

Enumerating Ordered Trees Lexicographically

M.C.ER

Department of Computer Science, University of Western Australia, Nedlands, W.A. 6009, Australia

A 1-1 mapping between the set of extended ordered trees with n internal nodes and the set of feasible binary bit-patterns with $2n$ bits is established. By manipulating the feasible bit-patterns, the set of ordered trees with n nodes can be enumerated lexicographically. The ranking and unranking functions are also described. It has been shown that the bit-pattern representation of ordered trees leads to simple construction and easy understanding of the enumerating, ranking and unranking algorithms.

1. INTRODUCTION

The problem of enumerating all ordered trees has attracted a great deal of attention in the recent literature.¹⁻⁹ Knott³ establishes a 1-1 correspondence between the ordered trees and the tree permutations using the in-order labelling and pre-order traversal of ordered trees. To generate successive ordered trees, it suffices to apply the unranking function to the corresponding successive natural numbers. Ruskey and Hu⁷ encode the ordered trees using the levels at which the leaves appear; the enumeration of ordered trees turns out to be a transformation of those strings of digits to their successors. Rotem and Varol⁶ show a 1-1 correspondence between the ordered trees and the stack-sortable permutations which can be represented as ballot sequences. These ballot sequences are then used for enumerating the ordered trees. Proskurowski⁴ generates the ordered trees by expanding the leaves of extended ordered trees. Solomon and Finkel⁸ give a detailed description of their enumerating algorithm based on the definition of (natural) ordered trees, without coding. Zaks⁹ proves the 1-1 correspondence between the ordered trees and the z -sequences. A successor function applied to the z -sequences is used for generating the ordered trees.

Of these great many methods, some methods are rather ad hoc. Predictably, their algorithms are very complex. The simplest idea of encoding the ordered trees as binary bit-patterns has not been fully exploited, though de Bruijn and Morselt,¹ Er,² Proskurowski,⁴ Read⁵ and Zaks⁴ have touched on it. We shall show in what follows that the binary bit-pattern representation of the extended ordered tree indeed simplifies the enumerating function.

Algorithms constructed from this enumerating function are simple and efficient. Furthermore, the ranking and unranking functions are also detailed. Algorithms implementing the ranking and unranking functions turn out to be shorter and simpler compared with others, owing to the conceptual simplicity of the encoding scheme of extended ordered trees.

2. DEFINITIONS AND NOTATIONS

There are many ways of permuting the ordered trees. The two common ones are natural and local orderings.

Let T be an ordered tree. Then by definition, T is either empty, or it has a node linking to two subtrees, each of which is an ordered tree. We use $\text{root}(T)$, $\text{left}(T)$ and $\text{right}(T)$ to indicate the root, left and right subtrees of

T respectively. Furthermore, the number of nodes in an ordered tree is denoted by $|T|$.

We now define the two commonly used orderings of trees.

Definition 1 (natural ordering)

Given two ordered trees T and T' , we define that $T < T'$ if

- (1) $|T| < |T'|$, or
- (2) $|T| = |T'|$ and $\text{left}(T) < \text{left}(T')$, or
- (3) $|T| = |T'|$ and $\text{left}(T) = \text{left}(T')$ and $\text{right}(T) < \text{right}(T')$.

Definition 2 (local ordering)

Given two ordered trees T and T' , we define that $T < T'$ if

- (1) T is empty and T' is not, or
- (2) both T and T' are not empty, and $\text{left}(T) < \text{left}(T')$, or
- (3) both T and T' are not empty, and $\text{left}(T) = \text{left}(T')$ and $\text{right}(T) < \text{right}(T')$.

These two orderings differ sharply in that the natural ordering takes a global view of the tree structures whereas the local ordering takes a local view of the tree structures. The permutations of ordered trees according to these two orderings will therefore be different. We shall deal with an enumeration of ordered trees according to the local ordering exclusively in this paper. By lexicographical ordering, we mean local ordering.

