ON THE SHUFFLE-EXCHANGE NETWORK AS A PERMUTATIONS GENERATOR

by

Tuvi Etzion

Technical Report #489
January 1988
On the Shuffle-Exchange Network
as a Permutations Generator

Tuvi Etzion

ABSTRACT

This paper considers the shuffle-exchange network as a permutations generator. We present a proof that the $N!$ permutations can be generated using $N!$ passes in the network. We give efficient algorithms for generating a few interesting classes of permutations, namely bit-permute permutations, bit-permute-complement permutations, row cyclic shifts, and classes of derangements, in the minimal passes which are possible to generate these classes of permutations in the network.

* Computer Science Department, Technion, Israel Institute of Technology, Haifa, ISRAEL.

Name and mailing address for correspondence:

Tuvi Etzion
Dept. of Computer Science
Technion - Israel Institute of Technology
Haifa 32000
ISRAEL
1. INTRODUCTION

The shuffle-exchange (SE) network is an efficient tool for implementing various types of parallel processes [12],[17]. The SE network is composed of \( N = 2^n \) processors, where each processor is represented by a binary \( n \)-tuple \((x_1, x_2, \ldots, x_n)\). In the SHUFFLE-operation processor \((x_1, x_2, \ldots, x_n)\) transfers information to processor \((x_2, \ldots, x_n, x_1)\). In the EXCHANGE-operation processors \((x_1, x_2, \ldots, x_n, 0)\) and \((x_1, x_2, \ldots, x_{n-1}, 1)\) may exchange information, independent of other pairs of this form.

One SHUFFLE followed by one EXCHANGE is called a pass. Between the SHUFFLE phase and the EXCHANGE phase of a pass there is a computational phase during which the active pairs of the upcoming EXCHANGE are determined. Prior to the first pass there is normally a preprocessing stage. The overall procedure consisting of the preprocessing stage and all the passes is often referred to as the routing algorithm.

An important problem in this context is the design of efficient routing algorithms that implement permutations in a SE network in a minimal number of passes. In general, a permutation in a SE network associates with each processor a destination processor for the purpose of information transfer.

In this paper we consider the following problem of generating permutations in the SE network. Given a class of permutations (or all the permutations), generate all the permutations of the class in minimal number of passes. The problem of generating all the permutations in a sequential algorithm was handled by many authors and a survey can be found in [16].

In Section 2 we give two different representations to the problem. The first using matrix representation was first presented by Linial and Tarsi[13], and the second uses some graph theory approach is first presented here. We prove that a solution to problem of generating all the \( N! \) permutations can be given using exactly \( N! \) passes.

In Section 3, 4, and 5, we give a routing algorithm using \( O(n) \) or less steps per pass to generate all the bit-permute permutations, bit-permute-complement permutations, and row cyclic shifts, respectively, in the minimal number of passes which is required to generate each of these classes of permutations.

In Section 6 we give a routing algorithm using \( O(1) \) steps per pass to generate a special class of derangements.
2. Representation of the problem

In this section we give a proof that all the \( N! \) permutations can be generated in the SE network using exactly \( N! \) passes. To generate a permutation \( P(0), P(1), \ldots, P(N-1) \) we have to transfer the information from each processor \( i \), \( 0 \leq i \leq N-1 \), to its destination processor \( P(i) \). There are a few approaches to represent the problem of generating permutations in the SE network. We will use a matrix approach and graph theory approach. First we give the matrix approach[13].

A \( 0-1 \) matrix \( A \), of order \( N \times n, N=2^m, m \geq n \), is said to be balanced if all the rows in any \( n \) consecutive columns of \( A \) are distinct. The standard matrix is an \( N \times n \) matrix \( D \) whose \( i \)-th row is the base-2 representation of \( i, 0 \leq i \leq N-1 \).

In terms of these definitions our problem can be stated as follows: Generate a balanced \( 0-1 \) matrix \( A \) of order \( N \times (N!+1) \) such that the first \( n \) columns form the standard matrix \( D \) and all the other \( N! \) consecutive \( n \) columns are distinct. Hence, each \( n \) consecutive columns represent a distinct permutation. The \( N \times n \) matrix which corresponds to an arbitrary permutation will be called the destination matrix. Assume that in one of the rows of \( A \) we have the following \( n+1 \) consecutive bits: \( x_1, x_2, \ldots, x_n, x_{n+1} \). In the SE network the implementation is that after the SHUFFLE phase in which processor \( (x_1, x_2, \ldots, x_n) \) transfers its information to processor \( (x_2, \ldots, x_n, x_1) \) then processor \( (x_2, \ldots, x_n, x_1) \) performs EXCHANGE if and only if \( x_1 \neq x_{n+1} \).

