Constant Time Generation of Trees with Specified Diameter

Shin-ichi Nakano[1] and Takeaki Uno[2]

[1] Gunma University, Kiryu-Shi 376-8515, Japan
nakano@cs.gunma-u.ac.jp
[2] National Institute of Informatics, Tokyo 101-8430, Japan
uno@nii.jp

Abstract. Many algorithms to generate all trees with n vertices without repetition are already known. The best algorithm runs in time proportional to the number of trees. However, the time needed to generate each tree may not be bounded by a constant, even though it is "on average". In this paper we give a simple algorithm to generate all trees with exactly n vertices and diameter d, without repetition. Our algorithm generates each tree in constant time. It also generates all trees so that each tree can be obtained from the preceding tree by at most three operations. Each operation consists of a deletion of a vertex and an addition of a vertex. By using the algorithm for each diameter $2, 3, \cdots, n-1$, we can generate all trees with n vertices.

1 Introduction

It is useful to have the complete list of graphs for a particular class. One can use such a list to search for a counter-example to some conjecture, to find the best graph among all candidate graphs, or to experimentally measure the average performance of an algorithm over all possible input graphs.

Many algorithms to generate a particular class of graphs, without repetition, are already known [2, 6, 7, 9, 8, 10, 15]. Many excellent textbooks have been published on the subject [3, 5, 14].

Algorithms to generate all trees with n vertices without repetition are already known. The best algorithm [15] runs in time proportional to the number of trees. However, the time needed to generate each tree may not be bounded by a constant, even though it is "on average".

In this paper we give a simple algorithm to generate, without repetition, all trees with exactly n vertices and diameter d. Our algorithm generates each tree in constant time. It does not output each tree entirely, but outputs the difference from the preceding tree.

The main idea of our algorithm is to define a simple relation among the trees, that is "a family tree" of trees (see Fig. 1), and outputs trees by traversing the family tree. *The family tree*, denoted by $T_{n,d}$, is the tree such that the vertices of $T_{n,d}$ correspond to the trees with n vertices and diameter d, and each edge corresponds to some relation between trees. By traversing the family tree with

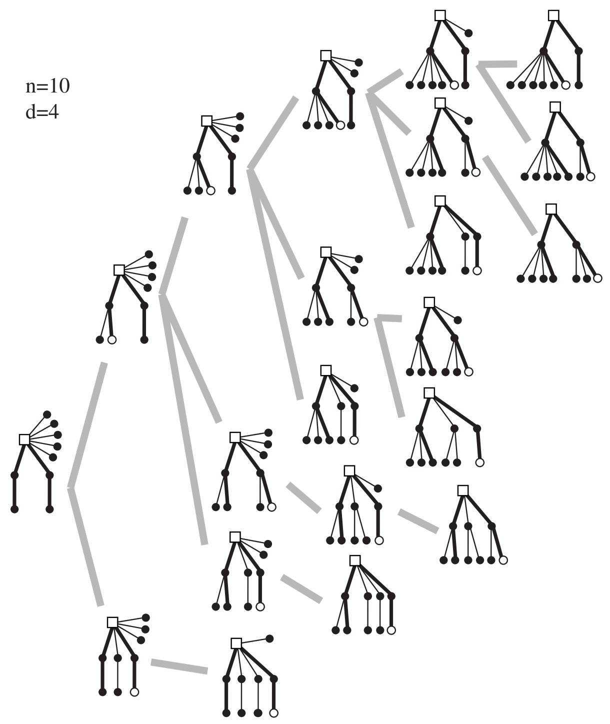

Fig. 1. The family tree $T_{10,4}$.

some ideas we can generate all trees corresponding to the vertices of the family tree, without repetition.

Furthermore, the algorithm generates all trees so that each tree can be obtained from the preceding tree by at most three operations, where each operation consists of a deletion of a vertex and an addition of a vertex. Therefore the derived sequence of trees is a kind of combinatorial Gray code [4, 12, 14] for trees with n vertices and diameter d. A Gray code [11] is a cyclic sequence of all 2^k bitstrings of length k, such that each bitstring differs from the preceding one in a small number of bit entries.

The rest of the paper is organized as follows. Section 2 gives some definitions. Section 3 introduces the family tree. Section 4 presents our first algorithm for the even diameter case. In Section 5 we sketch our algorithm for the odd diameter case. The algorithm generates each tree in $O(1)$ time on average. In Section 6 we improve the algorithm so that it generates each tree in $O(1)$ time. Finally Section 7 is a conclusion.

2 Preliminaries

In this section we give some definitions.

