

A SIMPLE SYSTOLIC ALGORITHM FOR GENERATING COMBINATIONS IN LEXICOGRAPHIC ORDER

IVAN STOJMENOVIC

Computer Science Department, University of Ottawa
Ottawa, Ontario, Canada K1N 9B4

(Received April 1991 and in revised form August 1991)

Abstract—A systolic algorithm is described for generating, in lexicographically ascending order, all combinations of m objects chosen from $\{1, \dots, n\}$. The algorithm is designed to be executed on a linear array of m processors, each having constant size memory, and each being responsible for producing one element of a given combination. There is a constant delay per combination, leading to an $O(C(m, n))$ time solution, where $C(m, n)$ is the total number of combinations. The algorithm is cost optimal (assuming the time to output the combinations is counted), does not deal with very large integers, and is much simpler than the previously known solutions that enjoy same properties.

1. INTRODUCTION

In this paper, we investigate generating the $C(m, n)$ combinations of m objects chosen from the set $\{1, 2, \dots, n\}$, in lexicographically ascending order. We call these combinations (m, n) -combinations.

Various sequential algorithms have been given for this problem [1-8]. Comparisons of combination generation techniques are given in [9-10]. Note that, since each of the $C(m, n) = n!/(m!(n-m)!)$ combinations requires $O(m)$ time to be produced as output, the best possible sequential algorithm runs in $O(mC(m, n))$ time (this is valid if the time to output combinations is taken into account; otherwise there are algorithms for generating combinations without producing them as output, whose running time is $O(C(m, n))$).

Recently, the fast generation of combinations in parallel has been studied in the literature [11-18]. The algorithms in [15-18] satisfy the following criteria:

- (1) The combinations are listed in lexicographic order, i.e., if $A = (a_1, a_2, \dots, a_m)$ and $B = (b_1, b_2, \dots, b_m)$ are two objects, then A precedes B lexicographically, if and only if, for some $j \geq 1$, $a_i = b_i$ when $i < j$, and $a_j < b_j$.
- (2) The algorithm is cost-optimal, i.e., 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.
- (3) The time required by the algorithm between any two consecutive objects 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.
- (4) 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 ($1 \leq i \leq m$) is connected by bidirectional links to its immediate left and right neighbors, $i - 1$ and $i + 1$ (if exist). This model is practical, as it is amenable to VLSI implementation [14]. The linear array operates in systolic fashion: all processors execute the same algorithm simultaneously, with each processor passing data to neighboring ones in a regular rhythmic pattern.

This research is supported by the Natural Science and Engineering Research Council of Canada.

Typeset by $\text{\AA}M\text{-S-TEX}$

- (5) Each processor needs as little memory as possible, preferably a constant number of words, each of $\log n$ bits and hence capable of storing an integer no larger than n . Thus, no processor can store an array of size m , or a counter up to $C(m, n)$.

In [15], it is stated that parallel algorithms [11–14] do not meet all of above criteria. In algorithms [15–18], each processor is responsible for producing one element of each combination. The algorithms by Lin and Tsay [16–18] are rather lengthy and sophisticated, while the algorithm [15] is concise but is based on sophisticated mathematical arguments.

In this paper we describe a new combination generation technique that satisfies all desirable criteria (1)–(5), is simpler than those in [15–18], and is based on a rather apparent mathematical fact.

2. PRELIMINARIES

A (m, n) -combination can be represented as a sequence $c_1 c_2 \dots c_m$ where $1 \leq c_1 < c_2 < \dots < c_m \leq n$.

In [5], a correspondence between (m, n) -combinations and combinations with repetitions of m out of $n - m + 1$ elements (where multiple choice of the same element is possible) is established in the following way. Let $x_i = c_i - i + 1$. Then, $1 \leq x_1 \leq x_2 \leq \dots \leq x_m \leq n - m + 1$, and $x_1 x_2 \dots x_m$ is a combination with repetitions of m out of $n - m + 1$ elements; we call it $(m, n - m + 1)$ - r -combination. The number of (p, q) - r -combinations is $R(p, q) = C(p + q - 1, p)$.

Because of the simple relation, any algorithm that generates $(m, n + m - 1)$ -combinations may be used to generate (m, n) - r -combinations, and vice versa. The only difference is in the output. In particular, the output x_i of a (m, n) - r -combination can be replaced by $x_i + i - 1$ to yield a $(m, n + m - 1)$ -combination, while the generating algorithm remains same. This will be exploited in our algorithm, to make the facts used in generating even more apparent.

3. PARALLEL GENERATION OF COMBINATIONS IN LEXICOGRAPHIC ORDER

The well known sequential algorithm [5] for generating (m, n) - r -combinations determines the next r -combination by a backtrack step that finds an element x_t with the greatest possible index t such that $x_t < n$, therefore, increasable. The element x_t is increased by one, and all following elements x_i for $i > t$ become equal to x_t .

