Fast Generation of Unlabelled Free Trees using Weight Sequences

Paul Brown   Trevor Fenner

Department of Computer Science and Information Systems,
Birkbeck, University of London, London WC1E 7HX, U.K

Abstract. In this paper, we introduce a new representation for *ordered* trees, the *weight sequence* representation. We then use this to construct new representations for both *rooted* trees and *free* trees, namely the *canonical weight sequence* representation. We construct algorithms for generating the weight sequence representations for all rooted and free trees of order n, and then add a number of modifications to improve the efficiency of the algorithms. Python implementations of the algorithms incorporate further improvements by using generators to avoid having to store the long lists of trees returned by the recursive calls, as well as caching the lists for rooted trees of small order, thereby eliminating many of the recursive calls. We further show how the algorithm can be modified to generate adjacency list and adjacency matrix representations for free trees. We compared the runtimes of our Python implementation for generating free trees with the Python implementation of the well-known WROM algorithm taken from NetworkX. The implementation of our algorithm is over four times as fast as the implementation of the WROM algorithm. The runtimes for generating adjacency lists and matrices are somewhat longer than those for weight sequences, but are still over four times as fast as the corresponding implementations of the WROM algorithm.

1   Introduction

The enumeration of *trees*, whether *ordered*, *rooted* or *free*, has been well-studied. (Rooted trees are also called *oriented* trees, see [9]). Indeed, "Cayley's formula", which states that there are precisely n^{n-2} free trees on n *labelled* vertices, dates back to Carl Wilhelm Borchard in the middle of the nineteenth century [6]. In 1948, Otter [12] derived asymptotic estimates for the numbers of both *unlabelled* free and rooted trees. In addition, generating functions for the numbers of both unlabelled free and rooted trees have been obtained (see [9]). The exact counts for unlabelled free trees with n vertices, for $n \leq 36$, are listed as Sequence $A000055$ in the OEIS [2].

One of the first efficient algorithms for *generating* unlabelled rooted trees was developed by Beyer and Hedetniemi [4] using a *level sequence representation*. This algorithm was extended by Wright, Richmond, Odlyzko and McKay [14] to generate all unlabelled free trees. This algorithm is referred to informally as the WROM algorithm, and is by far the most commonly used algorithm to generate non-isomorphic free trees. An alternative algorithm was constructed by Li and Ruskey [11] using the *parent sequence representation*. Indeed, a good survey of this topic can be found in Li's thesis [10]; see also [8]. Other work in this area has recently been conducted by Sawada [13], who presented algorithms to generate both rooted and free *plane* (i.e., ordered) trees.

In this paper, we construct new recursive algorithms for generating rooted trees and free trees, taking a different approach from previous authors. We introduce a new representation, the *canonical weight sequence*, and use this rather than level or parent sequences. Unlike the latter representations, the weight sequence representation preserves *referential transparency* for subtrees. This makes possible major improvements in efficiency by enabling us to efficiently cache the weight sequences of rooted trees of small order, thereby eliminating many of the recursive calls.

We introduce a number of modifications to improve the efficiency of the basic algorithms. We implemented the algorithms in Python, incorporating further improvements by using generators to avoid having to store the long lists of trees returned by the recursive calls. We also show how the algorithm can be amended to generate adjacency list and adjacency matrix representations for free trees.

We compared the runtimes of our Python implementation for generating free trees with the Python implementation of the well-known WROM algorithm taken from NetworkX [1], the popular Python graph and network libary. The Python implementation of our new algorithm is over four times as fast as the corresponding implementation of the WROM algorithm.

The programs were all written in Python 3.7 and executed using the PyPy3 compiler, although the pseudo-code we present can easily be translated into other languages. The Python code is available as additional material to this paper. Any graph-theoretic terminology and notation not explicitly defined can be found in Bondy and Murty's text [5].

In Section 2, we introduce the weight sequence representations for ordered trees, weighted trees and free trees. In Sections 3 and 4, we present our algorithms for generating rooted and free trees, respectively. Then, in Section 5, we discuss improvements to the algorithms and their implementations, as well as the modifications required to generate the adjacency list and matrix representations of the trees. In Section 6, we compare the runtimes of the Python implementations of our algorithm with those of the WROM algorithm, and Section 7 contains our concluding remarks.

2   Preliminaries

2.1   Notation

A *free tree* T is an connected undirected graph that contains no cycles (conventionally, just called a *tree* in the graph theory literature). The *degree* of a vertex v of T is the number of vertices adjacent to v. A *leaf* of T is a vertex of degree 1; all other vertices of T are called *branch* vertices. It is easy to show that there is a unique path between any pair of vertices of T.

A *rooted tree* R is a free tree with a distinguished vertex called its *root*. Let v be a vertex of R. Any other vertex u on the path from the root to v is an *ancestor* of v, and v is a *descendant* of u. A descendant w of v that is adjacent to v is a *child* of v, and v is the *parent* of w. Any other child of v is a *sibling* of w. By definition, the root has no parent.

Let v be any descendant of the root of R. The *subtree* of R that consists of v together with all of its descendants is a rooted tree with root v. We denote this subtree by $R(v)$ and define $wt(v)$, the *weight* of v, to be the order of $R(v)$; so the weight of the root of R is the order of R. If v is a leaf then $wt(v) = 1$ and $R(v)$ contains just the vertex v. $R - R(v)$ is the rooted tree, with the same root as R, obtained from R by deleting the subtree $R(v)$ together with the edge between v and its parent.

An *ordered tree*, sometimes called a *plane tree* [13], is a rooted tree in which there is an ordering defined on the children of each vertex. By convention, when drawing a rooted tree, the root is placed at the top of the diagram and, for an ordered tree, the order of the children is from left to right. So we may refer to the *first* (left-most) or *last* (right-most) child of its parent. Similarly, for any vertex v that is not the last child of its parent, we may refer to the *next* sibling of v. We note that, if R is an ordered tree, the subtrees $R(v)$ and $R - R(v)$ are considered to be ordered trees, inheriting the ordering of the sets of children from R. For convenience, when w is a child of v, instead of saying that $R(w)$ is subtree of $R(v)$, we often say that $R(w)$ is a subtree of v.

A tree is called a *labelled tree* if each vertex is assigned a unique label. For any unlabelled ordered tree R with n vertices, we conventionally label the vertices as $v_1, v_2, \ldots, v_n$ in *preorder*, where v_1 is the root of the tree. *Preorder* is the total ordering of the vertices of R defined recursively as follows: for any vertex u with children $u_1, u_2, \ldots, u_p$, *preorder* for the subtree $R(u)$ starts with u, followed by the vertices of $R(u_1)$ in preorder (if $p \geq 1$), then the vertices of $R(u_2)$ in preorder (if $p \geq 2$), etc. We note that v_2 is the first child of the root v_1. It trivially follows that, for any vertex v_k of R, the preorder of the vertices of $R(v_k)$ is a contiguous subsequence of the preorder of the vertices of R.

Two labelled free trees are *isomorphic* if there is a bijection between their vertex sets that preserves adjacency and non-adjacency; two labelled rooted trees are isomorphic if there exists an isomorphism between their underlying free trees that *maps the root of one onto root of the other*; two labelled ordered trees are isomorphic if there exists an isomorphism between their underlying rooted trees *that preserves the orderings of the children of each vertex*. We say that two trees (whether ordered, rooted or free) are *f-isomorphic* if their underlying free trees are isomorphic, and that two trees (whether ordered or rooted) are *r-isomorphic* if their underlying rooted trees are isomorphic. For completeness, we will also say that two isomorphic ordered trees are *o-isomorphic*.

An *integer sequence* $\mathbf{s}$ is an (ordered) list of integers $s_1 s_2 \ldots s_n$. In this paper, we shall assume that every element s_i in $\mathbf{s}$ is positive, and denote the length of $\mathbf{s}$ by $|\mathbf{s}|$; so in this case $|\mathbf{s}| = n$. If $\mathbf{t} = t_1 t_2 \ldots t_m$ is another integer sequence, we denote the *concatenation* of the two sequences by $\mathbf{s} \oplus \mathbf{t}$, i.e., $\mathbf{s} \oplus \mathbf{t} = s_1 s_2 \ldots s_n t_1 t_2 \ldots t_m$. For simplicity, we do not distinguish between a sequence of length one and single integer, e.g., we may write $s_1 \oplus \mathbf{t}$.

We say that $\mathbf{s}$ is *lexicographically greater than or equal to* $\mathbf{t}$, denoted $\mathbf{s} \geq \mathbf{t}$, if and only if either (a) or (b) below hold:

(a) $s_i = t_i$, for $1 \leq i < j$, and $s_j > t_j$, for some j, $1 \leq j \leq \min(n, m)$;

(b) $s_i = t_i$ for $1 \leq i \leq \min(n, m)$ and $n \geq m$.

Strict lexicographical inequality $\mathbf{s} > \mathbf{t}$ holds if $\mathbf{s} \geq \mathbf{t}$ and $\mathbf{s} \neq \mathbf{t}$. We note that this defines a total ordering on the set of integer sequences.

2.2   Weight sequences of ordered trees

A common way to represent an ordered tree is by a suitable integer sequence obtained by traversing the tree in some specified order and recording some particular property of each vertex as it is visited. The resulting sequence is called a *representation sequence* for the tree. A *valid* representation for ordered trees is a representation by integer sequences such that any two ordered trees that have the same representation sequence are *o*-isomorphic.
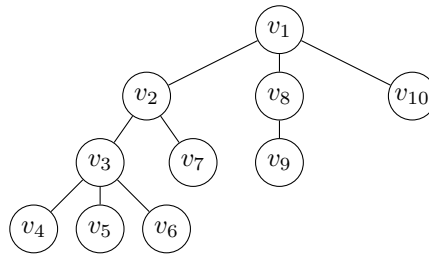

Figure 1: An ordered tree rooted at v_1 labelled in preorder.

