Theta(\log n)$ bits) of the representation can be decoded in constant time. Therefore the representation and the auxiliary data structures for supporting the queries can be split and it is easy to add new operations.

We introduce the *tree degree entropy* of an ordered tree. First let us consider the information-theoretic lower bound of an ordered tree, conditioned on the information that the tree comes from a class with specified degree multiplicities.

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

Let L denote this number. Then the information-theoretic lower bound for the condition is $\lceil \log L \rceil$. By using Stirling's approximation, we obtain $\lceil \log L \rceil = \sum_{i=0}^{n-1} n_i \log \frac{n}{n_i} - \Theta(\log n)$. Therefore we define the tree degree entropy as follows, which is an approximation of $\lceil \log L \rceil$.

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

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

[6] An ordered tree is a non-labeled rooted tree in which the order of children of each internal node is significant.

Note that $|nH^*(T) - \lceil \log L \rceil| = O(\log n)$.
Our main result is summarized as follows:

Theorem 1. *For any rooted ordered tree T with n nodes, there exists a data structure using $nH^*(T) + O(n \log\log n / \log n)$ bits such that any consecutive $\log n$ bits of DFUDS of T can be computed in constant time on word RAM.*

Note that $nH^*(T) \leqslant 2n$ for any tree, implying that the size of our data structure is never more than BP nor DFUDS. Theorem 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 an $o(n)$-bit auxiliary data structures for computing *lca*, *depth*, and *level-ancestor* on the original DFUDS. Geary et al. [9] proposed a data structure for ordered trees which supports *depth* and *level-ancestor*. However it does not support *lca*. Moreover, because it is based on a tree partition algorithm and the DFUDS is used only for representing subtrees, it is not guaranteed that any algorithm on the original DFUDS also works on this data structure. Therefore it is important to support these operations on the original DFUDS. We show the following:

Theorem 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.*

Our new auxiliary data structures have another benefit. Their size is smaller than the existing ones which use $O(n \log\log n / \sqrt{\log n})$ bits [8,9] or $O(n \log\log\log n / \log\log n)$ bits [3].

We also describe applications of our succinct representation of ordered trees. The first 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. The next one is space reduction of the compressed suffix trees [29] 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.

1.2. Organization of paper

The rest of the paper is organized as follows. Section 2 reviews existing succinct data structures. Section 3 proposes simple and space-efficient auxiliary data structures for *lca*, *depth*, and *level-ancestor* on the original DFUDS representation, as summarized in Theorem 2. Section 4 gives the data structure to compress the DFUDS into the tree degree entropy and thus proves Theorem 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 $\Theta(\log n)$ for representing an ordered tree with n nodes, where any arithmetic operation on two $\Theta(\log n)$-bit numbers and reading/writing consecutive $\Theta(\log n)$ bits of memory 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. For any $c \in \mathcal{A}$, $rank_c(S, i)$ is the number of occurrences of 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 [14,19,23,25]. A basic one [19] uses $n + o(n)$ bits for $\sigma = 2$ and supports rank/select in constant time on the 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, we use the following scheme to compress a string:

Theorem 3. *(See Ferragina and Venturini [7].) A string S of length n with alphabet size σ can be compressed into at most $nH_k(S) + O(n(k \log\sigma + \log\log n)/\log_\sigma n)$ bits for any $k \geqslant 0$ so that any substring of S of length $O(\log_\sigma n)$ (i.e., $O(\log n)$ bits) is decodable in constant time on word RAM.*

The rank/select functions for the case $\sigma = 2$ can be extended to counting occurrences of multiple characters [21]. 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

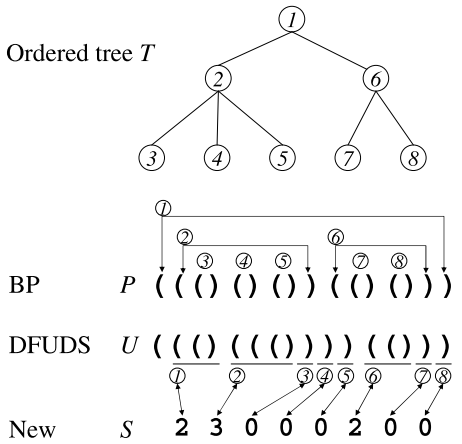

Fig. 1. Succinct representations of trees.

$S[1..i]$, 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.

A crucial technique for succinct data structures is *table lookup*. For small-size problems we construct a table which stores answers to 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 $\sqrt{n} \cdot \text{polylog}(n) = o(n)$ bits, which can be accessed in constant time on the 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 [20]. 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 [1,2,8,9,14,20–22,29].

