社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2003－AL－90 (4)
2003／5／23

A Simple Algorithm for Generating Unordered Rooted Trees

Shin-ichi Nakano[1] and Takeaki Uno[2]
[1] Gunma University, Kiryu-Shi 376-8515, Japan
email:nakano@cs.gunma-u.ac.jp
URL: http://www.msc.cs.gunma-u.ac.jp/~nakano/index.html
[2] National Institute of Informatics, Tokyo 101-8430, Japan
email:uno@nii.jp
URL: http://research.nii.ac.jp/~uno/

Abstract: We give a simple algorithm to enumerate all unordered rooted trees with at most n vertices in constant time for each on average. An unordered rooted tree is a rooted tree without ordering of children. No two rooted trees are output if they are isomorphic and their roots are the same. The algorithm also enumerates rooted trees with exactly n vertices in constant time for each on average.

Keywords: enumeration, generation, rooted tree, ordered tree, unordered tree

無順序木を列挙するシンプルなアルゴリズム

中野 眞一[1] and 宇野 毅明[2]
[1] 〒376-8515 群馬県桐生市群馬大学
email:nakano@cs.gunma-u.ac.jp
URL: http://www.msc.cs.gunma-u.ac.jp/~nakano/index.html
[2] 〒101-8430 東京都千代田区一ツ橋 国立情報学研究所
email:uno@nii.jp
URL: http://research.nii.ac.jp/~uno/

抄録: 本稿では, 頂点数が1からnの無順序木を列挙するシンプルなアルゴリズムを提案する. アルゴリズムの計算量は1つの木あたり定数時間である. 無順序木とは, 子供に対して順序が定義されていない木であり, 根が等しくかつ同型な根付き木は同一視され, 1度しか出力されない. 頂点数がnの無順序木も, このアルゴリズムを用いて, 1つあたり定数時間で列挙できる.

キーワード: 列挙, 生成, 根付き木, 順序木, 無順序木

1 Introduction

Enumeration algorithms have been derived for many problems. Many algorithms to generate given class of graphs are already known [B80] [LN01, N02, M98, W86]. Many nice textbooks have been published on the subject [G93, KS98].

Recently, these enumeration algorithms are often used for solving problems of data mining. For example, the problems of enumerating all the objects appearing more than α times in the data sets use enumeration algorithms. In these problems, trees, graphs, geometric objects and strings has to be enumerated in a depth first manner. The depth first manner means that the algorithm begins the enumertion from the empty set, and adds several elements in each iteration for finding other objects. In this paper, we address the problem of enumerating all

An algorithm to enumerate all rooted trees with exactly n vertices is known [B80]. The algorithm enumerates all rooted trees without duplications in constant time on average, thus it can also enumerate all rooted trees with at most n vertices, however, this way is not a depth first manner. Moreover, the algorithm is quite complicated.

In this paper, we give a more simple algorithm for our problem. The computation time of the algorithm is constant per tree on average, and the algorithm satisfies the above condition. Furthermore our algorithm clarifies a relation among the rooted trees, that is a family tree of the rooted trees (see Fig. 1), and outputs rooted trees by traversing the family tree.

The rest of the paper is organized as follows. Section 2 gives some definitions. Section 3 shows a tree structure over rooted trees. Section 4 presents our algorithm. Finally Section 5 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 *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 the *depth* of v, denoted by $dep(v)$, is k. If u is the vertex of $UP(v)$ adjacent to v, then u is the *parent* of v, and v is a *child* of u. If a vertex u is in $UP(v)$, then u is an *ancestor* of v and v is a *descendant* of u. A *leaf* is a vertex having no child.

A *rooted ordered tree* is a rooted tree with a left-to-right ordering specified for the children of each vertex. Two ordered rooted trees are said to be the same if there is a one-to-one mapping of vertices preserving the parent-child relationship and the left-to-right ordering. A rooted tree without orderings of children is called *unordered tree*.

For a rooted ordered tree T rooted at r_0, let $RP(T) = (r_0, r_1, \cdots, r_k)$ be the path such that r_i is the rightmost child of r_{i-1} for each i, $1 \leq i \leq k$, and r_k is a leaf of T. We call $RP(T)$ *the rightmost path* of T, and r_k *the rightmost leaf* of T. For a vertex v of T, we define $bro(v)$ by the the previous child of v in the left-to-right ordering. If v is the leftmost child or the root, we define $bro(v)$ by *nil*. Let $T(v)$ be the rooted ordered subtree consisting of vertex v and all descendant of v preserving the left-to-right ordering for the children of each vertex.

