

The Permutation Algorithm for Non-Sparse Matrix Determinant in Symbolic Computation

Dong Won Shin

*Control Information Systems Lab., School of Electrical Engr.
and Computer Science, Seoul National University, Seoul, 151-742, Korea*

Abstract: This paper considers the symbolic determinant computation. The aptness of permutation method for symbolic determinant is justified and a new efficient algorithm is proposed to implement the permutation method. Exploiting the fact computers perform arithmetic operations fast, the proposed algorithm generate all the permutations without manipulation on additional 2nd array or redundancies found in the existing permutation generating algorithms. The proposed algorithm is very efficient for computing symbolic determinant of a non-sparse matrix whose order is below 13.

Keywords: Permutation algorithm; Scramble; Matrix determinant; Symbolic computation;

1 INTRODUCTION

Several methods for computing determinant have been known in the literature [1, 2, 3]. Using one of them, the determinant of a matrix with various size can be computed undoubtedly only if sufficient time is given. With programmed computers, the speed of the computation has less importance in numerical determinant, especially when the order of matrix being calculated is less than thousands.

However, in symbolic computation, the calculation of a determinant is not that straightforward problem as it seems. Even with the powerful computers, it is still a troublesome task with many constraints and obstacles. If the symbolic matrix is full or non-sparse, computing its determinant becomes a difficult task both for human and for computer. Think over the fact a symbolic determinant of 10th order matrix with distinct nonzero entries is a polynomial with $10! = 3,628,800$ terms and 10 variables in each term. 36,288,000 symbols are required to express it explicitly. Even though the symbols for signs(plus/minus and multiplication) are not included in the figure, it is too enormous to deal with.

The permutation method is regarded as the most inadequate method with pencil and paper both for numerical and for symbolic because it is one of the NP-complete problems [4]. But it proves to be the most proper method for symbolic determinant. It produces

the determinant whose product terms have the entries in as-is form and, when implemented with computer, the speed of this method is decent for practical use. The key point is generating all the permutations from the column indices. Fortunately, computer can perform such task better and faster than human does. This paper justifies the permutation method for symbolic determinant computation and presents a new algorithm, *Partial Reversion Algorithm*, to implement the permutation method.

The following main part consists of 4 sections. In Section 2, Gauss elimination and cofactor method are reviewed with the reasons those methods are inadequate for symbolic determinant. In Section 3, the permutation method is analyzed and justified as an adequate method for symbolic determinant. In Section 4, the existing algorithms that generate permutations are analyzed. In Section 5, *Partial Reversion Algorithm*, a new algorithm of permutation generation is proposed and analyzed by comparisons with the existing algorithms. Finally, Section 6 concludes this paper.

2 REVIEWS ON GAUSS ELIMINATION AND COFACTOR METHOD

Unlike numerical determinant, symbolic determinant is not a numerical value but a symbolic expression, which

makes a huge difference. In fact, symbolic determinant is “fabricated”, while numerical determinant is simply calculated. Computing symbolic determinant is not a sequential calculation process of values but elaborately organized manipulations on symbols. Thus, the selection of an efficient algorithm becomes very important. In viewpoint of this difference, two methods for determinant computation are reviewed in this section.

The execution time function is expressed in $T(n^2)$ because the number of entries of n th order square matrix is n^2 . Since the algorithm is a matter of concern, a couple of assumptions are introduced for simplicity. First assumption is that all the entries of matrix in initial state are the simplest monomials. Second one is that copying an entry takes a fixed amount of time regardless of its actual length and complexity.

2.1 Gauss Elimination Method

Gauss elimination method is efficient and very fast for numerical determinant. As this method applies the operations directly on the original matrix, only the number of arithmetic operations counts for the evaluation of algorithm performance. The execution time function for numerical determinant is $T(n^2) = \frac{2}{3}n^3$ (precisely, $T(n^2) = \frac{4n^3+3n^2-7n}{6}$) [5]. This is valid for symbolic determinant too because the arithmetic operations in numerical case are exactly matched to the memory allocating processes for new symbols in symbolic case. However, the method reveals the critical defect when applied to symbolic computation in practice. The method is not effective in that it produces fractions inherently. Even for the simplest symbolic matrix, the determinant obtained from Gauss elimination has many unnecessary fractions. When Gauss elimination is applied, a 3th order non-sparse matrix \mathbf{G} ,

$$\mathbf{G} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \quad (1)$$

is changed into the upper diagonal matrix,

$$\begin{bmatrix} a & b & c \\ 0 & e - \frac{bd}{a} & f - \frac{cd}{a} \\ 0 & 0 & i - \frac{cg}{a} - \frac{(f - \frac{cd}{a})(h - \frac{bg}{a})}{e - \frac{bd}{a}} \end{bmatrix}. \quad (2)$$