For example, consider the ordered tree of order 10 shown in Figure 1, in which the vertices are labelled in preorder. If we record the *level* (where we define the level of the root to be 1, the level of its children to be 2, etc.) of each vertex in a preorder traversal, we obtain the following sequence: $1\,2\,3\,4\,4\,4\,3\,2\,3\,2$. This is called the *level sequence* of the tree. Similarly, if we record the index of the label of the parent of each vertex in a preorder traversal, we obtain its *parent sequence*: $1\,2\,3\,3\,3\,2\,1\,8\,1$ (note there is no parent for the root in the parent sequence representation). Both of these sequence representations are well-known and have been shown to be valid representations for ordered trees (see [4] [7]). They have been used in the design of algorithms for generating rooted trees and free trees by Beyer and Hedetniemi [4], Wright et al. [14], Li and Ruskey [11], Sawada [13] and Cook [7].

In this paper, we introduce a new representation sequence. This is constructed by recording the *weight* of each vertex in a preorder traversal of the tree. We call this representation the *weight sequence* of the tree, and denote the weight sequence of any ordered tree R by $\mathbf{ws}(R)$. For example, for the tree R in Figure 1, $\mathbf{ws}(R) = 10\ 6\,4\,1\,1\,1\,1\,2\,1\,1$.

Lemma 1 Let R be an ordered tree of order n with weight sequence $\mathbf{ws}(R) = s_1\, s_2 \ldots s_n$, where the vertices are labelled v_1, $v_2, \ldots, v_n$ in preorder. Then

(a) $s_1 = n$;

(b) for all vertices v_k of R,

      (i) $s_k = wt(v_k)$, i.e., the order of $R(v_k)$;

      (ii) $\mathbf{ws}(R) = \mathbf{x} \oplus \mathbf{ws}(R(v_k)) \oplus \mathbf{y}$ for some integer sequences $\mathbf{x}$ and $\mathbf{y}$;

      (iii) $\mathbf{ws}(R(v_k)) = s_k\, s_{k+1} \ldots s_{k+s_k-1}$;

(c) $\mathbf{ws}(R - R(v_2)) = t\, s_{s_2+2}\, s_{s_2+3} \ldots s_n$ where $t = n - s_2$.

Proof (a), (b)(i) and (b)(ii) follow immediately since the vertices of R are labelled in preorder.

(b)(iii) The weight of any vertex in $R(v_k)$ is the same as in R. Since $R(v_k)$ is of order s_k, it then follows that $\mathbf{ws}(R(v_k)) = s_k\, s_{k+1} \ldots s_{k+s_k-1}$.

(c) Since $R(v_2)$ is of order s_2, the result follows easily from (b)(iii). $\qquad\square$

Corollary 1 Let R be an ordered tree of order n. Suppose that u_1, $u_2, \ldots, u_p$ are the children of the root of R, where u_{i+1} is the next sibling of u_i for all i, $1 \leq i \leq p-1$. Then

$$\mathbf{ws}(R) = n \oplus \mathbf{ws}(R(u_1)) \oplus \mathbf{ws}(R(u_2)) \oplus \ldots \oplus \mathbf{ws}(R(u_p)). \qquad (1)$$

It therefore follows from Lemma 1(b) that, for any ordered tree R, the subsequence of $\mathbf{ws}(R)$ that corresponds to $R(v_k)$ is just $\mathbf{ws}(R(v_k))$, where $R(v_k)$ is considered as an ordered tree in its own right. This is the main reason why the weight sequence is a particularly useful representation for the generation of trees of order n: we can construct the weight sequence of any ordered tree of order n directly from the weight sequences of its subtrees. So, if r is the order of $R(u_1)$, it follows from Lemma 1 and Corollary 1 that one way to accomplish this is to take the weight sequence of an ordered tree of order r (corresponding to $\mathbf{ws}(R(u_1))$), and combine it appropriately with the weight sequence of an ordered tree of order $n - r$ (corresponding to $\mathbf{ws}(R - R(u_1))$). We shall elaborate on this in Sections 3 and 4.

We note that, since the weight sequence of a tree is well defined, any o-isomorphic trees must have the same weight sequence.

Lemma 2 The weight sequence is a valid representation for ordered trees.

Proof By inspection, the result holds when the order is less than four. So suppose that the result holds for all ordered trees of order less than n, where $n \geq 4$. Let R and R' be labelled ordered trees of order n such that $\mathbf{ws}(R) = \mathbf{ws}(R')$, where the vertices of the trees are labelled v_1, $v_2, \ldots, v_n$ and v'_1, $v'_2, \ldots, v'_n$, respectively, in preorder.

Consider the subtrees $R(v_2)$ and $R - R(v_2)$ of R, and $R'(v'_2)$ and $R' - R'(v'_2)$ of R'. By parts (b)(iii) and (c) of Lemma 1, $\mathbf{ws}(R(v_2)) = \mathbf{ws}(R'(v'_2))$ and $\mathbf{ws}(R - R(v_2)) = \mathbf{ws}(R' - R'(v'_2))$. Since these trees are of order less than n, it follows from the inductive hypothesis that $R(v_2)$ is o-isomorphic to $R'(v'_2)$, and $R - R(v_2)$ is o-isomorphic to $R' - R(v'_2)$. So, since v_2 and v'_2 are the first children of the roots of R and R', respectively, it follows that R is o-isomorphic to R'. Hence the weight sequence is a valid representation for ordered trees. $\qquad\square$

The following lemma will be used in Section 2.3.

Lemma 3 Let $\mathbf{s}$ and $\mathbf{t}$ be weight sequences of trees. If $\mathbf{s} > \mathbf{t}$ then $\mathbf{x} \oplus \mathbf{s} \oplus \mathbf{y} > \mathbf{x} \oplus \mathbf{t} \oplus \mathbf{z}$, for any integer sequences $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$.

Proof This follows immediately from Lemma 1(a) and the definition of lexicographical order. $\qquad\square$

2.3   Canonical weight sequences of rooted trees

We extend the definition of a valid representation by integer sequences to rooted trees: a *valid representation for rooted trees* is a well-defined representation such that any two rooted trees that have the same representation sequence are r-isomorphic.

Now, since the weight sequence is a valid representation for ordered trees by Lemma 2, two r-isomorphic ordered trees that are not o-isomorphic must have different weight sequences. For example, the two r-isomorphic ordered trees in Figure 2 have weight sequences 10 1 6 1 4 1 1 1 2 1 and 10 2 1 1 6 1 4 1 1 1, respectively (they are also r-isomorphic but not o-isomorphic to the ordered tree in Figure 1). So, in order to define a valid representation for rooted trees using weight sequences, we need to choose a unique representative from each r-isomorphism class of ordered trees.

An ordered tree R of order n is *canonically ordered* if $\mathbf{ws}(R(u)) \geq \mathbf{ws}(R(v))$, for each vertex u of R having a next sibling v. Clearly, if R is canonically ordered then so is $R(v)$, for each vertex v of R. It is easy to see that the ordered tree in Figure 1 is canonically ordered, but those in Figure 2 are not.
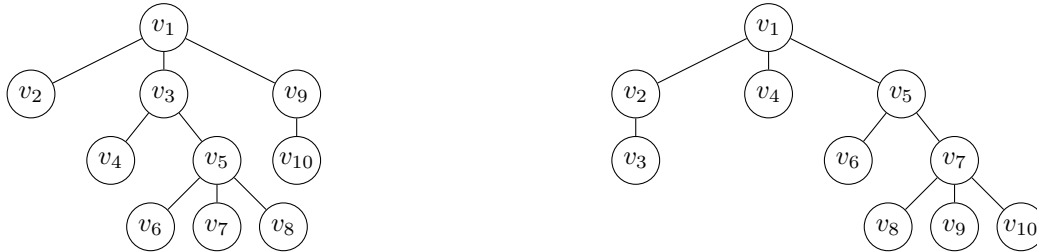

Figure 2: Two trees that are r-isomorphic but not o-isomorphic.

Lemma 4 Let R and R' be r-isomorphic canonically ordered trees. Then $\mathbf{ws}(R) = \mathbf{ws}(R')$, and R and R' are o-isomorphic.

Proof Let n be the order of R and R'. It is easy to see, by inspection, that the result holds when $n \leq 3$. So suppose that $n \geq 4$ and that the result holds for all pairs of trees of order less than n. Let $u_1, u_2, \ldots, u_p$ be the children of the root of R, where u_{i+1} is the next sibling of u_i for each u_i. Let θ be an r-isomorphism from R to R'. Clearly, θ maps the children of the root of R to the children of the root of R'. So the subtrees of the root of R' are precisely the subtrees $\theta(R(u_i))$ in some order.

For each u_i, both $R(u_i)$ and $\theta(R(u_i))$ are canonically ordered. Therefore, since $R(u_i)$ is r-isomorphic to $\theta(R(u_i))$, it follows that $R(u_i)$ is o-isomorphic to $\theta(R(u_i))$ by the inductive hypothesis. Hence $\mathbf{ws}(R(u_i)) = \mathbf{ws}(\theta(R(u_i)))$ for each u_i. Since R and R' are both canonically ordered, it is now easy to see from (1) that $\mathbf{ws}(R) = \mathbf{ws}(R')$. Hence R and R' are o-isomorphic by Lemma 2. $\square$

Clearly, for any ordered tree R, by suitably permuting the subtrees of each vertex, we can obtain a canonically ordered tree that is r-isomorphic to R. We therefore define $\mathbf{cws}(R)$, the *canonical weight sequence* of R, to be the weight sequence of any canonically ordered tree that is r-isomorphic to R. By Lemma 4, $\mathbf{cws}(R)$ is well defined.

Lemma 5 The canonical weight sequence is a valid representation for rooted trees.