2.2.1. Balanced parenthesis encoding (BP)

The most well-known representation of ordered trees is the balanced parenthesis representation [20], referred to as 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 Fig. 1 for an example). To allow tree navigational operations, the following operations are supported in constant time on the word RAM [20]:

- *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 *findclose*, *findopen*, and *enclose*, the following additional operations are supported [2,16,20,22,29]:

- *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.

These operations are done in constant time using $o(n)$-bit auxiliary data structures.

2.2.2. Depth-first unary degree sequence (DFUDS)

The DFUDS (depth-first unary degree sequence) representation [1,9] 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 has k subtrees $T_1, \ldots, T_k$, the DFUDS of T

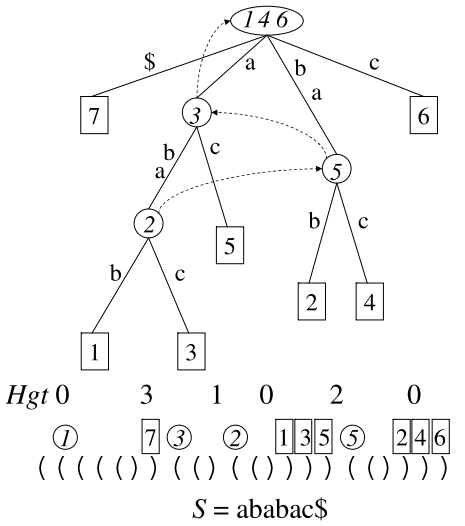

Fig. 2. The suffix tree for the string "ababac$" and its DFUDS representation. Suffix links are shown in dotted lines.

is the concatenation of $k+1$ [(, a [), and DFUDS's of $T_1, \ldots, T_k$, with the first [(for each $T_1, \ldots, T_k$ being omitted. Then the resulting DFUDS also forms a balanced parenthesis sequence (see Fig. 1 for an example). The leftmost [(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 [(]'s, followed by a [)]. We use the position of the leftmost parenthesis of the encoding of a node as its representative. Such parenthesis is [(for internal nodes, and [)] for leaves. We assume the position of parentheses begins with 0. Therefore the position of the root node is 1. The DFUDS [1] 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. Range min–max trees

Sadakane and Navarro [31] gave a time and space efficient data structure for tree navigational operations on BP sequences. A BP sequence represents depths of nodes in the order of depth-first traversal of a tree. Let $P[1..2n]$ be the BP sequence of a tree with n nodes, and let $E[i] = rank_{[(}(P, i) - rank_{[)]}(P, i)$. Then the array E stores node depths in preorder and it is called *excess array*. Most of tree navigational operations such as finding the parent and the next sibling are reduced to a linear scan on E. For example, to find $w = LA(v, d)$, we scan the excess array from the position of v in the BP to the left to find the first value $E[x]$ such that $E[x] = d - 1$. To speed up the scan, a range min–max tree is used. It is a balanced tree and leaves of the tree correspond to equally-partitioned blocks of the BP sequence. A leaf stores the minimum and the maximum of E values in a block. An internal node stores the minimum and the maximum of the values in its child nodes. Then a linear scan on E is reduced to a finger search on the range min–max tree, which is done in time proportional to tree height. Precisely, for the BP sequence P, we assume the word length is $w = \Theta(\log n)$. We partition the BP sequence into huge blocks of length w^c for some constant $c > 0$, and construct a range min–max tree for each huge block. Each internal node of a range min–max tree has $\Theta(w/(c \log w))$ children, and the height of the tree is $O(c)$. By using lookup tables, a finger search is done in $O(c^2)$ time, which is constant.

2.4. Suffix trees and compressed suffix trees

The *Suffix tree* [17,33] is a useful data structure for string matching. Here, a given string S of length s is preprocessed in $O(s)$ 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 can be 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 Fig. 2.

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 [29]. 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 $2(s+t) + o(s)$ bits, $2s + o(s)$ bits, and $|CSA(S)|$ bits, respectively, where $t \leqslant s - 1$ 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 $|CSA(S)| + 4s + 2t + o(s)$ bits [29].

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 in O(1) time. 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 version of DFUDS. However, the data structure is complicated and is difficult to compress. In contrast, we propose the first data structures that support *depth* and *level-ancestor* directly 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 [9,22], 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 $(select_{)}(x-1))+1$ if $x>1$, or 2 if $x=1$.

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.

We have the following property of the excess values.