The problem can be represented using graph theory approach. This representation can be considered for every even \( N \). In this case, we have \( N \) processors numbered from 0 to \( N-1 \). In the SHUFFLE operation processor \( i \) transfers information to processor \( 2i \) if \( 0 \leq i \leq \frac{N}{2} - 1 \), and to processor \( 2i+1 \) modulo \( N \) if \( \frac{N}{2} \leq i \leq N-1 \). In the EXCHANGE operation, for each even \( i \), processors \( i \) and \( i+1 \) may exchange information, independent of the other pairs of this form. For more information about the SHUFFLE operation in this case the reader is referred to [4]. Now, for each even \( N \) we associate a graph \( G(N) \). The graph has \( N! \) vertices each one represented by a permutation of the integers \( 0,1,\ldots,N-1 \). The graph has \( \frac{N}{2}N! \) directed edges. From vertex \( v \) there is a directed edge to vertex \( u \) if the permutation \( u \) can be obtained from \( v \) by one pass. Clearly the in-degree and the out-degree of each vertex is \( \frac{N}{2} \). Each path of \( k \) vertices in this graph represents a sequence of \( k-1 \) distinct permutations which begins in another permutation which is the first vertex of the path, and each permutation was obtained by one pass from the permutation which is represented by previous vertex. A cycle of length \( k \) represents a sequence of \( k \) distinct permutations which begins and ends with the
same permutation.

This graph has a generalized Buddy property. We say that a graph $G$ has the generalized Buddy property if when two vertices have a common son, all their sons are identical. This is a generalization of the Buddy property which is defined in [1].

An immediate result from the fact that every permutation can be realized in the SE network[13],[17] is the following lemma.

Lemma 2.1: $G(N)$ is a strongly connected graph.

Theorem 2.2: Let $C_1$ and $C_2$ be two cycles in a graph $G$ with the generalized Buddy property. If there is an edge from a vertex on $C_1$ to a vertex on $C_2$, then there is a cycle in $G$ which contains all the vertices of $C_1$ and $C_2$.

Proof: Let $v_1$ and $u_1$ be two consecutive vertices on $C_1$ ($v_1 \rightarrow u_1$), and $v_2$ and $u_2$ two consecutive vertices on $C_2$ ($v_2 \rightarrow u_2$). Furthermore there is a directed edge from $v_1$ to $u_2$. Hence by the generalized Buddy property there is an edge from $v_2$ to $u_1$. Therefore by taking the edges $v_1 \rightarrow u_2$ and $v_2 \rightarrow u_1$ (instead of $v_1 \rightarrow u_1$ and $v_2 \rightarrow u_2$) and all the other edges of $C_1$ and $C_2$, we obtain a cycle $C_3$ which contains all the vertices of $C_1$ and $C_2$. Q.E.D.

A factor in a directed graph is a set of vertex disjoint directed cycles which includes all the vertices of the graph. Using Lemma 2.1 and Theorem 2.2 we can form a hamiltonian cycle in $G(N)$. Initially we take a factor in $G(N)$ with many short cycles, e.g., this factor can be the one defined by the property that $X$ and $Y$ are on the same cycle if and only if we can realize the permutation represented by $Y$ only with SHUFFLES from the permutation represented by $X$. Using Lemma 2.1 and Theorem 2.2 we can join two cycles together and apply this process until we join all the cycles into a hamiltonian cycle. Hence we have the following theorem.

Theorem 2.3: All the $N!$ permutations can be generated in the SE network using $N!$ passes.

Another result which follows from the above description is the following theorem.

Theorem 2.4: If $G$ is a directed graph with the generalized Buddy property then there is a hamiltonian cycle in $G$.

It is still an open problem to generate all the $N!$ permutations in the SE network using an efficient routing algorithm.
3. Bit-permute permutations

A bit-permute permutation is a permutation with a destination matrix \( D(p_1) \ D(p_2) \ldots \ D(p_n) \), where \( p_1, p_2, \ldots, p_n \) is a permutation of \( 1, 2, \ldots, n \), and \( D(i) \) is the \( i \)-th column of \( D \). This class of permutations was considered in [8] for efficient algorithm to generate arbitrary permutations from this class.

**Lemma 3.1** [13]: Let \( A \) be a matrix of order \( N \times n \), and let \( T \) be a nonsingular matrix of order \( n \). Then \( AT \) is balanced if and only if \( A \) is balanced.