Its determinant is just the product of diagonal entries.

$$\det \mathbf{G} = a \left(e - \frac{bd}{a} \right) \left(i - \frac{cg}{a} - \frac{(f - \frac{cd}{a})(h - \frac{bg}{a})}{e - \frac{bd}{a}} \right) \quad (3)$$

It is true that (3) is correct and identical with the expected expression,

$$\det \mathbf{G} = aei - bdi + cdh - afh + bfg - ceg. \quad (4)$$

However, the additional work is inevitable to arrange (3) into (4). The additional ‘simplifying’ work is not simple at all especially for computers. In fact, the work becomes a considerable load on the program performance. This is the reason Gauss elimination is not adequate for symbolic determinant. It does not make sense to use the method that returns the complex expression even for the simplest inputs. Apart from the other factors to be considered in implementation of this method, this fraction-making characteristic alone can exclude Gauss elimination from candidates list for symbolic determinant computation methods.

2.2 Cofactor Method

The cofactor method also known as ‘expansion by the entries of column’ is feasible for symbolic determinant. Unlike Gauss elimination method, this method combines the entries in their as-is form into the product. As a result, if all the entries are monomials, determinant from this method takes the form of organized sum-of-products. The problem of this method, however, comes from its extremely low speed. This method is useful when computing symbolic determinant of relatively low order matrix with pencil and paper. But it turns out to be inefficient as the order of matrix grows higher. The mathematical definition of this method is $\det \mathbf{C} = \sum_{i=1}^n a_{ij} C_{ij}$, where $C_{ij} = (-1)^{i+j} \det \mathbf{C}(i|j)$ is the cofactor of a_{ij} and $\mathbf{C}(i|j)$ is the matrix of order $n-1$ obtained from \mathbf{C} by deleting the i th row and j th column of \mathbf{C} [1]. The strategy of this method is the typical “divide-and-conquer.” A determinant of order n matrix is divided into n multiplications between $(n-1)$ th order sub-matrix determinants and the entries of a specific column. This process is iterated recursively. On the contrary to Gauss elimination, most of the execution time is spent in allocating memory to the entries of sub-matrices. The execution time function is deducted as follows.

$$T(n^2) = n(n-1)^2 T((n-1)^2) + n \quad (5)$$

$$= n(n-1)^2 \{ (n-1)(n-2)^2 T((n-2)^2) + n-1 \} + n$$

$$\vdots$$

$$= n(n-1)^3 \dots 3^3 2^2 T(2^2) + n \sum_{p=1}^{n-2} \frac{((n-1)!)^3}{((n-k)!)^3} \quad (6)$$

$$= n(n-1)^3 \dots 2^3 1^2 T(1^2) + n \sum_{p=1}^{n-1} \frac{((n-1)!)^3}{((n-k)!)^3} \quad (7)$$

$$= n((n-1)!)^3 T(1^2) + n \sum_{p=1}^{n-1} \frac{((n-1)!)^3}{((n-k)!)^3} \quad (8)$$

The most influential term, $(n-1)^2$ in (5) accounts for memory allocation of sub-matrices($(n-1)^2$ entries). Notice the last n term in (5) is actually supposed to be $3n$ implying the generation of the coefficients composed of plus/minus sign, the entry from the specified column and multiplication sign. Because these 3 elements go with just like a cluster, the constant coefficient 3 can be omitted for simplicity. Assuming the base case $T(1^2)$ is constant so that it can be ignored, the most significant term is obtained as the execution time function,

$$T(n^2) = 2n((n-1)!)^3. \quad (9)$$

The base case can be set higher in the consecutive recursion formula starting from (5). When $T(p^2)$ is the base case, the time function becomes

$$T(n^2) = \frac{p^2 + 1}{p^3} (n((n-1)!)^3). \quad (10)$$

Between (9) and (10), only coefficients differ and the most influential term, $n((n-1)!)^3$ remains same. It is obvious that the speed of the algorithm is extremely low for relatively large n because the growth rate of $((n-1)!)^3$ is seriously huge. Due to its low speed, this method becomes the worst one among the methods feasible for symbolic determinant.

