Vol.2010-AL-131 No.6
2010/9/22

Enumerating All Rooted Trees Including k Leaves

Masanobu Ishikawa,[†1] Katsuhisa Yamanaka,[†2]
Yota Otachi[†3] and Shin-ichi Nakano[†1]

This paper presents an efficient algorithm to generate all (unordered) rooted trees with exactly n vertices including exactly k leaves. We design a simple tree structure among such trees, then by traversing the tree structure we generate all such trees in constant time per tree in the worst case. By repeatedly applying the algorithm for each $k = 1, 2, \ldots, n-1$, we can also generate all rooted trees with exactly n vertices.

1. Introduction

It is useful to have a complete list of objects for a particular class. Such a list can be used to search a counter-example for a hypothesis; to obtain the best object, among all the candidates, with respect to some criterion; or to experimentally measure the average performance of an algorithm for all possible inputs.

Several algorithms are known to generate all objects for a particular class without repetition [1, 2, 4, 8–16, 18–21]; Several textbooks have been published on the subject [3, 5–7, 17].

In this paper, we focus on (unordered) trees. Trees are fundamental models, frequently used in various fields such as searching for keys, modeling computations, and parsing a program, etc. Several enumeration algorithms for trees are already proposed [2, 4, 8, 10, 12, 13, 16, 18, 20].

A *rooted* tree refers to a tree with one designated "root" vertex. Note that no ordering is defined among the children of each vertex. Fig. 1(a) shows all (unordered) rooted trees with 5 vertices including 3 leaves. If we define an ordering among the children of each vertex, then the resulting tree is called an *ordered* tree. Fig. 1(b) shows all ordered trees with 5 vertices including 3 leaves.

†1 Department of Computer Science, Gunma University.
ishikawa@nakano-lab.cs.gunma-u.ac.jp, nakano@cs.gunma-u.ac.jp
†2 Graduate School of Information Systems, University of Electro-communications.
yamanaka@is.uec.ac.jp
†3 Graduate School of Information Sciences, Tohoku University.
otachi@dais.is.tohoku.ac.jp

Some results for enumerating all trees are already known. Beyer and Hedetniemi [2] gave an algorithm to generate all rooted trees with n vertices. Their algorithm is the first one to generate all rooted trees in constant time per tree on an average. Li and Ruskey [8] also gave an algorithm to generate all rooted trees, and claimed that it was easily modified to generate restricted classes of rooted trees. The possible restrictions include (1) upper bound on the number of children and (2) lower and upper bounds on the height of a rooted tree.

Due to the absence of a root vertex, the generation of nonisomorphic free trees is a more difficult problem. Wright *et al.* [18] and Li and Ruskey [8] presented algorithms to generate all free trees in $O(1)$ time per tree on an average. Nakano and Uno [12] improved the running time to $O(1)$ time in the worst case. Also, they generalized the algorithm to generate all "colored" trees [13], where a colored tree refers to a tree whose each vertex has a color.

An *ordered* tree is a rooted tree with a left-to-right ordering specified for the children of each vertex. An algorithm to generate all ordered trees has been proposed by Nakano [10]. He also provided a method to generate nonrooted ordered trees in [10]. Sawada [16] handled the enumeration problem for a similar but another class of trees, called *circular*-ordered trees. A circular-ordered tree refers to a rooted tree with a circular ordering specified for the children of each vertex. Sawada [16] presented algorithms to generate circular-ordered trees and nonrooted ones in $O(1)$ time per tree on an average.

Pallo [14] as well as Nakano [10] presented an algorithm to generate all ordered trees with n vertices including k leaves in $O(n-k)$ time per tree on an average. Yamanaka *et al.* [20] improved the running time to constant per tree in the worst case.

In this paper, we design a simple algorithm to generate all rooted trees with exactly n vertices including k leaves. Due to the absence of orderings of children, this problem is more difficult than the one with orderings. Our algorithm generates each such tree in constant time in the worst case.