Let G be a connected graph with n vertices. An edge connecting vertices x and y is denoted by (x, y). The *degree* of a vertex v, denoted by $d(v)$, is

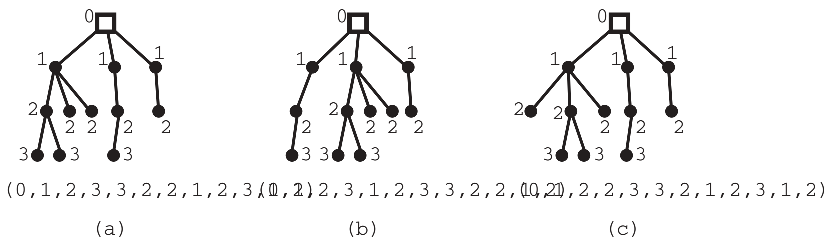

Fig. 2. The depth sequences.

the number of neighbors of v in G. A *path* is a sequence of distinct vertices $(v_0, v_1, \cdots, v_k)$ such that (v_{i-1}, v_i) is an edge for $i = 1, 2, \cdots, k$. The *length* of a path is the number of edges in the path. The *distance* between a pair of vertices u and v is the minimum length of a path between u and v. The *diameter* of G is the maximum distance between two vertices in G.

A *tree* is a connected graph without cycles. A *rooted* tree is a tree with one vertex r chosen as its *root*. For each vertex v in a rooted tree, let $UP(v)$ be the unique path from v to the root r. If $UP(v)$ has exactly k edges then we say that the *depth* of v is k, and write $dep(v) = k$. The *parent* of $v \neq r$ is its neighbor on $UP(v)$, and the *ancestors* of $v \neq r$ are the vertices on $UP(v)$ except v. The parent of the root r and the ancestors of r are not defined. We say that if v is the parent of u then u is *a child* of v, and if v is an ancestor of u then u is *a descendant* of v. A *leaf* is a vertex that has no child.

An *ordered tree* is a rooted tree with left-to-right ordering specified for the children of each vertex. We denote by $T(v)$ the ordered subtree of an ordered tree T consisting of a vertex v and all descendants of v that preserve the left-to-right ordering for the children of each vertex.

Let T be an ordered tree with n vertices, and $(v_1, v_2, \cdots, v_n)$ be the list of the vertices of T in preorder [1]. Let $dep(v_i)$ be the depth of v_i for $i = 1, 2, \cdots, n$. Then, the sequence $L(T) = (dep(v_1), dep(v_2), \cdots, dep(v_n))$ is called the *depth sequence* of T. Some examples are shown in Fig. 2. Note that those trees in Fig. 2 are isomorphic as rooted trees, but non-isomorphic as ordered trees.

Let T_1 and T_2 be two ordered trees, and $L(T_1) = (a_1, a_2, \cdots, a_n)$ and $L(T_2) = (b_1, b_2, \cdots, b_m)$ be their depth sequences. If either (1) $a_i = b_i$ for each $i = 1, 2, \cdots, j-1$ (possibly $j = 1$) and $a_j > b_j$, or (2) $a_i = b_i$ for each $i = 1, 2, \cdots, m$ and $n > m$, then we say that $L(T_1)$ is *heavier* than $L(T_2)$, and write $L(T_1) > L(T_2)$.

3 The Family Tree

In Section 3 and 4 we only consider the case where the diameter is even.

If a tree has $n \geq 3$ vertices and diameter 2, then the number of such a tree is exactly one, which is $K_{1,n-1}$. In the rest of the section we assume that the diameter is $2k \geq 4$.

Let T be a tree with the diameter $2k$. Let $v_0, v_1, \cdots, v_{2k}$ be a path in T having length $2k$. One can observe that T may have many such paths, but the vertex v_k, called *the center* of T, is unique [13, p72]. We assign to T the rooted tree R derived from T by choosing v_k as the root. Then we assign to R a unique ordered tree as follows.

Given a rooted tree R, since we can choose many left-to-right orderings for the children of each vertex, we can observe that R corresponds to many non-isomorphic ordered trees. Let H be the ordered tree corresponding to R that has the heaviest depth sequence $L(H)$. Then we say that H is the *left-heavy embedding* of R. For example, the ordered tree in Fig. 2(a) is the left-heavy embedding of a rooted tree, however the trees in Fig. 2(b) and (c) are not, since the one in Fig. 2(a) is heavier. We assign the ordered tree H to R.

Given a tree T, we have assigned to T a unique distinct rooted tree R, and then we have assigned to R a unique distinct ordered tree H, which is the left-heavy embedding of R. Note that T, R and H have the same diameter $2k$. One can observe that the assignment is a one-to-one mapping. Let $S_{n,2k}$ be the set of all left-heavy embeddings with exactly n vertices and diameter $2k$. If we generate all ordered trees in $S_{n,2k}$, then it also means the generation of all trees with exactly n vertices and diameter $2k$. We are going to generate all ordered trees in $S_{n,2k}$.

We have the following lemma.

