Pergamon

Computers Math. Applic. Vol. 28, No. 9, pp. 67–71, 1994
Copyright©1994 Elsevier Science Ltd
Printed in Great Britain. All rights reserved
0898-1221/94 $7.00 + 0.00

0898-1221(94)00178-2

The Representation of Permutations by Trees

P. BHATTACHARYA
Department of Computer Science and Engineering
University of Nebraska-Lincoln, Lincoln, NE 68588-0115, U.S.A.
prabir@cse.unl.edu

(Received April 1992; revised and accepted September 1993)

Abstract—Representation of permutations and combinations of N elements in lexicographical order by elements of a tree are considered. An algorithm for generating the nodes is presented and some examples are given. The algorithm could be implemented in any programming language that allows for recursive calls.

1. INTRODUCTION

Permutations arise naturally in connection with problems in many fields such as data encryption, parallel processing, computer networking and computational algebra. Extensive work has been done in the past to represent and generate permutations (see e.g., [1,2] for surveys). It is of considerable interest to store and retrieve permutations (and combinations) in an efficient manner in the computer memory. Recently, Arnow [3] developed an interesting method to store permutations, combinations and dihedral elements in a "tree"-structure, and gave some algorithms to generate the nodes and traverse these trees. In this paper, we develop algorithms to represent permutations and combinations in the *lexicographic order.* Lexicographic (or dictionary) order has numerous practical applications—in discrete mathematics (e.g., [4, p. 427]), in data structures (e.g., [5, p. 78], and [6, p. 117,181]), and in parallel processing and networking (e.g., [7, Chapter 6]). The reader may refer to, e.g., Liu [8, p. 78] or Rosen [9, p. 284] for algorithms to generate permutations in lexicographic order. However, these (and apparently, other) authors do not address the problems of storing and retrieving permutations which are certainly problems of considerable practical interest. The main contribution of this paper is to explore these problems using the tree structure. Our algorithms can be implemented using any programming language which is capable of recursive calls.

2. PERMUTATION TREES

We assume that the reader is familiar with the basic concepts of data structure and algorithms (e.g., [5,6,10]). We first review briefly the interesting algorithm considered in [3]. Suppose we wish to generate all the $N!$ permutations of a set of N elements. The "permutation tree" of N generations is shown in Figure 1 for $N = 3$. The leaves of the tree contain all the $N!$ permutations. The children of any node of the ith generation can be obtained from the node by inserting the integer $i + 1$ to the node element at all possible positions and thus generating new permutations. For example, for $i = 2$, the children of the node 12 of the tree in Figure 1 are obtained by inserting 3 in all possible ways into 12, obtaining 312, 132 and 123. Arnow [3] gives

This research is supported by the Grant AFOSR F49620-92-J-0286 from the Air Force Office of Scientific Research.

Typeset by $\mathcal{A}_{\mathcal{M}}\mathcal{S}$-TEX

an algorithm to generate all the $N!$ permutations in this way. To save the tree representation of the set of $N!$ permutations in the computer memory, we need to allocate enough memory for the $N!$ elements. These elements may be integers or characters—it may be easier to think of them in terms of integers but characters will take less memory. We shall consider the elements as integers but the method would work equally for characters.

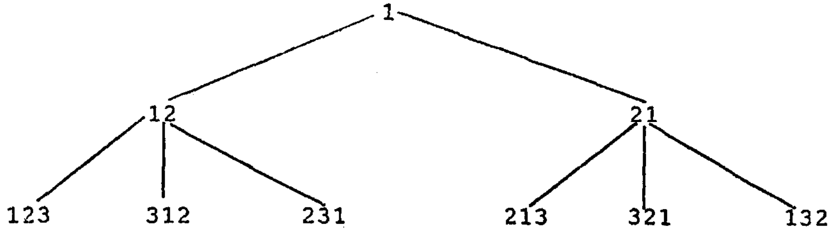

Figure 1.