If we modify the algorithm in [20] so that it outputs only "left-heavy" trees

© 2010 Information Processing Society of Japan

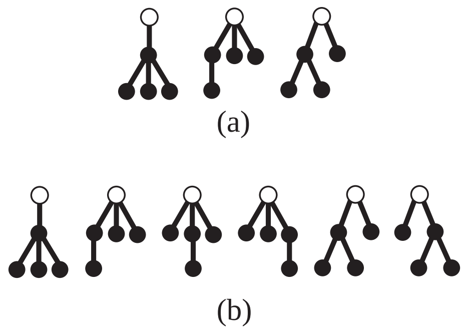
Fig. 1 All (a) (unordered) rooted trees and (b) ordered rooted trees with 5 vertices including 3 leaves.

(which will be defined in Section 2), then we can generate all rooted trees with exactly n vertices including exactly k leaves. However it may take much time to check whether each generated tree is left-heavy or not, and resulting algorithm enumerates all such trees in $O(nk)$ time for each. We will explain the detail in Section 3.

The concept of our algorithms is as follows. We first define a tree structure among trees, called family tree, in which each vertex corresponds to each tree to be enumerated. By traversing the tree structure we can enumerate all trees. Based on this concept several enumerating algorithms are designed [11, 19, 20]. However to enumerate all rooted trees with exactly n vertices including k leaves, we have to carefully design a new tree structure among them, and it is not so easy.

The rest of the paper is organized as follows. Section 2 provides some definitions; Section 3 introduces a left-heavy embedding of a rooted tree; Section 4 defines a family tree among rooted trees with n vertices including k leaves; and Section 5 presents an algorithm to generate all such trees. Finally Section 6 concludes the study.

2. Definitions

Let G be a connected graph with n vertices. A *path* is a sequence of distinct vertices $(v_1, v_2, \ldots, v_p)$ such that (v_{i-1}, v_i) is an edge for $i = 2, 3, \ldots, p$. The

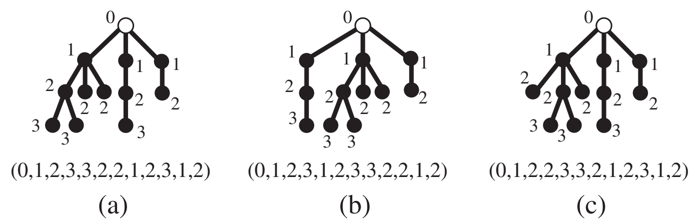
Fig. 2 Examples of the depth sequences.

length of a path is the number of edges in the path.

A *tree* is a connected graph with no cycle. A *rooted* tree is a tree with a vertex r chosen as its *root*. For each vertex v in a rooted tree, let $UP(v)$ be the unique path from v to r. If $UP(v)$ has exactly p edges then we say that the *depth* of v is p and write $dep(v) = p$. 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 r and the ancestors of r are not defined. We say 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. The *height* of a vertex v, denoted by $height(v)$, is the number of edges on the longest path from v to a descendant of v. A *leaf* is a vertex having no child. If a vertex is not a leaf, then it is called an *inner* vertex.

An *ordered tree* is a rooted tree in which a left-to-right ordering is specified for the children of each vertex. Let $C(v) = (c_1, c_2, \ldots, c_{d(v)})$ be the left-to-right ordering of the children of v from left to right, where $d(v)$ is the number of children of v. We call it the *child sequence* of v. A vertex c_i is the *next sibling* of c_{i-1}.

Let T be an ordered tree with n vertices, and $(v_1, v_2, \ldots, v_n)$ be the list of the vertices of T in preorder. Then, the sequence of the depth $L(T) = (dep(v_1), dep(v_2), \ldots, dep(v_n))$ is called the *depth sequence* of T. See Fig. 2 for examples. The three 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 $L(T_1) = (a_1, a_2, \ldots, a_n)$ and $L(T_2) = (b_1, b_2, \ldots, b_m)$ be their depth sequences. If either (1) $a_i = b_i$ for each $i = 1, 2, \ldots, j-1$ and $a_j > b_j$, or (2) $a_i = b_i$ for each $i = 1, 2, \ldots, m$ 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. Left-heavy Embedding of Rooted Trees

In this section we define the left-heavy embedding [12, 13] of a rooted tree.

Given a rooted tree T, by choosing some left-to-right ordering among the children of each vertex we can generate many ordered trees. The heaviest ordered tree among them is called the *left-heavy embedding* of T, and if ordered tree is the left-heavy embedding of some rooted tree then it is called *a left-heavy ordered tree*. We can observe there is a one-to-one mapping between the set of rooted trees and the set of left-heavy ordered trees. Thus if an algorithm can generate all left-heavy ordered trees then it also generates all rooted trees.