Let $T(n)$ be the set of ordered trees with n nodes, i.e.

$$T(n) = \{T : |T| = n\}.$$

Let $|T(n)|$ denote the number of distinct ordered trees with n nodes. It is well known that:

$$|T(n)| = C_n = \frac{1}{n+1} \binom{2n}{n}$$

where C_n is the n th Catalan number.¹⁰

Suppose $T(n) = \{T_1, T_2, \dots, T_{C_n}\}$, and its elements are arranged according to the lexicographical order. Then $T_i < T_j$ when $i < j$. A lexicographical listing of $T(4)$ can be found in Appendix A.

Given an ordered tree with n nodes, T , it can be converted to an *extended* ordered tree \hat{T} by adding another $(n+1)$ leaves to T (see Refs 5 and 10). An extended ordered tree can be encoded as a binary bit-pattern using the usual technique: representing an

internal node as a 1 and a leaf as a 0 in pre-order traversal. The resulting bit-pattern comprises $(2n+1)$ bits. As the last bit of every encoding bit-pattern is a 0, it can therefore be ignored without affecting the representation. We use $B_n = (b_1 b_2 b_3 \dots b_{2n})$, where b_i is either 1 or 0, to denote such a bit-pattern. B_n is said to possess the *dominating property* if the number of 1s is not less than the number of 0s at any state while scanning from b_1 to b_{2n} . Furthermore, B_n is said to be *feasible* if and only if it has the dominating property and the numbers of 1s and 0s in B_n are equal. The set of extended ordered trees with n internal nodes and the set of feasible binary bit-patterns with n 1s are denoted as $\hat{T}(n)$ and $B(n)$ respectively.

3. ENUMERATION

It is obvious that the encoding of an extended ordered tree always yields a feasible B_n . Thus the mapping between the set of extended ordered trees with n internal nodes, $\hat{T}(n)$, and the set of feasible binary bit-patterns, $B(n)$, is 1-1. As the mapping between $T(n)$ and $\hat{T}(n)$ is clearly 1-1, therefore, the mapping between $T(n)$ and $B(n)$ is also 1-1. The following theorem is simple to prove but is important for our enumerating, ranking and unranking algorithms to function.

Theorem 0

The lexicographical order of $T(n)$ is maintained in the lexicographical order of $B(n)$. In other words, the mapping between $T(n)$ and $B(n)$ is isotone.

Our strategy of enumerating all ordered trees takes advantage of this isotone property: instead of generating all ordered trees directly, $B(n)$ is enumerated in the lexicographical order by manipulating the bit-pattern.

Now we describe an enumerating algorithm for generating $B(n)$. This algorithm is based on a grid traversal approach. Let $T(x, y)$ be a point at coordinates (x, y) in a grid. $B(n)$ can be generated lexicographically by calling the following procedure as GenOrdTrees($n, 0$).

procedure GenOrdTrees(x, y :integer);

{ This procedure generates bit-patterns consisting of x 1s and $(x+y)$ 0s in the lexicographical order. To generate $B(n)$ lexicographically, it should be activated as GenOrdTrees($n, 0$). }

begin

if $x = 0$ **and** $y = 0$ **then** PrintBitPattern ()

else begin

if $y \neq 0$ **then begin**

$b_{2n-(2x+y)+1} := 0$;

GenOrdTrees($x, y-1$)

end;

if $x \neq 0$ **then begin**

$b_{2n-(2x+y)+1} := 1$;

GenOrdTrees($x-1, y+1$)

end

end

end { GenOrdTrees };

Note that the procedure PrintBitPattern simply prints a complete bit-pattern when it is activated.