3 Unique Sequence for Unordered Tree

Let T be a rooted ordered tree with n vertices, and $(v_1, v_2, \cdots, v_n)$ be the vertices of T in preorder[A95]. The preorder is the visiting order of a depth first search such that the search visit the children in the left-to-right order. The sequence $L(T) = (dep(v_1), dep(v_2), \cdots, dep(v_n))$ is called the *depth sequence* of T. Two rooted ordered trees have the same depth sequences iff they are isomorphic. For examples, see Fig. 2. Note that those trees in Fig. 2 are the same as rooted unordered trees, but not the same as rooted ordered trees.

Suppose that $L(T_1) = (a_1, a_2, \cdots, a_c)$ and $L(T_2) = (b_1, b_2, \cdots, b_d)$ be the depth sequences of T_1 and T_2. If $a_i = b_i$ for each $i = 1, 2, \cdots, k-1$ (possibly $k = 1$) and either $a_k > b_k$ or $c > k - 1 = d$, then we say $L(T_1)$ is *heavier* than $L(T_2)$.

Given a rooted unordered tree T, we can observe that T corresponds to many non-isomorphic rooted ordered trees, since we can choose many left-to-right orderings for the children of each vertex. Let *left-heavy embedding* of T be the rooted ordered tree having the heaviest depth sequence among them. For example the rooted ordered tree in Fig. 2(a) is the left-heavy embedding of a rooted unordered tree, however Fig. 2(b) and (c) are not.

Let $\mathcal{S}_n^1$ be the set of all left-heavy embeddings of rooted unordered trees having at most n vertices. We are going to enumerate all trees of $\mathcal{S}_n^1$ instead of all rooted unordered trees with

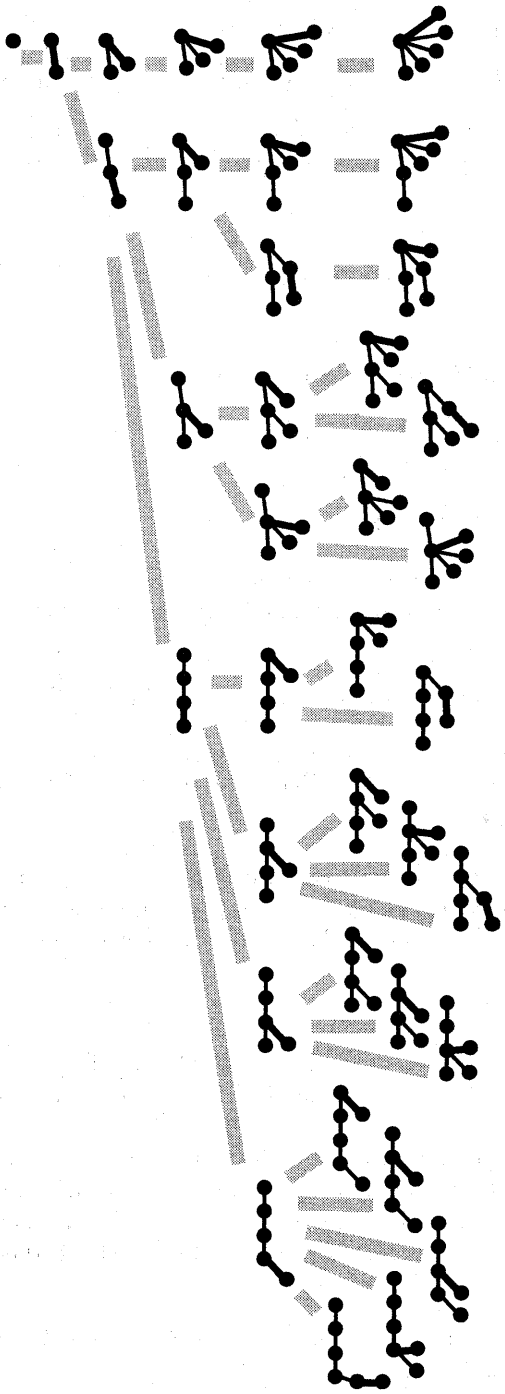

Figure 1: The family tree of rooted ordered trees having at most six vertices.

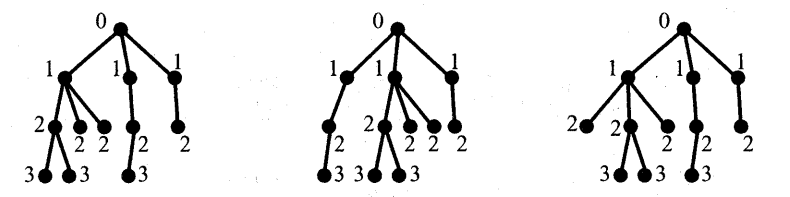

Figure 2: Depth sequences.

at most n vertices.

