Generating Colored Trees
(Extended Abstract)

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. A c-tree is a tree such that each vertex has a color $c \in \{c_1, c_2, \cdots, c_m\}$. In this paper we give a simple algorithm to generate all c-trees with at most n vertices and diameter d, without repetition. Our algorithm generates each c-tree in constant time. By using the algorithm for each diameter $2, 3, \cdots, n-1$, we can generate all c-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 are already known [B80, LN01, LR99, M98, N02, R78, W86]. Many excellent textbooks have been published on the subject [G93, KS98, W89].

Algorithms to generate all trees with n vertices without repetition are already known. The algorithm [LR99, W86, NU03] generates each tree in $O(1)$ time on average, and the algorithm [NU04] generates each tree in $O(1)$ time.

Let $C = \{c_1 = a, c_2 = b, c_3 = c, \cdots, c_m\}$ be a set of colors. A c-tree is a tree such that each vertex has a color $c \in C$.

In this paper we give a simple algorithm to generate, without repetition, all c-trees with at most n vertices and diameter d. Our algorithm generates each c-tree in constant time. It does not output each c-tree entirely, but outputs the difference from the preceding c-tree. Our algorithm is based on our algorithm in [NU03], and completely different from [W86].

The main idea of our algorithm is first to define a simple relation among the c-trees, that is "a family tree" of c-trees (see Fig. 1), then outputs c-trees by traversing the family tree. *The family tree*, denoted by $T_{n,d,m}$, is the (huge) tree such that the vertices of $T_{n,d,m}$ correspond to the c-trees with at most n vertices and diameter d, and each edge corresponds to some relation between two c-trees. We give a formal definition in Section 4. By traversing the family tree we can generate all c-trees corresponding to the vertices of the family tree without repetition.