GenOrdTrees can be seen as a grid traversal algorithm operating on a grid shown in Fig. 1. Viewing the algorithm in this light, the problem of enumerating the ordered trees lexicographically turns out to be equivalent

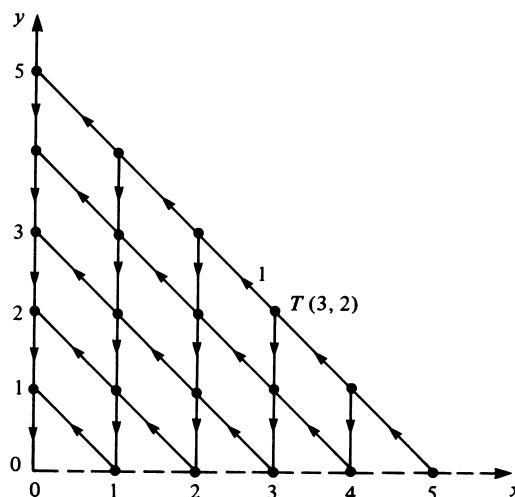


Fig. 1.

to the problem of generating systematically all paths between $(n, 0)$ and $(0, 0)$.

The parameters of GenOrdTrees are coordinates in the grid, and they have the following meaning: x regulates the number of 1s yet to be generated in B_n , and y controls the number of 0s that can be generated in succession beyond the state concerned.

GenOrdTrees is obviously convergent, because all paths from (x, y) are oriented towards $(0, 0)$ and bounded by the x and y axes.

We can prove that the procedure GenOrdTrees is correct. Firstly, we show that exactly n 1s and n 0s are generated in B_n when it is called with parameters $(n, 0)$. There is only one place in the procedure a 1 may be generated, and when it does, x is decremented by one. Therefore the total number of 1s generated in B_n is n . Furthermore, from the structure of the algorithm, the total number of 0s in the tail of B_n following bit $b_{2n-(2x+y)}$ is $(x+y)$. Thus, by induction, the total number of 0s in B_n is n . Secondly, we show that a B_n so generated is feasible. Starting from GenOrdTrees($n, 0$), a 0 cannot be generated until a 1 is generated. In a more general case, when GenOrdTrees(x, y) is about to be activated, the total numbers of 1s and 0s generated in between b_1 and $b_{2n-(2x+y)}$ are $(n-x)$ and $(n-x-y)$ respectively. As $(n-x-y) \leq (n-x)$ when x and y are non-negative integers, a B_n so generated is always feasible. Note that these two properties can also be obtained from the grid directly. Finally, we show that the successive B_n s generated are in lexicographical order. It may be seen from the procedure GenOrdTrees that when two B_n s with same prefix are generated, a 0 is always assigned to b_i before a 1 is assigned to it, where b_i is the leftmost bit that makes the two B_n s distinct. Therefore, we have proved that GenOrdTrees always generates the encoded form of extended ordered trees in lexicographical order.

If GenOrdTrees is involved in an application, then an application routine should be called from within GenOrdTrees replacing PrintBitPattern. If GenOrdTrees is built as a library routine, then one way of establishing communication with an application routine is to use co-routine construct. Moreover, it is sometimes required to generate the successor of a given ordered tree. GenOrdTrees always generates ordered trees starting from the first one in lexicographical order; this may not

be desired sometimes. We now give an algorithm which will generate the successor of a given B_n , without generating all the predecessors.

procedure GenSuc;

{ Given a feasible bit-pattern $b_1 b_2 \dots b_{2n}$, this procedure manipulates the trailing block of 1s so that the resulting bit-pattern is the immediate successor of the given one. }

var i,j,k,m:integer;

begin

i := 2n - 1;

while $b_i = 0$ **do** i := i - 1;

j := i - 1;

while j > 0 **and** $b_j = 1$ **do** j := j - 1;

k := i - j - 1;

for m := j + 1 **to** 2(n - k) - 1 **do** $b_m := 0$;

while k > 0 **do begin**

$b_{2(n-k)+1} := 1$;

$b_{2(n-k)} := 0$;

k := k - 1

end;

if j = 0 **then** $b_1 := 1$ **else** $b_j := 1$

end{ GenSuc };

Define a *block* of 1s to be a sequence of consecutive 1s such that it is surrounded by 0s or empty bit. Let $b_p \dots b_q$ be the rightmost block of 1s in B_n . The basic strategy of GenSuc is to move b_p to b_{p-1} , which must be a 0 or empty by definition of the block, and spread $b_{p+1} \dots b_q$ to the rightmost positions such that the resulting B_n is feasible. The former operation moves b_p to its next significant position, and the latter operation spreads $b_{p+1} \dots b_q$ to their least significant positions. Of course, if $p = q$, the latter operation amounts to a null operation. It can be shown that the resulting B'_n is the immediate successor of the given B_n . Note that when the last B_n in the lexicographical order is given as input, GenSuc will return the first B_n as a result. In this sense, the algorithm indeed generates $B(n)$ in the lexicographical order cyclically.