Left heavy embeddings satisfy the following conditions.

Lemma 1 *A rooted ordered tree T is in $\mathcal{S}_n^1$ iff $L(T(v)) \geq L(T(bro(v)))$ holds for any v satisfying $bro(v) \neq nil$.*

Proof: By contradiction. Omitted. ∎

For a tree $T \in \mathcal{S}_n^1$ having two or more vertices, we define the *the parent tree* $P(T)$ of T by the rooted ordered tree derived from T by removing its rightmost leaf. T is called a *child tree* of $P(T)$.

Lemma 2 *For any $T \in \mathcal{S}_n^1$ having two or more vertices, $P(T) \in \mathcal{S}_n^1$.*

Proof: By removing the rightmost leaf from T, $L(T(v))$ does not change if $v \notin RP(T)$, and removed the last depth if $v \in RP(T)$. Hence the condition of $L(T(bro(v))) \geq L(T(v))$ in Lemma 1 are not violated. ∎

The parent-child relationship among trees of $\mathcal{S}_n^1$ forms a tree, whose vertex set is $\mathcal{S}_n^1$ and two vertices of the tree are connected iff the corresponding rooted trees are a parent and its child. We call this tree *family tree* of $\mathcal{S}_n^1$. An instance is shown in Fig. 1.

4 Algorithm

If we can generate all child trees of a given tree of $\mathcal{S}_n^1$, then in a recursive manner we can traverse the family tree, and can enumerate all the trees of $\mathcal{S}_n^1$. In this section, we give an algorithm for this.

Let T be a rooted ordered tree of $\mathcal{S}_n^1$, and $RP(T) = \{r_0, ..., r_a\}$. Let $T + i$ be a rooted ordered tree derived from T by adding a new vertex v as the rightmost child of r_i. We can observe that each child tree of T is in $\{T + 0, T + 1, \cdots, T + a\}$, however not all trees in $\{T + 0, T + 1, \cdots, T + a\}$ are child trees of T. We need to check whether each $T + k$ is a child tree of T or not.

Lemma 3 *$T + k$ is a child tree of T iff for each i, $i = 1, 2, \cdots, k$, either $bro(r_i) = nil$ or $L(T(bro(r_i))) \geq L(T(r_i))$ holds in $T + k$.*

Proof: Since $T \in \mathcal{S}_n^1$, the condition $L(T(bro(v))) \geq L(T(v))$ in Lemma 1 is hold in $T + k$ if $bro(v) \neq nil$ and $v \notin RP(T)$. The claim checks all of these possible changes. ∎

If we generate each possible child tree $T + k$ of T and check whether it is actually a child tree or not based on the lemma above, then we need much running time. However, we can save the running time as follows.

We say that T is *active* at depth i if
(i) $RP(T)$ contains a vertex r_{i+1} having depth $i+1$,
(ii) $bro(r_{i+1}) \neq nil$,
(iii) $L(T(r_{i+1}))$ is a prefix of $L(T(bro(r_{i+1})))$.
Intuitively, if T is active at depth i, then we are copying subtree $T(r_{i+1})$ form $T(bro(r_{i+1}))$.

Note that if T is a path $(r_0, r_1, \cdots, r_a)$ then T is active at none of depth. For convenience, we imaginary construct a rooted tree consisting of a path $(x_1, x_2, \cdots, x_{n-1})$ and consider x_1 as the first child vertex of the root r_0 of T, then we regard that T is active at depth 0. Otherwise, if the last depth of $L(T)$ is k, T is active at depth $k-1$. Thus, T is active at least one depth. We define the *copy-depth* of T by the minimum k such that T is active at depth k.

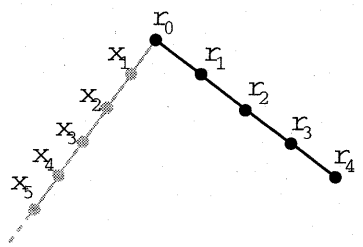

Figure 3: A path is active at depth 0.

Now we characterize the indices i satisfying that $T+i$ is a child of T. We consider the following three cases. Suppose that the copy-depth of T is k, and $RP(T) = (r_0, r_1, \cdots, r_a)$.
Case 1: T has n vertices, then T is a leaf of the family tree, and T has no child tree.