Let $S_{n,k}$ be a set of all left-heavy ordered trees with exactly n vertices including exactly k leaves. If we generate all ordered trees in $S_{n,k}$, then by ignoring the left-to-right orderings we can also generate all rooted trees with exactly n vertices including exactly k leaves.

We denote by $T(v)$ the subtree of an ordered tree T rooted at v. We have the following lemma.

Lemma 3.1 ([12]) An ordered tree T is a left-heavy ordered tree if and only if $L(T(c_{i-1})) \geq L(T(c_i))$ holds for every pair of a vertex c_{i-1} and its next sibling c_i.

From Lemma 3.1, we can check whether T is a left-heavy ordered tree or not, as follows. For every pair of a vertex c_{i-1} and its next sibling c_i, we traverse $T(c_{i-1})$ and $T(c_i)$ with depth first manner, then we check whether $L(T(c_{i-1})) \geq L(T(c_i))$ holds. Observe that for any c_{i-1} and its next sibling c_i, $L(T(c_{i-1})) \geq L(T(c_i))$ can be checked in $O(n)$ time. Furthermore, it is not difficult to see that the number of pairs (c_{i-1}, c_i) is $O(k)$. Thus, this naive method can be done in $O(nk)$ time.

Combining this method and Yamanaka's enumeration [20], one can enumerate all left-heavy ordered trees with n vertices including k leaves in $O(nk)$ time for each. Therefore it seems to be difficult to accomplish efficient, say $O(1)$ time, enumeration following this approach.

In this paper, we define a new tree structure among left-heavy ordered trees with n vertices including k leaves, then propose an efficient algorithm to enumerate such trees.

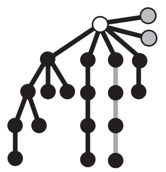
Fig. 3 Examples for non-branching paths and root leaves.

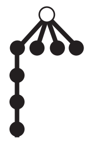
Fig. 4 Root tree $R_{8,4}$

4. Family Tree

In this section, we define a tree structure among the trees in $S_{n,k}$, in which each vertex corresponds to a tree in $S_{n,k}$ and each edge corresponds to a relation between two trees in $S_{n,k}$.

If either $k = 1$ or $k = n-1$, then $|S_{n,k}| = 1$ and its enumeration is trivial. So we assume $1 < k < n-1$.

We need some definitions.

Let T be an ordered tree, and r its root. If a leaf v is a child of the root then v is called *a root leaf*. A path P in T is called *a non-branching path* if (1) P starts at a child of r, (2) P ends at a leaf of T, and (3) all internal vertices of P has exactly one child in T. Note that P may be a root leaf. If T has a non-branching path, then the rightmost path among the longest non-branching paths is denoted by $P_L(T)$. See Fig. 3. The root leaves are depicted by gray circles and $P_L(T)$ is drawn as gray lines.

$R_{n,k}$ is the ordered tree consisting of a path with $n-k$ edges and $k-1$ leaves attaching at the root so that those leaves appear on the right of the path. See Fig. 4 for an example. $R_{n,k} \in S_{n,k}$ holds.

We now define the *parent tree* $P(T)$ for each $T \in S_{n,k} \setminus \{R_{n,k}\}$ by the following two cases. Let T' be the tree obtained from T by removing (1) all root leaves and

© 2010 Information Processing Society of Japan

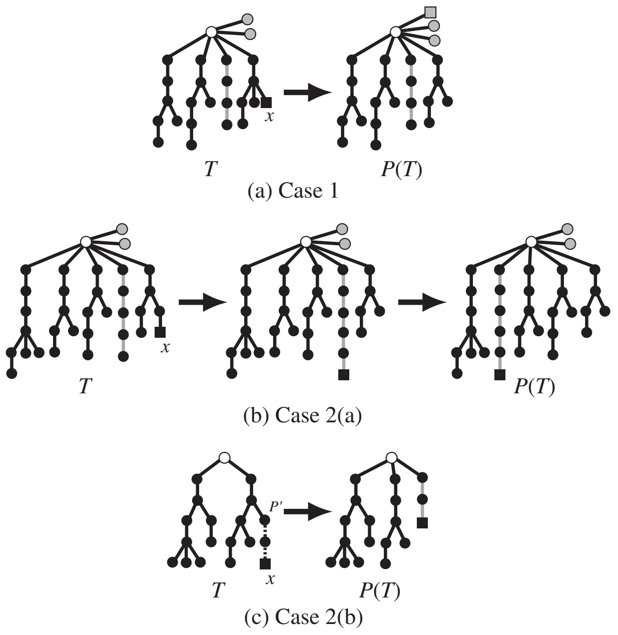

Fig. 5 Illustration for the parents.

