Nordic Journal of Computing 3(1996), 63–71.

GENERATING t-ARY TREES IN PARALLEL *

SELIM G. AKL
Department of Computing
and Information Science
Queen's University
Kingston, Ontario
Canada K7L 3N6

IVAN STOJMENOVIC
Computer Science Department
University of Ottawa
Ottawa, Ontario
Canada K1N 9B4
ivan@csi.UOttawa.CA

Abstract. We present a cost-optimal parallel algorithm for generating t-ary trees. Using a known inversion table representation, our algorithm generates all tree sequences in lexicographic order. It uses a linear array of n processors (where n is the number of nodes in the tree), each having a constant number of registers (each storing an integer of size at most tn), and each being responsible for producing one element of a given tree sequence.

CR Classification: G.1.0, G.2.1, G.2.2

Key words: binary tree, t-ary tree, parallel algorithms, breadth first search, linear array of processors, parallel random access machine, lexicographic order

1. Introduction

The combinatorial problem of generating binary and, in general, t-ary trees is concerned with generating all different shapes of t-ary trees with n nodes in some order. The number of t-ary trees with n nodes is [Knuth 1968, Zaks 1980]

$$B(n,t) = \frac{(tn)!}{n!(tn-n)!}\frac{1}{(t-1)n+1}.$$

A list of all t-ary trees might be used to search for a counter-example to some conjecture, or to test and analyze an algorithm for its correctness or computational complexity.

A significant body of knowledge exists concerning sequential tree generation. There are over 30 ingenious algorithms for generating binary trees (for example, Knott [1977], Lucas *et al.* [1993], Proskurowski [1980], Roelants van Baronaigien and Ruskey [1988], Roelants van Baronaigien and Ruskey [1987], Roelants van Baronaigien [1991], Trojanowski [1978], Zaks and Richards [1979], Zaks [1980] and references listed in Mäkinen [1991]) and t-ary trees [Er 1987, Roelants van Baronaigien and Ruskey 1988, Ruskey 1978, Trojanowski 1978, Zaks and Richards 1979, Zaks 1980]. In most of

*This research was partially supported by the NSERC

Received December 22, 1994; revised October 19, 1995; accepted December 13, 1995.

these algorithms, the trees are encoded as integer sequences and then those t-ary tree sequences are generated lexicographically.

In this paper we describe a parallel algorithm for generating t-ary tree sequences. To our knowledge this is the first parallel algorithm for generating all $B(n,t)$ t-ary trees with n nodes.

A well-known notation for binary trees is the inversion table representation [Knott 1977]. In this paper we use this notation as well as a new representation based on Breadth First Search (BFS). Our algorithm generates tree sequences in lexicographic order but lists trees in two different orders depending on whether the inversion table or BFS representation is chosen.

Our parallel algorithm satisfies several desirable properties with respect to the model of parallel computation and the speed of computation. The model of parallel computation should be as simple as possible. Arguably, the simplest such model is a linear array of m processors, indexed 1 through m, where each processor $i(2 \geq i \geq m-1)$ is connected by bidirectional links to its immediate left and right neighbors, $i-1$ and $i+1$, and processors 1 and m are each connected to one neighbor. This model is practical, as it is amenable to VLSI implementation [Akl 1989]. Each processor has a constant number of registers in its local memory, each capable of storing an integer of size $O(tn)$. This implies that no processor can store an array of size n, or a counter up to $n!$.

The time required by our algorithm between any two consecutive trees it produces is constant. A constant time delay between outputs is particularly important in applications where the output of one computation serves as input to another.

An algorithm is *cost-optimal* if the number of processors it uses multiplied by its running time matches - up to a constant factor - a lower bound on the number of operations required to solve the problem sequentially. This property can be further specified according to the way in which the lower bound is defined. We identify two such definitions:

a) The time required to "create" the trees, without actually listing the n elements of each tree, is counted. Optimal sequential algorithms in this sense generate trees in $O(B(n,t))$ time, i.e. time linear in the number of trees of n elements. The loopfree algorithms [Roelants van Baronaigien 1991, Lucas *et al.* 1993] generate the next binary tree sequence with constant delay from the current one, exclusive of the output. Akl [1987] describes a parallel algorithm for generating permutations and combinations which uses an unranking procedure to subdivide equally all permutations or combinations among a given number of processors, and then each of these generates its portion using a sequential algorithm. This approach can be used to generate t-ary trees in parallel; however, the constant memory and constant delay properties are not satisfied.

b) The time to output each tree in full is counted. Here, optimal sequen-

tial algorithms run in $O(n * B(n,t))$ time, since it takes $O(n)$ time to output a tree. In this paper we adopt this measure and design a cost-optimal parallel algorithm for generating t-ary trees; designing an optimal parallel algorithm for generating binary trees under measure (a), and with processors having constant size local memory, remains an open problem.