The values of the preceding levels of permutations can be easily accessed from the tree representation. For example, to obtain the values of the tree level for the permutations on $N-1$ elements, we would take the first $N-1$ elements of each set of permutations of N elements. For permutations on $N-X$ elements ($X = 2, \dots N-1$), we would take the first $N-X$ elements of each set of $N(N-1)\cdots(N-X+1)$ sets of permutations of N elements. For example, for $N = 4$, the memory representation is given in Figure 2 (spaces are added to distinguish the different sets).

1234	4123	3412	2341	2134	4213	3421	1342
2314	4231	1423	3142	3124	4312	2431	1243
3214	4321	1432	2143	1324	4132	2413	3241

Figure 2.

3. ORDERED REPRESENTATION

We now present a method to store the permutation tree representation in the lexicographic order. First, we give examples in Figure 3 to arrange lexicographically the permutation tree for $N = 3, 4$. We would arrange the permutations of one-element sets pattern followed by the permutations of two-element sets, etc., ending with the permutations of four-element sets. Obviously, we do not need to arrange the entire permutation tree to see a small portion of the tree. The only limitation on the portion would be that it is in the permutation tree for less than or equal to N elements.

We shall use an array of "pointers" (see, e.g., [5,10]). Using the array of pointers implementation, we can determine directly any lexicographic entry on the specified pattern level. Here, the first pointer points to the first column, the second pointer points to the second column, etc. We look collectively at all the entries in the ith position of each of the columns. For example, at position 0, we have 1-1; at position 1, we have 2-3; at position 2, we have 2-1; etc. (Here, position 0 corresponds to the first value in the lexicographic order, position 1 to the second, etc.) Further, each column defines the exact position of the proper value for the lexicographic ordering. There are $N-1$ columns, denoted by $0, 1, \dots N-2$. The first column divides the tree into halves, the second column divides each half into three parts, a third column will divide the thirds into four parts, etc. There are $N-1$ columns numbered $0, 1 \dots N-2$. Each "higher" column in the pattern is a further division of the tree. Notice that the first column only has 1's and 2's; this is always the case. The first entry of $N = 4$ in the lexicographic ordering of the tree is located at 1-1-1, or in the first half, the first third of that half, and finally the first permutation

$N=3$		$N=4$		
1	1	1	1	1
2	3	1	2	4
2	1	2	3	1
1	3	2	1	4
1	2	1	3	3
2	2	2	2	3
		2	1	1
		2	2	4
		1	3	1
		1	1	4
		2	3	3
		1	2	3
		1	2	1
		1	3	4
		2	2	1
		2	3	4
		1	1	3
		2	1	3
		1	1	2
		2	3	2
		2	1	2
		1	3	2
		1	2	2
		2	2	2

Figure 3.

of that third. The sixteenth permutation of the $N = 4$ level is located at 2-3-4, or the the fourth permutation of the third third of the second half. We must keep in mind how the permutation tree and its entries are arranged in the memory to calculate the appropriate advancement values of a pointer (which is an array) and the level (the number of permutations being ordered along with corresponding pattern to arrange the permutations). In the next section, we shall continue with the general implementation of the algorithm to generate the patterns.

4. LEXICOGRAPHIC ORDERING

Arnow [3] suggested the use of an array of pointers to make the retrieval of necessary information easier. The allocated memory would be divided in the following manner. An array of $N-1$ pointers would be used and each pointer points to an amount of space equal to N elements. For N equal to $1, 2$ or 3, the pattern of lexicographic order has to be defined entirely. The pattern of lexicographic order for $N = 4$ is more difficult. The key to the whole generation process is the $N-3$ column. The pattern of two of the columns is easy to see, the first (0 column) and the last ($N-2$ column). Any columns in between shall also have a pattern which has been already mentioned. The easiest pattern to program is the last column representation.

We give in the Appendix a code to generate the last column of the lexicographic pattern described in Section 3. Our code is written in the standard programming language C which is widely used (see e.g., [11] for a background). It is, however, straightforward to translate our code into a code in any programming language which is capable of recursive calls; we leave the details as an exercise to the interested reader. The last column of $N = 4$ generated by code is shown in Figure 4.