Lemma 2. *Consider an internal node v of T, which has $k>0$ subtrees $T_1,\ldots,T_k$ as its children. Suppose that $U[l_0..r_0]$ stores the DFUDS for the subtree rooted at v. We also assume that $U[l_i..r_i]$ stores the DFUDS for T_i for $1\leqslant i\leqslant k$ ($l_i=r_{i-1}+1$). Then we have the following.*

$$E[r_i]=E[r_{i-1}]-1=E[r_0]-i \quad (1\leqslant i\leqslant k),$$
$$E[j]>E[r_i] \quad (l_i\leqslant j<r_i).$$

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 $1\leqslant i\leqslant k$. Hence, we have $E[r_i]=E[r_0]-i$. The second property $E[j]>E[r_i]$ ($l_i\leqslant j<r_i$) is obvious because the sequence is balanced if an open parenthesis is added at the beginning. □

The following lemma shows that computing lca is reduced to a *range minimum query*. A range minimum query $RMQ_E(i,j)$ returns the position of the smallest element in $E[i..j]$. If there is a tie, *RMQ* returns the leftmost position. For a balanced parenthesis sequence of length n, *RMQ* can be computed in constant time using an $O(n(\log\log n)^2/\log n)$-bit auxiliary data structure [29].

Lemma 3. *Let x and y be two nodes with $x<y$ and x is not an ancestor of y. Let $x'=select_{)}(x)$ and $y'=select_{)}(y)$ be the ending positions of nodes x and y in DFUDS, respectively. Then the lowest common ancestor $z=lca(x,y)$ can be computed in constant time by*

$$w=RMQ_E\big(x',y'-1\big)+1,$$
$$z=parent(w).$$

Note that it is easy to check if x is an ancestor of y or not in constant time by $findclose(U,enclose(U,x))\leqslant y$, and if it is, $lca(x,y)=x$.

Proof. 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 ($1\leqslant i\leqslant k$). 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 2, $E[r_{\beta-1}]=d+1$ and $E[i]>d+1$ for $l_1\leqslant i<r_{\beta-1}$ and $l_\beta\leqslant i\leqslant r_\beta-2$. 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 $lca(x,y)$. □

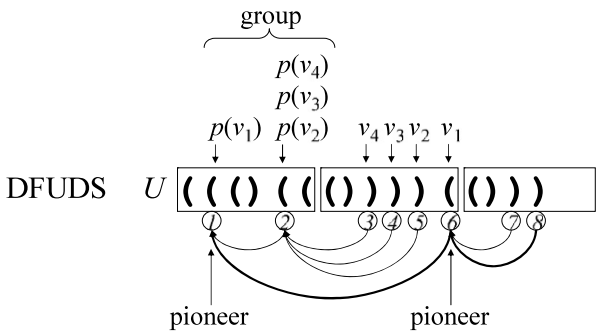

Fig. 3. Pioneers for blocks of size $B = 6$ in the DFUDS sequence for the tree shown in Fig. 1.

3.2. Depth

We represent the depth information using a two-level data structure. First we 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.

We identify a node with its starting position in DFUDS. For a node v, we denote by $f(v)$ the 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 the 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 $(v_i, p(v_i))$ $(1 \leqslant i \leqslant k$, $v_i \in M$, $v_1 > v_2 > \cdots > v_k)$ 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 $p(v_1) \leqslant p(v_2) \leqslant \cdots \leqslant p(v_k)$ 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 between the depth of $p(v_i)$ and that of its pioneer. We call this information I_2.

We show that the number of pioneers is at most $2n/B - 3$ as in [8,14,20]. 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. Fig. 3 shows the pioneers for the example tree in Fig. 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 $\mathit{select}_1(J_s, \mathit{rank}_1(J_s, v))$ 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.

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 as follows. Let b be the bit-pattern of the small block. This information is not enough to compute $f_L(v)$ because the first node in the block is not determined by only b. Precisely, assume that a prefix of b is k [(]'s, followed by a [)]. There are two cases: (1) this represents a node r with k children, (2) this is a part of the representation of a node u with more than k children, whose starting position is in another small block. In case (1), the first node of the small block is r, and we can determine which nodes in b are descendants of r from the bit pattern of b. For such node v, $f_L(v) = r$, and for other nodes in b, $f_L(v) = v$ itself. In case (2), the first node is encoded from the $(k+2)$-th position of b, and we can also determine its descendants in b from the bit pattern of b by ignoring the first $k+1$ bits. We can determine which case occurs in constant time from the last bit of a small block which is in the left of the small block of v. If the position of the first node in the small block is known, we can find $f_L(v)$ in constant time by a table lookup with b.

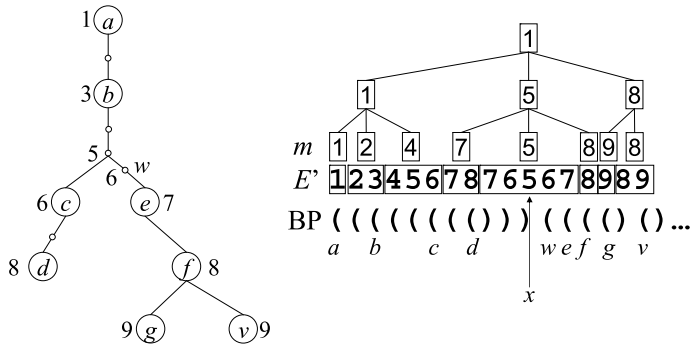

Fig. 4. The BP sequence and the excess array of a tree. In the left figure, big circles represent lower level pioneers, and numbers along with nodes represent their depths. In the right figure, E' is the excess array and m is an array of minimum excess values of small blocks. All the excess values belong to the same large block, and the range min–max tree for the block is built on m.