Lemma 1. *An ordered tree H is the left-heavy embedding of a rooted tree if and only if for every pair of consecutive child vertices v_1 and v_2, that appear in this order in the left-to-right ordering, $L(T(v_1)) \geq L(T(v_2))$ holds.*

Proof. By contraction. □

In the rest of the paper the condition "$L(T(v_1)) \geq L(T(v_2))$ for each consecutive child vertices v_1 and v_2", is called *the left-heavy condition.*

Let H be a left-heavy embedding in $S_{n,2k}$ with root r_k. Let $c_1, c_2, \cdots, c_{d(r_k)}$ be the children of r_k. Assume they appear in this order in the left-to-right ordering. We say that c_i, $3 \leq i \leq d(r_k)$ is a *waiting vertex* if $c_i, c_{i+1}, \cdots, c_{d(r_k)}$ are leaves. Since H has a path of lenght $2k$ with the center r_k, one can observe that c_1 and c_2 have a descendant at depth k, respectively. Thus, neither c_1 nor c_2 are leaves. We denote by $A(H)$ the ordered tree derived from H by removing all (possibly none) waiting vertices. We say that $A(H)$ is *the active tree* of H. Note that the diameter of $A(H)$ is also $2k$.

Let c_a be the rightmost child of the root r_k in $A(H)$. Let $P_{right} = (v_0 = r_k, v_1 = c_a, v_2, \cdots, v_x)$ be the path in $A(H)$ such that v_i is the rightmost child of v_{i-1} for each i, $1 \leq i \leq x$, and v_x is a leaf in $A(H)$. We call P_{right} *the right path* of H. If $v_1 = c_2$ and $H(v_1)$ is a path, then we say H is *right empty.* Note that $H(v_1)$ is the ordered subtree of H induced by v_1 and all descendants of v_1. Similarly, let $P_{left} = (u_0 = r_k, u_1 = c_1, u_2, \cdots, u_y)$ be the path in $A(H)$ such that u_1 is the leftmost child of u_0, and u_i is the rightmost child of u_{i-1} for each i, $2 \leq i \leq y$, and u_y is a leaf in $A(H)$. We call P_{left} *the left path* of H. If $H(u_1)$ is a path, then we say H is *left empty.* The right and left paths are depicted as thick lines in Fig. 1.

If H is not right empty then v_x is called *the active leaf* of H. Otherwise, if H is not left empty then u_y is called *the active leaf* of H. Otherwise, $A(H)$ is a path of lenght $2k$, and H has no active leaf.

We have the following lemma.

Lemma 2. *Let H be an ordered tree in $S_{n,2k}$ that has an active leaf. Then the ordered tree derived from H by (i) removing the active leaf of H, then (ii) adding one leaf as the rightmost child of the root, is also in $S_{n,2k}$. Moreover, H is heavier than the derived ordered tree.*

Proof. Removing the active leaf and then adding one leaf as the rightmost child of the root never destroys the left-heavy condition. And the number of vertices in the derived tree is still n. Furthermore the diameter of the derived tree is again $2k$. Thus any derived tree is also in $S_{n,2k}$.

The proof for the second half of the claim is omitted. □

Assume that H is an ordered tree in $S_{n,2k}$ that has an active leaf. We denote by $P(H)$ the ordered tree derived from H by (i) removing the active leaf of H, then (ii) adding one leaf as the rightmost child of the root. We say that $P(H)$ is *the parent tree* of H and H is *a child tree* of $P(H)$. By the lemma above, $P(H)$ is also in $S_{n,2k}$. Given an ordered tree H in $S_{n,2k}$, we can have the unique sequence $H, P(H), P(P(H)), \cdots$ of ordered trees in $S_{n,2k}$, which eventually ends with the ordered tree that has no active leaf. That is the ordered tree consisting of a path of length $2k$ and $(n-2k-1)$ waiting vertices. By merging these sequences we can have *the family tree* of $S_{n,2k}$, denoted by $T_{n,2k}$, such that the vertices of $T_{n,2k}$ correspond to the trees in $S_{n,2k}$, and each edge corresponds to each relation between some H and $P(H)$. For instance, $T_{10,4}$ is shown in Fig. 1.

4 Algorithm

In this section we give an algorithm to construct $T_{n,2k}$.

If we can generate all child trees of a given ordered tree in $S_{n,2k}$, then in a recursive manner we can construct $T_{n,2k}$. This means we can generate all trees with exactly n vertices and diameter $2k$. Now we are going to generate all child trees of a given ordered tree.

Let H be an ordered tree in $S_{n,2k}$. Let $P_{right} = (v_0 = r_k, v_1, \cdots, v_x)$ be the right path of H, and $P_{left} = (u_0 = r_k, u_1, \cdots, u_y)$ be the left path of H. We construct some ordered trees by slightly modifying H as follows. Set $x' = \min\{x, k-1\}$ and $y' = \min\{y, k-1\}$.

If H has at least one waiting vertex and H is right empty then we define $H[i]$, $1 \leq i \leq y'$, as the ordered tree derived from H by (i) removing the rightmost waiting vertex, then (ii) adding a new vertex as the rightmost child of u_i. See Fig. 3 for some examples. Note that the constraint $i \leq y' \leq k-1$ ensures that the diameter of $H[i]$ remains $2k$.