In order to generate all the bit-permute permutations we are going to consider a solution for generating all the \((n-1)\)! permutations of \( n-1 \) elements using an algorithm based on adjacent exchanges. An algorithm for generating permutations is based on adjacent exchanges if between two consecutive permutations which are generated, only two adjacent elements were swapped. There are several algorithms which are based on adjacent exchanges ([3],[5],[6],[11],[18]).

Given an algorithm for generating permutations based on adjacent exchanges. We will call this algorithm "perm" and since our permutations are on \( n \) elements we add the element \( n \) to the end of each permutation generated by "perm". We will generate a balanced matrix \( A \) such that for each permutation \( p(1), p(2), \ldots, p(n) \) of the integers \( 1, 2, \ldots, n \) there exist \( n \) consecutive columns of \( A \) of the form \( D(p(1)), D(p(2)), \ldots, D(p(n)) \). Given the permutation \( p(1), p(2), \ldots, p(n-1), n \) we will generate an \( N \times (2n-1) \) balanced matrix such that each \( n \) consecutive columns corresponds to a distinct cyclic shift of \( D(p(1)), \ldots, D(p(n-1)), D(n) \). The first permutation of "perm" is \( 1, 2, \ldots, n \) and hence the first \( 2n-1 \) columns of \( A \) are \( D(1), D(2), \ldots, D(n), D(1), D(2), \ldots, D(n-1) \). Now assume that the \( n \) cyclic shifts of the permutation \( p(1), p(2), \ldots, p(n-1), n \) are represented by the \( N \times (2n-1) \) balanced matrix \( M_1 = [D(Y_1) \ldots D(Y_n), D(Y_1) \ldots D(Y_n)] \). Let \( All = D(1)+D(2)+\ldots+D(n) \) and assume that the algorithm "perm" exchanges \( p(k) \) with \( p(k+1) \) which correspond to an adjacent pair(cyclicly) in the \( Y_i \)’s. Then we distinguish between two cases:

**Case 1:** \( Y_j = p(k) \) for \( j \neq n-1 \). In this case we generate the \( N \times 2n \) matrix, \( M_2 = [All \ D(Y_1) \ D(Y_n), D(Y_1) \ D(Y_n), D(Y_1) \ D(Y_2) \ D(Y_n) \ D(Y_1) \ D(Y_2) \ D(Y_n) \ D(Y_1) \ D(Y_2) \ D(Y_n)] \).

**Case 2:** \( Y_{n-1} = p(k) \). In this case we generate the \( N \times 2n \) matrix, \( M_2 = [All \ D(Y_1) \ D(Y_n), D(Y_1) \ D(Y_n), D(Y_1) \ D(Y_n) \ D(Y_1) \ D(Y_n)] \).

It is easy to verify that in both cases \( M_2 \) contains the \( n \) cyclic shifts of \( D(p(1)), D(p(2)), \ldots, D(p(n-1)), D(n) \), and using Lemma 3.1 we also have that the \( N \times (4n-1) \) matrix...
Based on this discussion we propose a procedure to realize all the bit-permute permutations. In this procedure each processor has at each stage the following information:

1. An n-tuple \( S = (s(1), \ldots, s(n)) \) whose initial value represents the ID of the said processor. In the SHUFFLE and EXCHANGE operations that follow each processor transfers its current \( S \) and receives a new value for \( S \).

2. A variable, \( \text{parity} \) whose value represents the parity of the sum of the bits of \( S \), i.e., \( \text{parity} = s(1) + s(2) + \cdots + s(n) \mod 2 \).

3. The permutation \( P = (p(1), \ldots, p(n-1)) \) whose value represents the current permutation by the algorithm "perm", and a variable \( p(n) = n \).

4. Two variables, \( f\text{pointer} \) and \( n\text{fpointer} \) whose values represent the index for which \( f(\text{pointer}) = Y_1 \), and \( n\text{fpointer} \) will be the new value of \( f\text{pointer} \). Hence, all the computations which involve addition to these two variables will be performed modulo \( n \) (the result will be between 1 and \( n \)).