Proof Let R_1 and R_2 be rooted trees such that $\mathbf{cws}(R_1) = \mathbf{cws}(R_2)$. Let $\tilde{R}_1$ and $\tilde{R}_2$ be canonically ordered trees that are r-isomorphic to R_1 and R_2, respectively. Then

$$\mathbf{ws}(\tilde{R}_1) = \mathbf{cws}(R_1) = \mathbf{cws}(R_2) = \mathbf{ws}(\tilde{R}_2).$$

So $\tilde{R}_1$ and $\tilde{R}_2$ are o-isomorphic by Lemma 2, and thus r-isomorphic. Therefore R_1 and R_2 are r-isomorphic. $\qquad\square$

It immediately follows from this result that, subject to labelling, we may represent any rooted tree by a unique canonically ordered tree.

It is straightforward to show that the ordered tree R_{max} that has the lexicographically largest weight sequence of all ordered trees r-isomorphic to R is canonically ordered, and that $\mathbf{cws}(R) = \mathbf{ws}(R_{max})$.

2.4   Free trees

We extend the definition of a valid representation by integer sequences to free trees: a *valid* representation for free trees is a well-defined representation such that any two free trees that have the same representation sequence are f-isomorphic.

Now, since the canonical weight sequence is a valid representation for rooted trees by Lemma 5, two f-isomorphic rooted trees that are not r-isomorphic must have different canonical weight sequences. So, in order to define a valid representation for free trees using weight sequences, we need to choose a unique representative from each f-isomorphism class of rooted trees.

Let T be a free tree of order n. Most algorithms for generating free trees of a given order choose the root of T to be a *central* vertex (T contains either a single central vertex or two adjacent central vertices). Instead, in keeping with our choice of the use of the weight sequence rather than the level or parent sequence, we choose the root of T to be the *centroid* when T is *unicentroidal*; when T is *bicentroidal*, we represent T as an ordered pair of subtrees rooted at the two centroidal vertices.

A *centroidal* vertex u of T is a vertex such that each component of the forest $T - u$ is of order at most $\frac{n}{2}$. It is well known that a tree is either unicentroidal, having a single centroidal vertex (in which case the largest component of $T - u$ is of order at most $\frac{n-1}{2}$), or bicentroidal, having two adjacent centroidal vertices (in which case the largest component of $T - u$ is of order $\frac{n}{2}$); see [5]. Moreover, it is easy to show that the centroids of two f-isomorphic free trees must map to each other under any f-isomorphism. We therefore consider the two types of free tree separately.

Suppose first that T is unicentroidal. We now define the *free weight sequence* $\mathbf{fws}(T)$ of T to be the canonical weight sequence of any tree R that is rooted at its centroid and is f-isomorphic to T; so $\mathbf{fws}(T) = \mathbf{cws}(R)$. We note that, since the centroid consists of a single vertex and the canonical weight sequence is well defined, the free weight sequence is well defined for all unicentroidal trees. It immediately follows from Lemma 5 that, subject to labelling, we may represent any unicentroidal tree by a unique canonically ordered tree *rooted at its centroid*.

For example, suppose that the tree in Figure 1 is a free tree T (so not rooted). It is easy to see that v_2 is the unique centroidal vertex of T, and therefore T is f-isomorphic to the canonically ordered tree in Figure 3, which is rooted at its centroid u. Therefore $\mathbf{fws}(T) = 10\ 4\,2\,1\,1\,4\,1\,1\,1\,1$.
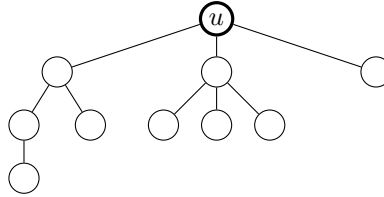
Figure 3: Unicentroidal free tree representation of the tree in Figure 1

Lemma 6 Let T and T' be two unicentroidal free trees. If $\mathbf{fws}(T) = \mathbf{fws}(T')$ then T is f-isomorphic to T'.

Proof Let R and R' be two rooted trees, rooted at their centroids, that are f-isomorphic to T and T', respectively. Suppose that $\mathbf{fws}(T) = \mathbf{fws}(T')$. Then $\mathbf{cws}(R) = \mathbf{cws}(R')$, so R is r-isomorphic to R' by Lemma 5. Hence T is f-isomorphic to T'. □

We now consider the case when T is bicentroidal with centroidal vertices u and v. If we delete the edge between u and v, we obtain disjoint trees T_u and T_v of order $\frac{n}{2}$, which we may consider to be rooted at u and v, respectively. We may therefore represent T as the ordered pair $<T_u, T_v>$ when $\mathbf{cws}(T_u) \geq \mathbf{cws}(T_v)$, or $<T_v, T_u>$ when $\mathbf{cws}(T_v) \geq \mathbf{cws}(T_u)$. We define $\mathbf{fws}(T)$, the *free weight sequence* of T, to be $\mathbf{cws}(T_u) \oplus \mathbf{cws}(T_v)$ in the former case, and $\mathbf{cws}(T_v) \oplus \mathbf{cws}(T_u)$ in the latter case. We note that the first and $\frac{n+2}{2}$th elements of $\mathbf{fws}(T)$ correspond to u and v, and are both equal to $\frac{n}{2}$.

Since the canonical weight sequence is well defined for rooted trees, it follows that the free weight sequence is well defined for bicentroidal trees. It immediately follows from Lemma 5, that, subject to labelling, we may represent any bicentroidal tree of order n by a unique ordered pair of canonically ordered trees of order $\frac{n}{2}$ (not generally rooted at their centroids). For example, the path P_8 is f-isomorphic to the tree in Figure 4 with centroidal vertices u and v. Therefore $\mathbf{fws}(P_8) = 4\,3\,2\,1\,4\,3\,2\,1$.
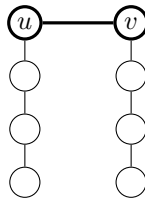
Figure 4: Representation of P_8 as a ordered pair of canonically ordered trees of order 4

Lemma 7 Let T and T' be two bicentroidal free trees. If $\mathbf{fws}(T) = \mathbf{fws}(T)$ then T is f-isomorphic to T'.

Proof Let $\{u, v\}$ and $\{u', v'\}$ be the centroidal vertices of T and T', respectively, and let $<T_u, T_v>$ and $<T'_{u'}, T'_{v'}>$ be the representations of T and T', respectively, described above. Suppose that $\mathbf{fws}(T) = \mathbf{fws}(T')$. Then $\mathbf{cws}(T_u) = \mathbf{cws}(T'_{u'})$ and $\mathbf{cws}(T_v) = \mathbf{cws}(T'_{v'})$. So, by Lemma 5, T_u and T_v are r-isomorphic to $T'_{u'}$ and $T'_{v'}$, respectively. Since we can recover T from T_u and T_v by adding an edge between u and v, and similarly for T', it immediately follows that T is f-isomorphic to T'. □

Lemma 8 The free weight sequence is a valid representation for free trees.

Proof If two free trees are isomorphic then they are both unicentroidal or both bicentroidal. The result then follows from Lemmas 6 and 7. □

3   Rooted tree generation

By Lemma 5, the canonical weight sequence is a valid representation for rooted trees. So, to generate all rooted trees of order n, we only need to generate every possible canonical weight sequence of length n.

An ordered set of integer sequences $[\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_p]$ is said to be *reverse lexicographically* (*relex*) *ordered* if $\mathbf{a}_i \geq \mathbf{a}_j$ when $i < j$, for all i and j. Let $\mathcal{B}(n)$ denote the relex ordered set of the canonical weight sequences of all rooted trees of order n. It follows from Lemmas 5 and 1 that, for each element $\mathbf{s} = s_1 s_2 \ldots s_n$ of $\mathcal{B}(n)$, there exists a unique canonically ordered tree R, with vertices labelled $v_1, v_2, \ldots, v_n$ in preorder, such that $\mathbf{ws}(R) = \mathbf{s}$, where $s_k = wt(v_k)$ for all k, $1 \leq k \leq n$.

If $\mathbf{s} = s_1 s_2 \ldots s_n$ is an integer sequence, we let $\mathbf{s}^{\#} = s_2 s_3 \ldots s_n$, i.e., $\mathbf{s}^{\#}$ is $\mathbf{s}$ with the first element s_1 removed. So if $\mathbf{s}$ is the weight sequence of an ordered tree R, then $\mathbf{s}^{\#}$ is the weight sequence of the ordered *forest* obtained by removing the root of R.

We write $\mathbf{s} \succcurlyeq \mathbf{t}$ if $\mathbf{t}$ is some other integer sequence such that either $\mathbf{s} \geq \mathbf{t}$ or $\mathbf{s}$ is a prefix of $\mathbf{t}$, i.e., $\mathbf{t} = \mathbf{s} \oplus \mathbf{x}$ for some integer sequence $\mathbf{x}$.

Let $\mathcal{A}_q(n)$ be the set of all ordered pairs $<\mathbf{a}, \mathbf{b}>$ in $\mathcal{B}(q) \times \mathcal{B}(n - q)$ such that $\mathbf{a} \succcurlyeq \mathbf{b}^{\#}$, and let $\mathcal{A}(n, maxq) = \bigcup\limits_{q=1}^{maxq} \mathcal{A}_q(n)$. We recall that if $<\mathbf{a}, \mathbf{b}>$ is in $\mathcal{A}_q(n)$ then the first element of $\mathbf{a}$ is q, $|\mathbf{a}| = q$, the first element of $\mathbf{b}$ is $n - q$ and $|\mathbf{b}| = n - q$.

Lemma 9 There is a bijection β from $\mathcal{A}(n, n - 1)$ to $\mathcal{B}(n)$ defined by

$$\beta(<\mathbf{a}, \mathbf{b}>) = n \oplus \mathbf{a} \oplus \mathbf{b}^{\#}. \tag{2}$$

Proof Suppose that $<\mathbf{a}, \mathbf{b}>$ is in $\mathcal{A}_q(n)$, for some q, $1 \leq q \leq n - 1$. We first show that $\beta(<\mathbf{a}, \mathbf{b}>) \in \mathcal{B}(n)$.