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. Obviously the query takes O(1) time.

3.3. Level-ancestor

We give auxiliary data structures for computing level-ancestors in O(1) time. We use the range min–max tree for the BP sequence representing the given tree, instead of the DFUDS sequence. Note that this BP sequence is only conceptual and not stored.

In the same way as when computing depths, we partition the DFUDS sequence U of length $2n$ for a tree T into small blocks of length $B_L = \frac{1}{2}\log n$. Consider computing $w = LA(v, d)$. If w belongs to the same small block as v, it is found in constant time by a table lookup, similarly to computing depths. Therefore we consider only the other case. Let z be the lower level pioneer of $p(f_L(v))$ where $f_L(v)$ is the farthest ancestor of v that belongs to the same small block, the same as computing depths. If w is on the path between v and z, it is found in constant time by table lookups. Therefore we consider only the case that w is an ancestor of z. If we find an ancestor of z which is a lower level pioneer and which belongs to the same block as w, the answer is also found in constant time by table lookups. Therefore it is enough to consider to compute level-ancestors only for the lower level pioneers for the small blocks. Let $v_1, v_2, \ldots$ denote the lower level pioneers in preorder.

We consider the excess array E' of the BP sequence $P[1..2n]$ representing the given tree, and partition it into small blocks E'_i. Recall that $E'[x] = rank_{\square}(P, x) - rank_{\square}(P, x)$ and it is equal to the depth of node x. Here small blocks have variable length and defined such that E'_i is a substring of E' which begins just after the position of v_{i-1} and ends at the position of v_i (E_1 begins at position 1). The number of small blocks is equal to the number of lower level pioneers, which is $O(n/\log n)$. Then, we make large blocks for E by merging consecutive small blocks E'_i. Each large block contains the maximum number of small blocks such that the length of the large block does not exceed $B_U = \log^3 n$. If a small block has length more than B_U, we partition it into blocks of length at most B_U. For each small block, we store the minimum excess value in it. The difference between any excess value in E'_i and the first excess value in the large block containing E'_i is in the range $[-B_U, B_U]$. Therefore the difference can be stored in $O(\log B_U) = O(\log\log n)$ bits. The first excess value in each large block is stored explicitly in $O(\log n)$ bits. Thus the total space to store the minimum excess values is $O(n \log\log n/\log n)$ bits.

As shown in [31], to find $w = LA(v, d)$, we search the rightmost index x such that $E'[x] = d-1$ and $x < v$. Then, $w = x+1$. Fig. 4 shows an example. To find $w = LA(v, 6)$, we search for the excess value $E'[x] = 6-1$. Then it holds $w = x+1$.

There are two cases: (1) x is in the same large block as v, (2) x is not in the large block. To determine which case occurs, we first search the same large block as v for a lower level pioneer which is an ancestor of v and whose depth is d or less. This is equivalent to find the rightmost index x such that $E'[x] \leqslant d-1$ and $x < v$. To perform not a linear search but a logarithmic search in a large block, we construct the range min–max tree for each large block. We construct a balanced tree storing the minimum values of small blocks in the leaves. Because we compute only level-ancestors, we need not to store maximum values. Each internal node stores the minimum value among those of its children. Each internal node has $\Theta(\log n/\log\log n)$ children, and therefore the height of the tree is constant. We store not absolute values of E' but relative values from the first value in the large block. Therefore each value is stored in $O(\log\log n)$ bits, and the total space for the entire tree is $O(n \log\log n/\log n)$ bits. The search takes constant time because the height of the tree is constant and for each internal node the objective child is found by a table lookup.

Assume that we found a small block E'_i containing $E'[x] \leqslant d-1$. Note that $E'[x+1] = E'[x]+1$ and $x+1$ is an ancestor of v whose depth is d or less. Then $x+1$ is also an ancestor of v_i and therefore the answer $w = LA(v,d)$ exists on the path between v_i and $x+1$. It is guaranteed that w belongs to the same small block of the BP sequence as $x+1$. However the nodes may belong to different small blocks of the DFUDS sequence. There are two cases: (1) $x+1$ and v_i belong the same small block of the DFUDS sequence, (2) they belong different blocks. For case (1), w is found in constant time by a table lookup. For case (2), w belongs to either the small block of v_i or that of $x+1$ because there is no pioneers between $x+1$ and v_i. Therefore w is also found in constant time.