In summary, our parallel algorithm for generating t-ary tree sequences is optimal in more than one sense: the simple model of computation (linear array of processors), the constant size of the local memory, the constant delay between consecutive trees, and the cost-optimality. We note in passing that parallel algorithms exist that satisfy these criteria for generating other combinatorial objects such as permutations [Akl *et al.* 1994], combinations [Akl *et al.* 1989/90], derangements [Akl *et al.* 1992], subsets and equivalence relations [Stojmenovic 1990] etc.

This paper generalizes the results for binary trees that we presented in the preliminary conference version of the paper [Akl and Stojmenovic 1992].

2. Existing t-ary tree representations

The t-ary trees are data structures consisting of a finite set of n nodes which is either empty ($n = 0$), or consists of a root and t disjoint children. Each child is a t-ary subtree, recursively defined. A node is the parent of another node if the latter is a child of the former. For $t = 2$ one gets the special case of binary trees, where each node has a left and a right child, where each child is either empty or is a binary tree. A computer representation of t-ary trees with n nodes can be achieved by an array of n records, each record consisting of several data fields, t pointers to children and a pointer to the parent. All pointers to empty trees are nil. A more convenient data structure for t-ary trees would be a single data field, an array of pointers for the children, and, perhaps, a pointer to the parent.

Lexicographic order of tree sequences is defined as follows. If $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_n)$ are representations of trees, then A precedes B lexicographically if and only if, for some $j \geq 1, a_i = b_i$ when $i < j$, and $a_j < b_j$. In most references, tree sequences are generated in lexicographic order. Each of tree generation algorithms causes trees to be generated in a particular order. Thus the lexicographic order of trees refers, more precisely, to the lexicographic order of the corresponding tree sequences.

The t-ary tree representation, called the inversion table, is introduced in Knuth [1968] (for binary trees) and used in Martin and Orr [1989]. Given a t-ary tree T having n nodes, let the root be labeled with 0. Inductively we define a labeling for every node of T as follows: if a node u is an i-th ($1 \leq i \leq t$) child of its parent, then label u with $t - i$ plus the value of the label of its parent. After having labeled the nodes of T, do a preorder traversal of T, writing down the labels as each node is visited, to form a sequence $x_1 \ldots x_n$ representing T. The sequence $x_1 \ldots x_n$ is an inversion

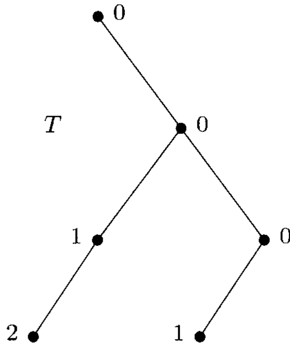

Fig. 1: T

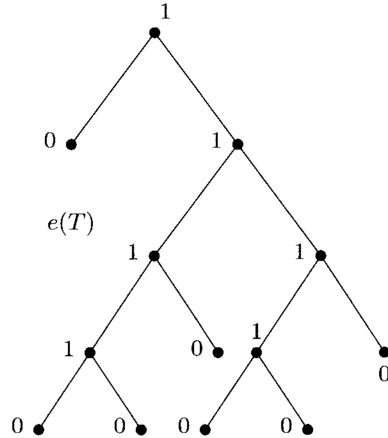

Fig. 2: $e(T)$

table sequence if and only if $x_1 = 0$ and $0 \leq x_j \leq x_{j-1} + t - 1$ for $2 \leq j \leq n$. The x-sequences have been used previously to generate binary trees Mäkinen [1987]. For example, for $t = 2$ and $n = 3$ the following is the list of all binary tree sequences with 3 nodes: $000, 001, 010, 011, 012$. The tree in Fig. 1 is labeled as 001201. The parallel tree sequence algorithm described in section 3 uses this representation.