Let R_1 be a canonically ordered tree rooted at v such that $\mathbf{ws}(R_1) = \mathbf{a}$, and let R_2 be a canonically ordered tree rooted at u such that $\mathbf{ws}(R_2) = \mathbf{b}$. Let $u_1, u_2, \ldots, u_p$ be the children of u in order, and let R be a new ordered tree rooted at u with children $v, u_1, u_2, \ldots, u_p$, i.e., R is obtained from R_2 by adding R_1 as the new first subtree of u. Now $<\mathbf{a}, \mathbf{b}>$ is in $\mathcal{A}_q(n)$, so $\mathbf{a} \succcurlyeq \mathbf{b}^{\#}$, and thus $wt(v) \geq wt(u_1)$. Therefore R is canonically ordered as both R_1 and R_2 are canonically ordered. So $\mathbf{ws}(R)$ is in $\mathcal{B}(n)$ and, moreover, $\mathbf{ws}(R) = n \oplus \mathbf{a} \oplus \mathbf{b}^{\#}$ by Corollary 1. Therefore $\beta(<\mathbf{a}, \mathbf{b}>) \in \mathcal{B}(n)$.

Suppose that $<\mathbf{a}_0, \mathbf{b}_0>$ is in $\mathcal{A}_r(n)$, for some r, and that $n \oplus \mathbf{a}_0 \oplus \mathbf{b}_0^{\#} = n \oplus \mathbf{a} \oplus \mathbf{b}^{\#}$. Then $r = q$ since the first element of $\mathbf{a}_0$ must be equal to the first element of $\mathbf{a}$. It follows that $\mathbf{a}_0 = \mathbf{a}$ and $\mathbf{b}_0 = \mathbf{b}$, as $|\mathbf{a}_0| = |\mathbf{a}|$. Hence β is injective.

Now suppose that $\mathbf{s} = s_1 s_2 \ldots s_n$ is an element of $\mathcal{B}(n)$, and let R be the unique canonically ordered tree such that $\mathbf{ws}(R) = \mathbf{s}$. By Lemma 1(b) and (c), $\mathbf{ws}(R(v_2)) = s_2 s_3 \ldots s_{s_2+1}$ and $\mathbf{ws}(R - R(v_2)) = t\, s_{s_2+2}\, s_{s_2+3} \ldots s_n$ where $t = n - s_2$. Since R is canonically ordered, so are $R(v_2)$ and $R - R(v_2)$. Hence $\mathbf{ws}(R(v_2)) \in \mathcal{B}(s_2)$ and $\mathbf{ws}(R - R(v_2)) \in \mathcal{B}(n - s_2)$. Moreover, since R is canonically ordered, it follows from Corollary 1 and the definition of $\succcurlyeq$ that $s_2 s_3 \ldots s_{s_2+1} \succcurlyeq s_{s_2+2}\, s_{s_2+3} \ldots s_n$. Therefore $<\mathbf{ws}(R(v_2)), \mathbf{ws}(R - R(v_2))>$ is in $\mathcal{A}_{s_2}(n)$. Hence β is onto, and is therefore a bijection. □

Corollary 2 For any n, $\mathcal{B}(n)$ can be constructed from the sets $\mathcal{B}(q)$, where $1 \leq q \leq n-1$.
   Proof It is easy to construct all rooted trees, and therefore $\mathcal{B}(n)$, when $n \leq 3$. The result then follows using equation (2) and induction on n.   □

The image of $\mathcal{A}_q(n)$ under the bijection β defined in (2) is denoted by $\mathcal{B}_q(n)$, i.e., $\mathcal{B}_q(n)$ corresponds to those rooted trees of order n for which the first subtree of the root is of order q. So $\mathcal{B}_q(n)$ contains those sequences in $\mathcal{B}(n)$ for which the second element is q. Clearly $\mathcal{B}(n) = \bigcup\limits_{q=1}^{n-1} \mathcal{B}_q(n)$.

   Following along the lines of the proofs of Lemma 9 and Corollary 2, we now construct a simple recursive algorithm to generate the elements of $\mathcal{B}(n)$. For each q, $1 \leq q \leq n-1$, and for each $\mathbf{a}$ in $\mathcal{B}(q)$, we need to find those elements $\mathbf{b}$ in $\mathcal{B}(n-q)$ for which $\mathbf{a} \succcurlyeq \mathbf{b}^{\#}$. We then form the integer sequence $n \oplus \mathbf{a} \oplus \mathbf{b}^{\#}$ to obtain the appropriate element of $\mathcal{B}(n)$. We can avoid searching the whole of $\mathcal{B}(n-q)$ for those elements $\mathbf{b}$ for which $\mathbf{a} \succcurlyeq \mathbf{b}^{\#}$, by noting that we only need to consider those elements that are in $\mathcal{B}_r(n-q)$, where $1 \leq r \leq \min(n-q-1, q)$.

   In the pseudocode we use in the rest of the paper, we represent lists in square brackets; we use $\bigoplus$ for concatenating lists, and continue to use $\oplus$ for concatenating integer sequences. If L is a list, then L[*start* ...] denotes the sublist beginning at element L[*start*] and ending at the last element of L.

   The following function RootedTrees(n) generates $\mathcal{B}(n)$. It makes use of the helper function RTHelper(n,q) that generates $\mathcal{B}_q(n)$.

Function RootedTrees(n)
   if $n = 1$ **then return** $[1]$
   Bn $\leftarrow [\,]$
   for q **from** $n-1$ **downto** 1 **do**
      Bn $\leftarrow$ Bn $\bigoplus$ RTHelper(n,q)
   return Bn

Function RTHelper(n,q)
   Bqn $\leftarrow [\,]$
   $newq \leftarrow \min(n-q-1, q)$
   if $newq = 0$ **then**
      for a **in** RootedTrees(q) **do** Bqn $\leftarrow$ Bqn $\bigoplus [n \oplus$ a $]$
   else
      for a **in** RootedTrees(q) **do**
         for r **from** $newq$ **downto** 1 **do**
            for b **in** RTHelper($n-q,r$) **do**
               if a $\succcurlyeq$ b$^{\#}$ **then** Bqn $\leftarrow$ Bqn $\bigoplus [n \oplus$ a $\oplus$ b$^{\#}]$
   return Bqn

   There are two key points to note about the recursive calls in RTHelper. Firstly, the length of the subsequence corresponding to the first subtree of the root must be smaller than the order of the tree itself; so we always have $q < n$. Secondly, if $q = n-1$ then the sequence represents a tree in which the root has only one subtree; so we simply return n concatenated with the subsequence that corresponds to this subtree.

We note that $\mathcal{B}_r(n-q)$, the list returned by RTHelper$(n-q, r)$, will clearly require too much space for most values of r when $n-q$ is large. This problem is addressed by returning a generator instead of a list (see Section 5.3). We note further that, in the loops in RootedTrees and RTHelper, the variables q and r are counting down, so Bn and Bqn will be relex ordered, as required.

For example, consider the sets $\mathcal{B}(4) = [4\,3\,2\,1,\ 4\,3\,1\,1,\ 4\,2\,1\,1,\ 4\,1\,1\,1]$, $\mathcal{B}(3) = [3\,2\,1,\ 3\,1\,1]$, $\mathcal{B}(2) = [2\,1]$ and $\mathcal{B}(1) = [1]$, which correspond to the canonically ordered trees in Figure 5 (we have omitted the vertex labels for simplicity).

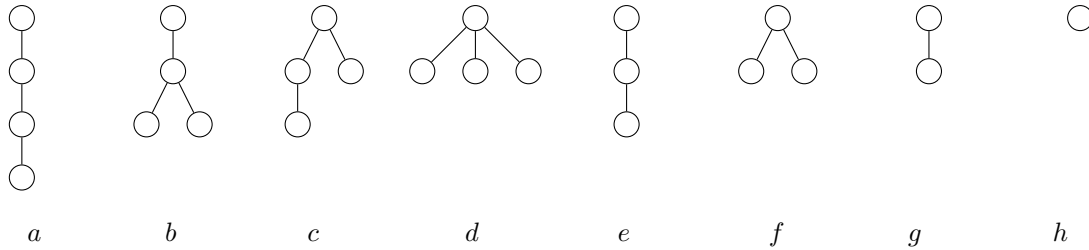$$a \qquad b \qquad c \qquad d \qquad e \qquad f \qquad g \qquad h$$

Figure 5: All canonically ordered trees of order at most 4

Recalling that $\mathcal{B}_q(n)$ contains those sequences in $\mathcal{B}(n)$ for which the second element is q, it is straightforward to show that the call RootedTrees(5) yields the set

$$\mathcal{B}(5) = [5\,4\,3\,2\,1,\ 5\,4\,3\,1\,1,\ 5\,4\,2\,1\,1,\ 5\,4\,1\,1\,1,\ 5\,3\,2\,1\,1,\ 5\,3\,1\,1\,1,\ 5\,2\,1\,2\,1,\ 5\,2\,1\,1\,1,\ 5\,1\,1\,1\,1].$$

These correspond to the canonically ordered trees in Figure 6, where the label indicates which pair of trees in Figure 5 - corresponding to **a** and **b** in equation (2) - are used to construct the tree.