3 PERMUTATION METHOD

The matrix determinant is defined by the permutations from the column indices. Thus, the permutation method obtains the determinant from its definition by generating all the permutations and copying the entries according to each permutation. Each permutation can be also used to determine a sign of the product. The problem is that it is not easy to get all the permutations from the set of integers. The generation of permutations is actually one of the NP-complete problems [4]. Thus, it is natural that the execution time function of this method has the term increasing exponentially as the number of entries increases. The permutation method produces a determinant without additional fractions just like cofactor method does. In fact, if cofactor method is implemented in non-recursive way, no substantial difference will be found between permutation method and cofactor method. Consider the 3th order matrix \mathbf{P} ,

$$\mathbf{P} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}. \quad (11)$$

To get the determinant of \mathbf{P} , the entire permutation set of column indices is necessary. In other words, (123) have to be scrambled. For the matrix \mathbf{P} ,

(123 132 231 213 312 321) forms the entire set of $3!=6$ permutations and the permutation set corresponds to $(a_{11}a_{22}a_{33} \ a_{11}a_{23}a_{32} \ \cdots \ a_{13}a_{21}a_{32} \ a_{13}a_{22}a_{31})$. The plus/minus signs of those permutations are determined by the number of interchanges required to transform the permutation into the original indices in order. For example, 321 returns to 123 by unscrambling steps of 3 interchanges shown below.

$$321 \longrightarrow 312 \longrightarrow 132 \longrightarrow 123$$

Thus, the sign for permutation 321 is minus and the product is determined as $-a_{13}a_{22}a_{31}$. With the sign consideration like this, the determinant of \mathbf{P}

$$\det \mathbf{P} = a_{11}a_{22}a_{33} - a_{11}a_{23}a_{32} + a_{12}a_{23}a_{31} - a_{12}a_{21}a_{33} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31} \quad (12)$$

can be obtained. As mentioned, this method is based on the definition of determinant. The determinant of n th order matrix \mathbf{A} is

$$\det \mathbf{A} = \sum_{j_1, \dots, j_n} \varepsilon_{j_1 j_2 \dots j_n} a_{1j_1} a_{2j_2} \cdots a_{nj_n}, \quad (13)$$

where $\varepsilon_{j_1 j_2 \dots j_n} = 1$ if unscrambling the permutation $(j_1 j_2 \cdots j_n)$ requires an even number of interchanges between adjacent integers and $\varepsilon_{j_1 j_2 \dots j_n} = -1$ if unscrambling the permutation requires an odd number of interchanges [1]. The column index numbers ranging from 1 to n are permuted or scrambled for determinant computation. As 3th order matrix is formed by $3! = 6$ products of 3 variables, the determinant of n th order matrix is generally formed by $n!$ products of n variables corresponding to each permutation [5]. Consequently, all the permutation methods have to copy the entries $n \cdot n!$ times to form all the products and must determine $n!$ signs for each of them. The fact is reflected on the time complexity and all the algorithms implementing the permutation method has the term $n \cdot n!$ for the most significant term. Therefore, the implementation of this method must not use the permutation generating algorithm whose execution time function has more influential term than $n \cdot n!$. The time complexity of the entire determinant computation has to be prevented from an increase resulting from permutation generating at all cost. Then, the remaining part is speeding-up the algorithm by lessening coefficients in the execution time function.