If no small block containing $E'[x] \leqslant d-1$ exists in the large block, we search other large blocks. To do so, we use the data structure for weighted level-ancestor queries [31]. Any query takes constant time and the space is $O(n \log^2 n / B_U)$ bits if large blocks are of length B_U. Because we set $B_U = \log^3 n$, the space is $O(n/\log n)$. In total, the space for computing level-ancestors is $O(n \log\log n / \log n)$ 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 $select_{]}(v) = 0$, 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:

$$select_{]}\big(rank_{]}\big(findopen(v-1)\big)+1\big) - findopen(v-1).$$

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. Compressing DFUDS into the tree degree entropy

We now consider how to compress the DFUDS U of a tree T with n nodes into the tree degree entropy. 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 [followed by a close parenthesis]. This is called a *unary code*. Hence, it is obvious how to convert between S and U in $O(n)$ time (see Fig. 1). We show how to compress S into $nH^*(T) + o(n)$ bits, and how to retrieve any consecutive $w = \Theta(\log n)$ bits of U from the compressed representation of S in constant time, thereby proving Theorem 1.

A similar approach is used in the original paper for DFUDS [1] to encode cardinal trees. They use prefix codes to encode node degrees for the special case $\sigma = 4$. Below, we propose data structures for ordered trees which achieve the entropy bound for a general alphabet.

Our compression technique can be applied to compressing not only the degree sequence of a tree, but also any sequence of unary codes. We first prove the following general result, and then use it to prove Theorem 1.

Lemma 4. *For a sequence S of n integers encoded by unary codes in a bit string U, let n_i denote the number of occurrences of integer $i \geqslant 0$. If the summation of all the integers is $O(n)$, the sequence is compressed in $\sum_i n_i \log \frac{n}{n_i} + O(n \log\log n / \log n)$ bits so that any $\log n$ bits of the string U can be retrieved in constant time on the word RAM with word length $\Theta(\log n)$.*

4.1. Sequences with small maximum values

We first consider the case where the maximum value in the sequence S is less than $\frac{1}{4}\log n$.

Let $w = \frac{1}{4}\log n$ and let U be the unary representation of S. Because the summation of all the integers is $O(n)$, the length of U is also $O(n)$. We divide the sequence U into blocks of variable lengths so that each block is of length between w and $2w$ and any unary code for $S[i]$ fits in a block. We add dummy bits to the last block if it is shorter than w. It is always possible that all blocks satisfy the condition because the maximum length of a unary code is w. The number of blocks is at most $\frac{n}{w}$.

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 $l_i \leqslant j \leqslant r_i$. For each $U[j]$ such that j is the starting position of a block ($1 \leqslant j \leqslant n/w$) we mark the position $m_j = f(j)$ of S. We can use FID to mark them using $O(n \log\log n / \log n)$ bits.

Let $s_1, s_2, \ldots, s_m$ be the node degrees in a block. Then each s_i appears in S with probability $\frac{n_{s_i}}{n}$ (recall that n_i is the number of nodes with i children). We consider them as a symbol appearing with probability $\prod_{i=1}^{m} \frac{n_{s_i}}{n}$, and construct a sequence S' consisting of those symbols. The length of the sequence is equal to the number of blocks and the alphabet size is at most 2^{2w} because each symbol is originally encoded in $2w$ bits.

We use a Huffman code to encode S'. However the maximum length of a Huffman code may be equal to the alphabet size and it is not possible to decode such a code in constant time. Therefore we add one-bit prefix to each code indicating that the following code is a Huffman code or the symbol is stored without compression. Then maximum code length is $1 + 2w$ bits. Let m_i be the number of occurrences of a symbol i in S', and m be the length of S'. By using the above code, S' can be represented by $m(H(S') + 1 + 1) = m(2 + \sum_i \frac{m_i}{m} \log \frac{m}{m_i})$ bits. Then regarding the original symbols of S, each symbol s_i is considered to be encoded in $\log \frac{n}{n_{s_i}}$ bits, and the total redundancy is $2m$ bits. Therefore the size of the encoding for S is $nH^*(T) + 2m = nH^*(T) + \mathrm{O}(\frac{n}{\log n})$. We have to use a decoding table for the Huffman code, but it uses only $\mathrm{O}(2^{1+2w} \cdot w) = \mathrm{O}(\sqrt{n} \log n)$ bits.

4.2. Sequences with unbounded values

First we divide the alphabet $\mathcal{A}$ of S into two sets; $\mathcal{A}_1$ for values larger than or equal to $w = \frac{1}{4} \log n$, 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 $\sigma = |\mathcal{A}_2| \leqslant \log n$ 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. S_2 is encoded by the Huffman code as described in the previous subsection.