Procedure 2:

Part 1

\{the first five lines are initialization in order that the first permutation of "perm" will be 1,2,\ldots,n \}

\( p(1) := 2; \)
\( p(2) := 1; \)
for \( i := 3 \) to \( n \) do \( p(i) := i; \)

\( f\text{pointer} := 1; \)
\( n\text{fpointer} := 1; \)

Part 2

for \( j := 1 \) to \( (n-1)! \) do

begin

for \( i := 1 \) to \( n \) do SHUFFLE; \{generating the columns \( D(Y_1), \ldots, D(Y_{n-1}) \) and one more SHUFFLE\}

if \( s(p(f\text{pointer}+n-1)) \neq \text{parity} \) then EXCHANGE; \{comparing the column \( D(Y_1) \) with the column \( A[Y_i] \)\}

\{the next four lines perform the adjacent exchange of the previous step of the algorithm "perm"\}

\( \text{temp} := p(k); \)
\( p(k) := p(k+1); \)
\( p(k+1) := \text{temp}; \)
Compute the $k$ for which $p(k)$ and $p(k+1)$ should be swapped by the algorithm "perm".

if $k \neq fpointer + n - 1$ then $\{ Y_j = p(k) \text{ for } j \neq n - 1 \}$

begin
  $nfpointer := fpointer - 1$; \{ since the column $D(Y_n)$ is generated after the column All \}
  $i := 0$;
  while $i < n - 1$ do
    begin
      SHUFFLE;
      if $(nfpointer + i) = k$ then \{ $D(Y_i)$ was generated \}
      begin
        SHUFFLE;
        $i := i + 1$;
        if $i \neq n - 1$ then
          begin
            if $s(fpointer + i) \neq s(nfpointer + i - 1)$ then EXCHANGE; \{ comparing the column $D(Y_{i+1})$ with the column $D(Y_{i-1})$ \}
            end
          else if parity $\neq s(p(nfpointer + n - 2))$ then EXCHANGE; \{ comparing the column $All$ with the column $D(Y_{n-2})$ since $Y_{n-2}$ and $Y_{n-1}$ are $p(k)$ and $p(k+1)$ \}
          end
        else if $s(fpointer + i) \neq s(nfpointer + i)$ then EXCHANGE; \{ comparing the column $D(Y_{i+1})$ with the column $D(Y_i)$ \}
        $i := i + 1$;
      end;
    end;
  if $i = n - 1$ then \{ the column $D(Y_{n-1})$ was not generated \}
  begin
    SHUFFLE;
    if parity $\neq s(p(nfpointer + n - 1))$ then EXCHANGE; \{ comparing the column $All$ with the column $D(Y_{n-1})$ \}
  end;
end
else begin \( Y_{n-1} = p(k) \)

\quad nfpointer := nfpointer; \{the column \( D(Y_i) \) will be generated after the column \text{All} \}

\quad for \( i := 1 \) to \( n-1 \) do SHUFFLE; \{generating the columns \( D(Y_1), \ldots, D(Y_{n-2}) \) and one more SHUFFLE\}

\quad if \( s(p(fpointer+n-2)) \neq s(p(fpointer+n-1)) \) then EXCHANGE; \{comparing the column \( D(Y_{n-1}) \)
\quad with the column \( D(Y_n) \)\}

\quad SHUFFLE;

\quad if \( parity \neq s(p(nfpointer+n-2)) \) then EXCHANGE; \{comparing the column \text{All}
\quad with the column \( D(Y_{n-1}) \)\}

end;

end;

**Theorem 3.2:** (a) Procedure BP generates the class bit-permute permutations in \( 2n! \) passes and \( O(1) \) steps per pass in average.

(b) All the bit-permute permutations cannot be generated in less than \( 2n! \) passes.

**Proof:** (a) The correctness of the procedure follows immediately from the discussion in Section 1, the discussion before the procedure, and the comments in the procedure. The claimed complexity of Procedure BP is obtained as follows. The complexity of the next permutation in the algorithm "perm" is at most \( O(n) \). Between the computation of two permutations in the algorithm "perm", we perform \( 2n \) passes in the SE network. It is easy to verify that the number of the other operations in a pass is constant. Thus there are \( O(1) \) steps per pass in average.

(b) This is an immediate result from the observation that after \( n \) SHUFFLES without EXCHANGE we are in the first permutation and a column different from \( D(1), D(2), \ldots, D(n) \), must be in the balanced matrix which implies that \( n \) permutations which are not bit-permute permutations must be generated.

Q.E.D.
4. Bit-permute-complement permutations

A *bit-permute-complement permutation* is a permutation with a destination matrix of the form
\[
[D(p_1), b_1] [D(p_2), b_2] \ldots [D(p_n), b_n],
\]
where \( p_1, p_2, \ldots, p_n \) is a permutation of \( 1, 2, \ldots, n \), \( b_i \in \{0, 1\} \), \( 1 \leq i \leq n \), and
if \( b_i = 0 \) then \( D(p_i) \) is taken and if \( b_i = 1 \) then the binary complement of \( D(p_i) \) is taken. It can be easily verified that there are \( n! 2^n \) bit-permute-complement permutations. This class of permutations was considered by
many authors (see for example [14],[15],[19]).

