Malaysian Journal of Fundamental and Applied Sciences Vol.9, No.4 (2013) 191-195

ISSN 1823-626X

Malaysian Journal of Fundamental and Applied Sciences

available online at http://mjfas.ibnusina.utm.my

SEALiP: A Simple and Efficient Algorithm for Listing Permutation via Starter Set Method

Haslinda Ibrahim*, Lugean Zake, Zurni Omar and Sharmila Karim

School of Quantitative Sciences, College of Arts and Sciences, Universiti Utara Malaysia, 06010 Sintok, Kedah Malaysia

Received 20 January 2013, Revised 1 April 2013, Accepted 25 April 2013, Available online 22 May 2013

ABSTRACT

Algorithm for listing permutations for *n* elements is an arduous task. This paper attempts to introduce a novel method for generating permutations. The fundamental concept for this method is to seek a starter set to begin with as an initial set to generate all distinct permutations. In order to demonstrate the algorithm, we are keen to list the permutations with the special references for cases of three and four objects. Based on this algorithm, a new method for listing permutations is developed and analyzed. This new permutation method will be compared with the existing lexicographic method. The results revealed that this new method is more efficient in terms of computation time.

| Permutation| Starter set| Lexicographic|

http://dx.doi.org/10.11113/mjfas.v9n4.108

1. INTRODUCTION

Permutation algorithms are one of the component from which all field of computer science grow. Permutation is the different arrangements that can be made with a given number of things taking some or all of them at a time. The applications of permutation are used in various fields such as mathematics, group theory, statistics, and computing. In general, permutation algorithms have received much attention in the literature [1,3,8-10]. Permutation algorithms can be classified into two main categories namely algorithms based on non-exchanged (random) and algorithms based on exchanged (lexicographic) [8]. In addition to the random and lexicographic algorithms for generating permutation, there are other permutations algorithms such as: using sequences of permutation, choosing any elements of permutations, and using permutation matrices [1,5,6]. Based on the methods mentioned above, we found that the most challenging task dealing with permutation is when *n* is large enough. For example, the value of $n = 10!$ is 3628800. However, in real application problems we usually are dealing with more than ten or even hundred items. Thus, to design a simple and efficient algorithm for permutation generation is fundamental and a practical issue.

Most of the existing algorithms concentrate on listing all *n!* permutations. The advantage of this proposed algorithm is to produce some sets to begin with and these sets finally generate all the *n!* permutations. However, a work on generating starter set as basis to list *n!* was proposed in 2010 [2]. Thus this paper aims to provide a method to list *n!* permutations based on starter set that will be discussed in section 2.3. The present paper is divided into two parts. In the first part various categories of permutation methods are reviewed whilst the second part focuses on the computation time. The comparison between proposed method and classical method such as lexicographic will be carried out at the end of the study.

2. PERMUTATION METHODS

This section will present discussion on permutation algorithms based on non-exchanged (random) and exchanged (lexicographic). Then we will provide another permutation algorithm based on starter set.

2.1 Permutation Algorithms Based on Non-Exchanged (random)

In this section, we are concerned on a variety of algorithms which not based on exchanges. These algorithms are for generating a random permutation that may be used to generate random cycles of length *n* instead. So some algorithms use random cyclic rotations to obtain the *n*! distinct permutations. The basic idea for generating permutation at random is to generate at random one of the *n*! sequences of integers $k_1, k_2, ..., k_n$ satisfying $0 \leq k_i < i$ (since k_i is always zero, it may be omitted), and to convert it to a permutation through a bijective correspondence. For the latter correspondence, one could interpret the (reverse) sequence as a Lehmer code. The above algorithm for generating permutation was first published in 1938 by Fisher and Yates [4].

*Corresponding author. E-mail: linda@uum.edu.my

The basic algorithm given for generating a random permutation of the numbers 1- *n* goes as follows:

Step 1: Write down the numbers from 1 to *n*.
Step 2: Pick a random number *k* between one and the number of unstruck numbers remaining (inclusive).
Step 3: Counting from the low end, strike out the *k*th number not yet struck out, and write it down elsewhere.
Step 4: Repeat from step 2 until all the numbers have been struck out.
Step 5: The sequence of numbers written down in step 3 is now a random permutation of the original numbers.

We show an example of how the steps for this algorithm work. To produce 4! permutations from (1 2 3 4), it has 24 permutations as shown below:
(1 2 3 4), (2 3 1 4), (3 1 2 4), (1 3 2 4),
(2 1 3 4), (3 2 1 4), (2 3 4 1), (3 1 4 2),
(1 2 4 3), (3 2 4 1), (1 3 4 2), (2 1 4 3),
(3 4 1 2), (1 4 2 3), (2 4 3 1), (2 4 1 3),
(3 4 2 1), (1 4 3 2), (4 1 2 3), (4 2 3 1),
(4 3 1 2), (4 1 3 2), (4 2 1 3), (4 3 2 1).