Let $C(x, y)$ be the number of calls of GenOrdTrees when x and y are given as parameters. From the algorithm, we may derive the following recurrence equation,

$$C(x, y) = C(x, y - 1) + C(x - 1, y + 1) + 1$$

with the following boundary condition,

$$C(x, y) = 0 \quad \text{if } x < 0 \quad \text{or } y < 0.$$

Solving the above recurrence equation for $C(n, 0)$ and then dividing C_n into it, the average time-complexity of GenOrdTrees is $O(1)$ or approximately '1.33 + lower order terms'.

Note also that GenSuc manipulates the same number of bits as GenOrdTrees on average. Hence the average time-complexity of GenSuc is the same as GenOrdTrees. The only programming difference between them is that GenSuc computes the 'backtracking points' explicitly by iteration, whereas GenOrdTrees carries out the same task implicitly by recursion.

4. RANKING FUNCTION

Let $I(T)$ be the position index of an ordered tree T in the lexicographical ordering. More specifically, $I(T_i) = i$. It is often necessary to associate such a position index with an ordered tree for some applications. Our approach is to

establish a 1-1 mapping from $B(n)$ to the set of position indices. This 1-1 mapping is termed the ranking function. As there is a 1-1 mapping between $B(n)$ and $T(n)$, therefore $I(T)$ equals the position index of B_n . The ranking algorithm is given below.

function Rank(B_n :bitpattern):integer;

{ This function computes the position index of B_n as a result. }

var cnt, index, i, posn:integer;

begin

cnt := n - 1;

index := 1;

for i := 2 **to** 2n - 1 **do**

if $b_i = 1$ **then begin**

posn := 2n - i;

index := index + $\binom{\text{posn}}{\text{cnt}} - \binom{\text{posn}}{\text{cnt} - 1}$;

cnt := cnt - 1

end;

Rank := index

end { Rank };

The basic ideas behind the ranking algorithm are explained in the following. Suppose we are given a feasible $B_n = (b_1 b_2 \dots b_{2n})$. Let $f(i)$ be the number of 1s between b_i and b_{2n-1} inclusively. Then the ranking algorithm computes precisely the following summation.

$$\text{Rank}(B_n) = 1 + \sum_{b_i=1} \left(\binom{2n-i}{f(i)} - \binom{2n-i}{f(i)-1} \right) \quad (1)$$

Equation (1) can be shown to be correct. Let $V(i)$ be the number of feasible B_n s such that they all have the same prefix $b_1 \dots b_{i-1}$ as B_n but preceding the B_n concerned lexicographically, when $b_i = 1$.

Lemma 1

$$V(i) = \binom{2n-i}{f(i)} - \binom{2n-i}{f(i)-1}$$

Proof

$V(i)$ is indeed equal to the number of permutations of $f(i)$ 1s and $(2n-i-f(i))$ 0s between b_{i+1} and b_{2n} inclusively, such that the resulting B_n s are feasible. The number of permutations of $f(i)$ 1s and $(2n-i-f(i))$ 0s is

$$\binom{2n-i}{f(i)}$$

However, the number of non-feasible B_n s amongst them is

$$\binom{2n-i}{f(i)-1}$$

Therefore the number of feasible permutations is

$$\binom{2n-i}{f(i)} - \binom{2n-i}{f(i)-1} \quad [\text{QED}]$$

We now apply lemma 1 to prove equation (1).