Case 2: If $L(T(s_{k+1})) = L(T(r_{k+1}))$ (Intuitively the copy is completed.)
The child trees of T are $T+0, T+1, \cdots, T+k$. Note that for each of $T+i$, $i = 0, 1, \cdots, k$, the left-heavy condition of Lemma 1 still holds. However, for each of $T+i$, $i = k+1, k+2, \cdots, a$, $L(T(bro(r_{k+1}))) < L(T(r_{k+1}))$ holds, thus it is not left-heavy. Since $L(T(bro(r_i))) < L(T(r_i))$ or $bro(r_i) = nil$ for any $i \leq k$, the copy-depth of $T+i$ is i for $i = 0, 1, \cdots, k-1$.

Case 3: If $L(T(bro(r_{k+1}))) \neq L(T(r_{k+1}))$ (Intuitively the copy is not completed yet.)
Let d be the depth of $L(T(bro(r_{k+1})))$ following to $L(T(r_{k+1}))$. Note that $L(T(r_{k+1}))$ is a prefix of $L(T(bro(r_{k+1})))$. We call the vertex giving the depth d the *copying vertex* of T. The child trees of T are $T+0, T+1, \cdots, T+(d-1)$. Note that for each of $T+d, T+(d+1), \cdots, T+a$, $L(T(bro(r_{k+1}))) < L(T(r_{k+1}))$ holds, so it is not left-heavy. The copy-depth of $T+i$ is i for $i = 0, 1, \cdots, d-2$, and the copy-depth of $T+i$ is k for $i = d-1$.

In this case we have to explain in details. We can observe that the copy-depth of $T+i$ is never greater than k, and $T+i$ is active at i. So the copy-depth of $T+i$ is somewhere between i and k.

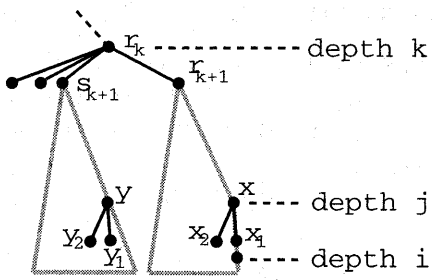

Figure 4: Illustration for Case 3.

Assume for the contradiction that the copy-depth of $T+i$ is $j < i$. Let $dep(x)$ be the last occurrence of depth j in $L(T+i)$. By the assumption $bro(x) \neq nil$. Let x_1 be the rightmost child of x. See Fig. 4. Let y and y_1 be the vertices in $T(bro(r_{k+1}))$ corresponding to x and x_1 in $T(r_{k+1})$. (Note that we are copying $T(r_{k+1})$ from $T(bro(r_{k+1}))$.) Now since $T \in \mathcal{S}_n^1$, $L(T(y_2)) \geq L(T(y_1))$ holds. By the choice of i, $L(T(y_1)) > L(T(x_1))$ holds. Since the copy-depth of T is k, $L(T(y_2)) = L(T(x_2))$. Thus, we have

$$L(T(x_2)) = L(T(y_2)) \geq L(T(y_1)) > L(T(x_1)).$$

Therefore, $L(T(x_1))$ is not the prefix of $L(T(x_2))$, and the copy-depth of $T+i$ is not j, a contradiction. Thus the copy-depth of $T+i$ is i for $i = k+1, k+2, \cdots, d-2$.

Based on the case analysis above we have the following algorithm. The algorithm inputs a left heavy embedded tree T, the copy depth k of T, the depth sequence $L(T)$, and the copying vertex v, and output all the descendants trees of T, thus by giving a rooted tree composed of one vertex to the algorithm, we can enumerate all the rooted trees of $\mathcal{S}_n^1$.

Procedure find-all-children$(T, L(T), k, v)$
1 **output** T by the difference from the previous output tree
2 **if** T has n vertices **then return** {Case 1}
3 **else if** {Case 3}
4 $u :=$ the vertex next to v in the preorder of T
5 **for** $i = 0$ **to** $dep(u) - 2$
6 **find-all-children**$(T+i, L(T)+(i+1), i, r_{i+1})$
7 **find-all-children**$(T+(d-1), L(T)+d, k, u)$
8 **End if**

We note that Case 2 is included in Case 3. An execution of the algorithm is shown in Fig. 5.

Theorem 1 *The algorithm uses $O(n)$ space and runs in $O(|\mathcal{S}_n^1|)$ time.*

Proof : Since we traverse the family tree and output each tree at every vertex, we can generate all rooted trees having at most n vertex. The algorithm takes $O(k)$ time to generate k recursive calls, and constant time for other parts. Thus, the algorithm takes constant time for each edge of the family tree, and the algorithm runs in $O(|\mathcal{S}_n^1|)$ time. The algorithm does not output each of entire tree but the difference from the previous output tree. It takes time proportional to the number of edges of the family tree traced after outputting the previous tree. Hence, the total computation time to output is $O(|\mathcal{S}_n^1|)$.