If H has at least one waiting vertex, then we define $H[i+]$, $1 \leq i \leq x'$, as the ordered tree derived from H by (i) removing the rightmost waiting vertex, then (ii) adding a new vertex as the rightmost child of v_i. See some examples in Fig. 3.

If H has at least two waiting vertices, then we define $H[+]$ as the ordered tree derived from H by (i) removing the rightmost waiting vertex, then (ii) adding a new vertex as the only child vertex of the leftmost waiting vertex. See Fig. 3.

We can observe that each child tree of H is in $\{H[1], H[2], \cdots, H[y']\} \cup \{H[1+], H[2+], \cdots, H[x'+]\} \cup \{H[+]\}$. However, not all trees in $\{H[1], H[2], \cdots, H[y']\} \cup \{H[1+], H[2+], \cdots, H[x'+]\} \cup \{H[+]\}$ are child trees of H, so we need to check whether each possible child tree is actually a child tree of H.

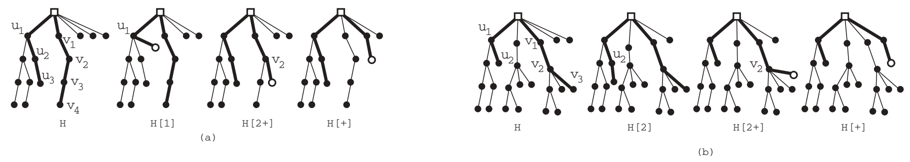

Fig. 3. The possible child trees.

We need some notations here. If vertex v_{i-1} has two or more children in the active tree $A(H)$, then we denote by v'_i the child of v_{i-1} that precedes v_i. Thus v'_i is the 2nd last child of v_{i-1} in $A(H)$. Similarly, for u_{i-1}, we denote by u'_i the 2nd last child of u_{i-1}. Note that $H(v)$ is the ordered subtree of H induced by v and all descendants of v.

We now have the following lemma.

Lemma 3. *Let H be an ordered tree in $S_{n,2k}$ with the right path $(v_0 = r_k, v_1, \cdots, \cdots, v_x)$ and the left path $(u_0 = r_k, u_1, \cdots, u_y)$.*

(1) $H[i]$, $i \leq \min\{y, k-1\}$, is a child tree of H if and only if H has at least one waiting vertex and is right empty, and for each j, $j = 1, 2, \cdots, i$, either u_{j-1} has only one child u_j in H, or $L(H(u'_j)) \geq L(H(u_j))$ holds in $H[i]$.

(2) $H[i+]$, $i \leq \min\{x, k-1\}$, is a child tree of H if and only if H has at least one waiting vertex, and for each j, $j = 1, 2, \cdots, i$, either v_{j-1} has only one child v_j in H, or $L(H(v'_j)) \geq L(H(v_j))$ holds in $H[i+]$.

(3) $H[+]$ is a child tree of H if and only if H has at least two waiting vertices.

Proof. (1) Since $H \in S_{n,2k}$ the left heavy condition has held in H. Then, only for vertex $u = u_0, u_1 \cdots, u_i$, $L(H(u))$ in $H[i]$ is heavier than $L(H(u))$ in H. The claim checks all of these possible changes that may destroy the left-heavy condition.

(2) (3) Omitted. □

If we generate each tree in $\{H[1], H[2], \cdots, H[y']\} \cup \{H[1+], H[2+], \cdots, H[x'+]\} \cup \{H[+]\}$ and check whether it is actually a child tree or not based on the lemma above, then we need considerable running time. However, we can save running time as follows. We need some definitions here.

Let H be an ordered tree in $S_{n,2k}$. We define "active at depth" in the following three cases. First, assume that H is not right empty. We say that H is *active* at depth i if (i) the right path contains a vertex v_i with depth i, (ii) v_i has two or more child vertices, and (iii) $L(H(v_{i+1}))$ is a prefix of $L(H(v'_{i+1}))$. Intuitively, if H is active at depth i, then we are copying subtree $H(v_{i+1})$ from $H(v'_{i+1})$. Then, assume that H is right empty but not left empty. We say that H is *active* at depth i if (i) the left path contains a vertex u_i with depth i, (ii) u_i has two or more child vertices, and (iii) $L(H(u_{i+1}))$ is a prefix of $L(H(u'_{i+1}))$. Then assume that H is right and left empty. We say that H is *active* at depth 0. Note that $L(H(v_1))$ is a prefix of $L(H(u_1))$.

We can show that H is always active at some depth as follows. If H is not right empty, then let j be the maximum index such that vertex v_j has two or more child vertices. Since H is not right empty, H always has such a vertex. Now since H is left-heavy and $H(v_{j+1})$ is a path, $L(H(v_{j+1}))$ is a prefix of $L(H(v'_{j+1}))$. Thus H is active at depth j. Otherwise, H is right empty. Then if H is not left empty, in a similar manner as above, we can show that H is active at some depth. Otherwise, H is right and left empty. In this case H is active at depth 0. Therefore H is always active at some depth.