We discuss some optimisations of the function RTHelper in Section 5.

$$ah \qquad bh \qquad ch \qquad dh \qquad eg \qquad fg \qquad ge \qquad gf \qquad hd$$
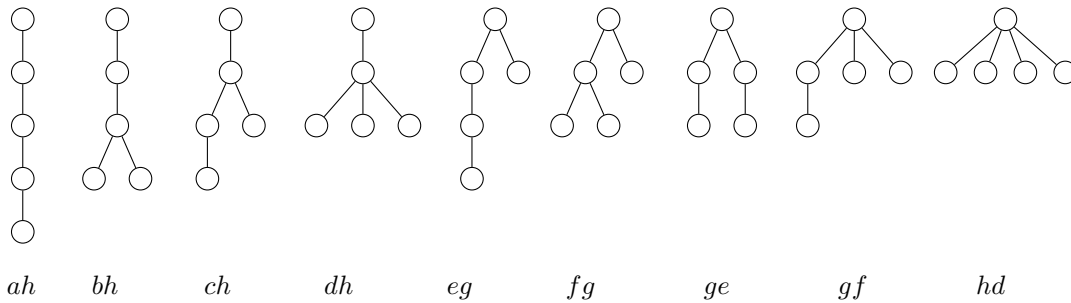
Figure 6: Canonically ordered trees of order 5

4   Free tree generation

By Lemma 8, the free weight sequence is a valid representation for free trees. So, in order to generate all free trees of order n, we only need to generate every possible free weight sequence of length n. We recall that a free tree is either unicentroidal or bicentroidal, which have slightly different definitions of the free weight sequence.

We denote the relex ordered set of the free weight sequence representations of all free trees, unicentroidal free trees and bicentroidal free trees of order n by $\mathcal{F}(n)$, $\mathcal{F}_U(n)$ and $\mathcal{F}_B(n)$, respectively. So $\mathcal{F}(n) = \mathcal{F}_U(n) \oplus \mathcal{F}_B(n)$, i.e., the elements of $\mathcal{F}_U(n)$ followed by those of $\mathcal{F}_B(n)$.

4.1   Unicentroidal

We recall from Section 2.4 that the free weight sequence $\mathbf{fws}(T)$ of a unicentroidal free tree T is the canonical weight sequence of any tree rooted at its centroid that is f-isomorphic to T. So $\mathcal{F}_U(n) \subseteq \mathcal{B}(n)$. We can therefore generate $\mathcal{F}_U(n)$ using a simple modification of the algorithm RootedTrees from Section 3: the canonically ordered tree R that represents a unicentroidal free tree T is rooted at its centroid, so the sub-trees of the root are of order at most $\frac{n-1}{2}$. It follows that $|\mathbf{a}| \leq \lfloor \frac{n-1}{2} \rfloor$ for every pair $<\!\mathbf{a}, \mathbf{b}\!>$ in $\mathcal{A}(n, n-1)$ for which $\beta(<\!\mathbf{a}, \mathbf{b}\!>)$ is in $\mathcal{F}_U(n)$.

Lemma 10 The mapping β defined in equation (2) is a bijection from $\mathcal{A}(n, \lfloor \frac{n-1}{2} \rfloor)$ to $\mathcal{F}_U(n)$.

  Proof We may represent any unicentroidal free tree T by a unique canonically ordered tree R in which the weight of each child of the root of R is at most $\lfloor \frac{n-1}{2} \rfloor$. So the result can be proved in a similar manner to Lemma 9, with the additional restriction that $|\mathbf{a}| \leq \lfloor \frac{n-1}{2} \rfloor$, i.e., we use $\mathcal{A}(n, \lfloor \frac{n-1}{2} \rfloor)$ instead of $\mathcal{A}(n, n-1)$. $\qquad\square$

Corollary 3 For any n, $\mathcal{F}_U(n)$ can be constructed from the sets $\mathcal{B}(q)$, where $1 \leq q \leq n-1$. $\qquad\square$

The following function UFT(n) generates the set $\mathcal{F}_U(n)$. It also makes use of the helper function RTHelper(n, q).

Function UFT(n)
    if $n = 1$ **then return** $[1]$
    UFn $\leftarrow$ [ ]
    $maxq \leftarrow \lfloor \frac{1}{2}(n-1) \rfloor$
    for q **from** $maxq$ **downto** 1 **do**
        UFn $\leftarrow$ UFn $\bigoplus$ RTHelper(n, q)
    return UFn

For example, we can construct $\mathcal{F}_U(8)$ using the call UFT(8) to obtain

$$\mathcal{F}_U(8) = \quad [\,83213211, 83213111, 83212121, 83212111, 83211111]$$
$$\bigoplus \quad [83113111, 83112121, 83112111, 83111111]$$
$$\bigoplus \quad [82121211, 82121111, 82111111, 81111111].$$

This corresponds to the set of canonically ordered trees in Figure 7.

4.2   Bicentroidal

We recall from Section 2.4 that the free weight sequence of a bicentroidal free tree with centroidal vertices u and v is $\mathbf{cws}(T_u) \oplus \mathbf{cws}(T_v)$, where $\mathbf{cws}(T_u) \geq \mathbf{cws}(T_v)$.

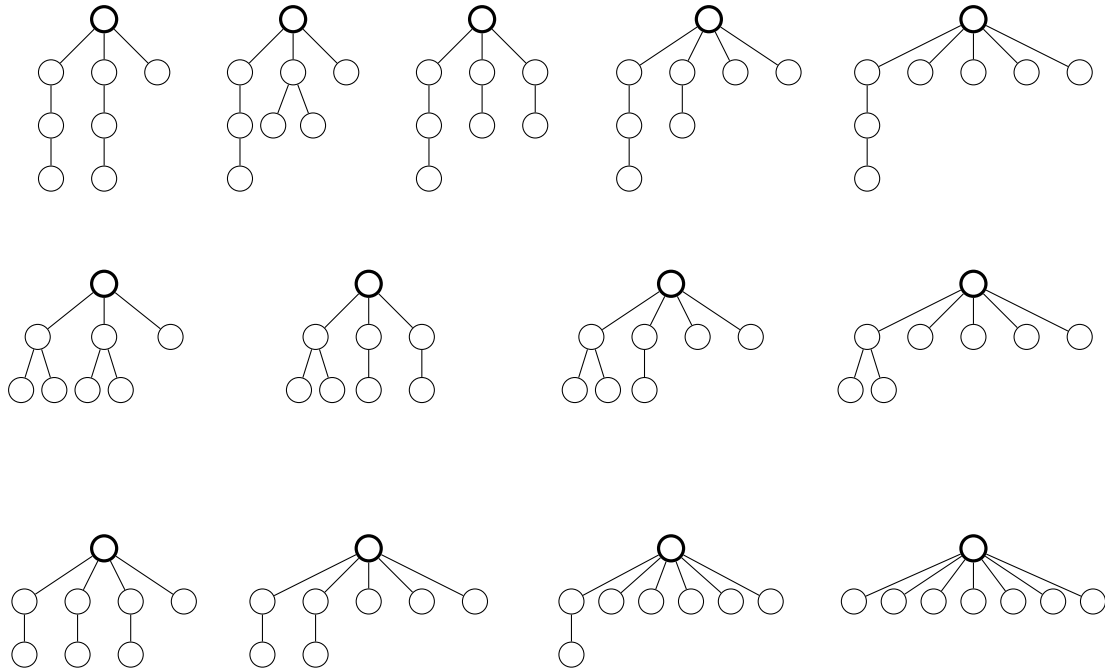**Lemma 11** For any n, $\mathcal{F}_B(n)$ can be constructed from the set $\mathcal{B}(\frac{n}{2})$. $\qquad\square$

Figure 7: Canonically ordered trees corresponding to unicentroidal free trees of order 8

The following function BFT(n) generates the set $\mathcal{F}_B(n)$.

Function BFT(n)
    BFn ← [ ]
    for a1 **in** RootedTrees($\frac{n}{2}$) **do**
        for a2 **in** RootedTrees($\frac{n}{2}$) **do**
            if a1 $\geq$ a2 **then** BFn ← BFn $\bigoplus$ [a1 $\oplus$ a2]
    return BFn

For example, we can construct $\mathcal{F}_B(8)$ using the call BFT(8) to obtain

$$
\begin{aligned}
\mathcal{F}_B(8) = \quad & [\,4\,3\,2\,1\,4\,3\,2\,1,\ 4\,3\,2\,1\,4\,3\,1\,1,\ 4\,3\,2\,1\,4\,2\,1\,1,\ 4\,3\,2\,1\,4\,1\,1\,1] \\
\oplus \quad & [4\,3\,1\,1\,4\,3\,1\,1,\ 4\,3\,1\,1\,4\,2\,1\,1,\ 4\,3\,1\,1\,4\,1\,1\,1] \\
\oplus \quad & [4\,2\,1\,1\,4\,2\,1\,1,\ 4\,2\,1\,1\,4\,1\,1\,1,\ 4\,1\,1\,1\,4\,1\,1\,1]
\end{aligned}
\tag{3}
$$

This corresponds to the set of ordered pairs of canonically ordered rooted trees of order 4 (see Figure 5), with an additional edge joining their roots, as shown in Figure 8.

By combining the unicentroidal and bicentroidal free tree algorithms, we can generate all free trees of order n using the following function FreeTrees(n).