Our algorithm to generate the (m, n) - r -combinations uses a linear array of m processors, indexed 1 to m , and m variables x_1, \dots, x_m , where $1 \leq x_1 \leq x_2 \leq \dots \leq x_m \leq n$. Each processor i is responsible for maintaining x_i by reading only data from processors $i - 1$ and $i + 1$. The processors act in lock-step fashion, and each step produces a new (m, n) - r -combination.

Table 1 shows $(4, 6)$ -combinations (the first column) and the corresponding $(4, 3)$ - r -combinations (the second column).

Processors 1 to $m - 1$ will always produce the same element unless they are advised to change their output. Processor m produces elements between x_{m-1} and n . This is called a run of processor m . For convenience, let $x_0 = n$ and $x_{m+1} = n$. Whenever $x_t = n - 1$ and $x_{t-1} < n - 1$ processor t (such an element is called the turning point) initiates a message that informs processors indexed $t + 1, \dots, m$ that a change in their value is about to happen. Processor i designates registers w_i and s_i for counting waiting time and recording the message, respectively. Each step in the message path corresponds to producing a combination. The message s_i , in fact, contains the value of $x_{t-1} + 1$ since it will become the new value in processors i for $t \leq i \leq m$. The new value of x_i becomes effective $m - i + 2$ steps following the receipt of the message (when w_i reaches 0). In Table 1, the message path is marked in bold. Whenever $x_{i+1} = n$ but $x_i < n$, x_i increases by one in the next step. Such elements are underlined in Table 1. There is enough time for message passing because the message path is $m - t + 1$, which is one less than the length of the current run of processor m ; the current run of processor m consists of $m - t + 2$ combinations ending with $x_t \dots x_m = n - 1, n - 1, \dots, n - 1, n, \dots, n$. When $x_1 = n - 1$ the message is also initiated, for the last time, since at the time of update the new combination is the last combination $nn \dots n$, and all processors terminate simultaneously.

Table 1.

combinations	r-combinations	$s_1 w_1$	$s_2 w_2$	$s_3 w_3$	$s_4 w_4$
1 2 3 4	1 1 1 1	00	00	00	00
1 2 3 5	1 1 1 2	00	00	00	22
1 2 3 6	1 1 <u>1</u> 3	00	00	00	21
1 2 4 5	1 1 2 2	00	00	23	00
1 2 4 6	1 1 <u>2</u> 3	00	00	22	22
1 2 5 6	1 <u>1</u> 3 3	00	00	21	21
1 3 4 5	1 2 2 2	00	24	00	00
1 3 4 6	1 2 <u>2</u> 3	00	23	23	00
1 3 5 6	1 <u>2</u> 3 3	00	22	22	22
1 4 5 6	<u>1</u> 3 3 3	00	21	21	21
2 3 4 5	2 2 2 2	45	00	00	00
2 3 4 6	2 2 <u>2</u> 3	44	44	00	00
2 3 5 6	2 <u>2</u> 3 3	43	43	43	00
2 4 5 6	<u>2</u> 3 3 3	42	42	42	42
3 4 5 6	3 3 3 3	41	41	41	41
		00	00	00	00

The above algorithm can be coded as follows. Each iteration consists of five **if** statements. If given criteria are satisfied, these statements correspond to decreasing waiting time, forwarding the message, increasing the element preceding n , updating all elements following the turning point, and initiating the message from the turning point, respectively.

```

For  $i \leftarrow 1$  to  $m$  do in parallel {
   $x_i \leftarrow 1$ ;  $x_0 \leftarrow n$ ;  $x_{m+1} \leftarrow n$ ;  $w_i \leftarrow 0$ ;  $s_i \leftarrow 0$ ;
  Repeat
    output  $x_i + i - 1$ ;
    if  $w_i \geq 1$ , then  $w_i \leftarrow w_i - 1$ ;
    if  $s_i = 0$  and  $s_{i-1} > 0$ , then  $\{s_i \leftarrow s_{i-1}; w_i \leftarrow m - i + 2\}$ ;
    if  $x_{i+1} = n$  and  $x_i < n$ , then  $x_i \leftarrow x_i + 1$ ;
    if  $w_i = 0$  and  $s_i > 0$  then  $\{x_i \leftarrow s_i; s_i \leftarrow 0\}$ ;
    if  $x_i = n - 1$  and  $x_{i-1} \neq n - 1$  and  $s_i = 0$  then  $\{s_i \leftarrow x_{i-1} + 1; w_i \leftarrow m - i + 2\}$ 
  until  $x_i = n + 1$ 
}

```

The algorithm given above generates all $(m, m + n - 1)$ -combinations. It will generate (m, n) - r -combinations if the current output is simply x_i instead of $x_i + i - 1$. The dynamic of variables s_i and w_i is illustrated in above Table 1 (last four columns).