Last Column				Column 0			
1	1	1	2	1	2	1	1
4	4	4	2	1	2	1	2
1	1	1	2	2	1	2	2
4	4	4	2	2	1	2	1
3	3	3	2	1	2	1	1
3	3	3	2	2	1	2	2

Figure 4.

5. COMBINATIONS

Combinations will be handled in a different way than permutations. For combinations of N elements, we need only to reserve space for these N elements. The algorithm for generating the combination tree proposed by Arnow [3] could be altered to have the procedure work with the arranged data. Also, the lexicographic ordering would be straightforward. For example, take the elements $1,\dots 4$. We start with the individual elements and then following this with the combinations of two with the first element and the remaining elements: 12 13 14, followed by the combinations of two of the second element and the remaining elements after it: 23, 24 which would be followed by 34. Then we do the same thing by taking combinations of three with the first element and those elements remaining. The interpretation of the remainder of the lexicographic ordering is now straightforward. If we want to just take a certain range, we have to compute the number of entries for each set of combinations and adjust starting appropriately.

6. CONCLUSION

We have developed an algorithm for generating a tree structure to represent permutations in a lexicographic order. Our proposed algorithm is executable in any programming language which allows for recursion. It would be of significant interest to develop more efficient algorithms to represent permutations in the computer memory. It would also be very interesting to consider the representation of permutations for parallel and distributed computing environments. We hope that this work would generate further interest in this practical problem.

APPENDIX

We give here a code to generate the last column of the lexicographic pattern described in Section 3. In our code, "maxint" denotes the N value of the permutation tree, "branch" is the array of pointers, "pattern" is used to copy the similar pattern that occurs in the column, "maxcopy" is the number of times the pattern should be copied.

```
-------------------------------------------------------------------------------
begin
if (maxint > 2) { /* Maxint ≥ 3 for this work.  */
pattern =1;
index = 0;
branch [maxint-2] [index] =1;
branch [maxint-2] [index +1] = maxint;
newindex = index +2;
maxcopy = 2;
while (pattern < maxint) {
count = 0;
tempindex = index;
while (count < (pattern * maxcopy)) {
branch [maxint -2] [newindex] = branch [maxint-2] [tempindex;
newindex = newindex +1;
```

```
count = count + 1;
} /* end while (count <pattern * maxcopy) */
if (pattern < (maxint-1)) {
for (count = 0; count < maxcopy;count++) {
branch [maxint-2] [newindex] = branch maxint - pattern;
newindex = newindex +1;
} /* end for (count < maxcopy) */
maxcopy = maxcopy * (pattern +2);
} /* end of if (pattern < (maxint-1)) */
pattern = pattern + 1;
} /* while (pattern < maxint) */
} /* end if (maxint > 2);
end
--------------------------------------------------------------------------------
```

REFERENCES

1. D.H. Lehmer, The machine tools of combinatorics, In *Applied Combinatorial Mathematics*, (Edited by E.F. Beckenbach), John Wiley, New York, (1964).
2. R. Sedgewick, Permutation generation methods, *ACM Computing Surveys* **19**, 137–164 (1977).
3. B.J. Arnow, Representation of permutations, combinations and dihedral elements as trees, *Computers Math. Applic.* **20** (3), 63–67 (1990).
4. R.L. Graham, D.E. Knuth and O. Patashnik, *Concrete Mathematics*, Addison-Wesley, Reading, MA, (1989).
5. A.V. Aho, J.E. Hopcroft and J.D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, (1983).
6. G.H. Gonnet and R.B. Yates, *Handbook of Algorithms and Data Structures*, 2nd edition, Addison-Wesley, Reading, MA, (1991).
7. S.G. Akl, *The Design and Analysis of Parallel Algorithms*, Prentice Hall, Englewoods Cliffs, NJ, (1989).
8. C.L. Liu, *Elements of Discrete Mathematics*, 2nd edition, McGraw Hill, New York, (1985).
9. K.H. Rosen, *Discrete Mathematics*, 2nd edition, McGraw Hill, New York, (1991).
10. T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, (1990).
11. B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, 2nd edition, Prentice Hall, Englewood Cliffs, NJ, (1988).