2.2 Permutation Algorithms Based on Exchanged (Lexicographic)

Permutation algorithms that based on exchanged *n*! permutations of *n* are usually obtained by a series of (*n*!–1) exchanges [8]. Thus, this algorithm when applied to permutations, the lexicographic order increases in numerical order (or equivalently, alphabetic order for lists of symbols). There are many ways to systematically generate permutations of a given sequence. One classical algorithm, which is both simple and flexible, is based on finding the next permutation is lexicographic ordering. This technique generates the distinct permutations by converting to permutations, one starts by sorting the sequence in increasing order (which gives its lexicographically minimal permutation), and then repeats to the next number.

The following steps show how to generate permutation lexicographically and transform the given permutation:

Step 1: Find the largest index *j* such that $a[j] < a[j+1]$. If no such index exists, the permutation is the last permutation.
Step 2: Find the largest index *l* such that $a[j] < a[l]$. If *l* exists and satisfies $j < l$, $j+1$ is such an index.
Step 3: Swap $a[j]$ with $a[l]$.
Step 4: Reverse the sequence from $a[j+1]$ up to the final element $a[n]$.

Strictly speaking, if the *n* items going through permutations are ordered by a precedence relation "<", then permutations $\pi_a = (\pi_{a,1}\ \pi_{a,2} \ldots \pi_{a,n})$ precedes permutation $\pi_b = (\pi_{b,1}\ \pi_{b,2} \ldots \pi_{b,n})$ if and only if, for some $i \leq 1$, we have $\pi_{a,j} = \pi_{b,j}$ for all $j < i$ and $\pi_{a,i} = \pi_{b,i}$.

For example, to generate all permutations of integers (1 2 3 4) for 4 elements, a lexicographic algorithm produces permutations in order just like dictionaries contain. The lexicographic order of 4! permutations of four distinct items {1 2 3 4} is (1 2 3 4) < (1 2 4 3) < (1 3 2 4) < (1 3 4 2) < (1 4 2 3) < (1 4 3 2) < (2 1 3 4) < (2 1 4 3) < ... < (4 1 2 3) < (4 1 3 2) < 4 2 1 3) < (4 2 3 1) <(4 3 1 2) < (4 3 2 1).
Here are the permutations based on the steps explained above:
(1 2 3 4), (1 2 4 3), (1 3 2 4), (1 3 4 2),
(1 4 2 3), (1 4 3 2), (2 1 3 4), (2 1 4 3),
(2 3 1 4), (2 3 4 1), (2 4 1 3), (2 4 3 1),
(3 1 2 4), (3 1 4 2), (3 2 1 4), (3 2 4 1),
(3 4 1 2), (3 4 2 1), (4 1 2 3), (4 1 3 2),
(4 2 1 3), (4 2 3 1), (4 3 1 2), (4 3 2 1).
The lexicographic algorithm requires more effort and time to generate all permutations as the size of the elements get larger [1,6].

2.3 Permutation Algorithms Based on Starter sets

A new algorithm for generating permutations by choosing any elements of permutations was found by Rohl [6]. The purpose of this algorithm is to give a simple, general algorithm which will produce arrangements of *n* symbols taken *r* at a time where the symbols may or may not be distinct. Various procedures based on the algorithm are presented, some producing the arrangements in lexicographical order, some not. The algorithm is easy to use as the basis for a solution to some combination problems. But we noticed that no timing information had been provided. The reason is that this algorithm is not designed specifically for its execution time but for its adaptability specifically to a wide variety of problems.

An algorithm for generating permutations by using permutation matrices was proposed by Mani [5]. The algorithm is proposed in an original technique, by using a cyclic of permutation matrices' rotations. The only restrictions on the implementation language are the availability of a function and efficient pointer manipulations.

This section will examine the other algorithm for generating all permutations by using combinatorial approach that was developed by Ibrahim et al. [2]. Notably, this new algorithm for listing all permutations for *n* elements was developed based on distinct starter sets. Hence, once the starter sets are obtained, each starter set is then cycled to obtain the first half of the distinct permutations. The complete list of permutations is achieved by reversing the order of the first half of the permutation. This algorithm has advantages over the other algorithms due to its simplicity and ease of use.

The followings definitions are needed in the following sequel for algorithm development.

Definition 2.3.1. *A starter set is a set that is used as a basis to enumerate other permutations.*

Definition 2.3.2. *A reversed (inverse) set is a set that is produced by reversing the order of permutation set.*

Definition 2.3.3. *An equivalence starter set is a set that can produce the same permutation from any other starter set.*

We will provide two different cases to generate 3! and 4! permutations before we develop the general algorithm for this method.