D. Kratsch (Ed.): WG 2005, LNCS 3787, pp. 249–260, 2005.

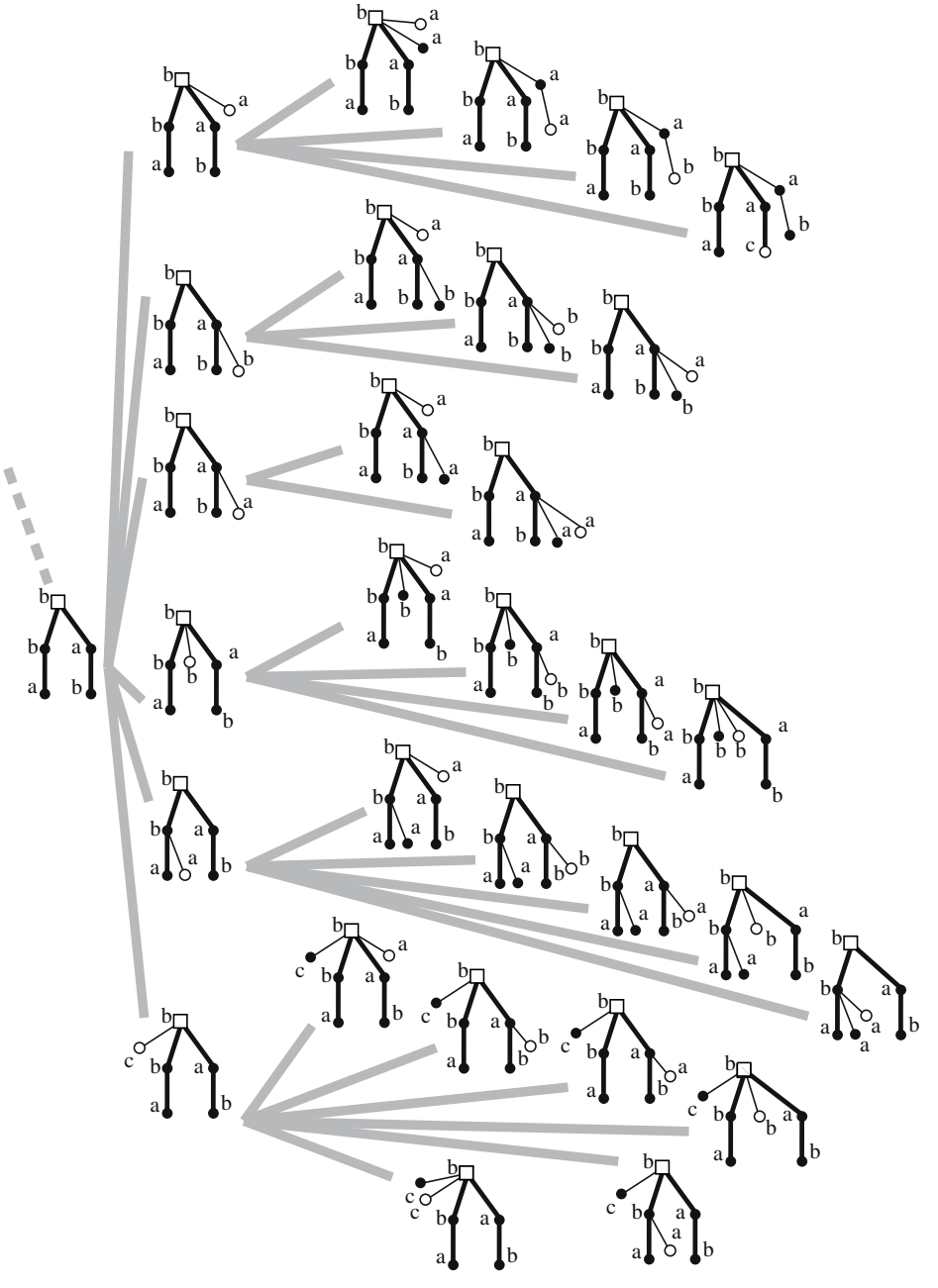

Fig. 1. The family tree $T_{7,4,3}$ sharing c-spine (a, b, b, a, b)

We have designed several generation algorithms based on the family trees [N02, NU03]. In this paper we first extend the method for c-trees.

Our algorithm has an application for a problem in tree mining. Many tree mining algorithms based on systematic enumeration of subtrees are already known. See a survey paper [C05]. Given a huge size of XML data, we wish to discover frequent patterns in the data. The frequent "patterns" are candidates for new "knowledge" [A03]. We can model XML data as a tree, where each data object is represented by a node with a label (=color), and each relationship between data objects by an edge. If we restrict patterns to frequent occurrences of the same colored subtrees, then we can solve the problem by (1) generating every c-tree, (2) then count the occurrences of each c-tree as a subgraph in the given XML tree, (3) then output the frequently occurred c-trees. By using our algorithm to generate every c-tree based on the family tree, we can efficiently prune rarely occurred c-trees, since in the family tree every "child" c-tree contains its "parent" c-tree as a subtree, so if the occurrence of a c-tree T is rare then the occurrence of any "descendant" c-tree of T is also rare. Thus we need not count the occurrences of each descendant rare c-tree of T.

The rest of the paper is organized as follows. Section 2 gives some definitions. Section 3 assigns a unique ordered c-tree H for each c-tree T, by choosing the root of T and the ordering of each child vertices. Section 4 introduces the family tree. Section 5 generates all c-paths, which are colored paths. Section 6 presents our algorithm to generate all c-trees for the even diameter case. In Section 7 we sketch our algorithm for the odd diameter case. Finally Section 8 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). 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* . A *c-tree* is a tree such that each vertex has a color $c \in \{c_1, c_2, \cdots, c_m\}$. 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

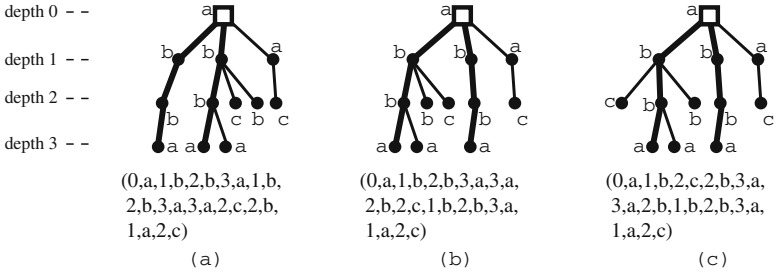

Fig. 2. The *dc* sequences

tree T consisting of a vertex v and all descendants of v with preserving the left-to-right ordering for the children of each vertex.