The permutation method takes moderate position between speed and simplicity. It produces the determinant in which the entries exist in their as-is form unlike Gauss elimination. Also, its speed is higher than that of cofactor method because the growth rate of $n \cdot n!$ is smaller than that of $n((n-1)!)^3$. It is most adequate

order n	Gauss $O(n^3)$	Cofactor $O(n((n-1)!)^3)$	Permutation $O(n \cdot n!)$
1	1	1	1
2	8	2	4
3	27	24	18
4	64	864	96
5	125	69120	600
6	216	10368000	4320
7	343	2612736000	35280
8	512	1024192512000	322560
9	729	589934886912000	3265920
10	1000	477847258398720000	36288000

Table 1: The execution time transitions

among the methods for computing symbolic determinant. Table 1 shows how the execution time increases as the order of matrix increases from 1 to 10. The performance of each method is well summarized in the table. Notice the coefficients of time functions are omitted for simplicity. What to be heeded in Table 1 is not the absolute values but the transition pattern among those values. When Gauss elimination method is excluded due to its fraction problem, the second best permutation method can be selected as the proper method for symbolic determinant. The details of the permutation method are presented throughout the following sections.

4 DEMERITS OF EXISTING ALGORITHMS FOR PERMUTATION GENERATING

Once the permutation set becomes available, the determinant can be obtained by iterating the entry-copying process according to each permutation with sign consideration. However, storing $n \cdot n!$ integers for $n!$ permutations of n variables is not efficient regarding memory management especially for large n . The alternative for memory saving is copying entries after each process of permuting indices and scrambling the current indices to get the next permutation. With this strategy, the prototype of execution time function,

$$T(n^2) = P(\text{algorithm}) + n!S(\text{algorithm}) + (n-1) \cdot n! \quad (14)$$

is determined, where $P(\text{algorithm})$ and $S(\text{algorithm})$ are also the time functions that have dependencies on the algorithm used to generate the permutations and $(n-1) \cdot n!$ is the time for generating multiplication signs between entries. $P(\text{algorithm})$ implies the time required to get all the permutations from column indices and to copy the entries. Similarly, $S(\text{algorithm})$

implies the time for determining plus/minus sign for a product. Therefore, selecting/devising fast and simple algorithms for those two time functions implies the successful implementation of this method.

Several algorithms that generates permutations have existed for centuries [6]. Some of those are lexicographic algorithm(denoted Algorithm L), cyclic shift algorithm(Algorithm C), plain changes algorithm(Algorithm P), plain change transition algorithm(Algorithm T), Ehrlich swaps(Algorithm E) and so forth [6]. All these algorithm have both merits and demerits. In viewpoint of the efficiency evaluation, these algorithms can be categorized into 2 groups. One is the group shows redundancy and the other is the group uses the additional second array/table that has to be manipulated.

Algorithm L and Algorithm C fall within the former which have redundancies. Algorithm L produces permutations in order just like dictionaries contain words. This sorted permutations are indispensable for some work occasionally, however, symbolic determinant does not require the permutations in exact order. In viewpoint of symbolic determinant, Algorithm L wastes some execution time to make the strictly-ordered permutations. With Algorithm L, re-visiting same permutation happens quite often and the re-visited permutation has to be discarded in vain. For example, Algorithm L visits the permutations from (1234) in the pattern like

1234 1243 (1342) 1324 1342 (1432) 1423 1432 (2431) 2134...

where parenthesized permutations are discarded. Algorithm C has the same problem. For example, to produce $4!$ permutations from (1234), it has to discard 9 permutations as shown below.

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

On the other hand, Algorithm P, Algorithm T and Algorithm E belong to the latter which have the auxiliary array/table that has to be manipulated. Algorithm P and Algorithm E use 2 auxiliary arrays of size n respectively to generate all the permutations. The values within those 2 arrays have to be manipulated and the manipulation adds to the execution time. Algorithm T, one of the variants of Algorithm P, does not use auxiliary arrays but instead it makes a table of size $n! - 1$ before it actually permutes. Algorithm T uses 5 additional integer variables to make the table and this table of size $n! - 1$ works as a overhead to the algorithm performance. These algorithms use the additional spaces.

Algorithm T, in particular, uses relatively large space of $n! - 1$ size array and has to keep it until all the products of entries are made. Though the permutations does not need to be in order, the set of permutations made both by Algorithm P and by Algorithm T are too much scattered. Table 2 shows the output of Algorithm P/T and Algorithm E for (1234).