Summarizing, we obtain the following theorem that clearly follows from the above description.

THEOREM. *The algorithm described above generates all combinations (or combinations with repetitions) of m objects chosen from $\{1, \dots, n\}$ in lexicographic order and with constant delay per combination on a linear array of m processors, thus achieving an optimal cost of $O(mC(m, n))$; furthermore each processor has a memory of constant size and can generate elements without the need to deal with large integers such as $C(m, n)$.*

We now compare our algorithm with the formerly most concise solution [15]. The algorithm [15] has five **if then else** statements that are nested inside each other; our algorithm has five **if**

then statements that follow each other in a sequence. In [15], each processor needs five variables D_i , r_i , rl_i , dl_i , and $d2_i$; plus three references x , y , xy to variables shared with neighboring processors; our algorithm uses three variables x_i , w_i and s_i per processor plus one more variable that is sufficient to communicate with neighbors (in both algorithms each processor reads three data from its neighbors). The number of assignment statements inside the inner loop in [15] is 12 versus 8 in our solution. Finally, [15] proved two mathematical lemmas to support the correctness of the algorithm while we do not need any (a simple map to the case of combinations with repetitions suffices).

4. CONCLUSION

We derived a simple cost-optimal algorithm for generating (m, n) -combinations on a linear array of processors. The algorithm uses m processors and produces combinations in constant time per combination.

As pointed out in [15], a combination generation algorithm can be made adaptive, i.e., to run on a linear array consisting of an arbitrary number k of processors. If $k < n$ then each of k processors can simulate the work of n/k processors. This will obviously require $O(n/k)$ memory per processor. If $k > n$ then the processors are divided into k/n groups of n processors each such that each group produces an interval of consecutive combinations. The first and the last combination in each group can be determined in a preprocessing step by applying known unranking function ([19] described a one-to-one function that maps integers between 1 and $C(m, n)$ onto the set of (m, n) -combinations). However, the function involves very large integers. Another scheme that does not deal with large integers and yet divides job evenly onto groups is described in [20]. Therefore we may obtain an adaptive algorithm which, at the same time, keeps all desirable properties (1)–(5).

REFERENCES

1. J. Kurtzberg, Combination (Algorithm 94), *Comm. ACM* 5, 344 (1962).
2. C.J. Misfud, Combination in lexicographic order (Algorithm 154), *Comm. ACM* 6 (3), 103 (1963).
3. P.J. Chase, Combinations of m out of n objects (Algorithm 382), *Comm. ACM* 13 (6), 368 (1970).
4. C.M. Liu and D.T. Tang, Enumerating combinations of m out of n objects (Algorithm 452), *Comm. ACM* 16 (8), 485 (1973).
5. S. Even, *Algorithmic Combinatorics*, Macmillan, New York, (1973).
6. E.M. Reingold, J. Nievergelt and N. Deo, *Combinatorial Algorithms*, Prentice Hall, Englewood Cliffs, New Jersey, (1977).
7. I. Semba, A note on enumerating combinations in lexicographic order, *J. of Inf. Proc.* 4 (1), 35–37 (1981).
8. F. Ruskey, Adjacent interchange generation of combinations, *J. of Algorithms* 9 (2), 162–180 (1988).
9. W.H. Payne and F.M. Ives, Combination generators, *ACM Trans. on Math. Software* 5 (2), 163–172 (1979).
10. S.G. Akl, A comparison of combination generation methods, *ACM Trans. on Math. Software* 7 (1), 42–45 (1981).
11. B. Chan and S.G. Akl, Generating combinations in parallel, *BIT* 26 (1), 2–6 (1986).
12. G.H. Chen and M.S. Chern, Parallel generation of permutations and combinations, *BIT* 26 (3), 277–283 (1986).
13. S.G. Akl, Adaptive and optimal parallel algorithms for enumerating permutations and combinations, *The Comp. J.* 30 (5), 433–436 (1987).
14. S.G. Akl, *The Design and Analysis of Parallel Algorithms*, Prentice Hall, Englewood Cliffs, New Jersey, (1989).
15. S.G. Akl, D. Gries and I. Stojmenovic, An optimal parallel algorithm for generating combinations, *Inf. Process. Lett.* 33 (3), 135–139 (1989/90).
16. C.J. Lin, A parallel algorithm for generating combinations, *Computers and Mathematics with Applications* 17 (12), 1523–1533 (1989).
17. C.J. Lin and J.C. Tsay, A systolic generation of combinations, *BIT* 29, 23–36 (1989).
18. J.C. Tsay and C.J. Lin, A systolic design for generating combinations in lexicographic order, *Parallel Computing* 13, 119–125 (1990).
19. G.D. Knott, A numbering system for combinations, *Comm. ACM* 17 (1), 45–46 (1974).
20. I. Stojmenovic, On random and adaptive parallel generation of combinatorial objects, *Int. J. Computer Mathematics* (to appear).