Function FreeTrees(n)
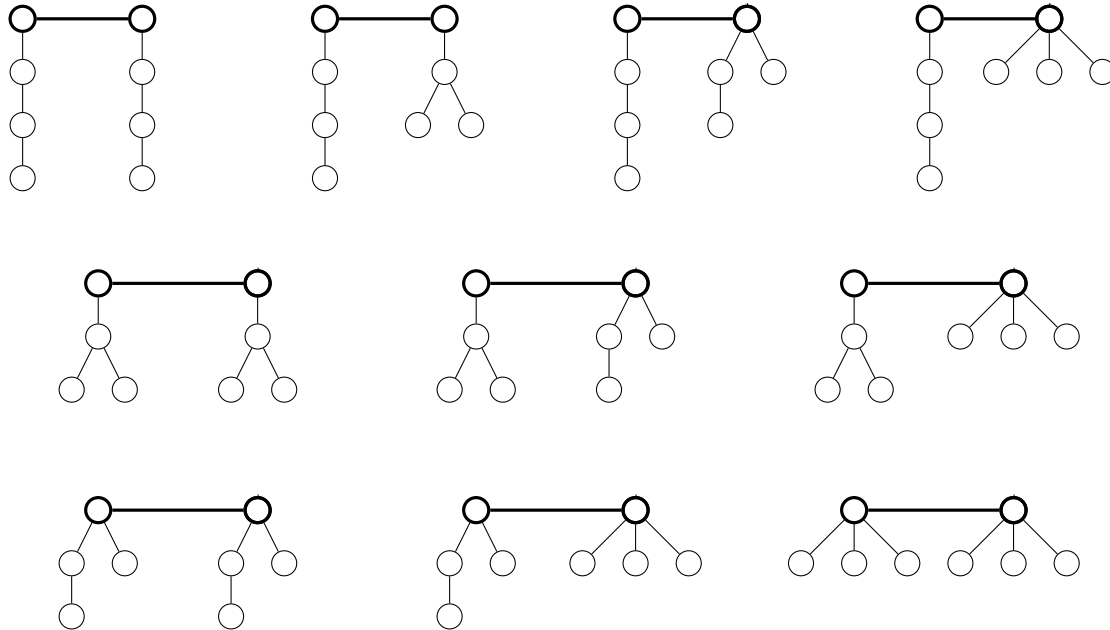   **return** UFT(n) $\bigoplus$ BFT(n)

Figure 8: Bicentroidal free trees of order 8 constructed from canonically ordered pairs of rooted trees of order 4

For example, the 23 free trees of order 8 is the ordered union of $\mathcal{F}_U(8)$ and $\mathcal{F}_B(8)$, i.e., the trees in Figures 7 and 8.

5   Improvements and implementation of the algorithms

We now outline some of the changes we have made to improve the efficiency of the functions described in Sections 3 and 4 above, and their implementations in Python. We use $\mathbf{a}^k$ to denote the integer sequence that is formed by the concatenation of k copies of the integer sequence $\mathbf{a}$.

5.1   Improvements to RTHelper

Firstly we note that, since there is only one rooted tree of order 1 and one of order 2, having canonical weight sequences 1 and 2 1, respectively, we may compute the result in a more efficient and explicit manner when q is 1 or 2. If $q = 1$ then the function should return the single sequence $n \oplus 1^{n-1}$, and if $q = 2$ then it should return the ordered set of sequences $(2\,1)^t \oplus 1^{n-1-2t}$ for t from $\left\lfloor \frac{n-1}{2} \right\rfloor$ down to 1. We also note that, when $q = n - 2$, the second subtree of the root contains just a single vertex, so $\mathbf{b}$ is just 1 in this case. These observations enable us to remove the recursive calls to RTHelper when $q \in \{1, 2, n-2\}$, as in the more efficient function RTHelper2(n, q) below. (In practice, in the implementation, we subsume the case $q = 1$ into the case $q = 2$ by reducing the lower limit of t from 1 to 0, and correspondingly increasing the lower limit of q from 1 to 2 in RootedTrees.)

Next we note that, during the execution of RTHelper, checking whether $\mathbf{a} \succcurlyeq \mathbf{b}^{\#}$ is only necessary when the order r of the second child of the root is the same as the order q of the first child. We further note that after $\mathbf{a} \succcurlyeq \mathbf{b}^{\#}$ for the first time, this will also hold for all subsequent sequences $\mathbf{b}$, since $\mathcal{B}(n)$ is relex ordered. This removes the necessity to check whether $\mathbf{a} \succcurlyeq \mathbf{b}^{\#}$ from then on.

Function RTHelper2(n, q)
    if $q = 1$ **then return** $n \oplus 1^{n-1}$
    Bqn $\leftarrow$ [ ]
    if $q = 2$ **then**
        for t **from** $\left\lfloor \frac{1}{2}(n-1) \right\rfloor$ **downto** 1 **do**
            Bqn $\leftarrow$ Bqn $\bigoplus \left[n \oplus (2\,1)^t \oplus 1^{n-1-2t} \right]$
    else if $q = n - 1$ **then**
        for a **in** RootedTrees(q) **do** Bqn $\leftarrow$ Bqn $\bigoplus [n \oplus$ a $]$
    else if $q = n - 2$ **then**
        for a **in** RootedTrees(q) **do** Bqn $\leftarrow$ Bqn $\bigoplus [n \oplus$ a $\oplus 1]$
    else
        $newq \leftarrow \min(n - q - 1, q)$
        for a **in** RootedTrees(q) **do**
            for r **from** $newq$ **downto** 1 **do**
                RTHelperList $\leftarrow$ RTHelper2$(n - q, r)$
                $start \leftarrow 1$
                if $r = q$ **then**
                    for b **in** RTHelperList **do**
                        if a $\succcurlyeq$ b$^{\#}$ **then break**
                        $start \leftarrow start + 1$
                for b **in** RTHelperList$[start \ldots]$ **do** Bqn $\leftarrow$ Bqn $\bigoplus [n \oplus$ a $\oplus$ b$^{\#}]$
    return Bqn

We shall assume from now on that the functions RootedTrees and UFT use the helper function RTHelper2 instead of RTHelper.

5.2   Caching of $\mathcal{B}(k)$ for smaller values of k

We now discuss how we can further improve the efficiency of the tree generation algorithms by caching $\mathcal{B}(k)$ for small values of k.

    RTHelper2(n, q) calls RootedTrees(q) and RTHelper2$(n - q, r)$, where $r \leq q$, and RootedTrees(q) calls RTHelper2(q, q'), where $q' < q$. It follows that $n' \leq q$ in all the calls to RootedTrees(n') made by RTHelper2(n, q), whether directly or indirectly. It therefore follows that we can obtain a significant increase in the efficiency of the function call RTHelper2(n,q) if we cache in memory $\mathcal{B}(k)$, for $1 \leq k \leq q$. This will increase the efficiency of both RootedTrees and UFT.

    For rooted tree generation, RootedTrees(n) makes calls to RTHelper2(n, q), where $1 \leq q \leq n - 1$. However, for large values of n, the space requirements to cache $\mathcal{B}(k)$, for $1 \leq k \leq n - 1$, would be prohibitive. For unicentroidal free tree generation, $q \leq \left\lfloor \frac{n-1}{2} \right\rfloor$ for all calls to RTHelper2(n, q) made by UFT(n). Furthermore, since BFT(n) only makes calls to RootedTrees$(\frac{n}{2})$, caching $\mathcal{B}(k)$ for $1 \leq k \leq \frac{n}{2}$ would avoid all calls of RootedTrees for both UFT(n) and BFT(n), and thus also FreeTrees(n). So, for example, to generate all $109, 972, 410, 221$ free trees of order 32, this would mean caching $\mathcal{B}(k)$ for $1 \leq k \leq 16$, which is perfectly feasible since there only $235, 381$ rooted trees of order 16. On the other hand, in order to generate all *rooted* trees of order 32 whilst avoiding all calls of RootedTrees, we would need to cache $\mathcal{B}(k)$ for $1 \leq k \leq 31$. Since there are nearly 10^{12} rooted trees of order 31, the cache space requirements for generating all rooted trees of order 32 would be of the order of at least 10 terabytes, and thus infeasible on practically all current computers (see [2] and [3] for tree counts).

Suppose that we cache $\mathcal{B}(k)$ for $1 \leq k \leq L$, and store the sets $\mathcal{B}(1)$, $\mathcal{B}(2), \ldots, \mathcal{B}(L)$ in the list RTList. We can then replace all calls of RootedTrees(q) in RTHelper2(n, q) by references to RTList[q], provided $L \geq q$.

As there are very few rooted trees of order less than five, we explicitly create $\mathcal{B}(1)$, $\mathcal{B}(2)$, $\mathcal{B}(3)$ and $\mathcal{B}(4)$ before calling RootedTrees(k) for larger values of k, as follows:

Procedure InitialiseRTList(L)
  RTList[1] = [1]
  RTList[2] = [2 1]
  RTList[3] = [3 2 1, 3 1 1]
  RTList[4] = [4 3 2 1, 4 3 1 1, 4 2 1 1, 4 1 1 1]
  for k **from** 5 **to** L **do** RTList[k] $\leftarrow$ RootedTrees(k)

In this initialisation, when computing RootedTrees(k), we note that we will already have computed the previous elements of RTList; so the recursive calls of RootedTrees in RTHelper2 may be replaced by references to RTList.

As explained above, we can replace all calls to RootedTrees from FreeTrees(n) by references to RTList if $L \geq \left\lfloor \frac{n}{2} \right\rfloor$. In practice, as explained below, in order to improve the efficiency of the code when $q = \left\lfloor \frac{n-1}{2} \right\rfloor$, we henceforth assume that $L \geq \left\lfloor \frac{n}{2} \right\rfloor + 1$.

To avoid all calls to RootedTrees(q) in RTHelper2(n, q), we require that $L \geq q$. This is clearly always true for FreeTrees. We will see that we can also avoid the recursive calls to RTHelper2($n - q, r$) when $q \geq n - L$.

When q, the order of the first subtree of the root, is at least $\left\lfloor \frac{n-1}{2} \right\rfloor$, then $newq = n - q - 1 \leq L$. So the r-loop (where r is the order of the second subtree of the root) can be dispensed with by letting **b** iterate through RTList[$n - q$]. Now, when q is at least $\left\lfloor \frac{n+1}{2} \right\rfloor$, then $r \leq newq < q$, so we can dispense with checking whether $\mathbf{a} \succcurlyeq \mathbf{b}^{\#}$. When $q = \left\lfloor \frac{n-1}{2} \right\rfloor$ and $r = q$, we can also avoid checking whether $\mathbf{a} \succcurlyeq \mathbf{b}^{\#}$ by skipping the initial elements of RTList[$n - q$], as we now explain.