We say the *copy-depth* of H is c if H is active at depth c but not active at any depth in $\{0, 1, \cdots, c-1\}$.

Now we are going to generate all child trees of an ordered tree H in $S_{n,2k}$. We have the following four cases.

We assume that H has the copy-depth c, the right path $P_{right} = (v_0 = r_k, v_1, \cdots, v_x)$ and the left path $P_{left} = (u_0 = r_k, u_1, \cdots, u_y)$.

Case 1: H has no waiting vertex.

Then H corresponds to a leaf in $T_{n,2k}$. Hence H has no child tree.

Case 2: Otherwise, and if H is not right empty.

In this case, for $H[i]$, $i = 1, 2, \cdots, \min\{y, k-1\}$, the active leaf of $H[i]$ is on the right path of $H[i]$. So $H[i]$ is not a child tree of H.

If H has two waiting vertices, then $H[+]$ is defined and is a child tree of H. The copy-depth of $H[+]$ is 0. Otherwise, H has exactly one waiting vertex and $H[+]$ is not defined.

We have two subcases for $H[i+]$. Note that since Case 1 does not occur, H has a waiting vertex.

Case 2a: $L(H(v'_{c+1})) = L(H(v_{c+1}))$. (Intuitively the copy has completed.)

First we show that $H[c+]$ is a child tree of H. Since H has the copy-depth c, for $j = 1, 2, \cdots, c$, $L(H(v'_j)) > L(H(v_j))$ holds in H and $L(H(v_j))$ is not a prefix of $L(H(v'_j))$. Since, for $j = 1, 2, \cdots, c$, $L(H(v_j))$ is not a prefix of $L(H(v'_j))$, $L(H(v'_j)) > L(H(v_j))$ still holds in $H[c+]$. Thus by Lemma 4.1 $H[c+]$ is a child tree of H. The copy-depth of $H[c+]$ remains at c.

Similarly, $H[i+]$, $i = 1, 2, \cdots, c-1$, is a child tree of H, and the copy-depth of $H[i+]$ is i.

However, for each $H[i+]$, where $i = c+1, c+2, \cdots, \min\{x, k-1\}$, the left-heavy condition is destroyed because of $L(H(v'_{c+1})) < L(H(v_{c+1}))$ in $H[i+]$. Thus, they are not child trees.

Case 2b: Otherwise. (Now $L(H(v'_{c+1})) > L(H(v_{c+1}))$ holds. Intuitively the copy has not completed yet.)

Let $L(H(v'_{c+1})) = (dep(s_1), dep(s_2), \cdots, dep(s_{n'}))$, $L(H(v_{c+1})) = (dep(t_1), dep(t_2), \cdots, dep(t_{n''}))$, and set $z = dep(s_{n''+1})$. (Intuitively we are copying $H(v_{c+1})$ from $H(v'_{c+1})$ and $s_{n''+1}$ is the next vertex to be copied.)

First, $H[(z-1)+]$ is a child tree of H, and the copy-depth of $H[(z-1)+]$ remains at c.

Similarly, $H[1+], H[2+], \cdots, H[(z-2)+]$ are child trees of H, and we will prove in a lemma below that the copy-depth of $H[i+]$ is i for $i = 0, 1, \cdots, z-2$.

For each of $H[i+]$, where $i = z, z+1, \cdots, \min\{x, k-1\}$, $L(H(v'_{c+1})) < L(H(v_{c+1}))$ holds in $H[i]$. Therefore, they are not left-heavy.

Case 3: Otherwise, and if H is not left empty.

Now H is right empty and H has a waiting vertex. Let z' be the $(k+1)$-th depth in $L(H)$.

Then $H[i+]$, $i = 1, 2, \cdots, z'-1$, is a child tree of H. The copy-depth of $H[i+]$ is i for $i = 1, 2, \cdots, z'-2$, and 0 for $z'-1$. On the other hand, $H[i+]$, where $i = z, z+1, \cdots, \min\{x, k-1\}$, is not a child tree of H, since $L(T(u_1)) < L(T(v_1))$ and so $H[i+]$ is not left-heavy.

If H has two waiting vertices, then $H[+]$ is a child tree of H and the copy-depth of $H[+]$ is 0. Otherwise, $H[+]$ is not defined.

We have two subcases for $H[i]$. Note that H has a waiting vertex.

Case 3a: $L(H(u'_{c+1})) = L(H(u_{c+1}))$.

$H[i]$, $i = 1, 2, \cdots, c$, is a child tree of H, and the copy-depth of $H[i]$ is i.

However, $H[i]$, where $i = c+1, c+2, \cdots, y$, is not a child tree of H.

Case 3b: Otherwise.