A t-ary tree is full if each node has either 0 or t children. In Knuth [1968] a one-to-one correspondence between full trees with $tn + 1$ nodes and t-ary trees with n nodes is established by matching t-ary tree T with extended tree $e(T)$, obtained by replacing empty subtrees of T with real nodes (see Fig. 1 and 2 for an example with $t = 2$).

A related bitstring representation has been known since it was realized that well-formed parentheses and binary trees are both counted by the Catalan numbers. A t-ary tree T having n nodes can be represented by a bitstring $b_1 b_2 \ldots b_{tn}$ with $(t-1)n$ zeros and n ones that satisfies the prefix property: the number of 0's is always at most $t - 1$ times the number of 1's while scanning from b_1 to b_{tn} (i.e. $b_1 + \ldots + b_i \geq i/t, 1 \leq i \leq tn$). Here each 1 corresponds to a node of T while 0's correspond to the leaves of $e(T)$. The bitstring encodes $e(T)$, in preorder traversal. For example, binary tree T in Fig. 1 is extended to tree $e(T)$ in Fig. 2 and is coded as 101110001100 (the last 0 is omited). If the positions of 1's in a given sequence $b_1 \ldots b_{tn}$ are marked in a separate sequence $z_1 \ldots z_n$ (called z-sequence in Zaks [1980]) then, due to the prefix property, we get: $z_1 = 1, z_{j-1} < z_j \leq tj - 1$ for $2 \leq j \leq n$. For example, $123, 124, 125, 134, 135$ correspond to bitstrings $111000, 110100, 110010, 101100$, and 101010, respectively; T of Fig. 1 is coded as $1, 3, 4, 5, 9, 10$.

Given a full tree P, let r_P be the number of children of the root of P, i.e. $r_P = t$ or $r_P = 0$. Let P_i denote the i-th subtree of P, $1 \leq i \leq t$. Two full

trees P and Q are in B-order Trojanowski [1978], $P < Q$, if:

1. $r_P < r_Q$,
2. $r_P = r_Q$, and $P_1 < Q_1$ or
3. $r_P = r_Q$, $P_j = Q_j$ for $1 \leq j < i \leq t$, and $P_i < Q_i$.

Two t-ary trees T' and T'' are in B-order, $T' < T''$ if $e(T') < e(T'')$.

The algorithms in Trojanowski [1978] and Zaks [1980] generate binary trees in B-order. In Zaks [1980] and Er [1987] it is proved that the B-order of trees matches the lexicographic order of bitstring sequences $b_1 \ldots b_{tn}$. It is an easy exercise to verify that the lexicographic order of inversion table sequences corresponds to the B-order of binary trees. The inversion table and z-sequences are related by $x_i = ti - t + 1 - z_i$. Therefore the sequential algorithms for generating trees [Zaks 1980, Proskurowski 1980, Er 1987] can be used to generate all inversion table sequences, using a simple transform. Note that there are algorithms for generating trees in orders that are different from the B-order (for example, Lucas *et al.* [1993], Roelants van Baronaigien and Ruskey [1988]). A more complete discussion of the relationship between various generation algorithms and their representations is discussed in Lucas *et al.* [1993].

3. Parallel generation of tree sequences

In this section we present a parallel algorithm that generates tree sequences of the form $x_1, \ldots, x_n$ as defined in section 2. If the inversion table representation is meant, then the algorithm generates tree sequences in B-order. The same algorithm can be used to generate tree sequences in lexicographic order in the new notation introduced in section 3 (recall that the same sequence refers to different trees and thus the order of generating trees is different) without any change, except that $p_i = ti - t + 1 - x_i$, for $1 \leq i \leq n$, is obtained easily once x_i is generated.

Our algorithm generates the tree sequences on a linear array of n processors, indexed 1 to n, and n variables $x_1, x_2, \ldots, x_n$. Each processor i is responsible for maintaining x_i by reading data from processors $i - 1$ and $i + 1$. Processor n successively produces the values $0, 1, ..., x_{n-1} + 1$, in that order. The production of those $x_{n-1} + 2$ values is called a run of processor n. Whenever $x_n = 0$ processor n initiates a message that searches for processor tp, called the turning point, that keeps an element x_{tp} which should be increased by 1 after processor n finishes producing its current run. tp is the maximum index such that $tp < n$ and $x_{tp} \leq x_{tp-1} + t - 2$. Each step in the search corresponds to producing a tree. After the backtracking message search finds the turning point tp, processors $j \geq tp$ wait for an appropriate number of steps, and then simultaneously x_{tp} increases by 1 while x_j for $j > t$ becomes 0.