Suppose that $r = q = \left\lfloor \frac{n-1}{2} \right\rfloor$ and $\mathbf{a} = \text{RTList}[q][k]$. When n is odd, $n - q - 1 = q$, so we can start with the element **b** for which $\mathbf{b}^{\#} = \mathbf{a}$; this is easily seen to be RTList[$n - q$][k]. When n is even, $n - q - 1 = q + 1 = \frac{n}{2}$, so we can skip the $|\mathcal{B}(\frac{n}{2})|$ elements for which the first subtree is of order $\frac{n}{2}$, and start with the element **b** for which $\mathbf{b}^{\#} = \mathbf{a} \oplus 1$; this is easily seen to be RTList[$n - q$][$k + s$], where $s = |\mathcal{B}(\frac{n}{2})|$.

When $\left\lfloor \frac{n-1}{2} \right\rfloor > q \geq n - L$, we can again dispense with the r-loop, letting **b** iterate through part of the RTList[$n - q$]. We pre-compute an array RTqstart, where RTqstart[n][q] is the index of the first element in the list RTList[n] for which the first subtree of the root of the corresponding rooted tree is of order q. We store in memory RTqstart[n][q], where $1 \leq q < n \leq L$. Since $\mathbf{a} \succcurlyeq \mathbf{b}^{\#}$ cannot hold for any element **b** in RTList[$n - q$] with index less than RTqstart[$n - q$][q], we only need to consider those elements **b** in RTList[$n - q$] from this index on.

We note that, following the above changes, we can replace $newq$ by q when $q < n - L$, since $q \leq L$. Making these changes to RTHelper2 yields the function RTHelper3. We shall assume from now on that the functions RootedTrees and UFT use the helper function RTHelper3 instead of RTHelper2.

Function RTHelper3(n, q) # This assumes that $L \geq q$ and $L \geq \left\lfloor \frac{n}{2} \right\rfloor + 1$.

    if $q = 1$ **then return** $n \oplus 1^{n-1}$

    Bqn $\leftarrow [\,]$

    if $q = 2$ **then**

        for t **from** $\left\lfloor \frac{1}{2}(n-1) \right\rfloor$ **downto** 1 **do**

            Bqn $\leftarrow$ Bqn $\bigoplus [n \oplus (2\,1)^t \oplus 1^{n-1-2t}]$

    else if $q = n - 1$ **then**

        for a **in** RTList$[q]$ **do** Bqn $\leftarrow$ Bqn $\bigoplus [n \oplus$ a $]$

    else if $q = n - 2$ **then**

        for a **in** RTList$[q]$ **do** Bqn $\leftarrow$ Bqn $\bigoplus [n \oplus$ a $\oplus 1]$

    else if $q \geq \left\lfloor \frac{n+1}{2} \right\rfloor$ **then**

        for a **in** RTList$[q]$ **do**

            for b **in** RTList$[n-q]$ **do** Bqn $\leftarrow$ Bqn $\bigoplus [n \oplus$ a $\oplus$ b$^{\#}]$

    else if $q = \left\lfloor \frac{n-1}{2} \right\rfloor$ **then**

        $start \leftarrow 1$

        if $n \equiv 0 \pmod 2$ **then** $start \leftarrow$ length(RTList$[\frac{n}{2}]$) $+ 1$

        for a **in** RTList$[q]$ **do**

            for b **in** RTList$[n-q][start \ldots]$ **do** Bqn $\leftarrow$ Bqn $\bigoplus [n \oplus$ a $\oplus$ b$^{\#}]$

            $start \leftarrow start + 1$

    else if $q \geq n - L$ **then**

        $start \leftarrow$ RTqstart$[n-q, q]$

        for a **in** RTList$[q]$ **do**

            for b **in** RTList$[n-q][start \ldots]$ **do**

                if a $\succcurlyeq$ b$^{\#}$ **then** break

                $start \leftarrow start + 1$

            for b **in** RTList$[n-q][start \ldots]$ **do** Bqn $\leftarrow$ Bqn $\bigoplus [n \oplus$ a $\oplus$ b$^{\#}]$

    else

        for a **in** RTList$[q]$ **do**

            for r **from** q **downto** 1 **do**

                RTHelperList $\leftarrow$ RTHelper3$(n-q, r)$

                $start \leftarrow 1$

                if $r = q$ **then**

                    for b **in** RTHelperList **do**

                        if a $\succcurlyeq$ b$^{\#}$ **then** break

                        $start \leftarrow start + 1$

                for b **in** RTHelperList$[start \ldots]$ **do** Bqn $\leftarrow$ Bqn $\bigoplus [n \oplus$ a $\oplus$ b$^{\#}]$

    return Bqn

In practice, as well as caching the $\mathcal{B}(k)$ in RTList(k), we also cache, in RFList(k), the relex ordered set of sequences that is obtained by replacing each sequence $\mathbf{b}$ in $\mathcal{B}(k)$ by $\mathbf{b}^{\#}$. We can then make $\mathbf{b}^{\#}$ iterate through the corresponding forest sequences instead of $\mathbf{b}$ iterating through the tree sequences, obviating the need to remove the first element of $\mathbf{b}$ each time. We therefore return $\mathbf{a} \oplus \mathbf{b}^{\#}$ instead of $n \oplus \mathbf{a} \oplus \mathbf{b}^{\#}$ in the function RFHelper that generates the sequence $\mathbf{s}^{\#}$ for each sequence $\mathbf{s}$ generated by RTHelper3. We prepend the weight of the root to each forest in the functions RootedTrees and UFT. Although we have not included the pseudocode for RFHelper, we use this function in our Python implementations instead of RTHelper3.

We note that, we may improve the efficiency of the function BFT by using the same idea as that used for the case $q = \left\lfloor \frac{n-1}{2} \right\rfloor$ in the function RTHelper3. This yields the following function BFT2.

> **Function** BFT2(n)
>   BFn $\leftarrow$ [ ]
> $start \leftarrow 1$
>   **for** a1 **in** RTList$\left[\frac{n}{2}\right]$ **do**
>   **for** a2 **in** RTList$\left[\frac{n}{2}\right][start...]$ **do** BFn $\leftarrow$ BFn $\bigoplus$ [a1 $\oplus$ a2]
> $start \leftarrow start + 1$
>   **return** BFn

We shall assume from now on that the function FreeTrees uses the function BFT2 instead of BFT.

5.3   Generators

The size of $\mathcal{B}(n)$ grows exponentially, so the list Bqn may become prohibitively large for large values of n, except when q is small. Therefore, to avoid creating and returning the list Bqn in RTHelper3, we instead return a generator. The changes necessary to effect this are, in essence, to simply replace all the assignments of the form Bqn $\leftarrow$ Bqn $\bigoplus$ [c] by the statement yield $\mathbf{c}$, and make corresponding changes to the other algorithms.

5.4   Strings for sequences

We store the weight sequences of the trees as alphanumeric strings, instead of lists, both to save storage and to create the canonical weight sequences more efficiently. We use the digits 1 to 9 for the corresponding weights, and the letters A, B, C, ... for weights 10, 11, 12, ... So the weight sequence of the free tree T in Figure 3 is denoted by the string "$A421141111$" instead of the sequence 10 4 2 1 1 4 1 1 1 1.

5.5   Adjacency lists and matrices

Although weight sequences are useful for generating trees, for most purposes a more conventional representation is required, such as adjacency lists or adjacency matrices. Most other tree generation algorithms also initially generate the trees using non-conventional representations (e.g., level sequences or parent sequences, as mentioned in the introduction). The adjacency lists or matrices are then constructed from the particular representation used.

We now give a brief explanation of how we can incorporate the construction of the adjacency lists of the free trees of order n into our algorithm FreeTrees, using a caching approach similar to that outlined in Section 5.2. We assume that the vertices are labelled 1 to n in preorder.

The algorithm AdjListFromWS below returns the adjacency list of a single free tree given its weight sequence. In the algorithm, we denote the jth element of the weight sequence ws by ws[j], and the list of n empty lists by $[\,]^{n}$.

Function AdjListFromWS(ws)
    $n \leftarrow$ length(ws)
    $\mathsf{A} \leftarrow [\ ]^n$
    for i **from** 1 **to** n **do**
        $j = i + 1$
        while $j < i + \mathsf{ws}[i]$ **do**
            $\mathsf{A}[i] \leftarrow \mathsf{A}[i] \bigoplus [j]$
            $\mathsf{A}[j] \leftarrow \mathsf{A}[j] \bigoplus [i]$
            $j \leftarrow j + \mathsf{ws}[j]$
    if $\mathsf{ws}[1] = \frac{n}{2}$ **then**
        $hn = \frac{n}{2} + 1$
        $\mathsf{A}[1] \leftarrow \mathsf{A}[1] \bigoplus [hn]$
        $\mathsf{A}[hn] \leftarrow \mathsf{A}[hn] \bigoplus [1]$
    return A

We note that, given the weight sequence of any *ordered* tree (or indeed any ordered forest), whether canonically ordered or not, this algorithm will return its adjacency list if we remove the assignment $\mathsf{A}[j] \leftarrow \mathsf{A}[j] \bigoplus [i]$ and the if statement (which, for a bicentroidal tree, adds the edge between the two centroids).

We extend the procedure InitialiseRTList to construct the adjacency list representations of the rooted trees, by calling the function AdjListFromWS on each weight sequence in RTList[k]. We store these representations in a hash table (implemented as a Python dictionary RTDict) using the weight sequence as the key.

We can now construct the adjacency list representation of all the free trees of order n while we construct their weight sequences: for each subtree of the root, we look up its adjacency list representation in the hash table, and then increase the label of each vertex by a suitable offset value. For a unicentroidal free tree represented by the integer sequence $n \oplus \mathbf{a} \oplus \mathbf{b}^{\#}$, we offset the labels of the vertices of the subtree corresponding to $\mathbf{a}$ by 1, and those of the vertices of the forest corresponding to $\mathbf{b}^{\#}$ by $|\mathbf{a}| + 1$. For a bicentroidal free tree, we only need to offset the labels of the vertices corresponding to the subtree rooted at the second bicentroid by $\frac{n}{2}$.