We describe how to compress S_1. We use a bit-vector D_1 of the same length as U, which is $\mathrm{O}(n)$, to indicate if $S[i] \in \mathcal{A}_1$ by setting $D_1[i] = 1$. Because there are at most $\mathrm{O}(n)/w = \mathrm{O}(n/\log n)$ 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.

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 where n' is the length of S_2. Let n_i be the number of occurrences of integer i in S. Then

$$n' H_0(S_2) = \sum_{i \in \mathcal{A}_2} n_i \log \frac{n'}{n_i} \leqslant \sum_{i \in \mathcal{A}} n_i \log \frac{n}{n_i}.$$

Therefore the total space is $\sum_{i \in \mathcal{A}} n_i \log \frac{n}{n_i} + \mathrm{O}(\frac{n \log \log n}{\log n})$ bits. This proves Lemma 4. This also proves Theorem 1 because $\sum_{i \in \mathcal{A}} n_i \log \frac{n}{n_i} = nH^*(T)$.

5. Applications

We now describe some applications of our compressed DFUDS representation of ordered trees and Lemma 4.

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 4. *(See Ferragina et al. [5].) Let $\mathcal{C}$ be a compressor that compresses any string w into $|w|H_0(w) + \mu|w|$ bits. The string* **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 compressing the other string: S_{last}. It encodes the degrees of the nodes of T by unary codes after the stable sort. Therefore by using Lemma 4, we can compress it into the tree degree entropy $H^*(T)$. We obtain the following theorem:

Theorem 5. *The string* **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.*

Proof. For the compressor $\mathcal{C}$ of Theorem 4 we use the one in Theorem 3 for $k = 0$. Then $\mu = \log\sigma \log\log n / \log n$ and S_α is compressed into $nH_k(T) + o(n\log\sigma) + g_k$ bits. The string S_{last} is compressed into $nH^*(T) + o(n)$ bits by Lemma 4. In both compression algorithms, any consecutive $O(\log n)$ bits can be decoded in constant time. □

5.2. Ultra-succinct compressed suffix trees

The size of the compressed suffix tree of a string S of length s is $|CSA(S)| + 6s + o(s)$ bits in the worst case [29]. For the size $|CSA(S)|$ of the compressed suffix array, one implementation achieves the asymptotically optimal size $sH_k(S) + o(s)$ 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 [29], the tree topology is encoded by the BP in $2(s+t)$ bits where t is the number of internal nodes of the tree. 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 degree 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:

- *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).

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

Definition 2. (See Sadakane [29].) 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 every node in a suffix tree has at least two children. Therefore each internal node has at least one inorder rank. A node with d children has $d - 1$ different inorder ranks. Note that the preordering includes the leaves whereas the inordering does not. To compute inorders we use this lemma:

Lemma 5. *(See Sadakane [29].) There is a one-to-one correspondance between the leaves and the inorder ranks for the internal nodes, and an inorder rank of v is equal to the number of leaves that have smaller preorder ranks than v.*

Proof. Inorder ranks are assigned during a depth-first traversal, which is divided into upgoing and downgoing paths. An inorder rank 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. □

To support the above functions, we need the following in addition to *lca* [29], which can be also computed on the DFUDS:

- *leaf_rank*(v): Return the number of leaves before or equal to the node v in the preordering of the tree.
- *leaf_select*(i): Return the ith leaf in the preordering of the tree.
- *preorder_rank*(v): Return the preorder rank of the node v.
- *preorder_select*(i): Return the node whose preorder rank is i.
- *inorder_rank*(v): Return the smallest inorder rank of the internal node v.
- *inorder_select*(i): Return the node whose inorder rank is i.
- *leftmost_leaf*(v), *rightmost_leaf*(v): Return the leftmost (rightmost) leaf in the subtree rooted at v.

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

- $leaf\_rank(v) = rank_{\square\square}(v)$
  [Each leaf is represented by an occurrence of the pattern $\square\square$, 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.]
- $leaf\_select(i) = select_{\square\square}(i)$
  [Find the i-th leaf according to the above.]
- $preorder\_rank(v) = (rank_{\square}(v-1)) + 1$
  [The description of each node consists of a consecutive sequence of $\square$ symbols followed by a single $\square$, and the nodes appear in the sequence according to their preorder. Therefore $rank_{\square}(v-1)$ gives the preorder of the receding node of v. By adding 1 we obtain the answer.]
- $preorder\_select(i) = (select_{\square}(i-1)) + 1$
  [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 *preorder_rank*.]