(2) $P_L(T)$ if T has one or more non-branching paths. Let x be the last vertex of T' in preorder. Since $T \neq R_{n,k}$ such x always exists.

Case 1: x has one or more siblings.

$P(T)$ is the tree obtained from T by (1) removing x, then (2) attaching a new root leaf as the rightmost child of r. See Fig. 5(a).

Case 2: x has no sibling.

We have the following two subcases

Case 2(a): T has $P_L(T)$.

$P(T)$ is the tree obtained from T by (1) removing x, (2) attaching a new leaf to the end vertex of $P_L(T)$, then (3) re-embedding the resulting tree to be left-heavy. See Fig. 5(b). The extended non-branching path may move to the left. Note that $P_L(P(T))$ has one or more edges, although $P_L(T)$ may be a root leaf.

Case 2(b): T has no $P_L(T)$. Thus T has no non-branching path.

Let $P = (v_0 = r, v_1, \ldots, v_q = x)$ be the path in T starting at r and ending at x. Let $P' = (v_p, v_{p+1}, \ldots, v_q)$ be the subpath of P such that $d(v_{p-1}) \geq 2$, $d(v_p) = d(v_{p+1}) = \cdots = d(v_{q-1}) = 1$. $P(T)$ is the tree obtained from T by (1) removing P', then (2) appending P' to r so that P' becomes the rightmost non-branching path. See Fig. 5(c). Note that $P(T)$ has exactly one or two non-branching paths. If $P(T)$ has exactly one non-branching path P, then P has one or more edges, and the starting vertex of P is the rightmost child of the root. Otherwise $P(T)$ has exactly two non-branching path P_1, P_2, then P_1 and P_2 have one or more edges respectively, and the starting vertices of P_1 and P_2 are the rightmost and the second rightmost child of the root.

We say T is *a child tree* of $P(T)$. If $P(T)$ is derived in Case 1 then T is called a Type 1 child. Similarly if $P(T)$ is derived in Case 2(a) or 2(b) then it is called a Type 2(a) or 2(b) child, respectively.

We have the following lemma.

Lemma 4.1 For any $T \in S_{n,k} \setminus \{R_{n,k}\}$, $P(T) \in S_{n,k}$ holds.

Proof. In each case $P(T)$ has n vertices including k leaves, and $P(T)$ is the left-heavy embedding. $\square$

By repeatedly finding the parent tree of the derived tree, we can have the unique sequence $T, P(T), P(P(T)), \ldots$ of trees in $S_{n,k}$. We have the following lemma.

Lemma 4.2 For any $T \in S_{n,k}$ the sequence $T, P(T), P(P(T)), \ldots$ always ends up with $R_{n,k}$.

Proof. Let T be a tree in $S_{n,k}$, and $nb(T)$ be the number of edges in non-branching paths in T. We can observe $nb(P(T)) > nb(T)$ always holds, and for any $T \in S_{n,k} \setminus \{R_{n,k}\}$, $nb(R_{n,k}) > nb(T)$ holds.

Therefore by repeatedly finding the parent tree of derived tree, we eventually obtain $R_{n,k}$ on which $nb(T)$ is maximized. $\square$

By merging those sequences of trees in $S_{n,k}$, we can have the *family tree*, denoted by $T_{n,k}$, of $S_{n,k}$, in which each vertex corresponds to a tree in $S_{n,k}$ and