Let T be an ordered c-tree with n vertices, and $(v_1, v_2, \cdots, v_n)$ be the list of the vertices of T in preorder [A95]. Let $dep(v_i)$ be the depth of v_i and $c(v_i)$ be the color of v_i for $i = 1, 2, \cdots, n$. Then, the sequence $L(T) = (dep(v_1), c(v_1), dep(v_2), c(v_2), \cdots, dep(v_n), c(v_n))$ is called *the dc-sequence* of T. Some examples are shown in Fig. 2. Note that those trees in Fig. 2 are isomorphic as unordered c-trees, but non-isomorphic as ordered c-trees.

Let T_1 and T_2 be two ordered c-trees, and $L(T_1) = (a_1, b_1, a_2, b_2, \cdots, a_n, b_n)$ and $L(T_2) = (x_1, y_1, x_2, y_2, \cdots, x_z, y_z)$ be their *dc*-sequences. If there is some j such that $a_i = x_i$ and $b_i = y_i$ for each $i = 1, 2, \cdots, j-1$ (possibly $j = 1$) and either (i) $a_j > x_j$, (ii) $a_j = x_j$ and $b_j > y_j$, or (iii) $n > z = j - 1$, then we say that $L(T_1)$ is *heavier* than $L(T_2)$, and write $L(T_1) > L(T_2)$. For example, in Fig. 2, (a)<(b)<(c).

3 The Left-Heavy Embeddings

In Section 3–6, we only consider the case where the diameter is even.

Let T be a c-tree with diameter $2k$, and $(v_0, v_1, \cdots, v_{2k})$ 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 [W01–p72]. We assign to T the rooted c-tree R derived from T by choosing v_k as the root. Then we assign to R a unique ordered c-tree H as follows.

Given a rooted c-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 c-trees. Let H be the ordered c-tree corresponding to R that has the heaviest *dc* sequence $L(H)$. Then we say that H is the *left-heavy embedding* of R. For example, the ordered c-tree in Fig. 2(c) is the left-heavy embedding of a rooted c-tree, however the ordered c-trees in Fig. 2(a) and (b) are not, since the one in Fig. 2(c) is heavier. We assign the ordered c-tree H to R.

Given a c-tree T, we have assigned to T a unique distinct rooted c-tree R, and then we have assigned to R a unique distinct ordered c-tree H, which is the left-heavy embedding of R. Note that T, R and H have the same diameter $2k$. Let $S_{n,2k,m}$ be the set of all left-heavy embeddings of c-trees with at most n vertices and diameter $2k$. If we generate all ordered c-trees in $S_{n,2k,m}$, then

it also means the generation of all c-trees with at most n vertices and diameter $2k$. We are going to generate all ordered c-trees in $S_{n,2k,m}$.

We have the following lemma.

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

Proof. By contradiction. □

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.*

4 The Family Tree of c-Trees Sharing a c-Spine

Let H be a left-heavy embedding in $S_{n,2k,m}$ with root r. Let p_k be the first leaf of H at depth k in preorder, and $P_L = (r = p_0, p_1, \cdots, p_k)$ be the path between $r = p_0$ and p_k. We say that P_L is *the left spine* of H. Let $H^{'}$ be the ordered tree derived from H by removing $T(p_1)$, that is the subtree rooted at p_1. We can observe that $H^{'}$ is also a left-heavy embedding. Let q_k be the first leaf in $H^{'}$ at depth k in preorder, and $P_R = (r = q_0, q_1, \cdots, q_k)$ be the path between $r = q_0$ and q_k. We say that P_R is *the right spine* of H. We call $P_L \cup P_R$ *the spine* of H. We can observe that $P_L \cup P_R$ corresponds to a path with $2k$ edges. Since the diameter of H is $2k$, such p_k and q_k always exist.

An left-heavy embedding H in $S_{n,2k,m}$ is *trivial* if it consists of only $P_L \cup P_R$. Observe that any non-trivial $H \in S_{n,2k,m}$ has at least three leaves, so we can choose one leaf except p_k and q_k.

Assume $H \in S_{n,2k,m}$ is non-trivial. The last leaf x of H in preorder except p_k and q_k is called *the removable vertex* of H. Let $P(H)$ be the ordered c-tree derived from H by removing x.

Now we consider whether the left-heavy condition still holds in $P(H)$ or not. We have the following seven cases, depending on the location of x in H. Let $r_1, r_2, \cdots, r_{d(r)}$ be the children of r. Assume that they appear in this order in the left-to-right ordering of them. Also assume that p_k in P_L is a descendant of r_y and q_k in P_R is a descendant of r_z. See Fig. 3.