In our method of generating all the bit-permute-complement permutations we will use again the algorithm "perm" for generating \((n-1)! \) permutations using adjacent exchanges. Given a permutation \( p(1), \ldots, p(n-1), n \) of the integers \( 1, 2, \ldots, n \), we will generate all the cyclic shifts of
\[
[D(p(1)), b_1], \ldots, [D(p(n-1)), b_{n-1}], [D(n), b_n]
\]
for all the possible \( 2^n \) distinct combinations of the \( b_i \)'s. Hence, for each permutation \( p(1), \ldots, p(n-1) \) of the algorithm "perm", we will generate \( n 2^n \) bit-permute-complement permutations.

For each permutation \( \Pi = p(1), p(2), \ldots, p(n-1), n \) of the integers \( 1, 2, \ldots, n \) we associate a graph \( G(\Pi) \). The graph has \( n 2^n \) vertices each one represented by an \( n \)-tuple \((x_1, b_1), (x_2, b_2), \ldots, (x_n, b_n)\), where \( x_1, x_2, \ldots, x_n \) is some cyclic shift of \( p(1), p(2), \ldots, p(n-1), n \), and \( b_i \in \{0, 1\}, 1 \leq i \leq n \). The graph has \( n 2^{n+1} \) directed edges. From vertex \( v=((v_1, b_1), (v_2, b_2), \ldots, (v_n, b_n)) \) there are two directed edges, one to vertex
\[
((v_2, b_2), (v_3, b_3), \ldots, (v_n, b_n), (v_1, 0)),
\]
and a second edge to the vertex \((v_2, b_2), (v_3, b_3), \ldots, (v_n, b_n), (v_1, 1))\). Clearly the in-degree and the out-degree of each vertex is two, and \( G(\Pi) \) has the generalized buddy property. By Theorem 2.4 there is a hamiltonian cycle in this graph. The order of the vertices in this hamiltonian cycle will
determine when we are taking the complement of the column from \( D \) and when we are taking the column
without complementing.

The *companion* \( X' \) of a vertex \( X=((x_1, b_1), (x_2, b_2), \ldots, (x_{n-1}, b_{n-1}), (x_n, b_n)) \) is defined by

\[
X'=((x_1, b_1), (x_2, b_2), \ldots, (x_{n-1}, b_{n-1}), (x_n, \overline{b}_n))
\]

where \( \overline{y} \) denotes the binary complement of \( y \).

The *necklaces factor* (NF) is a factor of \( G(\Pi) \) which is defined by the following property:
\( X=((x_1, c_1), (x_2, c_2), \ldots, (x_n, c_n)) \) and \( Y=((y_1, c_1), (y_2, c_2), \ldots, (y_n, c_n)) \) are on the same NF-cycle iff \( X \) is a cyclic
shift of \( Y \).

The *weight* of a vertex \( X=((x_1, b_1), (x_2, b_2), \ldots, (x_n, b_n)) \) is the number of ONES in \( B=(b_1, b_2, \ldots, b_n) \).
The weight of a cycle from NF is the the weight of each of its vertices.
Lemma 4.1: Let $C_1$ be a cycle of weight $k > 0$ from NF. Then there exists a vertex $X$ on $C_1$ such that its companion $X'$ is on a cycle $C_2$ whose weight is $k - 1$.

Proof: Since $W(C_1) > 0$ there exists a vertex of the form $X = ((x_1, b_1), (x_2, b_2), \ldots, (x_{n-1}, b_{n-1}), (x_n, 1))$ on $C_1$. Hence $X' = ((x_1, b_1), (x_2, b_2), \ldots, (x_{n-1}, b_{n-1}), (x_n, 0))$ and $W(X') = W(X) - 1 = k - 1$. Therefore $X'$ is on an NF-cycle $C_2$, with $W(C_2) = k - 1$.

Q.E.D.