It is fairly straightforward to modify the above procedure in order to generate adjacency matrices instead of adjacency lists in a similar manner. The Python code for generating both the adjacency list and matrix representations is available as additional material to this paper.

6 Free tree generation runtimes and comparisons

As n, the order of the trees we are generating, is never large, we assume a uniform cost criterion, i.e., that the string operations, such as concatenation, take constant time. It is straightforward to prove, by induction, that the runtimes of RootedTrees(n) and RTHelper(n, q) would be at most proportional to the number of sequences returned by these functions if we removed from RTHelper the test $\mathbf{a} \succcurlyeq \mathbf{b}^{\#}$. We have computed empirically that the additional time taken for scanning along the list until $\mathbf{a} \succcurlyeq \mathbf{b}^{\#}$ is equivalent to increasing the runtime by less than 5%. Therefore, under the uniform cost criterion assumption, the generation time per tree would be approximately constant.

We now present an empirical comparison of our algorithm with the popular WROM algorithm. We implemented our algorithms in Python and compared these with the Python implementation of the WROM algorithm taken from NetworkX. All computations were performed using Python 3.7 and the JIT compiler PyPy3.6-v7.3.1, running on a Pentium i7 with 16GB RAM. We set L, the order of the largest tree for which we cache the representations, to be $\frac{n}{2} + 1$.

Table 1 shows the times in seconds to generate all free trees of order n and return the count of the number of trees, without saving the representations. BRFE refers to the algorithm FreeTrees described above and WROM to the algorithm described in [14]. BRFE(ls) and BRFE(mat) include converting the weight sequences into the adjacency list and matrix representations, respectively; WROM(ls) and WROM(mat) are defined similarly.

n	No. of trees	BRFE	BRFE(ls)	BRFE(mat)	WROM	WROM(ls)	WROM(mat)
18	123867	0.02	0.06	0.08	0.22	0.33	0.38
19	317955	0.04	0.10	0.10	0.29	0.53	0.69
20	823065	0.08	0.24	0.24	0.53	1.14	1.58
21	2144505	0.21	0.58	0.55	1.10	2.62	3.82
22	5623756	0.54	1.46	1.47	2.64	6.79	10.12
23	14828074	1.49	4.07	3.96	6.50	17.95	27.34
24	39299897	3.30	11.28	10.49	17.72	48.41	75.49
25	104636890	9.87	30.26	26.38	49.73	129.57	207.03
26	279793450	23.27	87.00	76.48	127.31	364.85	594.39
27	751065460	71.33	244.83	199.45	336.57	–	–
28	2023443032	161.89	–	–	–	–	–
29	5469566585	600.90	–	–	–	–	–

Table 1: Runtimes in seconds of the implementations of the BRFE and WROM algorithms

As can be seen, the runtimes for generating the weight sequences using BRFE are less than a quarter of those for generating the level sequences using WROM. The speed-ups for the times to create the adjacency list and matrix representations are similar. Due to the excessive times involved, we have not run some of the algorithms for the larger values of n.

We note that the runtimes for BRFE are about four times as long using the standard CPython implementation as those in Table 1, and the run times for WROM are about ten times as long. We further note that, by increasing the value of L, we could significantly reduce the runtimes of our algorithms for larger values of n, e.g., the runtime when $n = 29$ and $L = 19$ is around 556 seconds.

Li and Ruskey presented an alternative algorithm in [10] [11] that generates parent sequences, and compared a PASCAL implementation of their algorithm and the WROM algorithm. It can be seen from Table 5.2 in [10] that the runtime of their algorithm is about 70% of that of WROM. We can deduce from this that BRFE would take about a third of the time of their algorithm.

In Table 2, we show the corresponding generation times per tree in nanoseconds. For the smaller values of n, the times are inflated by the start-up times of the PyPy3 JIT compiler. These times seem to be approximately constant for $n \geq 22$, although for the BFRE algorithms there is some additional fluctuations between odd and even n, as the bicentroidal tree algorithm is faster than the unicentroidal. The increases in the times for the largest values of n for the BRFE algorithms are probably due to an increase in the number of hardware cache misses.

n	No. of trees	BRFE	BRFE(ls)	BRFE(mat)	WROM	WROM(ls)	WROM(mat)
18	123867	137.12	499.29	641.44	1766.34	2643.35	3101.38
19	317955	122.34	313.76	323.00	900.57	1665.94	2175.35
20	823065	96.38	288.43	287.50	641.24	1386.23	1919.2
21	2144505	99.76	272.52	254.87	515.16	1222.02	1779.22
22	5623756	95.87	259.45	260.88	469.36	1207.92	1799.06
23	14828074	100.77	274.29	266.75	438.12	1210.62	1843.91
24	39299897	83.93	287.13	267.04	450.94	1231.72	1920.81
25	104636890	94.28	289.22	252.06	475.30	1238.24	1978.52
26	279793450	83.18	310.95	273.34	455.02	1303.99	2124.39
27	751065460	94.97	325.98	265.56	448.12	–	–
28	2023443032	80.81	–	–	–	–	–
29	5469566585	109.86	–	–	–	–	–

Table 2: Generation times per tree in nanoseconds of the implementations of the algorithms

n	No. of trees	BRFE	BRFE(ls)	BRFE(mat)	WROM	WROM(ls)	WROM(mat)
18	123867	7.62	27.74	35.64	98.13	146.85	172.30
19	317955	6.44	16.51	17.00	47.40	87.68	114.49
20	823065	4.82	14.42	14.37	32.06	69.31	95.96
21	2144505	4.75	12.98	12.14	24.53	58.19	84.72
22	5623756	4.36	11.79	11.86	21.33	54.91	81.78
23	14828074	4.38	11.93	11.60	19.05	52.64	80.17
24	39299897	3.50	11.96	11.13	18.79	51.32	80.03
25	104636890	3.77	11.57	10.08	19.01	49.53	79.14
26	279793450	3.20	11.96	10.51	17.50	50.15	81.71
27	751065460	3.52	12.07	9.84	16.60		
28	2023443032	2.86	–	–	–	–	–
29	5469566585	3.79	–	–	–	–	–

Table 3: Ratio of the generation times per tree to n in nanoseconds

Although appoximately constant, there is an increase in the times in Table 2 as n increases, presumably because of the over-simplification in assuming a uniform cost criterion. If we assume the cost of string/list/matrix operations is proportional to the length, and divide the times in Table 2 by n, we obtain the times in Table 3. As we might expect, the times are now generally decreasing with n, since many of the operations involve smaller strings/lists/matrices.

7   Conclusion

In this paper we have presented new canonical representations for ordered, rooted and free trees. We constructed recursive algorithms for generating all rooted trees and all free trees of order n using these representations; each of these algorithms returns a list of the trees generated. We made a number of improvements to the algorithms and their Python implementations, including using generators to avoid having to explicitly construct and store the long lists of trees returned by the recursive calls. Moreover, in order to eliminate many of the recursive calls for small values of n, we cached the lists of rooted trees of small order. Our main interest is in the generation of free trees and, in this case, in order to eliminate a large proportion of the recursive calls, it is only necessary to cache the lists of rooted trees up to order

around $\frac{n}{2}$. We then described how the algorithm could be modified to generate the adjacency list or matrix representations of the trees.

We compared our Python implementation of the algorithm for generating free trees with the Python implementation of the well-known WROM algorithm taken from NetworkX. We used our algorithm to generate the free trees of order n, for $18 \leq n \leq 29$, but because of the longer runtimes, we only ran the WROM algorithm up to $n = 27$. It can be seen from Table 1 that the runtimes for the new algorithm are less than a quarter of those for the WROM algorithm (the improvement in the runtimes for the algorithms that generate adjacency lists or matrices is similar). From the comparisons in [10], we may deduce that our algorithm would take less than a third of the time of the algorithm presented there.

References

[1] Networkx documentation. http://networkx.github.io.

[2] On-line encyclopedia of integer sequences. https://oeis.org/A000055.

[3] On-line encyclopedia of integer sequences. https://oeis.org/A000081.

[4] T. Beyer and S. Hedetniemi. Constant time generation of rooted trees. *SIAM Journal of Computing*, 9:706–712, 1980. doi:10.1137/0209055.

[5] J. Bondy and U. Murty. *Graph Theory*. Springer, Berlin, 2008. doi:10.1007/978-1-84628-970-5.

[6] C. Borchardt. Über eine interpolationsformel für eine art symmetrischer functionen und über deren anwendung. *Math. Abh. der Akademie der Wissenschaften zu Berlin*, pages 1–20, 1860.

[7] R. Cook. A computational investigation of the universal reconstruction numbers of trees. Master's thesis, Birkbeck, University of London, London, UK, 2011.

[8] D. Knuth. *Combinatorial Algorithms Part 1, The Art of Computer Programming Vol 4A*. Addison-Wesley, Boston, Mass, 2011.

[9] D. Knuth. *Fundamental Algorithms, The Art of Computer Programming Vol 1*. Addison-Wesley, Boston, Mass, 2011.

[10] G. Li. Generation of rooted trees and free trees. Master's thesis, University of Victoria, British Columbia, CA, 1996.

[11] G. Li and F. Ruskey. The advantages of forward thinking in generating rooted and free trees. In *10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 939–940, New York, 1999. ACM.

[12] R. Otter. The number of trees. *Annals of Mathematics*, 49(3):583–599, 1948. doi:10.2307/1969046.

[13] J. Sawada. Generating rooted and free plane trees. *ACM Transactions on Algorithms*, 2(1):1–13, 2006. doi:10.1145/1125994.1125995.

[14] R. Wright, B. Richmond, A. Odlyzko, and B. McKay. Constant time generation of free trees. *SIAM Journal of Computing*, 15:540–548, 1986. doi:10.1137/0215039.