Case 1: 3! = 6 distinct permutations. Here we will provide steps to list all six distinct permutations.

Step 1: Fix one element from integers {1, 2, 3} for example "1", we have the following different permutations:

{1, 2, 3}, {1, 3, 2}

Step 2: Determine all equivalent starter sets

{1, 2, 3} ≅ {1, 3, 2)

Step 3: Delete the equivalent starter set, then we have only the following starter sets.

{1, 2, 3}

Step 4: Generate distinct permutations from this starter set.

1	2	3
2	3	1
3	1	2

Step 5: Find the reverse of the above permutations.

Starter set (1, 2, 3)		
1	2	3
2	3	1
3	1	2

Reverse		
3	2	1
1	3	2
2	1	3

Now we have six different permutations from the above steps.

{1, 2, 3}, {3, 2, 1}
{2, 3, 1}, {1, 3, 2}
{3, 1, 2}, {2, 1, 3}

Case 2: 4! = 24 distinct permutations. For example, to generate all permutations from the integers {1, 2, 3, 4} for 4 elements, the following steps are developed:

Step 1: Fix one element from the integers {1, 2, 3, 4}, for example "1", we have the following starter sets:

{1, 2, 3, 4}, {1, 2, 4, 3}, {1, 3, 2, 4},
{1, 3, 4, 2}, {1, 4, 2, 3}, {1, 4, 3, 2}

Step 2: Determine all equivalent starters sets and starter sets in the following:

{1, 2, 3, 4}≅{1, 4, 3, 2}
{1, 2, 4, 3}≅{1, 3, 4, 2}
{1, 3, 2, 4}≅{1, 4, 2, 3}

Step 3: Delete the equivalence starter sets, then we produce the following starter sets.

{1, 2, 3, 4}, {1, 2, 4, 3}, {1, 3, 2, 4}

Step 4: Generate all permutations from these starters set {(1, 2, 3, 4), (1, 2, 4, 3), (1, 3 ,2 ,4)}. All permutations are generated cyclically for every permutation as the following:

starter set {1, 2, 3, 4}

1	2	3	4
2	3	4	1
3	4	1	3
4	1	3	2

starter set {1, 2, 4, 3}

1	2	4	3
2	4	3	1
4	3	1	2
3	1	2	4

Starter set {1, 3, 2, 4}

1	3	2	4
3	2	4	1
2	4	1	3
4	1	3	2

Step 5: Find the inverses (reverse) of the above permutations,

Starter set (1, 2, 3, 4)			
1	2	3	4
2	3	4	1
3	4	1	2
4	1	2	3

Reverse			
4	3	2	1
1	4	3	2
2	1	4	3
3	2	1	4

Starter set (1, 2, 3, 4)			
1	2	4	3
2	4	3	1
4	3	1	2
3	1	2	4

Reverse			
3	4	2	1
1	3	4	2
2	1	3	4
4	2	1	3

Starter set (1, 2, 3, 4)			
1	3	2	4
3	2	4	1
2	4	1	3
4	1	3	2

Reverse			
4	2	3	1
1	4	2	3
3	1	4	2
2	3	1	1

From step 5 we can have all 24 distinct permutations for 4!.

For case 5! we have 120 distinct permutations. By employing this method we need to generate only 12 starter sets and apply the same steps as mentioned above to list all the permutations. Thus we can summarize the algorithms to generate permutation based on the starter sets as listed in the following.

Permutation algorithm based on starter set	
Step 1:	Fix one element
Step 2:	Determine other starter set
Step 3:	Determine the equivalence starter set and eliminate this set.
Step 4:	Repeat step 2-3 to produce all distinct starters.
	Cycle each starter to get *n*!/2 permutations
Step 5:	Reverse the order of each *n*!/2 permutation to get
Step 6:	another *n*!/2 permutations
	List all *n*! permutations
Step 7:	

3. PERFORMANCE OF PERMUTATION ALGORITHMS

Several algorithms to generate permutations have been presented such as lexicographic algorithm and random cyclic algorithm. All these algorithms have their strengths and weaknesses from the viewpoint of the efficiency evaluation. For example, lexicographic algorithm uses more execution time to make the strictly-ordered permutations. This algorithm happens to repeat the same permutation again, and the repeated permutation is actually not needed [1,6,8]. Random cyclic algorithm has the same problem in terms of execution time to generate permutation [5,8]. The most familiar and well-known algorithm used to find all permutations is called a lexicographic algorithm. A lexicographic algorithm as we mentioned earlier requires more time to find all permutations [1,6,8].

The proposed permutation algorithm based on starter set is as an endeavour to improve the performance of the algorithm to generate permutation [2]. In this section we compare the proposed permutation algorithm based on starter set with the lexicographic permutation algorithm.