- $inorder\_rank(v) = leaf\_rank(child(v, 2) - 1)$
  [According to Lemma 5, count the number of leaves to the left of the second child of v.]
- $inorder\_select(i) = parent((leaf\_select(i)) + 2)$
  [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 taking its parent, we obtain the answer.]
- $leftmost\_leaf(v) = leaf\_select(leaf\_rank(v - 1) + 1)$
  [We obtain the number of leaves appearing before v in preorder by $leaf\_rank(v-1)$. By finding the next leaf in preorder, we obtain the answer.]
- $rightmost\_leaf(v) = findclose(enclose(v))$
  [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 $findclose(v)$. 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 degree entropy. Furthermore, if the alphabet is binary, the tree topology is encoded in $2s + o(s)$ bits:

Theorem 6. *The tree topology of the suffix tree T with s leaves and t internal nodes can be encoded in $s \log \frac{s+t}{s} + t \log \frac{s+t}{t} + 2t + o(s)$ bits. Any operation on the compressed suffix tree is done in the same complexity as the one using the BP representation. In particular, if $\sigma = 2$, the tree topology can be encoded in $2s + o(s)$ 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.

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

The condition on the shape of the tree is only the number of leaves. Therefore the information-theoretic lower bound conditioned on the degree multiplicities is applied. Because the tree shape is encoded into the entropy bound by Theorem 1, the encoding size matches the lower bound. Therefore to give an upper bound on the encoding size, it is enough to give some code which encodes the tree in $s \log \frac{s+t}{s} + t \log \frac{s+t}{t} + 2t + o(s)$ bits.

We consider a code which first encodes whether a node is a leaf or not in preorder, then encodes the shape of the tree whose leaves are removed. The first part uses $s \log \frac{s+t}{s} + t \log \frac{s+t}{t} + o(s)$ bits, and the second part uses $2t$ bits by using the BP. This proves the claim.

If $\sigma = 2$, the suffix tree has s leaves and $s-1$ internal nodes, each of which has exactly two children. Therefore the size of the compressed DFUDS is $(2s-1)H^*(T) + o(s) = s \log \frac{2s-1}{s} + (s-1) \log \frac{2s-1}{s-1} + o(s) = 2s + o(s)$ bits. □

In the original compressed suffix tree [29], string depths of internal nodes are sorted by inorder ranks and conceptually stored in an array $Hgt[1..s]$, which is actually represented by a sequence of s increasing numbers $0 \leqslant x_1 \leqslant x_2 \leqslant \cdots \leqslant x_s \leqslant s$, 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 $2s$. 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 $\sum_i n_i = s$ and $\sum_i (i+1)n_i \leqslant 2s$. Therefore we can compress the sequence by using Lemma 4. 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 expect good compression in practice.

We state how to implement $string\_depth(v)$ and $sl(v)$ for completeness. For details and the correctness, please refer to [29].

- $string\_depth(v) = Hgt[inorder\_rank(v)]$
- $sl(v) = lca(leaf\_rank(\Psi[x]), leaf\_rank(\Psi[y]))$ where Ψ is a function of the compressed suffix array [11,28] that can be computed in constant time, and $x = leaf\_rank(leftmost\_leaf(v))$ and $y = leaf\_rank(rightmost\_leaf(v))$.

6. Concluding remarks

In this paper, we have given a natural definition of the entropy of tree topology called the tree degree entropy and proposed a succinct data structure for storing a tree whose size matches this entropy. We have also given the first auxiliary data structure for *lca*, and auxiliary data structures for depths and level-ancestors which use less space than existing ones [9].

We also showed applications to reduce the size of the compressed suffix trees [29] and labeled trees [5] further.

After the preliminary version of this paper [15], Farzan and Munro [3] gave another representation achieving the tree degree entropy based on the tree cover, but their auxiliary data structures are larger than ours. Very recently, Farzan et al. [4] gave a universal representation of trees which implicitly stores both BP and DFUDS, but this representation is not compressed into the tree degree entropy.

Acknowledgments

The authors would like to thank Johannes Fischer, who pointed out an error in an earlier draft, and the anonymous referees for their helpful comments.