Algorithm P/T	n	Algorithm E
1234	1	1234
1243	2	2134
1423	3	3124
4123	4	1324
4132	5	2314
1432	6	3214
1342	7	4213
1324	8	1243
3124	9	2143
3142	10	4123
3412	11	1423
4312	12	2413
4321	13	3412
3421	14	4312
3241	15	1342
3214	16	3142
2314	17	4132
2341	18	1432
2431	19	2431
4231	20	3421
4213	21	4321
2413	22	2341
2143	23	3241
2134	24	4231

Table 2: The permutations from Algorithm P/T and Algorithm E

5 PARTIAL REVERSION ALGORITHM

The permutation generating algorithms mentioned above are not too bad for symbolic determinant. However, a new algorithm is proposed as an endeavor to improve the performance. This algorithm is elaborately devised to avoid two demerits found in the existing algorithms. Firstly, this algorithm does not re-visit permutations. In other words, no redundant permutation is generated and no permutation is discarded. It can visit all the permutations exactly once. Secondly, though this algorithm requires 2 arrays of size n , one of which is not manipulated but fixed as reference table. This saves the execution time for scrambling. Also, the permutations from this algorithm will be in order up to some degree, which adds to the merits of this method.

5.1 The Framework of Partial Reversion Algorithm

This algorithm exploits the fact computer performs arithmetic operations fast and stores integer values with ease. Precisely speaking, 2 integer variables and a boolean variable are required for *Partial Reversion Algorithm*. One of two integer variables is used as a

counter which counts the number of the permutations generated to the point and the other variable is the pointing number for the column indices array. Boolean variable is used for the sign determination.

Unlike Algorithm P/T, this algorithm does not apply interchanges on permutations. This algorithm reverses a portion of array like Algorithm L does. Also, the condition evaluations for reversing is realized by a single counter variable. For the sign determination, this algorithm evaluates the pointing number for the column indices array. The reason *Partial Reversion Algorithm* becomes feasible is that arithmetic operations can be performed very fast with computers. Frequent modulus operations and counting the number of permutations which reaches to $n!$ are not easy task for human. But it is not the problem at all for computers. For this algorithm to work, the values of $k!$ for all k from 1 to n should be initialized and be stored in an array in advance. This is not a big overhead because the factorial values are constants that those can be directly initialized like ‘factorial_array[3]=6’ in program codes. The array has to be initialized just once and, as the reference table, the values within the factorial array will not be changed. *Partial Reversion Algorithm* works by following the steps described in Table 3, where n is the order of matrix, k is index number ($1 \leq k \leq n$) and j is the index-pointing number from which reversing starts.

R1. [Initialize.] Make an array F of size $n + 1$ and set $F[0] \leftarrow 0$, $F[k] \leftarrow k!$ for all k . Make an array P of size $n + 1$ and set $P[0] \leftarrow 0$, $P[k] \leftarrow k$ for all k . Set counter variable $c \leftarrow 0$ and $b \leftarrow true$.
R2. [Copy.] Copy entries $a_{kP[k]}$ for $1 \leq k \leq n$. If b is true, attach plus sign to copied entries. Otherwise attach minus sign. Set $c \leftarrow c + 1$ and $j \leftarrow n - 1$. If $c = n!$ terminate the entire process.
R3. [Compute the reverse point.] Execute modulus operation, $c \% F[n - j + 1]$. If the result is 0, set $j \leftarrow j - 1$ and repeat the modulus operation, otherwise terminates this step.
R4. [Get plus/minus sign.] Set $b \leftarrow not b$ if $\sum_{m=1}^{n-j} m$ is odd number. (This sum is equivalent to the number of interchanges from current permutation to next one. The number corresponds to the sign transition caused by reversion.)
R5. [Reverse a portion of the index array.] From $P[j]$ to $P[n]$, reverse the values of elements. (For example, if the values from $P[j]$ to $P[n]$ are 132465, the values of corresponding array cells become 564231 after reversion.) Go to R2 .