Theorem 2

$$\text{Rank}(B_n) = 1 + \sum_{b_i=1} \left(\binom{2n-i}{f(i)} - \binom{2n-i}{f(i)-1} \right)$$

Proof

The given B_n is preceded by other B_n s which have similar prefixes in the lexicographical order. For each $b_i = 1$ in the B_n concerned, $b_1 \dots b_{i-1}$ is a prefix. Therefore, the total number of B_n s preceding the B_n concerned in the lexicographical order is (by lemma 1):

$$\sum_{b_{i-1}} \binom{2n-i}{f(i)} - \binom{2n-i}{f(i)-1}$$

Hence the position index of B_n is:

$$1 + \sum_{b_{i-1}} \binom{2n-i}{f(i)} - \binom{2n-i}{f(i)-1} \quad [\text{QED}]$$

Corollary 3

$$V(1) = 0$$

Proof

By the property of feasible B_n ,

$$f(1) = n$$

Therefore

$$\begin{aligned} V(1) &= \binom{2n-1}{n} - \binom{2n-1}{n-1} \\ &= 0 \quad [\text{QED}] \end{aligned}$$

Theorem 2 and corollary 3 are used in Rank for setting the initial values of cnt and index. Furthermore, the expression of lemma 1 can be simplified into an expression consisting of one binomial coefficient. As there are n 1s in B_n , the running time of the ranking algorithm is $O(n)$ units of the computation time of binomial coefficient. Note that a binomial coefficient $\binom{n}{i}$ can be computed in $O(\lg n)$ arithmetic steps.¹¹

5. UNRANKING FUNCTION

The unranking function is the inverse of the ranking function. It maps a set of natural numbers to $B(n)$. The unranking algorithm is given in the following.

procedure Unrank(p :integer);

{ This algorithm constructs the bit-pattern of a given position index p . }

var cnt, i, V:integer;

begin

cnt := n;

for i := 1 **to** 2n **do begin**

V := $\binom{2n-i}{\text{cnt}} - \binom{2n-i}{\text{cnt}-1}$;

if $p > V$ **then begin**

b_i := 1;

cnt := cnt - 1;

p := p - V;

end

else b_i := 0

end

end { Unrank };

The unranking algorithm basically computes equation (1) in reverse. It compares the position index p with $V(i)$. If $p \leq V(i)$, obviously a 1 at b_i will make the position index of the constructed B_n larger than the given position

index; consequently, b_i should be a 0. Conversely, if $p > V(i)$, b_i should be assigned a 1; otherwise, the given position index will not fall within the range covered by the subset of B_n s having the prefix $b_1 \dots b_{i-1}$. By induction, we have established that Unrank is correct.

Here, we assume that $\binom{x}{y} = 0$ when $y < 0$. As there are $2n$

bits in B_n , the running time of the unranking algorithm is $O(n)$ units of the computation time of binomial coefficient.

6. CONCLUDING REMARKS

This paper exploits the advantages of encoding the extended ordered trees as binary bit-patterns. The enumerating algorithms of ordered trees, the ranking and unranking functions are shown to be very simple, based on this encoding scheme.

Note that generating random-ordered tree with n nodes becomes a trivial exercise with this encoding scheme. All we need to do is to generate a random number between 1 and C_n and then apply the unranking algorithm to it.

As a conclusion we compare the complexity of our algorithms with that of the published algorithms. Knott³ gives no enumerating algorithm at all, but specifies a ranking and unranking algorithms, both running at $O(n^2)$ units of time and requiring an array of Catalan numbers. Rotem and Varol⁶ present an enumerating algorithm running at constant units of time on average, a ranking algorithm at $O(n^2)$ and an unranking algorithm at $O(n \lg n)$ – both require a 2-D array filled with pre-calculated values which take $O(n^2)$ units of time to compute. Solomon and Finkel⁸ detail no algorithm at all, but argue that their enumerating and ranking algorithms run at $O(n)$ units of time, and their unranking algorithm takes $O(n \lg n)$ units of time. Proskurowski⁴ does not discuss a ranking and an unranking algorithms at all, but his enumerating algorithm requires to generate all ordered trees with nodes less than n even if they are not needed. Read⁵ describes no algorithm at all, but gives an example of enumerating ordered trees. Ruskey and Hu⁷ improve the running time of enumerating algorithm to $O(3)$ on average, and also present a ranking algorithm and an unranking algorithm – both run at $O(n \lg n)$ units of time and require a 2-D array of Catalan numbers. Zaks⁹ also details an enumerating algorithm running at constant units of time on average, a ranking algorithm running at $O(n)$ units of time and a non-trivial unranking algorithm – both require a 2-D array with pre-calculated values. In contrast, our enumerating algorithm runs at $O(1)$ units of time on average, and our ranking and unranking algorithms both run at $O(n)$ units of time with a 2-D array of Catalan numbers or at $O(n \lg n)$ units of time without the array. More importantly, our algorithms are conceptually simpler compared with the corresponding published algorithms. This is because our algorithms generate and manipulate bit-patterns, whereas some of the published algorithms generate and manipulate actual trees.