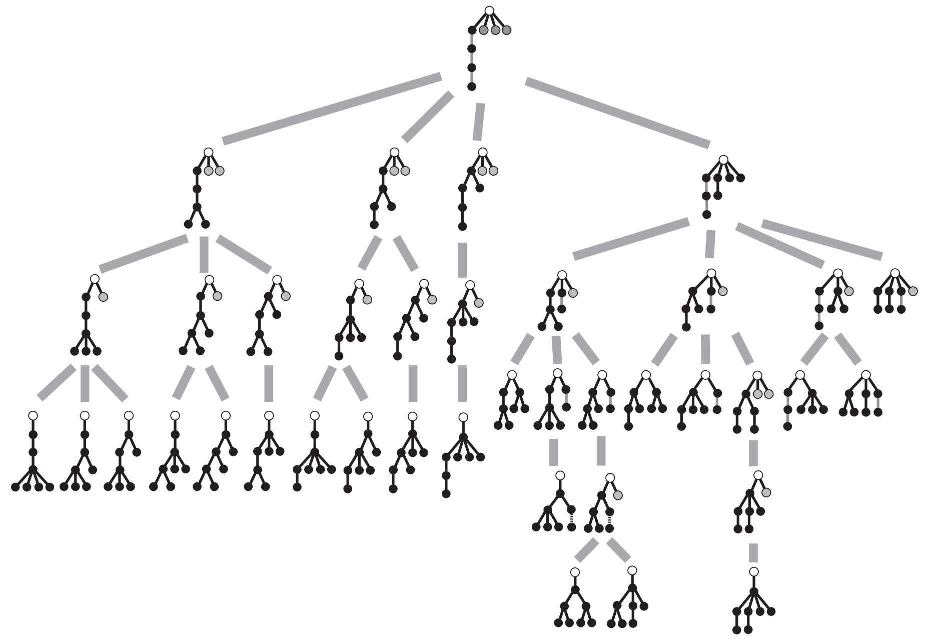

Fig. 6 Family tree $T_{8,4}$

each edge connects a tree T and its parent $P(T)$. See Fig. 6.

5. Enumerating All Child Trees

If we can enumerate all child trees of a given tree in the family tree $T_{n,k}$, then by applying the algorithm recursively we can enumerate all trees in $S_{n,k}$. We now give an algorithm to enumerate all child trees of a given tree in $S_{n,k}$.

For an ordered tree T with root r and its vertex v, we define trees $T_1(v)$, $T_{2a}(v)$, $T_{2b}(v)$ for some v. Every child tree of $T \in S_{n,k}$ is either $T_1(v)$, $T_{2a}(v)$, $T_{2b}(v)$, however the reverse is not always true. Later we classify those trees into the child trees and others.

Let T' be the tree obtained from T by removing (1) all root leaves and (2) $P_L(T)$ if T has $P_L(T)$. The *active path* $AP(T) = (a_1, a_2, \ldots, a_p)$ of T is the path in T' such that a_1 is the rightmost child of the root, and a_i is the rightmost child of a_{i-1} for each $i = 2, 3, \ldots, p$.

If T has a root leaf, then for each $i = 1, 2, \ldots, p-1$, $T_1(a_i)$ is the tree obtained from T by (1) removing the rightmost root leaf, then (2) attaching a new leaf to a_i as the rightmost child. See Fig. 7(a).

If T has a root leaf and $P_L(T)$, where $P_L(T) = (b_1, b_2, \ldots, b_q)$, then for each $i = 1, 2, \ldots, q-1$, $T_1(b_i)$ is the tree obtained from T by (1) removing the rightmost root leaf, then (2) attaching a new leaf to b_i as the rightmost child. See Fig. 7(a).

If T has $P_L(T)$ and $P_L(T)$ has one or more edges, where $P_L(T) = (b_1, b_2, \ldots, b_q)$, then $T_{2a}(a_p)$ is the tree obtained from T by (1) removing b_q, (2) attaching a new leaf to a_p, then (3) re-embedding the resulting tree to be left-heavy. See Fig. 7(b).

If (1) T has exactly one non-branching path, say $P_L(T) = (b_1, b_2, \ldots, b_q)$, (2) $P_L(T)$ has one or more edges, and (3) b_1 is the rightmost child of the root, then for each $i = 1, 2, \ldots, p-1$, $T_{2b}(a_i)$ is the tree obtained from T by (1) removing the $P_L(T) = (b_1, b_2, \ldots, b_q)$, then (2) attaching the $P_L(T)$ to a_i so that b_1 is the rightmost child of a_i. See Fig. 7(c).