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

5 Generating Rooted Trees with n Vertices

Let $\mathcal{S}_n$ be the set of rooted trees with exactly n vertices. The algorithm we proposed can be used to enumerate all the rooted trees of $\mathcal{S}_n$, by outputting only them. In this section we prove that the number of rooted trees with n vertices is more than twice the number of rooted trees with $n-1$ vertices, i.e.,

$$|\mathcal{S}_n| \geq 2|\mathcal{S}_{n-1}|$$

This implies that

$$|\mathcal{S}_n| \geq |\mathcal{S}_{n-1}^1|,$$

thus the computation time of the algorithm is $O(|\mathcal{S}_n|)$.

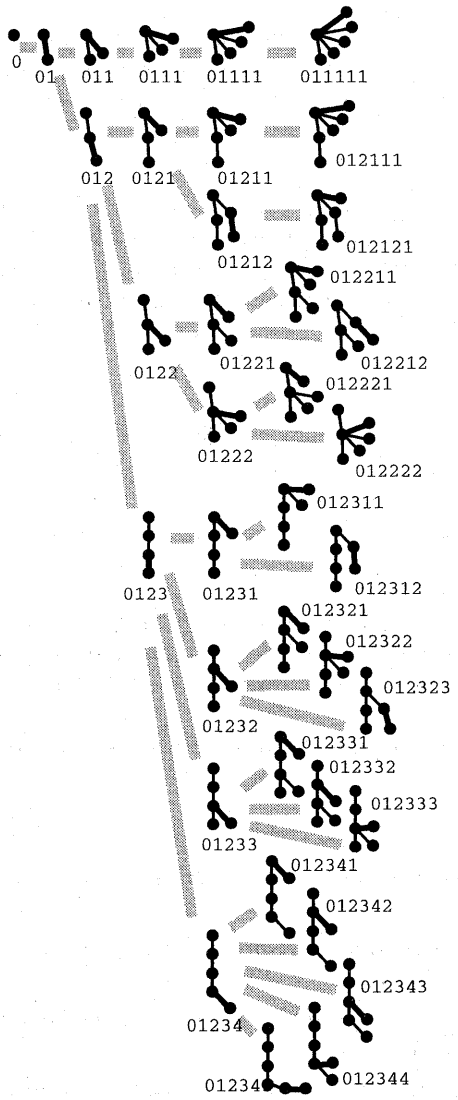

Figure 5: An execution of the algorithm.

Lemma 4

$$|\mathcal{S}_n| \geq 2|\mathcal{S}_{n-1}|$$

Proof: Let $\mathcal{S}'_n$ be the set of trees of $\mathcal{S}_n$ such that no child of the root is a leaf. For any tree $T \in \mathcal{S}_{n-1}$, we obtain a tree of $\mathcal{S}_n \setminus \mathcal{S}'_n$ by adding a vertex as the rightmost child of the root. Since no two trees of $\mathcal{S}_{n-1}$ generate the same rooted tree by this operation, we can see

$$|\mathcal{S}_n| - |\mathcal{S}'_n| \geq |\mathcal{S}_{n-1}|.$$

Suppose that the root of a tree $T \in \mathcal{S}_{n-1}$ has k children which are leaves (possibly $k = 0$). we obtain a tree of $\mathcal{S}_n$ by removing all these k leaves, and adding $k+1$ vertices as children of the leftmost leaf of T, where the leftmost leaf is the leaf any whose ancestor is the leftmost child of its parent. By this operation, no two trees of $\mathcal{S}_{n-1}$ generates the same tree, hence we obtain

$$|\mathcal{S}'_n| \geq |\mathcal{S}_{n-1}|.$$

Therefore,

$$|\mathcal{S}_n| \geq 2|\mathcal{S}_{n-1}|.$$

∎

6 Conclusion

In this paper, we propose a simple algorithm to enumerate all rooted trees with at most n vertices in constant time per rooted tree. The algorithm also enumerates all rooted trees with exactly n vertices in a constant time per rooted tree. We clarify a family tree of the rooted trees.

Can we generate efficiently all (rooted) trees with n vertices and with diameter d?

References

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

[B80] T. Beyer and S. M. Hedetniemi, *Constant time generation of rooted trees*, SIAM J. Comput., 9, (1980), pp.706-712.

[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.

[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.

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

[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.