Let $L(H(u'_{c+1})) = (dep(s_1), dep(s_2), \cdots, dep(s_{n'}))$, $L(H(u_{c+1})) = (dep(t_1), dep(t_2), \cdots, dep(t_{n''}))$, and set $z = dep(s_{n''+1})$.

$H[1], H[2], \cdots, H[(z-1)]$ are child trees of H. The copy-depth of $H[i]$ is i for $i = 0, 1, \cdots, z-2$, and c for $i = z-1$.

For each of $H[i]$, where $i{=}z, z{+}1, \cdots, \min\{y, k{-}1\}$, $L(H(v'_{c+1})){<}L(H(v_{c+1}))$ holds in $H[i]$, therefore they are not left-heavy.

Case 4: Otherwise. (Now H is right and left empty.)

$H[i+]$, $i = 1, 2, \cdots, \min\{x, k-1\}$, is not a child tree of H.

If H has two waiting vertices, then $H[+]$ is a child tree of H and the copy-depth of $H[+]$ is 0. Otherwise, $H[+]$ is not defined.

$H[i]$, $i = 1, 2, \cdots, k-1$, is a child tree of H, and the copy-depth of $H[i+]$ is i.

Lemma 4. *In Case 2(b) the copy-depth of $H[i]$ is i for $i = 1, 2, \cdots, z-2$.*

Proof. For $i = 1, 2, \cdots, c$ the claim is obvious, so we assume otherwise. We can observe that the copy-depth of $H[i]$, $c+1 \leq i \leq z-2$, is never smaller than c, and $H[i]$ is active at i. So the copy-depth of $H[i]$ is somewhere between i and c.

Assume for contradiction that the copy-depth of $H[i]$ is $j < i$. Let $dep(w)$ be the last occurrence of depth j in $L(H[i])$. By the assumption above, w has two or more child vertices. Let w_1 be the rightmost child of w, and w_2 be the child vertex of w preceding w_1. See Fig. 4 for examples. Let w' be the vertex in $H(v'_{c+1})$

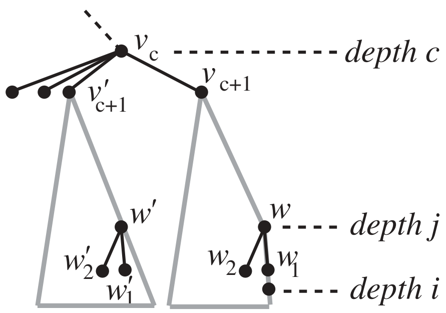

Fig. 4. Illustration for Lemma 4.2.

corresponding to w, and w'_1 and w'_2 be vertices in $H(v'_{c+1})$ corresponding to w_1 and w_2. (Note that we are copying $H(v_{c+1})$ from $H(v'_{c+1})$.) Now since $H \in S_{n,2k}$, $L(H(w'_2)) \geq L(H(w'_1))$ holds. By the choice of i, $L(H(w'_1)) > L(H(w_1))$ holds and $L(H(w_1))$ is not a prefix of $L(H(w'_1))$. Since the copy-depth of H is c, $L(H(w'_2)) = L(H(w_2))$. Then $L(H(w_2)) = L(H(w'_2)) \geq L(H(w'_1)) > L(H(w_1))$ holds, and $L(H(w_1))$ is not a prefix of $L(H(w'_1))$. Thus $L(H(w_1))$ is not a prefix of $L(H(w_2))$, and the copy-depth of $H[i]$ is not j, a contradiction.

Thus the copy-depth of $H[i]$ is i for $i = 1, 2, \cdots, z-2$. □

Based on the case analysis above, we have the following algorithm.

```
Procedure find-all-children($T$, $c$)
{ $T$ is the current tree, and $c$ is the copy-depth of $T$.}
begin
Output $H$ { Output the difference from the preceding tree.}
if $H$ has no waiting vertices {Case 1}
then return
else if $H$ is not right empty
then {Case 2}
  begin
    if $H$ has two waiting vertices then find-all-children($H[+]$, 0)
    if $L(H(v'_{c+1})) = L(H(v_{c+1}))$     then {Case 2a}
      for $i = 1$ to $c$
        find-all-children($H[i+]$, $i$)
    else {Case 2b} { $H(T(v'_{c+1})) > L(H(v_{c+1}))$ }
    { Let $z$ be the depth of the next vertex to be copied.}
      for $i = 1$ to $z - 2$
        find-all-children($H[i+]$, $i$)
      find-all-children($H[(z-1)+]$, $c$)
  end
else if $H$ is not left empty
then {Case 3}
  begin
    { Let $z'$ be the $(k+1)$-th depth in $L(H)$.}
```

```
21      for  $i = 1$ to $z' - 2$
22        find-all-children($H[i+]$, $i$)
23      find-all-children($H[(z'-1)+]$, 0)
24      if $H$ has two waiting vertices then find-all-children($H[+]$, 0)
25      if $L(H(u'_{c+1})) = L(H(u_{c+1}))$      then {Case 3a}
26        for  $i = 1$ to $c$
27          find-all-children($H[i]$, $i$)
28      else  {Case 3b} { $H(T(u'_{c+1})) > L(H(u_{c+1}))$ }
29      { Let $z$ be the depth of the next vertex to be copied.}
30        for  $i = 1$ to $z - 2$
31          find-all-children($H[i]$, $i$)
32        find-all-children($H[z-1]$, $c$)
33    end
34  else  {$H$ is right empty and left empty.}
35    begin
36      if $H$ has two waiting vertices then find-all-children($H[+]$, 0)
37      for  $i = 1$ to $k - 1$
38        find-all-children($H[i]$, $i$)
39    end
    end
```

```
Algorithm find-all-trees($n$)
begin
  Output the tree $H$ that consists of the path of length $2k$ and $(n - 2k - 1)$
of waiting vertices.
  find-all-children($H$, 0)
end
```