Furthermore, a similar encoding scheme is applicable to the extended t -ary ordered trees. The results discussed above can also be extended to the enumerating, ranking and unranking algorithms of t -ary ordered trees.

Acknowledgements

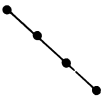

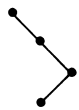
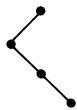

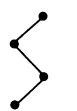
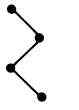

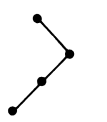
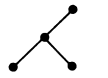

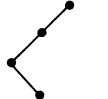
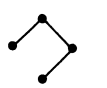
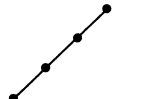
The author wishes to thank the referee for his many valuable suggestions and additional references. This

research was supported by the R.G.C. under grant 05-143-105.

REFERENCES

1. N. G. de Bruijn and B. J. M. Morselt, A note on plane trees, *Journal of Combinatorial Theory* **2**, 27–34 (1967).
2. M. C. Er, A note on generating well-formed parenthesis strings lexicographically. *The Computer Journal* **26**, 205–207 (1983).
3. G. D. Knott, A numbering system for binary trees. *Communications of the ACM* **20**, 113–115 (1977).
4. A. Proskurowski, On the generation of binary trees. *Journal of the ACM* **27**, 1–2 (1980).
5. R. C. Read, A survey of graph generation techniques. *Lecture Notes in Mathematics* **884**, 77–89 (1980).
6. D. Rotem and Y. L. Varol, Generation of binary trees from ballot sequences. *Journal of the ACM* **25**, 396–404 (1978).
7. F. Ruskey and T. C. Hu, Generating binary trees lexicographically. *SIAM Journal on Computing* **6**, 745–758 (1977).
8. M. Solomon and R. A. Finkel, A note on enumerating binary trees. *Journal of the ACM* **27**, 3–5 (1980).
9. S. Zaks, Lexicographic generation of ordered trees. *Theoretical Computer Science* **10**, 63–82 (1980).
10. D. E. Knuth, *Art of Computer Programming*, vol. 1. Addison-Wesley, Reading, Massachusetts (1973).
11. A. Shamir, Factoring numbers in $O(\log n)$ arithmetic steps. *Information Processing Letters* **8**, 28–31 (1979).

APPENDIX A: LEXICOGRAPHICAL LISTING OF ORDERED TREES

Position indices	Ordered trees	Encoding in binary bit-patterns	Position indices	Ordered trees	Encoding in binary bit-patterns
1		1 0 1 0 1 0 1 0	8		1 1 0 1 0 0 1 0
2		1 0 1 0 1 1 0 0	9		1 1 0 1 0 1 0 0
3		1 0 1 1 0 0 1 0	10		1 1 0 1 1 0 0 0
4		1 0 1 1 0 1 0 0	11		1 1 1 0 0 0 1 0
5		1 0 1 1 1 0 0 0	12		1 1 1 0 0 1 0 0
6		1 1 0 0 1 0 1 0	13		1 1 1 0 1 0 0 0
7		1 1 0 0 1 1 0 0	14		1 1 1 1 0 0 0 0