Case 1: $x \in T(r_i)$ for some $i > z$.

Then the left-heavy condition still holds in $P(H)$, since we remove the rightmost leaf, so a "right" subtree may loose some weight, but it never destroys the left-heavy condition.

Case 2: $x \in T(r_z)$, and x succeeds q_k in preorder.

Then the left-heavy condition still holds in $P(H)$. Similar to Case 1.

Case 3: $x \in T(r_z)$, and x precedes q_k in preorder.

Now there is no leaf x satisfying Case 1 or 2.

Let q_j on P_R be the ancestor of x having maximum depth, and $q_j = q_j^{'}, q_{j+1}^{'}, q_{j+2}^{'}, \cdots, q_s^{'} = x$ be the path between q_j and x. See Fig. 4. Note that by the

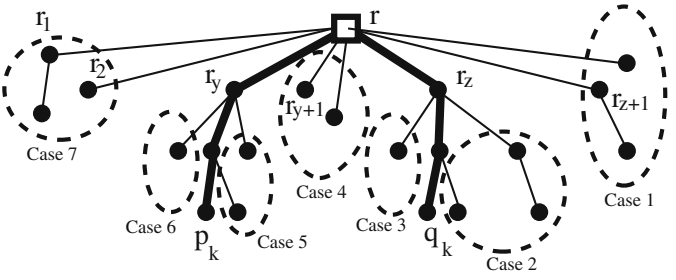

Fig. 3. Illustration for the seven cases

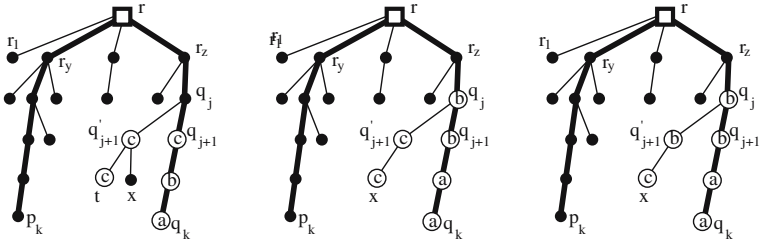

Fig. 4. Illustration for Case 3

definition of P_R, the depth of any descendant of q'_{j+1} is at most $k-1$. (Otherwise, q'_{j+1} has a descendant at depth k, and P_R must pass through q'_{j+1}. Now P_R is the path between r and the leftmost descendant of q'_{j+1} at depth k, a contradiction.)

We have the following two subcases.

Case 3(a): $T(q'_{j+1})$ is not a path.

Then the left-heavy condition still holds in $P(H)$. See Fig. 4(a), where the set of color is $\{c_1 = a, c_2 = b, c_3 = c\}$. Let t be the first leaf of $T(q'_{j+1})$ in preorder. Note that the dc sequence of the path from q'_{j+1} to t is heavier than the dc sequence of the path from q_{j+1} to q_k, since the left-heavy condition holds in H.

Case 3(b): $T(q'_{j+1})$ is a path.

Then we have two subcases.

If $c(q'_{j+1}) = c(q_{j+1}), c(q'_{j+2}) = c(q_{j+2}), \cdots c(q'_{s-1}) = c(q_{s-1})$ holds then the left-heavy condition destroyed in $P(H)$, since $L(T(q_{j+1}))$ is heavier than $L(T(q'_{j+1}))$ in $P(H)$. See Fig. 4(c). In this case, by swapping the order of q'_{j+1} and q_{j+1}, the left-heavy condition again holds. We re-define the resulting ordered c-tree as $P(H)$.

Otherwise the left-heavy condition still holds in $P(H)$. See Fig. 4(b).

Case 4: $x \in T(r_i)$ for some $i, y < i < z$.

Now r_{z-1} is the ancestor of x at depth one, and there is no leaf x satisfying Case 1, 2 or 3.

Case 4(a): $T(r_{z-1})$ is not a path.

Then the left-heavy condition still holds in $P(H)$. (Similar to Case 3(a).)

Case 4(b): $T(r_{z-1})$ is a path.

Similar to Case 3(b). We have two subcases as follows.

Let $q_0' = r, q_1', q_2', \cdots, q_s' = x$ be the path between r and x.