Theorem 1. *The algorithm uses $O(n)$ space and runs in $O(f(n))$ time, where $f(n)$ is the number of nonisomorphic trees with exactly n vertices and diameter $2k$.*

Proof. Since we traverse the family tree $T_{n,2k}$ and output each ordered tree at each corresponding vertex of $T_{n,2k}$, we can generate all trees with exactly n vertex and diameter $2k$.

We maintain the last two occurrences of each depth in each subtree $T(v_1)$ and $T(u_1)$ in four arrays of length k. We record the update of the four arrays and restore the arrays if return occur. Thus we can find v_i, v'_i, u_i and u'_i in constant time for each i.

We also maintain the current copy-depth c and the vertex next to be copied. Therefore with the help of the above arrays we can check the conditions in Lines 08 and 25 in constant time. Also, we can compute the value z and z' in constant time.

Other parts of the algorithm need only constant time of computation for each edge of $T_{n,2k}$.

Thus the algorithm runs in $O(f(n))$ time. Note that the algorithm does not output each tree entirely, but the difference from the preceding tree.

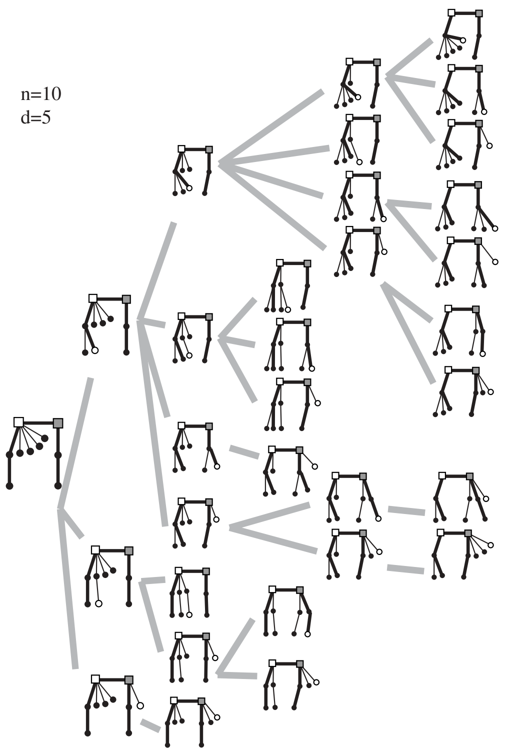

Fig. 5. The family tree $T_{10,5}$.

For each recursive call we need a constant amount of space, and the depth of the recursive call is bounded by n. Thus the algorithm uses $O(n)$ space. □

5 The Odd Diameter Case

In this section we sketch the case where the diameter is odd.

It is known that a tree with odd diameter $2k + 1$ may have many paths of length $2k + 1$, but all of them share a unique edge, called *the center* of T [13, p72].

Intuitively, by treating the edge as the root in a similar manner to the even diameter case, we can define the family tree $T_{n,2k+1}$. The detail is omitted. We only show $T_{10,5}$ in Fig. 5 as an example of the family tree.

6 Modification

The algorithm in Section 4 generates all trees with n vertices and diameter d in $O(f(n))$ time, where $f(n)$ is the number of nonisomorphic trees with n vertices and diameter d. Thus the algorithm generates each tree in $O(1)$ time "on average". However, after generating the tree corresponding to the last vertex in a large subtree of $T_{n,d}$, we have to merely return from the deep recursive call without outputting any trees. This may take $O(n)$ time. Therefore, we cannot generate each tree in $O(1)$ time.