Table 3: Reversing Algorithm

5.2 The analysis of Partial Reversion Algorithm

To evaluate the performance of *Partial Reversion Algorithm*, $P(\text{algorithm})$ and $S(\text{algorithm})$ in (14) has to be determined. **R3** makes the time for each permuting vary according to the counter value. Considering this, the execution time function in (14) is modified into

$$T(n^2) = n \cdot n! + 1 + \sum_{p=2}^n \frac{n!}{p(p-2)!} (2p \cdot (p-1)) + (n-1) \cdot n! + n!, \quad (15)$$

where $n \cdot n!$ signifies the time for copying all the entries, $\frac{n!}{p(p-2)!}$ the number of permutations requiring same modulus operations in **R3**, $2p$ the number of copying process when reversing a portion of array in **R5**, $(p-1)$ the number of divisions that determines reversing point in **R3**, $(n-1) \cdot n!$ the total number of multiplication signs generation between entries. The last term $n!$ is for the time to determine plus/minus signs of products and the reason of its simplicity is that the summing process of **R4** can be replaced by another reference table of integers. This time function is valid for $n \geq 4$. Computing this time function results in the estimation below.

$$\begin{aligned} T(n^2) &= n \cdot n! + 1 + 2n! + 4n! + 3n! \\ &+ \dots + 2n(n-1)(n-2)(n-3)^2 \\ &+ 2n(n-1)(n-2)^2 + 2n(n-1)^2 + (n-1) \cdot n! + n! \\ &\simeq n \cdot n! + 1 + \text{constant} \cdot n! + (n-1) \cdot n! + n! \simeq n \cdot n! \end{aligned} \quad (16)$$

As seen in (16), the proposed algorithm does not have the term which is more influential than $n \cdot n!$. It means that generation of permutations by this algorithm takes smaller amount of time compared to $n \cdot n!$ copying processes. $n \cdot n!$ remains same as the most significant term of the time function and this justifies the use of *Partial Reversion Algorithm*. Table 4 illustrates how this algorithm works for 4th order matrix. The factorial column in Table 4 is the transition of the factorial values, with which the result of *counter % factorial* becomes zero. $1!$ appears $12 = \frac{4!}{2 \cdot 0!}$ times, $2!$ does $8 = \frac{4!}{3 \cdot 1!}$ times, $3!$ does $3 = \frac{4!}{4 \cdot 2!}$ times and $4!$ does just once. The sequences in Figure 1 illustrates the formation of general term, $\frac{n!}{p(p-2)!}$ for factorial appearances in (15).

As each entry for a product is copied one by one, it is possible to check the entry is zero or one. With zero or one detection, copying process will be skipped or shortened. As a result, the execution time decreases for sparse matrices. Thus, the execution time function with the proposed algorithm can be expressed in Big-

permutation	reverse point	counter	factorial	sign sum
1234	3	1	1!	1
1243	2	2	2!	3
1342	3	3	1!	1
1324	2	4	2!	3
1423	3	5	1!	1
1432	1	6	3!	6
2341	3	7	1!	1
2314	2	8	2!	3
2413	3	9	1!	1
2431	2	10	2!	3
2134	3	11	1!	1
2143	1	12	3!	6
3412	3	13	1!	1
3421	2	14	2!	3
3124	3	15	1!	1
3142	2	16	2!	3
3241	3	17	1!	1
3214	1	18	3!	6
4123	3	19	1!	1
4132	2	20	2!	3
4231	3	21	1!	1
4213	2	22	2!	3
4312	3	23	1!	1
4321	0	24	4!	10

Table 4: The variable transitions for 4th order matrix

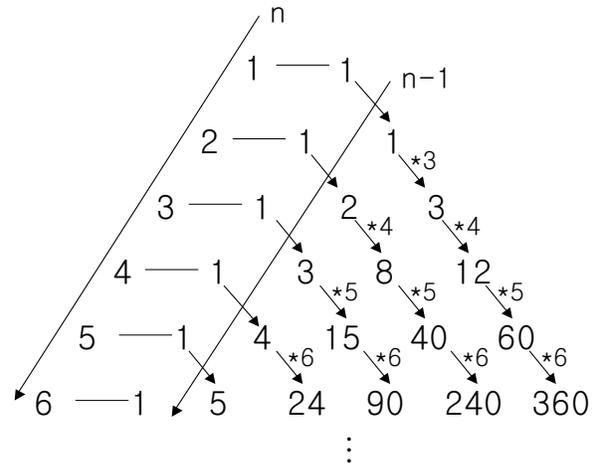


Figure 1: Sequences in triangle for $\frac{n!}{p(p-2)!}$

Oh notation as follows.