The algorithm is as follows. All processors i, for $i = 1, 2, \ldots, n$, synchronously run the following program.

$x_i \leftarrow 0$; $w_i \leftarrow 0$; $m_i \leftarrow 0$; $term_i \leftarrow$ false;
Repeat
  if $m_i = 1$ and $i < n$ and $x_i < x_{i-1} + t - 1$ **then** $m_i \leftarrow 2$;
  if $w_i = 1$ and $m_i = 0$ **then** $x_i \leftarrow 0$;
  if $w_i = 1$ and $m_i = 2$ **then** $\{x_i \leftarrow x_i + 1; m_i \leftarrow 0\}$
  if $m_{i+1} = 1$ and $i < n$ **then** $\{m_i \leftarrow 1; w_i \leftarrow w_{i+1}\}$
    else if $m_i = 1$ **then** $m_i \leftarrow 0$;
  if $w_i \geq 1$ **then** $w_i \leftarrow w_i - 1$;
  if $x_n = 0$ and $w_n = 0$ **then** $\{m_n \leftarrow 1; w_n \leftarrow x_{n-1} + t\}$
    else $x_n \leftarrow x_n + 1$;
  if $w_i = i + (n-1)(t-2)$ **then** $term_i \leftarrow$ true;
  output x_i
until $term_i$ =true and $w_i = 1$

The algorithm is traced for $t = 2, n = 4$, and the following values (taken at the time of each output) for variables x, m, w, and $term$ are obtained.

x_1	x_2	x_3	x_4	m_1	m_2	m_3	m_4	w_1	w_2	w_3	w_4	$term_1$	$term_2$	$term_3$	$term_4$
0	0	0	0	0	0	0	1	0	0	0	2	false	false	false	false
0	0	0	1	0	0	1	0	0	0	1	1	false	false	false	false
0	0	1	0	0	0	0	1	0	0	0	3	false	false	false	false
0	0	1	1	0	0	1	0	0	0	2	2	false	false	false	false
0	0	1	2	0	1	0	0	0	1	1	1	false	false	false	false
0	1	0	0	0	0	0	1	0	0	0	2	false	false	false	false
0	1	0	1	0	0	1	0	0	0	1	1	false	false	false	false
0	1	1	0	0	0	0	1	0	0	0	3	false	false	false	false
0	1	1	1	0	0	1	0	0	0	2	2	false	false	false	false
0	1	1	2	0	0	2	0	0	0	1	1	false	false	false	false
0	1	2	0	0	0	0	1	0	0	0	4	false	false	false	true
0	1	2	1	0	0	1	0	0	0	3	3	false	false	true	true
0	1	2	2	0	1	0	0	0	2	2	2	false	true	true	true
0	1	2	3	1	0	0	0	1	1	1	1	true	true	true	true

More details of the above algorithm are given in what follows. We assume that the processors operate synchronously; this means that the r-th instruction executed by a processor does not start until the $(r-1)$-st instruction of all the other processors has been executed. A new tree sequence is produced at the end of each iteration of the Repeat ... until loop. Processor i keeps the following variables in its local memory: x_i, w_i, m_i, and $term_i$. Variable m_i is used as the message to search for the turning point tp. At any time there is exactly one processor i for which $m_i \neq 0$. The value $m_i = 0$ indicates the inactive state of the processor in the search process. The search for tp

is started by processor n whenever $x_n = 0$, and processor n sets $m_n = 1$. After each step (producing a tree) the message is sent from processor i to processor $i-1$ (thus $m_{i-1} = 0$, $m_i = 1$ becomes $m_{i-1} = 1$, $m_i = 0$). Processor i searching for the turning point tp (i.e. $m_i = 1$) recognizes itself as such when $x_i \leq x_{i-1} + t - 2$, and indicates this case by the value $m_i = 2$. The variable w_i is used to count down from the moment the search for the turning point went through processor i until processor i is supposed to change its value. If processor i is the turning point then the new value is $x_i + 1$ else it is 0. Counting down starts from the value $x_{n-1} + t$ at processor n, and decreases to 0, when the new value of x_i takes effect. The count down value is broadcast to processors $n-1, n-2, \ldots, tp$ together with the message m_i looking for the turning point. Processor i keeps also variable $term_i$ which is initialized to false and is set to true when $w_i = i + (n-1)(t-2)$. The algorithm terminates when $term_i$ =true and $w_i = 1$.