Lemma 4.1 and Theorem 2.2 lead to a simple method of constructing a Hamiltonian cycle in $G(I)$. At each step we have a main cycle, obtained by joining a subset of NF-cycles, and the remaining NF-cycles. Initially the main cycle is chosen to be the unique NF-cycle of weight zero. Next, the main cycle is extended by joining to it all the cycles of weight one. In general step $i$, we extend the main cycle by joining to it all the NF-cycles of weight $i$ (in arbitrary order). This is always possible because the current main cycle contains all the vertices whose weight is less than $i$ and, since each NF-cycle of weight $i > 0$ has a vertex ending with $b_i = 1$, it can be joined (see Theorem 2.2 and Lemma 4.1) to the current main cycle. This procedure ends when all the NF-cycles have been joined together. A join of an NF-cycle to the main cycle is performed by means of a pair of companion vertices $X$ and $X'$, with $X$ on the next NF-cycle $C$ in line and $X'$ on the current main cycle. The vertices $X$ and $X'$ are called the bridging vertices of the join. The bridging vertex $X = ((x_1, b_1), \ldots, (x_{n-1}, b_{n-1}), (x_n, b_n))$ on $C$ is determined in a way such that $b_n = 1$ and $x_n$ is greater than the other $x_i$'s for which $b_i = 1$. It is easy to verify that by this definition a unique vertex is determined on each NF-cycle with weight greater than zero. This construction is similar to the one of constructing de Bruijn sequences [1,[2].

The formal steps for generating the Hamiltonian cycle are:

Construction A: Let $\alpha_0 = ((a_0, 0), (a_1, 0), \ldots, (a_{n-1}, 0))$ be a vertex in $G(I)$. Given $\alpha_i = ((a_i, b_i), (a_{i+1}, b_{i+1}), \ldots, (a_{i+n-1}, b_{i+n-1}))$ proceed to produce $\alpha_{i+1} = ((a_{i+1}, b_{i+1}), (a_{i+2}, b_{i+2}), \ldots, (a_{i+n}, b_{i+n}))$, as follows: If the largest $a_j, i + 1 \leq j \leq i + n - 1$, such that $b_j = 1$ is smaller than $a_i$ (or if for each $j$ in the range $b_j = 0$) set $a_{i+n} = a_i$ and $b_{i+n} = b_i$ else set $a_{i+n} = a_i$ and $b_{i+n} = b_i$.

Example 4.2: For $n = 3$ and $I = 1, 2, 3$ we generate the Hamiltonian cycle $123123123123123123123123$ where each three consecutive digits corresponds to a vertex, $(a, 0)$ was replaced by $a$, and $(a, 1)$ was replaced by $\bar{a}$.

Lemma 4.3: If the Hamiltonian cycle in $G(I)$ which was constructed by Construction A, begins with the vertex $((x_1, 0), \ldots, (x_{n-1}, 0), (x_n, 0))$ then it ends with the vertex $((x_n, 1), (x_1, 0), \ldots, (x_{n-1}, 0))$. 
Proof: Since the hamiltonian cycle begins with the vertex \((x_1,0), \ldots, (x_{n-1},0), (x_n,0))\) it should end with either the vertex \((x_n,0), (x_1,0), \ldots, (x_{n-1},0)\) or the vertex \((x_n,1), (x_1,0), \ldots, (x_{n-1},0)\). By Construction A the vertex \((x_n,0), (x_1,0), \ldots, (x_{n-1},0)\) should be followed by the vertex \((x_1,0), \ldots, (x_{n-1},0), (x_n,1)\) and the vertex \((x_n,1), (x_1,0), \ldots, (x_{n-1},0)\) should be followed by the vertex \((x_1,0), \ldots, (x_{n-1},0), (x_n,0)\).

Q.E.D.

It follows from Lemma 4.3 that when the algorithm "perm" generates a new permutation we will have in the balanced matrix the columns \(D(Y_n), D(Y_1), \ldots, D(Y_{n-1}), 0\), and the next \(n\) columns will be the same as in Procedure BP (see case 1 and case 2 before Procedure BP).

Now, the algorithm and its complexity depends on the amount of storage which is allocated to each processor. If the amount of storage is \(O(N)\) then since the structure of the hamiltonian cycle can be the same for all the permutations, the cycle can be computed only once. We will assume storage of \(O(n)=O(\log N)\) and hence each processor will have only about \(n\) digits of the cycle of Construction A and each time a new digit will be computed and an old digit will be erased.

In Procedure BPC given below we are using some ideas from Procedure BP, and some new ideas which correspond to generating the hamiltonian cycle in \(G(\Pi)\) for each permutation \(\Pi=(p(1), \ldots, p(n-1), n)\). We will give comments to the lines which corresponds to generating the hamiltonian cycles, and the corresponding columns. Each processor has at each stage the same information as in procedure BP, and also \(O(n)\) variables for implementing the \(a_i\)'s and the \(b_i\)'s of Construction A.