$$T(n^2) = O(n \cdot n!) \quad (17)$$

The idea of this algorithm is just same with making tree diagram. Ordinary tree diagram of integer set starts with placing the smallest number from integer set in the first place, putting the second smallest number in the second place and so on. This iterating enables tree diagram to make all the number of cases. *Partial Reversion Algorithm* works almost same way. The dividing counter by factorial value is for judging whether to include new number in the set of numbers to be reversed. For example, after visiting 1234 and 1243, it becomes impossible to make new cases from $12i_2i_1$ where i_n implies a movable number because $2 = 2!$ is maximum number of permutations by two integers. Thus,

one more movable number has to be introduced and 2 becomes the new number, i_3 . Then, new permutations can be made from $i_3i_2i_1$. To avoid redundancy, the numbers ranging from the new movable number to the last number has to be reversed first. Reversing a portion of array works just like traversing backwards in the tree diagram without visiting the number on the way back. When no new case is left in the branch, pointing index goes backwards until it find the right place to start again. Pointing index starts the traversing process with the new number which it did not meet at specific location before. Figure 2 shows the analogy between *Partial Reversion Algorithm* and the tree diagram by the simple set of 3 integer numbers.

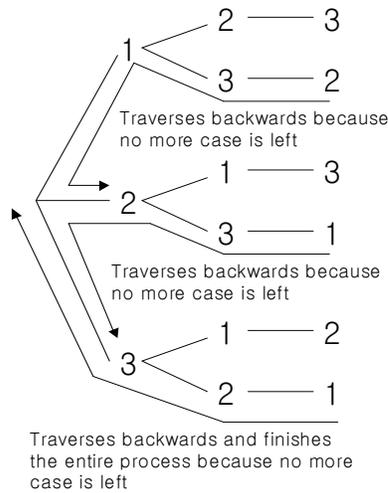


Figure 2: Traversing the tree diagram backwards

6 CONCLUSION

In this paper, the conventional methods for determinant computation are reconsidered in viewpoint of symbolic computation. Among three feasible methods, the permutation method is justified as the most suitable one for symbolic computation because it produces a determinant without fractions and its speed is very decent. To implement the permutation method, *Partial Reversion Algorithm* that generates permutations is proposed. It has reasonable speed for practical use.

The proposed algorithm for generating permutations has the edge over the other permutation generating algorithms. Unlike the existing algorithms, it has no redundancy and do not have to manipulate on auxiliary array/table. However, it definitely has trade-off. *Partial Reversion Algorithm* employs the factorial values as criteria for determining the range of reversion and the fact puts a limit to the order of matrix that can be computed by this method. As this algorithm is op-

timized for the non-sparse matrix whose order is less than 13, a different approach is required to compute the non-sparse symbolic matrix determinant whose order is greater than 12.

For the sparse matrices, it is possible to consider only non-zero entries to generate permutations. Then the algorithm can be applied to high-ordered sparse matrices with the criteria composed of less number of factorial values. The future works regarding symbolic determinant will target overcoming the constraint of this method regarding the order. At the same time, handling high-ordered sparse matrices with less number of factorial values will be studied for the improvement of *Partial Reversion Algorithm*.

REFERENCES

- [1] Martin Braun, *Differential Equations and Their Applications*, Springer-Verlag, 4th edition, 1993.
- [2] Erwin Kreyszig, *Advanced Engineering Mathematics*, John Wiley and Sons, 6th edition, 1988.
- [3] William H. Press/Brian P. Flannery/Saul A. Teulsky/William T. Vetterling, *Numerical Recipes, the art of scientific computing*, vol. 1, Cambridge University Press, 1st edition, 1986.
- [4] Dominique Roelants van Baronaigien, "A multi-stack method for the fast generation of permutations with minimal length increasing subsequences," *Information Processing Letters*, vol. 69, no. 3, pp. 123–126, February 1999.
- [5] Donald E. Knuth, *The art of computer programming, Seminumerical Algorithms*, vol. 2, Addison-Wesley, 3th edition, 1998.
- [6] Donald E. Knuth, *The art of computer programming*, vol. 4, unpublished, preview edition, 2002, online preview version.