If $c(q_1) = c(q_1), c(q_2') = c(q_2), \cdots, c(q_{s-1}') = c(q_{s-1})$ holds, then the left-heavy condition destroyed in $P(H)$, since $L(T(q_1))$ is heavier than $L(T(q_1'))$ in $P(H)$. In this case, by swapping the order of $q_1' = r_{z-1}$ and $q_1 = r_z$, the left-heavy condition again holds. We re-define the resulting ordered c-tree as $P(H)$.

Case 5: $x \in T(r_y)$, and x succeeds p_k in preorder.

Then the left-heavy condition still holds in $P(H)$. Similar to Case 1 and 2.

Case 6: $x \in T(r_y)$, and x precedes p_k in preorder.

Similar to Case 3.

Case 7: $x \in T(r_i)$ for some $i < y$.

Similar to Case 4.

Since we never remove p_k and q_k, the spine always remains as it was. Note that $P(H)$ is left-heavy unless Case 3(b), 4(b) or 6(b) occurs, and even if Case 3(b), 4(b) or 6(b) occurs, by a possible modification, the resulting $P(H)$ is left-heavy.

Now we have the following lemma.

Lemma 2. *For any non-trivial $H \in S_{n,2k,m}$, $P(H)$ is also in $S_{n,2k,m}$ (after possible modification in Case 3(b), 4(b) or 6(b)).*

Given an ordered c-tree H in $S_{n,2k,m}$, by repeatedly removing the removable vertex, we can have the unique sequence $H, P(H), P(P(H)), \cdots$ of ordered c-trees in $S_{n,2k,m}$, which eventually ends with the trivial ordered c-tree H_1. By merging these sequences we can have *the family tree* of $S_{n,2k,m}$, denoted by $T_{n,2k,m}$, such that the vertices of $T_{n,2k,m}$ correspond to the c-trees in $S_{n,2k,m}$ having the same c-spine, and each edge corresponds to each relation between some H and $P(H)$. For instance, $T_{7,4,3}$ with c-spine (a, b, b, a, b) is shown in Fig. 1.

We say that $P(H)$ is *the parent tree* of H and H is *a child tree* of $P(H)$. We also say $P(H)$ is *a Type i child* of H if Case i occurs to find $P(H)$ from H.

5 Algorithm for c-Paths

A *c-path* is a path such that each vertex has a color $c \in \{c_1, c_2, \cdots, c_m\}$. Given an integer $2k$, one can generate every c-path with length $2k$ in constant time for each on average[RS00]. The detail is not mentioned in [RS00], but we can design a naive recursive algorithm as follows.

Let $S_{2k,m}$ be the set of all c-path with length $2k$. Let $(v_0, v_1, \cdots, v_{2k})$ be a c-path with edge (v_{i-1}, v_i) for $1 \leq i \leq 2k$. Let $(c(v_0), c(v_1), \cdots, c(v_{2k}))$ be the sequence of colors. Given a c-path, since we can choose the direction of the path we have two such sequences of colors, each one is the reverse of the other. Assume

S_1 and S_2 be the two sequence of colors for a c-path. We say a sequence S_1 is *a forward sequence* if S_1 is lexicographically larger than or equal to S_2.

If we generate all forward sequences with length $2k+1$ over alphabet $\{c_1, c_2, \cdots, c_m\}$, then they correspond to all c-paths in $S_{2k,m}$.

Each forward sequence $(x_0, x_1, \cdots, x_{2k})$ with length $2k+1$ is one of the following two types.

Type 1: $x_0 > x_{2k}$.
Then subsequence $(x_1, x_2, \cdots, x_{2k-1})$ is any sequence.
Type 2: $x_0 = x_{2k}$.
Then subsequence $(x_1, x_2, \cdots, x_{2k-1})$ is any forward sequence corresponds to a c-path in $S_{2k-1,m}$.

Based on the recursive structure above we can generate every c-path in constant time for each on average.

6 Algorithm for c-Trees

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

Using the algorithm in [RS00] or Section 5, we can generate every c-path in constant time for each. During the generation above, at the time we generate each c-path P_c, we wish to generate all c-trees in $S_{n,2k,m}$ sharing the c-spine P_c.

All we need to do is, given a c-tree H having the c-spine P_c, to generate all "child" c-trees of H. Then in a recursive manner we can generate all c-trees in $T_{n,2k,m}$ sharing the c-spine P_c. Now we are going to give an algorithm to generate all child c-trees of a given ordered c-tree.