The efficiency of both algorithms can be computed in terms of the execution time. We have used permutation algorithm for starter set as described in section 2.3 and lexicographic algorithm as discussed in section 2.2. The pseudo code for both algorithms can be found in [7]. Table 1 enumerates algorithms for lexicographic and starter set for $n = 4$ elements. While the efficiency of both algorithms was computed in terms of the execution time. Based on several tests conducted on different order of n elements, the results stated in Table 2 show the value in seconds for the execution time of the lexicographic algorithm and starter set algorithm. We can see that results of the execution time of the starter set algorithm are less than the lexicographic algorithm (see Fig.1). This computation time have been implemented by using MATLAB version 7 and were performed in laptop Acer 5635Z (Pentium 800 MHz CPU, 3 MB of main memory).

Table 1. Lexicographic order and starter set order for n=4!

Lexicographic				Starter set (A)				Reverse from A			
1	2	3	4	1	2	3	4	4	3	2	1
1	2	4	3	2	3	4	1	1	4	3	2
1	3	2	4	3	4	1	2	2	1	4	3
1	3	4	2	4	1	2	3	3	2	1	4
1	4	2	3	1	2	4	3	3	4	2	1
1	4	3	2	2	4	3	1	1	3	4	2
2	1	3	4	4	3	1	2	2	1	3	4
2	1	4	3	3	1	2	4	4	2	1	3
2	3	1	4	1	3	2	4	4	2	3	1
2	3	4	1	3	2	4	1	1	4	2	3
2	4	1	3	2	4	1	3	3	1	4	2
2	4	3	1	4	1	3	2	2	3	1	4
3	1	4	2								
3	1	4	2								
3	2	1	4								
3	2	4	1								
3	4	1	2								
3	4	2	1								
4	1	2	3								
4	1	3	2								
4	2	1	3								
4	2	3	1								
4	3	1	2								
4	3	2	1								

Table 2. The execution time (in seconds) for Lexicographic and starter set algorithm .

n	Lexicographic algorithm	Starter set algorithm
2	0.0009	0.0001
3	0.0160	0.0010
4	0.0360	0.0160
5	0.0940	0.0310
6	0.1410	0.1070
7	0.4220	0.1820
8	3.6400	2.5690

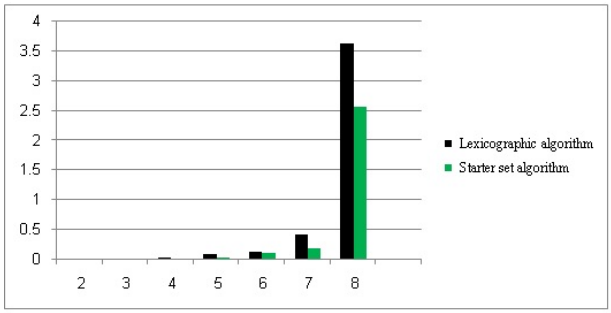

Fig. 1 Comparison of the execution times between the starter set algorithm and lexicographic.

4. CONCLUSION

We have presented a new method to list $n!$ permutations based on starter set. The results show that this new method has less computation time. Based on the discussion on this study these three following questions arise:

- Could we construct isomorphism classes to classify equivalence starter sets? We expect to use one-factorization or decomposition of compete graph (K_n).
- Could we employ these new algorithms (permutation based on starter set) in other fields, for example in cryptography (security), image processing?
- Could we determine the starter set without fixing any positions?

ACKNOWLEDGEMENT

This research was supported by Malaysian Ministry of Education under Fundamental Research Grant Scheme (FRGS) vot 11768.

REFERENCES

[1] C.T. Fike, The Computer Journal, 18 (1) (1975) 21-22.
[2] H. Ibrahim, Z. Omar, & A. M. Rohni, Modern Applied Science, 4 (2) (2010) 89-93.
[3] D. E. Knuth, The Art of Computer Programming, 4 (2005) 1-26.
[4] D. H. Lehmer, Proceedings of Symposium Applied Mathemaitcs, Combinatorial Analysis, (1960) 179-193.
[5] G. I. Mani, & G. Iye, Permutation generation using matrices. Retrieved on November 01, 2010, from http://www.drdobbs.com/184409671.
[6] J. S. Rohl, The Computer Journal, 21 (4) (1978) 302-305.
[7] L.M. Zake, New Algorithm for Determinant of Matrices Via Combinatorial Approach. (2011), Unpublished Thesis. Universiti Utara Malaysia
[8] R. Sedgewick, Journal Computer Science of Applied Mathematics, 9 (1977) 137-164.
[9] R.J. Ord-Smith, Part 1, The Computer Journal, 13 (1970) 152.
[10] R.J. Ord-Smith, Part 2, The Computer Journal, 15 (1970) 136.