References

[1] D. Benoit, E.D. Demaine, J.I. Munro, R. Raman, V. Raman, S.S. Rao, Representing trees of higher degree, Algorithmica 43 (4) (2005) 275–292.
[2] Y.-T. Chiang, C.-C. Lin, H.-I. Lu, Orderly spanning trees with applications, SIAM J. Comput. 34 (4) (2005) 924–945.
[3] A. Farzan, J.I. Munro, A uniform approach towards succinct representation of trees, in: Proc. SWAT, in: Lecture Notes in Comput. Sci., vol. 5124, 2008, pp. 173–184.
[4] A. Farzan, R. Raman, S.S. Rao, Universal succinct representations of trees?, in: Proc. ICALP, in: Lecture Notes in Comput. Sci., vol. 5555, 2009, pp. 451–462.
[5] P. Ferragina, F. Luccio, G. Manzini, S. Muthukrishnan, Compressing and indexing labeled trees, with applications, J. ACM 57 (1) (2009) 4, 1–4:33.
[6] P. Ferragina, G. Manzini, Indexing compressed texts, J. ACM 52 (4) (2005) 552–581.
[7] P. Ferragina, R. Venturini, A simple storage scheme for strings achieving entropy bounds, Theoret. Comput. Sci. 372 (1) (2007) 115–121.
[8] R.F. Geary, N. Rahman, R. Raman, V. Raman, A simple optimal representation for balanced parentheses, Theoret. Comput. Sci. 368 (December 2006) 231–246.
[9] R.F. Geary, R. Raman, V. Raman, Succinct ordinal trees with level-ancestor queries, ACM Trans. Algorithms 2 (October 2006) 510–534.
[10] R. Grossi, A. Gupta, J. S. Vitter, High-order entropy-compressed text indexes, in: Proc. ACM–SIAM SODA, 2003, pp. 841–850.
[11] R. Grossi, J.S. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, SIAM J. Comput. 35 (2) (2005) 378–407.
[12] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, 1997.
[13] W.K. Hon, K. Sadakane, W.K. Sung, Succinct data structures for searchable partial sums, in: Proc. of ISAAC, in: Lecture Notes in Comput. Sci., vol. 2906, 2003, pp. 505–516.
[14] G. Jacobson, Space-efficient static trees and graphs, in: Proc. IEEE FOCS, 1989, pp. 549–554.
[15] J. Jansson, K. Sadakane, W.-K. Sung, Ultra-succinct representation of ordered trees, in: Proc. ACM–SIAM SODA, 2007, pp. 575–584.
[16] H.-I. Lu, C.-C. Yeh, Balanced parentheses strike back, ACM Trans. Algor. (TALG) 4 (3) (2008), Article No. 28.
[17] E.M. McCreight, A space-economical suffix tree construction algorithm, J. ACM 23 (12) (1976) 262–272.
[18] D.R. Morrison, Patricia – practical algorithm to retrieve information coded in alphanumeric, J. ACM 15 (4) (1968) 514–534.
[19] J.I. Munro, Tables, in: Proceedings of the 16th Conference on Foundations of Software Technology and Computer Science, FSTTCS '96, in: Lecture Notes in Comput. Sci., vol. 1180, 1996, pp. 37–42.
[20] J.I. Munro, V. Raman, Succinct representation of balanced parentheses and static trees, SIAM J. Comput. 31 (3) (2001) 762–776.
[21] J.I. Munro, V. Raman, S.S. Rao, Space efficient suffix trees, J. Algorithms 39 (2) (May 2001) 205–222.
[22] J.I. Munro, S.S. Rao, Succinct representations of functions, in: Proceedings of ICALP, in: Lecture Notes in Comput. Sci., vol. 3142, 2004, pp. 1006–1015.
[23] R. Pagh, Low redundancy in static dictionaries with constant query time, SIAM J. Comput. 31 (2) (2001) 353–363.
[24] C.K. Poon, W.K. Yiu, Opportunistic data structures for range queries, in: Proceedings of COCOON, in: Lecture Notes in Comput. Sci., vol. 3595, 2005, pp. 560–569.
[25] R. Raman, V. Raman, S. S. Rao, Succinct indexable dictionaries with applications to encoding k-ary trees and multisets, in: Proc. ACM–SIAM SODA, 2002, pp. 233–242.
[26] S. Srinivasa Rao, Time-space trade-offs for compressed suffix arrays, Inform. Process. Lett. 82 (6) (2002) 307–311.
[27] G. Rote, Binary trees having a given number of nodes with 0, 1, and 2 children, Sem. Lothar. Combin. 38 (1996), B38b, 6 pp. http://www.emis.de/journals/SLC/wpapers/s38proding.html,.
[28] K. Sadakane, New text indexing functionalities of the compressed suffix arrays, J. Algorithms 48 (2) (2003) 294–313.
[29] K. Sadakane, Compressed suffix trees with full functionality, Theory Comput. Syst. 41 (4) (2007) 589–607.
[30] K. Sadakane, R. Grossi, Squeezing succinct data structures into entropy bounds, in: Proc. ACM–SIAM SODA, 2006, pp. 1230–1239.
[31] K. Sadakane, G. Navarro, Fully-functional succinct trees, in: Proc. ACM–SIAM SODA, January 2010, pp. 134–149.
[32] N. Välimäki, V. Mäkinen, W. Gerlach, K. Dixit, Engineering a compressed suffix tree implementation, J. Exp. Algorithmics (JEA) 14:2:4.2–2:4.23 (January 2010).
[33] P. Weiner, Linear pattern matching algorithms, in: Proceedings of the 14th IEEE Symposium on Switching and Automata Theory, 1973, pp. 1–11.