Let H be an ordered c-tree in $S_{n,2k,m}$. We have eight cases depending on the location of the removable vertex x in H as follows.

Again let $r_1, r_2, \cdots, r_{d(r)}$ be the children of the root r. Assume they appear in this order in the left-to-right ordering of them. Let $P_L = (p_0 = r, p_1, \cdots, p_k)$, and $P_R = (q_0 = r, q_1, \cdots, q_k)$. Also assume that p_k in P_L is a descendant of r_y and q_k in P_R is a descendant of r_z. See Fig. 3.

Case 0: H is trivial, that means H has only two leaves p_k and q_k.
Case 1: $x \in T(r_i)$ for some $i > z$.
Case 2: $x \in T(r_z)$, and x succeeds q_k in preorder.
Case 3: $x \in T(r_z)$, and x precedes q_k in preorder.
Case 4: $x \in T(r_i)$ for some $i, y < i < z$.
Case 5: $x \in T(r_y)$, and x succeeds p_k in preorder.
Case 6: $x \in T(r_y)$, and x precedes p_k in preorder.
Case 7: $x \in T(r_i)$ for some $i < y$.

For each case we can generate all child c-trees of H. In this paper we only explain for Case 2 and Case 3, since other cases are similar.

Case 2: $x \in T(c_z)$, and x succeeds q_k in preorder.

If H has a child c-tree H_c with Type 4, 5, 6 or 7, then $P(H_c) \neq H$, a contradiction. Thus H has no child c-tree with Type 4, 5, 6 or 7.

Then consider for child c-trees with Type 1, 2 and 3.

Case 2(1): Child c-trees with Type 1.

Let $H_1[i]$ be the c-tree derived from H by adding the rightmost child leaf of r with color c_i. Assume that r_z has color c_j. The child c-trees of H with Type 1 are $H_1[0], H_1[1], \cdots, H_1[j]$. Note that $H_1[j+1]$ is not left heavy.

Case 2(2): Child c-trees with Type 2.

We need some definitions here.

Let $P = (u_0 = r, u_1, \cdots, u_{dep(x)} = x)$ be the path between $r = u_0$ and x. Let u_y on P_R be the ancestor of x having maximum depth. Thus P and P_R share the subpath $u_0 = q_0, u_1 = q_1, \cdots, u_y = q_y$). Let s_{i+1} be the child vertex of u_i preceding u_{i+1} (if such s_{i+1} exists), for $0 \leq i \leq dep(x)$.

We say that H is *active at depth* i if (i)u_i has two or more child vertices, and (ii)$L(H(u_{i+1}))$ is a prefix of $L(H(s_{i+1}))$. Intuitively, if H is active at depth i, then we are copying subtree $H(u_{i+1})$ from $H(s_{i+1})$. We say the *copy-depth* of H is d if H is active at depth d but not active at any depth in $\{0, 1, \cdots, d-1\}$. If H is not active at any depth, then we say the copy-depth of H is $dep(x)$. Assume that H is active at depth d.

Let $H_2[i, j]$ be the c-tree derived from H by adding the rightmost child leaf s to u_j with color c_i. Thus u_{j+1} precede the new vertex s in $H_2[i, j]$, if $j + 1 \leq dep(x)$. Any child c-tree of H with Type 2 is $H_2[i, j]$ for some i, j, however not all of them are child c-trees of H with Type 2. We need to check each carefully.

For $j = 0, 1, \cdots, d-1$, if $c(u_{j+1}) \geq c_i$ then $H_2[i, j]$ is a child c-tree of H, and otherwise $H_2[i, j]$ is not a child c-tree of H, since it is not left heavy. The copy-depth of each derived c-tree is j if c_i equal to $c(u_{j+1})$, and is $j+1$ otherwise.

Then consider for $j = d, d+1, \cdots, dep(x)$. Let n_R be the number of vertices in the subtree $H(u_{j+1})$ rooted at u_{j+1}, and t be the $(n_R + 1)$-th vertex in the subtree $H(s_{j+1})$ rooted at s_{j+1}. Assume t has a color c_ℓ.