We assume that T has exactly two non-branching paths such that they contain the rightmost child of the root and the second rightmost child, respectively. Thus the two non-branching path are $AP(T) = (a_1, a_2, \ldots, a_p)$ and $P_L(T) = (b_1, b_2, \ldots, b_q)$. Then, for each $i = 1, 2, \ldots, q-1$ $T_{2b}(b_i)$ is the tree obtained from T by (1) removing the $AP(T)$, then (2) attaching the $AP(T)$ to b_i so that a_1 is the rightmost child of b_i. See Fig. 7(c).

Now we classify those trees into the child trees of T and others.

Type 1 child tree:

If T has no root leaf, $T_1(v)$ is not defined. Assume otherwise. For each $i = 2, 3, \ldots, p-1$, $T_1(a_i)$ is a Type 1 child of T if and only if $T_1(a_i)$ is a left-heavy ordered tree. For each $i = 1, 2, \ldots, q-1$, $T_1(b_i)$ is a Type 1 child tree of T if and only if (1) $P_L(T)$ has one or more edges, (2) b_1 is the rightmost child of the root vertex excluding root leaves, and (3) $T_1(b_i)$ is the left-heavy ordered tree.

Type 2(a) child tree:

If T has no $P_L(T)$ or $P_L(T)$ is a root leaf then $T_{2a}(a_p)$ is not defined. Assume

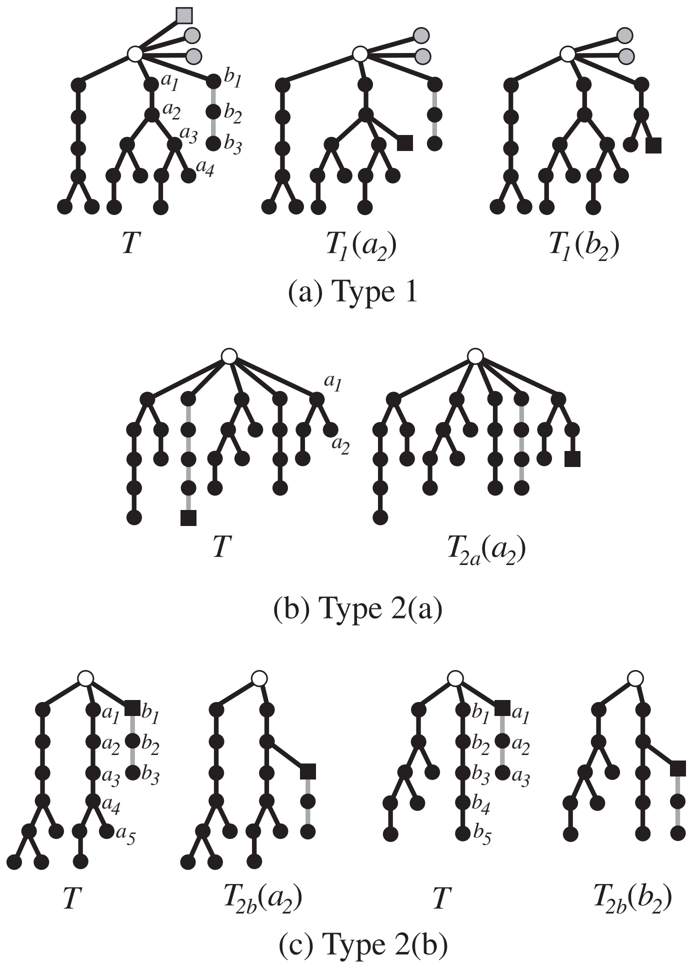

Fig. 7 Examples of child trees.

otherwise. $T_{2a}(a_p)$ is a Type 2(a) child of T if and only if $T_{2a}(a_p)$ is a left-heavy ordered tree.

Type 2(b) child tree:

We assume that T has no root leaf. If T has exactly one non-branching path such that it has one or more edges and b_1 on $P_L(T)$ is the rightmost child of the root, then for each $i = 1, 2, \ldots, p-q$, $T_{2b}(a_i)$ is a Type 2(b) child tree, since they are left-heavy. On the other hand, for each $i = p-q+1, p-q+2, \ldots, q$, $T_{2b}(a_i)$ is not left-heavy, so it is not a child tree of T.