However, a simple modification improves the algorithm to generate each tree in $O(1)$ time. The algorithm is as follows.

```
Procedure find-all-children2($T$, $c$, $depth$)
{ $T$ is the current tree, $c$ is the copy-depth of $T$, and $depth$ is the depth of
the recursive call.}
begin
  if $T$ has no waiting vertex
  then Output $T$ { $T$ is a leaf.}
  else
  begin
    if $depth$ is even
    then Output $T$ { before outputting its child trees.}
    Generate child trees $T_1, T_2, \cdots, T_x$ by the method in Section 4, and
    recursively call find-all-children2 for each child tree.
    if $depth$ is odd
    then Output $T$ { after outputting its child trees.}
  end
end
```

An execution of the algorithm is shown in Fig. 6.

One can observe that the algorithm generates all trees so that each tree can be obtained from the preceding tree by tracing at most three edges of $T_{n,k}$, each of which corresponds to an operation consisting of a deletion of a vertex and an addition of a vertex. Note that if T corresponds to a vertex v in $T_{n,k}$ with odd depth, then we may need to trace three edges to generate the next tree. Otherwise we need to trace at most two edges to generate the next tree. Thus, the derived sequence of the trees is a combinatorial Gray code [4, 12, 14] for rooted trees.

In Fig. 6 the added vertices are drawn as white circles, and the deleted, then added again, vertices are drawn as gray circles. (See the sixth tree in Fig. 6.) Each integer near an arrow mark is the number of edges in $T_{n,d}$ between the two vertices corresponding to the two trees. Each tree corresponding to a vertex in $T_{n,d}$ at odd depth is surrounded by a rectangle, and these trees are generated after all its child trees are generated.

Since $T_{10,4}$ has 21 vertices corresponding to the 21 trees in $S_{10,4}$, shown in Fig. 1, $T_{10,4}$ has 20 edges. In the algorithm we trace each edge twice, once for down and once for up. Therefore the sum is 40. This matches the sum of the integers near the arrow marks in Fig. 6.

7 Conclusion

In this paper we gave a simple algorithm to generate all trees with n vertices and diameter d. The algorithm generates each tree in constant time and clarifies the family tree of the trees.

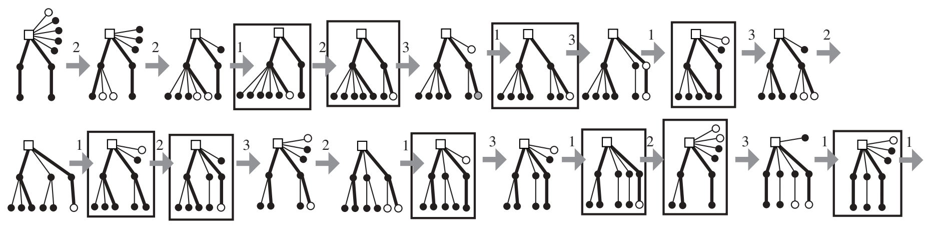

Fig. 6. An execution of the algorithm for $T_{10,4}$.

References

1. A.V. Aho and J.D. Ullman, *Foundations of Computer Science*, Computer Science Press, New York, (1995).
2. T. Beyer and S. M. Hedetniemi, *Constant Time Generation of Rooted Trees*, SIAM J. Comput., 9, (1980), pp. 706–712.
3. L.A. Goldberg, *Efficient Algorithms for Listing Combinatorial Structures*, Cambridge University Press, New York, (1993).
4. J.T. Joichi, D.E. White and S.G. Williamson, *Combinatorial Gray Codes*, SIAM J. Comput., 9, (1980), pp. 130–141.
5. D.L. Kreher and D.R. Stinson, *Combinatorial Algorithms*, CRC Press, Boca Raton, (1998).
6. Z. Li and S. Nakano, *Efficient Generation of Plane Triangulations without Repetitions*, Proc. ICALP2001, LNCS 2076, (2001), pp. 433–443.
7. G. Li and F. Ruskey, *The Advantage of Forward Thinking in Generating Rooted and Free Trees*, Proc. 10th Annual ACM-SIAM Symp. on Discrete Algorithms, (1999), pp. 939–940.
8. B.D. McKay, *Isomorph-free Exhaustive Generation*, J. of Algorithms, 26, (1998), pp. 306–324.
9. S. Nakano, *Efficient Generation of Plane Trees*, Information Processing Letters, 84, (2002), pp. 167–172.
10. R.C. Read, *How to Avoid Isomorphism Search When Cataloguing Combinatorial Configurations*, Annals of Discrete Mathematics, 2, (1978), pp. 107–120.
11. K.H. Rosen (Eds.), *Handbook of Discrete and Combinatorial Mathematics*, CRC Press, Boca Raton, (2000).
12. C. Savage, *A Survey of Combinatorial Gray Codes*, SIAM Review, 39, (1997) pp. 605–629.
13. D.B. West, *Introduction to Graph Theory, 2nd Ed*, Prentice Hall, NJ, (2001).
14. H.S. Wilf, *Combinatorial Algorithms: An Update*, SIAM, (1989).
15. R.A. Wright, B. Richmond, A. Odlyzko and B.D. McKay, *Constant Time Generation of Free Trees*, SIAM J. Comput., 15, (1986), pp. 540–548.