If $j > dep(t)$ then $H_2[i, j]$ is not a child c-tree of H, since it is not left heavy. If $j = dep(t)$ but $\ell < i$ then $H_2[i, j]$ is not a child c-tree of H, since it is not left heavy. If $j = dep(t)$ and $\ell = i$ then $H_2[i, j]$ is a child c-tree of H. The copy-depth of the derived c-tree is again d. If $j = dep(t)$ and $\ell > i$ then $H_2[i, j]$ is a child c-tree of H. The copy-depth of each derived c-tree is j if c_i equal to $c(s_{j+1})$, and is $j + 1$ otherwise. If $j < dep(t)$ then $H_2[i, j]$ is a child c-tree of H for any i. The copy-depth of each derived c-tree is j if c_i equal to $c(s_{j+1})$, and is $j + 1$ otherwise.

Case 2(3): Child c-trees with Type 3.

In this case we need to check the reverse of Case 3(b) in Section 4. Thus a c-tree with Type 2 may have a child c-tree with Type 3.

Define $P = (u_0 = r, u_1, \cdots, u_{dep(x)} = x)$, u_y, ℓ as in Case 2(2).

If H has only one leaf succeeding q_k in preorder, $H(u_{y+1})$ is a path, $H(q_{y+1})$ is a path, and $L(H(u_{y+1}))$ is a prefix of $L(H(q_{y+1}))$, then, for each $i > \ell$, $H_2[i, dep(x)]$ is a child c-tree with Type 3, after swapping the order of u_{y+1} and q_{y+1}.

Case 3: $x \in T(c_z)$, and x precedes q_k in preorder.

If H has a child c-tree H_c with Type 4, 5, 6 or 7, then $P(H_c) \neq H$, a contradiction. Thus H has no child c-tree with Type 4, 5, 6 or 7.

Then consider for child c-trees with Type 1, 2 and 3.

Case 3(1): Child c-trees with Type 1.

Omitted. Similar to Case 2(1).

Case 3(2): Child c-trees with Type 2.

Omitted. Similar to Case 2(2).

Case 3(3): Child c-trees with Type 3.

Let $P = (u_0 = r, u_1, \cdots, u_{dep(x)} = x)$ be the path between $r = u_0$ and x. Let u_y on P_R be the ancestor of x having maximum depth. Let s_{i+1} be the child vertex of u_i preceding u_{i+1} (if such $s_{i+1} exists$), for $0 \leq i \leq dep(x)$.

We say that H is *active at depth* i if (i)u_i has two or more child vertices, and (ii)$L(H(u_{i+1}))$ is a prefix of $L(H(s_{i+1}))$. We say the *copy-depth* of H is d if H is active at depth d but not active at any depth in $\{0, 1, \cdots, d-1\}$. If H is not active at any depth, then we say the copy-depth of H is $dep(x)$. Assume that H is active at depth d.

For $j \geq y$, let $H_3[i, j]$ be the c-tree derived from H by adding the new child leaf s to u_j succeeding u_{j+1} with color c_i.

Any child c-tree of H with Type 3 is $H_3[i, j]$ for some i, j, however not all of them are child c-trees of H with Type 3.

For $j = y$, if $s \leq i < t$, where $c_s = c(u_{j+1})$ and $c_t = c(q_{j+1})$, then $H_2[i, j]$ is a child c-tree of H.

For $j = y+1, y+2, \cdots, d-1$, if $c(u_{j+1}) \geq i$ then $H_3[i, j]$ is a child c-tree of H, and otherwise $H_3[i, j]$ is not a child c-tree of H, since it is not left heavy. The copy-depth of each derived c-tree is j if c_i equal to $c(u_{j+1})$, and is $j+1$ otherwise.

Then consider for $j = d, d+1, \cdots, dep(x)$. Let n_R be the number of vertices in the subtree $H(u_{j+1})$ rooted at u_{j+1}, and t be the (n_R+1)-th vertex in the subtree $H(s_{j+1})$ rooted at s_{j+1}. Assume t has a color c_ℓ.

If $j > dep(t)$ then $H_3[i, j]$ is not a child c-tree of H, since it is not left heavy. If $j = dep(t)$ but $\ell < i$ then $H_3[i, j]$ is not a child c-tree of H, since it is not left heavy. If $j = dep(t)$ and $\ell = i$ then $H_3[i, j]$ is a child c-tree of H. The copy-depth of the derived c-tree is again d. If $j = dep(t)$ and $\ell > i$ then $H_3[i, j]$ is a child c-tree of H. The copy-depth of each derived c-tree is j if c_i equal to $c(s_{j+1})$, and is $j+1$ otherwise. If $j < dep(t)$ then $H_3[i, j]$ is a child c-tree of H for any i. The copy-depth of each derived c-tree is j if c_i equal to $c(s_{j+1})$, and is $j+1$ otherwise.