Procedure BPC:

Part 1

\[
p(1):=2; \\
p(2):=1; \\
\text{for } i :=3 \text{ to } n \text{ do } p(i):=i; \\
\text{fpointer}:=1; \\
\text{nfpointer}:=1;
\]

Part 2

\[
\text{for } j :=1 \text{ to } (n-1)! \text{ do} \\
\begin{array}{l}
\text{begin} \\
\text{for } i :=1 \text{ to } n \text{ do } b_1:=0; \\
\text{for } i :=1 \text{ to } n \text{ do } a_i:=p(\text{nfpointer } +i-1); \quad \text{(note that the } b_1 \text{'s and the } a_i \text{'s can be implemented}
\end{array}
\]
using only $O(n)$ variables}

[the next four lines generates all the columns which correspond to all the cyclic shifts with all the possible complements of the previous $n$ columns]

for $i := 1$ to $n 2^n - 1$ do

begin

SHUFFLE;

If the largest $a_j$, $i+1 \leq j \leq i+n-1$, such that $b_j = 1$ is smaller than $a_i$ (or for each $j$ in the range $b_j = 0$) set $a_{i+n} = a_i$ and $b_{i+n} = b_i$ else set $a_{i+n} = a_i$ and $b_{i+n} = b_i$ ;

if $b_i \neq b_i+n$ then EXCHANGE;

end;

SHUFFLE;

if $s(p(fpointer + n - 1)) = parity$ then EXCHANGE; [comparing the column $D(Y_n)$ with the column All]

\[
temp := p(k);
\]

\[
p(k) := p(k + 1);
\]

\[
p(k + 1) := temp;
\]

\[
fpointer := npointer;
\]

Compute the $k$ for which $p(k)$ and $p(k+1)$ should be swapped by the algorithm "perm";

if $k \neq fpointer + n - 1$ then

begin

\[
fpointer := fpointer - 1;
\]

\[
i := 0;
\]

while $i < n - 1$ do

begin

SHUFFLE;

if $(npointer + i) = k$ then

begin

SHUFFLE;

$i := i + 1$;

if $i \neq n - 1$ then

begin

if $s(fpointer + i) \neq s(npointer + i - 1)$ then EXCHANGE;

end;

end;

end;
else if parity \( s (p\text{ (nfpointer +n -2)}) \) then EXCHANGE;
end

else if \( s (p\text{ (fpointer +i)}) \neq s(nfpointer+i) \) then EXCHANGE;
\( i := i + 1; \)
end;

if \( i = n - 1 \) then
begin
SHUFFLE;
if parity \( s (p\text{ (nfpointer +n -1)}) \) then EXCHANGE;
end;
end
else begin
\( nfpointer := fpointer; \)
for \( i := 1 \) to \( n - 1 \) do SHUFFLE;
if \( s (p\text{ (fpointer +n -2)}) \neq s (p\text{ (fpointer +n -1)}) \) then EXCHANGE;
SHUFFLE;
if parity \( s (p\text{ (nfpointer +n -2)}) \) then EXCHANGE;
end;
end;

The proof of the following theorem is similar to the one of Theorem 3.2.

Theorem 4.4: (a) Procedure BPC generates the class of bit-permute-complement permutations in \((2^n + 1)n!\) passes and \(O(n)\) steps per pass.

(b) All the bit-permute-complement permutations cannot be generated in less than \((2^n + 1)n!\) passes.
A permutation is called a \textit{row cyclic shift} if for a given integer \( k \), \( P(i) = P((i+k) \mod N) \), for each \( i \), \( 0 \leq i \leq N-1 \). The meaning is that the destination matrix is obtained from the standard matrix \( D \) by shifting the first \( k \) rows cyclically. Let \( SH(k) \) denote this destination matrix. Using results of Lawrie [12] we have the following lemma.

\textbf{Lemma 5.1:} \( [SH(m) \ SH(k)] \) is a balanced matrix for each \( 0 \leq m, k \leq N-1 \).

\textbf{Corollary 5.2:} Each permutation \( p_0, p_1, ..., p_{N-1} \) of the integers \( 0, 1, ..., N-1 \) induces a balanced matrix \( [D \ SH(p_0) \ SH(p_1) \ ... \ SH(p_{N-1})] \).