The correctness of the algorithm follows from the properties of inversion table representation. There is enough time for message passing because the message path has length (the number of steps) at most $x_{n-1} + t - 1$, which is exactly the length of the current run of processor n. Therefore there is enough time for the message to find the turning point and for all processors to receive the signal to change their value x_i to either $x_i + 1$ or 0 before the change is due. Processor i is also able to easily distinguish the last two cases; it is not affected by the "major" change if $i < tp$, i.e. when the search for the turning point does not reach processor i. Furthermore, all processors will terminate simultaneously. The equality $w_i = (t-1)(n-1) + 1 - (n-i) = i + (n-1)(t-2)$ is possible only when the turning point search reaches processor 1, in which case the algorithm is about to terminate.

Summarizing, the algorithm described above generates all tree sequences with n nodes in lexicographic order and with constant delay per tree on a linear array of n processors, thus achieving an optimal cost of $O(n * B(n,t))$; furthermore each processor has a memory of constant size and can generate elements without the need to deal with large integers such as $B(n,t)$.

4. Conclusion

The parallel algorithms presented in this paper can be made adaptive (i.e. to run on a parallel model of computation consisting of an arbitrary number k of processors) if the processors are divided into k/n groups of n processors each such that each group produces an interval of consecutive trees. The first and the last tree in each group can be determined in a preprocessing step by applying any known unranking function [Zaks 1980, Trojanowski 1978, Ruskey 1978, Er 1987] that follows the lexicographic order of z-sequences of trees. However, these function involve integers of size $O(B(n,t))$. Another scheme that deals only with integers of size $O(tn)$ and yet divides the job evenly among the groups is described in Stojmenovic [1992].

Sequential algorithms exist for generating AVL trees [Li 1986], 2-3 trees,

B-trees [Gupta *et al.* 1983], non-full trees [Er 1988], unordered trees [Pallo 1989], free trees [Wright *et al.* 1986], trees with n nodes and m leaves [Pallo 1987], trees with nodes of any degree [Skarbek 1988], and trees with bounded height [Lee *et al.* 1986]. It remains an open problem to describe parallel algorithms for generating these kinds of trees.

Acknowledgements

The authors are grateful to two referes for careful reading of the paper and improving its clarity and presentation.

References

AKL, S. G. 1987. Adaptive and Optimal Parallel Algorithms for Enumerating Permutations and Combinations. *The Computer Journal 30*, 5 (Oct.), 433–436.

AKL, S. G. 1989. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, Englewood Cliffs, New Jersey.

AKL, S. G., CALVERT, J., AND STOJMENOVIC, I. 1992. Systolic Generation of Derangements. In *Algorithms and Parallel VLSI Architectures II.* (P. Quinton, Y. Robert, ed.), Elsevier, 59–70.

AKL, S. G., GRIES, D., AND STOJMENOVIC, I. 1989/90. An Optimal Parallel Algorithm for Generating Combinations. *Information Processing Letters 33*, 3 (Nov.), 135–139.

AKL, S. G., MEIJER, H., AND STOJMENOVIC, I. 1994. An Optimal Systolic Algorithm for Generating Permutations in Lexicographic Order. *Journal of Parallel and Distributed Computing 20*, 1 (Jan.), 84–91.

AKL, S. G. AND STOJMENOVIC, I. 1992. Generating Binary Trees in Parallel. In *Proc. Allerton Conference on Communication, Control, and Computing*, 225–233.

ER, M.C. 1987. Lexicographic Listing and Ranking t-ary Trees. *The Computer Journal 30*, 6 (Dec.), 569–572.

ER, M.C. 1988. A Simple Algorithm for Generating Non-Regular Trees in Lexicographic Order. *The Computer Journal 31*, 1 (Feb.), 61–64.