If T has exactly two non-branching paths such that they respectively contain the rightmost child of the root and the second rightmost child, then for each $i = 1, 2, \ldots, q-b$, $T_{2b}(b_i)$ is a Type 2(b) child tree. Note that $P_L(T)$ is longer than $AP(T)$.

Based on the case analysis above we can enumerate all child trees of T by **Algorithm** 1. By recursively applying **Algorithm 1** from the root tree $R_{n,k}$, we can enumerate all trees in $S_{n,k}$.

Now we analyze running time of **Algorithm 1**.

With a help of suitable data structure in [12] we can check whether each of $T_1(a_i)$, $T_1(b_i)$ and $T_{2a}(a_p)$ is left-heavy or not in lines 4, 9 and 20, respectively in constant time.

We also maintain $P_L(T)$ and $AP(T)$ as a list.

To construct $T_{2a}(a_p)$ we need to re-embed the resulting tree to be left-heavy, in which $P_L(T)$ may move to right to a suitable place. To accomplish this in constant time we store the current tree T as the subtrees of T rooted at the children of the root as follows. We store the subtrees having the same height in a list, in which each subtree T_c appear in the order of $L(T_c)$. Also, we prepare an array $A[1..n]$ of pointers, in which $A[i]$ is a pointer to the list of the subtrees with height i. Thus in constant time we can move $P_L(T)$ to the end of the list having one less height. Also, we can check whether T has a root leaf or not in constant time.

We can compute each $T_{2b}(a_i)$ in constant time for each tree.

Thus we can enumerate all child trees in constant time for each tree.

Theorem 5.1 After constructing and outputting the root tree $R_{n,k}$ in $S_{n,k}$, one can enumerate all trees in $S_{n,k}$ in constant time per tree on an average.

By Theorem 5.1, our algorithm generates each tree in $S_{n,k}$ in constant time "on an average." However it may have to return from the deep recursive calls

© 2010 Information Processing Society of Japan

Algorithm 1: `find-all-children`(T)

```
1  Let $AP(T) = (a_1, a_2, \ldots, a_p)$ be the active path of $T$ and
   $P_L(T) = (b_1, b_2, \ldots, b_q)$ if $T$ has $P_L(T)$.
2  if $T$ has one or more root leaves then
3      for each $i = 1, 2, \ldots, p-1$ do
4          if $T_1(a_i)$ is a left-heavy ordered tree then
5              find-all-children($T_1(a_i)$)
6          else break
7      if (1) $P_L(T)$ has one or more edges, (2) $b_1$ is the rightmost child of the
       root excluding root leaves then
8          for each $i = 1, 2, \ldots, q-1$ do
9              if $T_1(b_i)$ is a left-heavy embedding then
               find-all-children($T_1(b_i)$)
10             else break
11 else
12     if $T$ has exactly one non-branching path, that is $P_L(T)$, such that $P_L(T)$
       has one or more edges, and $b_1$ on $P_L(T)$ is the rightmost child of the root
       then
13         for each $i = 1, 2, \ldots, (p-q)$ do
14             find-all-children($T_{2b}(a_i)$)
15     else
16         if $T$ has exactly two non-branching paths, each of them has one or
           more edges, and $a_1$ and $b_1$ are the rightmost and the second rightmost
           child of the root then
17             for each $i = 1, 2, \ldots, (q-p)$ do
18                 find-all-children($T_{2b}(b_i)$)
19 if $T$ has $P_L(T)$ and $P_L(T)$ has one or more edges then
20     if $T_{2a}(a_p)$ is a left-heavy ordered tree then
21         find-all-children($T_{2a}(a_p)$)
```

without outputting any tree in $S_{n,k}$ after generating a tree corresponding to the rightmost leaf of a large subtree in the family tree. Therefore the next tree in $S_{n,k}$ cannot be generated in constant time.

This delay can be canceled [6, 12] by outputting each tree before its children if the tree corresponds to a vertex at odd depth, and after its children otherwise.

Now we have the following corollary.

Corollary 5.2 After constructing and outputting the root tree $R_{n,k}$, one can enumerate all trees in $S_{n,k}$ in constant time per tree in the worst case.