In procedure RCS given below we will generate the balanced matrix \( [D \ SH(1) \ SH(2) \ ... \ SH(N-1) \ SH(0)] \). In this procedure each processor has at each stage the following information:

1. An \( n \)-tuple \( S = (s(1), s(2), ..., s(n)) \) whose value after each multiple of \( n \) passes is the ID of the said processor. In the \textit{SHUFFLE} and \textit{EXCHANGE} operations that follow each processor transfers its current \( S \) and receives a new value for \( S \).
2. An \( n \)-tuple \( U = (u(1), u(2), ..., u(n)) \) whose value after each multiple of \( n \) passes is \( S+1 \). In the \textit{SHUFFLE} and \textit{EXCHANGE} operations that follow each processor transfers its current \( U \) and receives a new value for \( U \).

\textbf{Procedure RCS:}

\begin{verbatim}
for j := 1 to N do
begin
  S := ID
  U := S + 1
  for i := 1 to n do
  begin
    if s(i) != u(i) then EXCHANGE;
  end
end;
\end{verbatim}
The following lemma is an immediate observation from the structure of the standard matrix.

**Lemma 5.3:** Given two integers, \( k, 1 \leq k \leq n \), and \( m, 0 \leq m \leq N - 1 \), there is no shift (of rows) of \( k \) consecutive columns of \( D \) which produces the same \( N \times k \) matrix as the one produced by \( m \) shifts (of rows) of the first \( k \) columns of \( D \).

**Corollary 5.4:** For each \( m, n + 1 \leq m \leq 2n - 1 \), there is no \( N \times m \) binary matrix, such that the first \( n \) columns are a row cyclic shift and the last \( n \) columns are another row cyclic shift.

By Lemma 5.3 we can also observe that there are exactly \( N! \) different solutions to generate the class of row cyclic shifts in \( nN \) passes.

**Theorem 5.5:**
(a) Procedure RCS generates the class of row cyclic shifts in \( nN \) passes and \( O(1) \) steps per pass.
(b) All the row cyclic shifts cannot be generated in less than \( nN \) passes.

**Proof:**
(a) An immediate result from Procedure RCS and Corollary 5.2.
(b) An immediate result from Corollary 5.4.

Q.E.D.
6. Derangements

A permutation $p_1, p_2, \ldots, p_k$ is called a derangement if for each $i$, $1 \leq i \leq k$, $p_i \neq i$. For more details on derangements the reader is referred to [9].

Assume that we want to obtain a set of permutations such that each processor has after each pass a different information, or in other words each information will be in a different processor after each pass. Each permutation that we will generate will be a derangement of the other permutations. In the matrix representation those permutations correspond to an $N \times (N+n-1)$ balanced matrix in which each row has the property that each $n$ consecutive bits are different. This imply that each row is really a de Bruijn sequence[2]. Hence we employ the following strategy for the solution. The $N \times (N+n-1)$ balanced matrix consists of all the $N$ shifts of a de Bruijn sequence, where the first $n$ columns are the standard matrix. Initially processor $(x_1, x_2, \ldots, x_n)$ will start to compute the bit which is followed by the bits $x_1, x_2, \ldots, x_n$ in the de Bruijn sequence. If this bit is $x_a$ then no EXCHANGE is needed, if this bit is $x_n$ then EXCHANGE is needed. At all the steps the processor will send its information to the same processor. For generating a de Bruijn sequence we will use the algorithm of Fredricksen[10]. Steps (D1) through (D3) given in Procedure DP below correspond to the problem of finding the previous bit in the de Bruijn sequence from $n$ given bits. This leads to procedure DP for generating a class of derangements. In this procedure, each processor uses the following information:

1. An $n$-tuple $ID = (id_1, id_2, \ldots, id_n)$ which represents the identification of the processor.
2. A variable $doEX$ from which the processor knows if to perform EXCHANGE.

Procedure DP:

(D1) Form $\beta^* = (id_1, id_2, \ldots, id_{n-1}, 1)$.

(D2) Let $M$ be the cyclic shift of $\beta^*$ with the largest base-2 value $|M| = l \cdot 2^r$, $l$ odd. Let $T$ be the cyclic shift of $\beta^*$ such that $|T| = l$.

(D3) If $T = \beta^*$ then $doEX := 1$ else $doEX := 0$;
   for $i := 1$ to $N$ do
     begin
       SHUFFLE;
       if $doEX = 1$ then EXCHANGE;
     end;
The following theorem is an immediate result of Procedure DP and the discussion before the procedure.

**Theorem 6.1:** Procedure DP generates a class of derangements in $2^n$ passes and $O(1)$ steps per pass.

To end this section we note that there are $2^{2^{n-1}}$ de Bruijn sequences of order $n$ [2] which implies that there exists at least $2^{2^{n-1}}$ different solutions to generate a class of $N$ permutations in which, each permutation is a derangement of all the other permutations in the class.
REFERENCES