GUPTA, U. I., LEE, D. T., AND WONG, C. K. 1983. Ranking and Unranking of B-Trees. *Journal of Algorithms 4*, 1 (Mar.), 51–60.

KNOTT, G. D. 1977. A Numbering System for Binary Trees. *Communications of ACM 20*, 2 (Feb.), 113–115.

KNUTH, D. E. 1968. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, Ma.

LEE, C. C., LEE, D. T., AND WONG, C. K. 1986. Generating Binary Trees of Bounded Height. *Acta Informatica 23*, 529–544.

LI, L. 1986. Ranking and Unranking of AVL-Trees. *SIAM Journal on Computing 15*, 4 (Nov.), 1025–1035.

LUCAS, J., ROELANTS VAN BARONAIGIEN, D., AND RUSKEY, F. 1993. On Rotations and the Generation of Binary Trees. *Journal of Algorithms 15*, 3 (Nov.), 343–366.

MÄKINEN, E. 1987. Left Distance Binary Tree Representations. *BIT 27*, 2, 163–169.

MÄKINEN, E. 1991. A Survey on Binary Tree Codings. *The Computer Journal 34*, 5 (Oct.), 438–443.

MARTIN, H. W. AND ORR, B. J. 1989. A Random Binary Tree Generator, Computing Trends in the 1990's. In *ACM Seventeenth Computer Science Conference.* Louisville, Kentucky, Feb., 33–38.

OLARIU, S., SCHWING, J. L., WEN, Z., AND ZHANG, J. 1991. Optimal Parallel Encoding and Decoding Algorithms for Trees. In *Proc. ACM 19th Annual Computer Science Conference.* San Antonio, Texas, 1–9.

PALLO, J. M. 1987. Generating Trees with n Nodes and m Leaves. *Int. J. Comp. Math. 21*, 133–144.

PALLO, J. M. 1989. Lexicographic Generation of Binary Unordered Trees. *Pattern Recognition Letters 10*, 4 (Oct.), 217–221.

PROSKUROWSKI, A. 1980. On the Generation of Binary Trees. *Journal of the ACM 27*, 1 (Jan.), 1–2.

ROELANTS VAN BARONAIGIEN, D. 1991. A Loopless Algorithm for Generating Binary Tree Sequences. *Information Processing Letters 39*, 4 (Aug.), 189–194.

ROELANTS VAN BARONAIGIEN, D. AND RUSKEY, F. 1988. Generating t-ary Trees in A-order. *Information Processing Letters 27*, 4 (Apr.), 205–213.

ROELANTS VAN BARONAIGIEN, D. AND RUSKEY, F. 1987. A Hamiltonian Path in the Rotation Lattice of Binary Trees. In *Congressus Numerantium 59*, 313–318.

RUSKEY, F. 1978. Generating t-ary Trees Lexicographically. *SIAM Journal on Computing 7*, 4 (Nov.), 424–439.

SKARBEK, W. 1988. Generating Ordered Trees. *Theoretical Computer Science 57*, 1 (Apr.), 153–159.

STOJMENOVIC, I. 1990. An Optimal Algorithm for Generating Equivalence Relations on a Linear Array of Processors. *BIT 30*, 3, 424–436.

STOJMENOVIC, I. 1992. On Random and Adaptive Parallel Generation of Combinatorial Objects. *Int. J. Comp. Math. 42*, 125–135.

TROJANOWSKI, A. E. 1978. Ranking and Listing Algorithms for k-ary Trees. *SIAM Journal on Computing 7*, 4 (Nov.), 492–509.

WRIGHT, R. A., RICHMOND, B., ODLYZKO, A., AND MCKAY, B. D. 1986. Constant Time Generation of Free Trees. *SIAM Journal on Computing 15*, 2 (May), 540–548.

ZAKS, S. 1980. Lexicographic Generation of Ordered Trees. *Theoretical Computer Science 10*, 1, 63–82.

ZAKS, S. 1982. Generation and Ranking of k-ary Trees. *Information Processing Letters 14*, 1 (Mar.), 44–48.

ZAKS, S. AND RICHARDS, D. 1979. Generating Trees and Other Combinatorial Objects Lexicographically. *SIAM Journal on Computing 8*, 1 (Feb.), 73–81.