For each $k = 1, 2, \ldots, n-1$, $R_{n,k}$ is constructed from $R_{n,k-1}$ in constant time by (1) removing the leaf on the longest non-branching path, then (2) attaching a new leaf as the rightmost child of the root. Thus by repeatedly applying the algorithm in Corollary 5.2 for each $k = 1, 2, \ldots, n-1$, we can enumerate all rooted trees with exactly n vertices. We have the following theorem.

Theorem 5.3 After constructing and outputting the root tree $R_{n,1}$, one can enumerate all rooted trees with exactly n vertices in constant time for each in the worst case.

6. Conclusion

In this paper we have given an efficient algorithm to generate all (unordered) rooted trees with exactly n vertices including exactly k leaves. Our algorithm generates each such tree in $O(1)$ time, while a modified version of known algorithms generates each such tree in $O(nk)$ time.

By repeatedly applying our algorithm for $k = 1, 2, \ldots, n-1$, we can also generate all rooted trees with exactly n vertices in constant time for each.

References

[1] D.Avis and K.Fukuda. Reverse search for enumeration. *Discrete Appl. Math.*, 65(1-3):21–46, 1996.
[2] T.Beyer and S.M. Hedetniemi. Constant time generation of rooted trees. *SIAM J. Comput.*, 9(4):706–712, 1980.
[3] L.A. Goldberg. *Efficient algorithms for listing combinatorial structures*. Cambridge University Press, New York, 1993.
[4] Y.Kikuchi, H.Tanaka, S.Nakano, and Y.Shibata. How to obtain the complete list of caterpillars. *Proc. The 9th Annual International Computing and Combinatorics*

© 2010 Information Processing Society of Japan

Conference, (COCOON 2003), LNCS 2697:329–338, 2003.

[5] D.E. Knuth. *The art of computer programming*, volume 4, fascicle 2, generating all tuples and permutations. Addison-Wesley, 2005.

[6] D.E. Knuth. *The art of computer programming*, volume 4, fascicle 4, generating all trees, history of combinatorial generation. Addison-Wesley, 2006.

[7] D.L. Kreher and D.R. Stinson. *Combinatorial algorithms.* CRC Press, Boca Raton, 1998.

[8] G. Li and F. Ruskey. The advantages of forward thinking in generating rooted and free trees. *Proc. 10th Annual ACM-SIAM Symp. on Discrete Algorithms, (SODA1999)*, 939–940, 1999.

[9] B.D. McKay. Isomorph-free exhaustive generation. *J. Algorithms*, 26(2):306–324, 1998.

[10] S.Nakano. Efficient generation of plane trees. *Inf. Process. Lett.*, 84(3):167–172, 2002.

[11] S. Nakano. Efficient generation of triconnected plane triangulations. *Comput. Geom. Theory and Appl.*, 27(2):109–122, 2004.

[12] S.Nakano and T.Uno. Constant time generation of trees with specified diameter. *Proc. the 30th Workshop on Graph-Theoretic Concepts in Computer Science, (WG 2004)*, LNCS 3353:33–45, 2004.

[13] S. Nakano and T. Uno. Generating colored trees. *Proc. the 31th Workshop on Graph-Theoretic Concepts in Computer Science, (WG 2005)*, LNCS 3787:249–260, 2005.

[14] J.Pallo. Generating trees with n nodes and m leaves. *International Journal of Computer Mathematics,*, 21(2):133–144, 1987.

[15] R.C. Read. Every one a winner or how to avoid isomorphism search. *Annuals of Discrete Mathematics*, 2:107–120, 1978.

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

[17] H.S. Wilf. *Combinatorial algorithms: An update.* SIAM, 1989.

[18] R.A. Wright, B.Richmond, A.Odlyzko, and B.D. McKay. Constant time generation of free trees. *SIAM J. Comput.*, 15(2):540–548, 1986.

[19] K.Yamanaka and S.Nakano. Listing all plane graphs. *Journal of Graph Algorithms and Its Applications*, 13(1):5–18, 2009.

[20] K. Yamanaka, Y. Otachi, and S. Nakano. Efficient enumeration of ordered trees with k leaves. *The 3rd International Workshop on Algorithms and Computation, (WALCOM 2009)*, LNCS 5431:141–150, 2009.

[21] S.Zaks and D.Richards. Generating trees and other combinatorial objects lexicographically. *SIAM J. Comput.*, 8(1):73–81, 1979.

© 2010 Information Processing Society of Japan