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

Theorem 1. *One can generate all c-trees in $O(f(n))$ time and $O(n)$ space, where $f(n)$ is the number of nonisomorphic c-trees with at most n vertices and diameter $2k$.*

Proof. Since we traverse the family tree $T_{n,2k,m}$ and output each ordered c-tree at each corresponding vertex of $T_{n,2k,m}$, we can generate all c-trees in $S_{n,2k,m}$.

We maintain the last two occurrences of each depth in two arrays of length k. We record the update of the arrays and restore the arrays if return occur. Thus we can find u_i in constant time for each i.

We also maintain the current copy-depth d and the vertex next to be copied.

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

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.

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. □

7 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 [W01–p72].

Intuitively, by treating the edge as the root, we can define the family tree $T_{n,2k+1,m}$ in a similar manner to the even diameter case. The detail is omitted.

8 Conclusion

In this paper we gave a simple algorithm to generate all c-trees with at most n vertices and diameter d. The algorithm generates each c-tree in constant time on average.

By slightly modifying the algorithm as shown below [NU03, NU04] we can improve the worst case running time. Since we traverse at most three edges to generate next c-tree, the algorithm generates each c-tree in constant time.

Procedure find-all-children(T, *depth*)
{ T is the current c-tree, and *depth* is the depth of the recursive call.}
begin
01 **if** *depth* is even
02 **then** Output T { before outputting its child c-trees.}
03 Generate child c-trees $T_1, T_2, \cdots, T_x$ by the method in Section 6 and 7, and
04 recursively call **find-all-children** for each child c-tree.
05 **if** *depth* is odd
06 **then** Output T { after outputting its child c-trees.}
end

References

[A95] A. V. Aho and J. D. Ullman, *Foundations of Computer Science*, Computer Science Press, New York, (1995).

[A03] T. Asai, H. Arimura, T. Uno, and S. Nakano, *Discovering Frequent Substructures in Large Unordered Trees*, The 6th International Conference on Discovery Science (DS'03) LNAI 2843, (2003), pp.47-61.

[B80] T. Beyer and S. M. Hedetniemi, *Constant Time Generation of Rooted Trees*, SIAM J. Comput., 9, (1980), pp.706-712.

[C05] Y. Chi, S. Nijsseny, R. R. Muntz and J. N. Kok, *Frequent Subtree Mining–An Overview*, Fundamenta Informaticae, Special Issue on Graph and Tree Mining, 2005. (to appear)

[G93] L. A. Goldberg, *Efficient Algorithms for Listing Combinatorial Structures*, Cambridge University Press, New York, (1993).

[KS98] D. L. Kreher and D. R. Stinson, *Combinatorial Algorithms*, CRC Press, Boca Raton, (1998).

[LN01] Z. Li and S. Nakano, *Efficient Generation of Plane Triangulations without Repetitions*, Proc. ICALP2001, LNCS 2076, (2001), pp.433–443.

[LR99] 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.

[M98] B. D. McKay, *Isomorph-free Exhaustive Generation*, J. of Algorithms, 26, (1998), pp.306-324.

[N02] S. Nakano, *Efficient Generation of Plane Trees*, Information Processing Letters, 84, (2002), pp.167–172.

[NU03] S. Nakano and T. Uno, *Efficient Generation of Rooted Trees*, NII Technical Report (NII-2003-005E) (2003). (http://research.nii.ac.jp/TechReports/03-005E.html)

[NU04] S. Nakano and T. Uno, *Constant Time Generation of Trees with Specified Diameter*, Proc. WG2004, LNCS, 3353, (2004), pp.33-45.

[RS00] F. Ruskey and J. Sawada, *A Fast Algorithm to Generate Unlabeled Necklaces*, Proc. of SODA (2000), pp.256–262

[R78] R. C. Read, *How to Avoid Isomorphism Search When Cataloguing Combinatorial Configurations*, Annals of Discrete Mathematics, 2, (1978), pp.107–120.

[W01] D. B. West, *Introduction to Graph Theory, 2nd Ed*, Prentice Hall, NJ, (2001).

[W89] H. S. Wilf, *Combinatorial Algorithms : An Update*, SIAM, (1989).

[W86] 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.